Reflective Relational Machines**

Serge Abiteboul* Christos H. Papadimitriou† V.Vianu‡

Abstract

We propose a model of database programming with reflection *(dynamic generation of queries within the host programming language), called the* reflective relational machine*, and characterize the power of this machine in terms of known complexity classes. In particular, the polynomial-time restriction of the reflective relational machine is shown to express PSPACE, and to correspond precisely to uniform circuits of polynomial depth and exponential size. This provides an alternative, logic-based formulation of the uniform circuit model, that may be more convenient for problems naturally formulated in logic terms, and establishes that reflection allows for more "intense" parallelism, which is not attainable otherwise (unless $P = PSPACE$). We also explore the power of the reflective relational machine subject to restrictions on the number of variables used, emphasizing the case of sublinear bounds.*

1 Introduction

A model of computation, called *relational machine*, was introduced in [AV91] to capture standard database computation with a relational query language embedded in a full-fledged programming language, in the spirit of C+SQL. Such programs use a fixed set of relational queries, specified *a priori* in the code. Recently, a mode of programming called "reflection" has made a comeback in programming languages. The basic idea is as old as Universal Turing Machines and early assembly languages: reflection allows the dynamic generation of code. Simple examples in modern programming languages are **eval** in Lisp and **clause** in Prolog. In recent database products, reflection allows the *dynamic* generation of queries within the host programming language. In this paper, we extend the relational machine by allowing the dynamic generation of

*INRIA, B.P. 105, 78153 Le Chesnay CEDEX, France. Serge.Abiteboul@inria.fr. Work supported in part by European Community under the Esprit Fide2 project.

†EECS Department, UC Berkeley, Berkeley CA 94720-1776, christos@cs.berkeley.edu. Work performed while visiting INRIA.

‡CSE 0114, UC San Diego, La Jolla, CA 92093-0114, vianu@cs.ucsd.edu. Work supported in part by the NSF under grant IRI-9221268. ** A preliminary version of this paper appeared in *LICS* 1994.

queries, thus modeling reflective database programming. The results concern the expressive power of the reflective relational machine with time and space restrictions on resources, and with limitations on the number of variables allowed in queries. The reflective relational machine also provides a model of parallel computation analogous to the uniform circuit model, only formulated in logic terms. The close correspondence to circuits, which does not hold for non-reflective machines, suggests that reflection allows to express more "intense" parallelism than non-reflective programming.

Database query languages are based on first-order logic over relations (FO). However, FO itself cannot express simple and useful database queries such as connectivity: this observation has created a dynamic field of research at the interface of Database Theory, Logic, and Complexity (see the surveys [Gu84, Im87, Lei89a]). Control capabilities (such as iteration and fixpoint) were added to logic [Ch81, Var82, Pa85, Im86, AV89], and familiar computational paradigms emerged as a result (Fagin's important characterization of NP [Fa74], although otherwise motivated, also falls in this framework). The relational machine of [AV91] is a more general, and at first inspection more powerful, model of computation, in which the control is provided by a full-fledged Turing machine interacting with a relational store via a logical query language. The relational machine helps address the thorny issue of *order* in computation. Database computations typically manipulate unordered data. On the other hand, all known models of computation assume an ordered domain, or introduce an order via their linear data structures. As a result of this fundamental incompatibility, characterizations of query languages in terms of complexity classes typically depend (in a rather artificial way) on the underlying domain being ordered[1]. Relational machines provide a credible computational model of order-free computation, which can then be related naturally to the expressive power of query languages. In particular, there is a nice match between "relational" complexity classes defined using this model, and query languages: relational polynomial-time (P_r) coincides with *fixpoint*, and relational polynomial space (PSPACE_r) is precisely *while*. It was also shown in [AV91] that important questions concerning the equivalence of query languages with iteration or fixpoint (in particular, inflationary vs. noninflationary fixpoint) coincide with important complexity questions (the P vs. PSPACE problem). Further results of the same flavor are obtained in [AVV97].

Computing devices in the spirit of relational machines have already been investigated by Friedman [Fri71] and Leivant [Lei89b], but with a different focus: Friedman's emphasis is on generalizing recursion theory, whereas Leivant's is on logic characterizations of computational complexity on *ordered* (or *enumerated*) structures. In contrast, the focus on unordered structures lies at the core of our investigation.

Relational machines capture database computation *and programming* in the style of C+SQL. Since the combined program is typically compiled, the database queries in it are fixed. This limitation is reflected in the relational machine's use of a fixed set of

[1]Fagin's Theorem avoids this because in existential second-order logic one can postulate the existence of an order.

queries. In particular, it implies a constant bound on the number of variables, a fact with important consequences on expressive power. For example, relational machines are subsumed by infinitary logic with finitely many variables, $L^{\omega}_{\infty\omega}$ [Ba77], and so have a 0-1 law [KV90] and cannot express certain "simple" queries such as evenness[2].

More recent database programming environments (such as the object-oriented *O2* [BDK92, O2]) do away with this limitation, and allow *programmable queries* constructed *dynamically* by the program, and not explicitly by the programmer. This *reflective* style of database programming seems to be useful and popular among users, for instance for programming user interfaces [Cl93]. Relational machines as originally defined in [AV91] fail to model reflective database programs. *In this paper we extend relational machines so that they interact with the relational store via a writable query tape.* We call such machines *reflective relational machines,* as they were intended to capture the novel reflective style of database programming alluded to above. Our main results characterize the power of such machines in terms of known complexity classes. In particular, we show that polynomial-time bounded reflective relational machines decide precisely the structure classes decidable in PSPACE. In contrast, polynomial-time bounded ordinary relational machines were shown in [AV91] to be a strict subset of P. Interestingly, our result relates reflective relational machines to all of PSPACE without the usual *order* assumption, a feature that makes it more directly relevant to database theory.

Another aspect of interest is the connection between the reflective relational machine and *parallel* models of computation. From a practical viewpoint, the model is of interest because it captures the computational paradigm occurring in increasingly common client/server architectures, where a client workstation interacts with a database server running on a highly parallel machine [DG92, Val93]. Thus, the core of the parallelization consists in the use of the database server. This is in the spirit of the reflective relational machine, where the Turing machine component is sequential, and the relational computation counted as one parallel step.

Since reflective relational machines answer complex relational queries in a single step, they can also be considered as a *uniform* model of parallelism. It turns out that time in the polynomially bounded reflective relational machine coincides with parallel time in the standard uniform circuit model. Thus, the reflective relational machine can be viewed as an alternative, logic-based formulation of the uniform circuit model. We believe that this model is more easily applicable to problems which are naturally formulated in terms of logic, since it circumvents the rather cumbersome translation to the circuit framework.

Seen in terms of parallel complexity, our result relating polynomial-time reflective machines and PSPACE is a little less surprising, as it falls within the tradition of "the second machine class" of [vE90], for which polynomial time always coincides with polynomial space. In terms of the computational style of which it is a model (a computer which occasionally interacts with a very powerful machine via powerful but

[2]The evenness query on a unary relation S is the query *even(S)*= **true** iff $|S|$ is even.

specialized operations), the reflective relational machine is quite close to the vector machines of [PS76]. Technically, and in terms of the kinds of operations used, our model is much closer to Immerman's models of parallelism in terms of iterated logic and inductive definitions [Im89]. (However, the languages considered by Immerman are all subsumed by the *non-reflective* relational machine, so reflection is not captured.) To simulate PSPACE by a relational machine, we simulate by a single query the alternation aspect of PSPACE [CKS81], not unlike the way that Immerman simulates lower-level parallelism [Im89]. For the other direction, the technically hard part is building a kind of "programmable logic array," which can be specialized to circuits computing any polynomially-bounded FO expression that the machine may submit to the relational store.

However, we do show that reflective relational machines are not altogether boring members of the second machine class, by pointing out that for them polynomial space coincides with Turing machine *exponential space* (instead of the expected exponential time, Theorem 4.5).

Finally, we consider the power of reflective machines subject to restrictions on the number of variables used in a computation, as functions of the input. The interesting case is when the bound is "sublinear": above $O(1)$ and below $O(n)$. This is because the $O(1)$ and $O(n)$ cases are two significant extremes: the $O(1)$ bound yields a restricted machine equivalent to the non-reflective relational machine, so reflection becomes purely cosmetic; and, at the other extreme, $O(n)$ variables are sufficient to completely identify the input, yielding devices which can define all recursive properties. We show that the power of sublinear machines lies strictly between the $O(1)$ and $O(n)$ machines, and that there is a hierarchy among them.

We also look at what happens when sublinear variable complexity restrictions are combined with time/space restrictions, and show that there is a mismatch between the classes so definable, and classical complexity classes. Thus, for machines with sublinear variable complexity, one cannot hope to obtain *without order* the kind of exact characterizations described above. In this respect, the behavior of reflective machines with sublinear variable complexity is similar to that of relational machines, and is also caused by the restriction on the number of variables. However, some subtleties arise because the bound on the variables is dynamic. Nonetheless, we manage to extend to sublinear reflective machines a normal form shown for relational machines [AV91], reducing computation over an unordered input to an ordered input. This bridge between computation with and without order provides a key technical tool. For reflective machines, the reduction is done via a EXPTIME computation. This is not as good as for relational machine, where the reduction takes only polynomial time. It remains open whether this can be improved.

There has been little previous formal work on reflection in database languages. Ross introduces in [R92] relational algebra with a limited form of reflection, allowing relations that contain relation names. In [VVV93], relational algebra is augmented with the ability to dynamically generate and evaluate queries encoded in relations,

yielding *Reflective Relational Algebra (RRA)*. The results provide a connection between RRA and relational algebra with bounded looping (introduced in [Ch81]). As a corollary, the data and expression complexity of RRA and of a restricted version of RRA are established.

The paper begins with an informal review of some query languages and of the relational machine. This is followed by a description of the reflective relational machine model, and some basic facts about it. The polynomial time/space restrictions of the machine are considered next in Section 4. The results on sublinear bounds on the number of variables are presented in Section 5.

2 Background

In this section we briefly review several query languages, some basic results on the relational machine, and pebble games for infinitary logic.

FO, *fixpoint*, *while*, and $L^{\omega}_{\infty\omega}$

In databases, only finite structures are considered. Most traditional query languages are based on first-order logic without function symbols (here FO). The simplicity of Codd's algebraization of FO and the fact that FO is in (uniform) AC_0 [Im87] (and, thus, in a reasonable sense, takes constant parallel time) explain the appeal of FO as a query language. However, FO cannot compute simple queries like connectivity of a graph. Intuitively, this is due to the lack of recursion.

Most of the extensions of FO with recursion that have been proposed converge towards two classes of queries, *fixpoint* and *while* [CH82]. The *fixpoint queries* (*fixpoint*) [CH82] are constructed using the first-order constructors as in FO together with a fixpoint operator (μ). The fixpoint operator binds a predicate symbol T that is free and that appears only positively (i.e., under an even number of negations) in the formula. The semantics is given by the least fixpoint of the formula, and convergence is guaranteed in polynomial time. For instance, the transitive closure of relation E is given by the *fixpoint* query $\mu_T\varphi(x, y, T)$ for $\varphi = E(x, y) \vee \exists z(E(x, z) \wedge T(z, y))$. *Fixpoint* expresses exactly P on ordered databases [Im86, Var82]. It cannot, however, express the evenness query. The *while* language was originally introduced in [CH82] in a procedural form. FO is extended with (i) sorted relational variables $(X, Y, ...)$, (ii) assignment of FO queries to variables, and (iii) a *while* construct allowing to iterate a program while some condition (e.g., $X = \emptyset$) holds. An alternative definition of *while* based on *partial fixpoint logic* is proposed in [AV89]. *While* expresses PSPACE on ordered databases [Var82], but, like *fixpoint*, cannot express the evenness query on an unordered set. An elegant proof uses the fact that all properties expressible by *while* have a *0-1 law*, i.e. the asymptotic probability that the property holds exists and is either 0 or 1 [KV87]. It is of interest whether this property holds for more powerful languages. Infinitary logic with finitely many variables is a very powerful extension of

while which still has a 0-1 law [KV90]. This provides insight into the factors that limit expressive power in query languages. Beyond this aspect, $L^{\omega}_{\infty\omega}$ provides an elegant unifying formalism for a wide variety of query languages [ACY91, KV90].

We first define unrestricted infinitary logic. $L_{\infty\omega}$ is first-order logic (FO) extended by allowing disjunctions and conjunctions of infinite sets of formulas.

Definition 2.1 The set of *infinitary formulas* over a database schema[3] σ is the smallest collection $L_{\infty\omega}$ of expressions such that

1. it contains all FO-formulas over σ;
2. if φ is an infinitary formula and x a variable, then $\neg(\varphi), \exists x(\varphi), \forall x(\varphi)$, are also infinitary formulas; and
3. if A is a set of infinitary formulas, then $\bigvee A$ and $\bigwedge A$ are also infinitary formulas.

The semantics is the natural extension of FO semantics. We are particularly interested by infinitary formulas in which the number of variables is finite. (Each variable may occur an unbounded number of times in the formula.) This logic has a 0-1 law, whereas the unrestricted logic expresses all queries. Let k be a positive integer. The set of *infinitary formulas with k variables* is the collection $L^k_{\infty\omega}$ of $L_{\infty\omega}$-formulas with at most k variables and

$$L^{\omega}_{\infty\omega} = \bigcup_{k=1}^{\infty} L^k_{\infty\omega}.$$

The collection of FO-formulas with at most k variables is denoted FO^k.

The relational machine

The *fixpoint* and *while* queries can be viewed as programming languages with simple control interacting with the databases via a fixed set of FO queries. A natural extension of this paradigm is to consider a computationally complete computing device interacting with the database via given FO queries. This is in the spirit of SQL queries embedded in a full programming language such as C. It is formally captured by a computational model called "relational machine". The relational machine turns out to capture precisely the effective fragment of $L^{\omega}_{\infty\omega}$ [AVV95].

A *relational machine* is a Turing machine augmented with a *relational store*. The relational store consists of a set of relations of certain arities. Some of these relations are designated as input relations and some are designated as output relations. The tape of the machine is a work tape and is initially empty. In addition to changing its internal state, moving the head on the tape and writing on the tape, the machine can check whether a relation in the store is empty or not and assign to a relation in

[3] A database schema is a finite set of relation names and their arities.

the store the result of a FO query. For example, the machine can have instructions such as:

> "If the machine is in state s_3, the head is reading the symbol 1, and relation R_1 is empty, then change the state to s_4, replace the symbol 1 by 0, move the head to the right, and replace R_2 by $\{x, y \mid \exists z(R_2(x, z) \wedge R_3(z, y))\}$".

The input is accepted iff the machine halts in an accepting state.

It is easy to see that this generalizes most query languages, including *fixpoint* and *while*.

Note that a relational machine uses a fixed set of queries; the *arity* of a relational machine is the maximum number of variables used in its FO queries.

Although the Turing machine component of relational machines provides full computational power, relational machines do *not* express all computable queries. Indeed, relational machines are subsumed by $L^{\omega}_{\infty\omega}$, and so have a 0-1 law and cannot express certain "simple" queries such as evenness.

The relational machine displays a puzzling range of expressive power: it collapses to FO on sets, but is complete on ordered inputs. As discussed in [AV91, AV95], this behavior is due to the fact that relational machines using some constant k number of variables have limited discerning power: given an input, different tuples may not be distinguishable using k variables. Thus, the machine really manipulates classes of the equivalence relation on k-tuples: $u \equiv_k v$ on input $\mathbf{I}$ iff u and v cannot be distinguished by any relational machine with k variables running on input $\mathbf{I}$. As implicit in [IL90, AV91] and shown in [AV95, KV92], $\equiv_k$ can be computed by a *fixpoint* query for each k. Moreover, the *fixpoint* query outputs the classes of $\equiv_k$ in some order. This allows reducing computations over unordered inputs to ordered inputs, and yields the following normal form for relational machines, which provides a key technical tool: For each relational machine M, there exists an equivalent machine which works in two phases. The first phase computes $\equiv_k$ and a set of *summary tables* describing the action of first-order queries with k variables on the classes of $\equiv_k$. For an input $\mathbf{I}$, let $summary_k(\mathbf{I})$ be the set of summary tables obtained for $\mathbf{I}$. To conclude the first phase, the summary tables are coded on the tape using the integers representing the equivalence classes. (The entire first phase is done in P.) Next, the computation is carried out exclusively on the tape, using the information provided by $summary_k(\mathbf{I})$. The content of a relation is represented at all times by a sequence of integers representing the equivalence classes it contains.

For Turing machines, the most natural measure of complexity is in terms of the size of the input. This is no longer so for relational machines, since such machines cannot measure the size of their input. Instead, relational machines are sensitive to the number of classes of $\equiv_k$ on input $\mathbf{I}$, called the *k-size* of $\mathbf{I}$. To see the difference between k-size and regular size, note that the k-size of sets is constant, that of complete binary trees is poly-logarithmic in their size, and for ordered structures it is polynomial in the size. This explains the fact that relational machines collapse to first-order on sets

but are complete on ordered inputs. This also suggests that the right measure of the input for a relational machine with k variables is the k-size of the input. The use of k-size as a measure of the input for a relational machine gives rise to a new notion of computational complexity, called *relational complexity*, resulting in classes such as P_r (relational polynomial time), and PSPACE_r. It is shown in [AV91] that $\mathrm{P}_r = \mathit{fixpoint}$ and $\mathrm{PSPACE}_r = \mathit{while}$, generalizing the results of [Im86, Var82] that $\mathrm{P} = \mathit{fixpoint}$ and $\mathrm{PSPACE} = \mathit{while}$, *on ordered inputs*.

Pebble games

The discerning power of relational machines of arity k, as well as that of the infinitary logic $L^k_{\infty\omega}$, are characterized by a two player pebble game [Im82, Po82]. The discerning power can refer to the ability to distinguish between structures, or between tuples in the context of a structure. It has been shown in [AV95] that in both cases, the discerning power of relational machines of arity k is the same as the discerning power of $L^k_{\infty\omega}$. We use the following notation: for two structures **I**,**J** over the same schema, $\mathbf{I} \equiv_k \mathbf{J}$ iff **I** and **J** cannot be distinguished by any relational machine of arity k (nor by any $L^k_{\infty\omega}$ sentence). Similarly, given a structure **I**, for two tuples u, v of elements of **I** of arity l, $l \leq k$, $u \equiv_k v$ iff u and v cannot be distinguished by any relational machine of arity k on input **I** (nor by any $L^k_{\infty\omega}$ formula evaluated on **I**).

We now briefly describe the k-pebble game on two structures **I** and **J**. The *k-pebble game* between the Spoiler and the Duplicator has the following rules. Each of the players has k pebbles, say $p_1, \ldots, p_k$ and $q_1, \ldots, q_k$, respectively. In each move, the Spoiler places/removes a pebble p_i on/from an element of **I** or **J**, and the Duplicator responds by placing/removing a pebble q_i on/from the opposite structure. Let a_i (resp., b_i), be the elements of **I** (resp, **J**) under pebble p_i or q_i, $1 \leq i \leq l \leq k$. If the mapping h with $h(a_i) = b_i$, $1 \leq i \leq k$, is not an isomorphism between the substructures of **I** and **J** with universes $\{a_1, \ldots, a_k\}$ and $\{b_1, \ldots, b_k\}$ respectively, then the Spoiler wins the game. The Duplicator wins the game if he can continue playing "forever", i.e. if the Spoiler can never win the game.

If the Duplicator wins the k-pebble game on structures **I** and **J**, then we say that **I** and **J** are equivalent with respect to the k-pebble game.

The k-equivalence characterizes the power of k-ary relational machines, as well as $L^k_{\infty\omega}$.

Proposition 2.2 [Im82, Po82] Let **I** and **J** be structures over the same schema. The following are equivalent.

(i) **I** and **J** are equivalent with respect to the k-pebble game.

(ii) $\mathbf{I} \equiv_k \mathbf{J}$.

The k-pebble game on a structure **I** and two tuples u, v of elements in **I** is played similarly, except that initially the pebbles are placed on the elements of the two tuples.

If the Duplicator wins the k-pebble game on structures $\mathbf{I}$ and tuples u, v, then we say that u and v are k-equivalent (with respect to structure $\mathbf{I}$). The analog of Proposition 2.2 holds:

Proposition 2.3 Let $\mathbf{I}$ be a structure and u, v be tuples of arity l, $l \leq k$, over the elements of $\mathbf{I}$. The following are equivalent.

(i) u and v are equivalent with respect to the k-pebble game (in the context of $\mathbf{I}$), and

(ii) $u \equiv_k v$.

3 The Model

A *reflective relational machine* M is a Turing machine with a standard Turing tape, a *query tape*, and a *relational store*, capable of storing an unbounded but finite number of arbitrary relations $R_0, R_1, R_2, \ldots$. Initially, the relational store contains only the *input relation*[4] R_0 whose arity r_0 depends only on M; all tapes are empty and the initial state prevails. M computes as an ordinary two-tape Turing machine, with the following unique exception: When state q, the "query state," is entered, then the contents of the query tape are interpreted as statements. We allow two kinds of statements:

1. yes-no queries of the form
$$\varphi(R_{i_1}, \ldots, R_{i_k})?$$
where $i_1, \ldots, i_k$ are natural numbers, and φ is a FO formula over the vocabulary $\{R_{i_1}, \ldots, R_{i_k}\}$. The query is evaluated *in a single step* on the relational store. Depending on the answer, the machine moves to one of the particular states *yes* and *no*.

2. updates of the form
$$R_j \leftarrow \varphi(R_{i_1}, \ldots, R_{i_k}),$$
where $j, i_1, \ldots, i_k$ are natural numbers, and φ is a FO formula over the vocabulary $\{R_{i_1}, \ldots, R_{i_k}\}$. Relation R_j is assigned the result of query φ. The machine moves to a particular state, say *update_done*.

If R_j does not yet exist in the relational store, it is added to the store. Errors can occur for various reasons: (i) undefined relations are mentioned in the query, (ii) the query tape contains some syntax error, (iii) the arity of a relation as used in the query does not match the arity of the relation in the store. When such an error occurs, the machine halts and rejects.

[4]Inputs consisting of several relations can also be considered without complication.

Remark 3.1 We allow general updates in addition to the yes/no queries only in order to model more faithfully reflective database programs. In our simulation of PSPACE in the proof of our main theorem only yes-no queries are employed. In fact, essentially a single such query is used. □

The *time* spent by a reflective relational machine on input R_0 is the total number of steps, including one step for each query execution, until the machine halts. The *space* used by a reflective relational machine is the maximum number of tape squares (of the two tapes) used during the computation.

Let $\mathcal{S}$ be a set of finite relations, all of the same arity. We say that the reflective relational machine M *decides* $\mathcal{S}$ if the following is true: M on input R_0 halts and accepts if $R_0 \in \mathcal{S}$; and M on input R_0 halts and rejects if $R_0 \notin \mathcal{S}$. We say that M decides $\mathcal{S}$ in time (or space) $f(n)$ if it decides $\mathcal{S}$, and furthermore on input R_0 it spends time (respectively, uses space) bounded by $f(n)$, where n is the number of distinct elements appearing in the tuples of R_0.

It should be clear that, if $\mathcal{S}$ is a set of structures decidable by a reflective relational machine M, then $\mathcal{S}$ must be *generic*, or *isomorphism-invariant;* that is, $S \in \mathcal{S}$ if and only if $S_\pi \in \mathcal{S}$, where S_π is S with its constants renamed by an arbitrary isomorphism π of the domain. Generic sets of structures are called *properties*.

We now make some simple remarks on the power of reflection. The *relational machine* model defined in[5] [AV91] used a finite set of queries specified *a priori* in the control of the machine. Therefore no reflection is allowed. However, it was shown in [AV95] that no additional power is gained if the relational machine is augmented with reflection, as long as the number of variables allowed on the query tape is constant. Thus, the power of reflection resides in the use of queries with unbounded number of variables. This naturally leads to considering the *variable complexity* of a reflective machine M on input R_0, which is the largest number of variables ever used in a query. Thus, ordinary relational machines are equivalent to reflective relational machines with $O(1)$ variable complexity.

It is immediate that reflective relational machines are more powerful than ordinary ones. In polynomial time, non-reflective relational machines can decide properties in the set P_r, that is, properties expressible in FO logic with fixpoint. This class excludes trivial properties, such as graphs with an even number of nodes. In contrast, there is a reflective relational machine M which decides graphs with an even number of nodes: Given a graph R_0, M goes on to ask queries of the form

$$\exists x_1 \ldots \exists x_m [\bigwedge_{1 \le i < j \le m} (x_i \neq x_j)]?$$

for $m = 1, 2, \ldots$ until it receives a negative answer. It then accepts if the last m was odd, and rejects otherwise. Hence reflective relational machines are indeed more

[5]More precisely, the machine is called "loosely coupled generic machine" in [AV91]. The term "relational machine" was first used in [AVV97].

powerful than ordinary ones. Just how powerful are they? Clearly, a reflective machine can completely identify (up to isomorphism) a relation with n elements, using n variables. Once complete information about the input is obtained, the machine can then decide acceptance by a regular Turing computation on the tape. Thus, it is easy to see that reflective relational machines can define all recursive properties.

The expressive power of reflective machines with sublinear variable complexity is trickier, since it seems that a structure cannot be identified with fewer than n variables. This issue is examined in Section 5.

In most of the paper we look at reflective machines of variable complexity at least linear, and with additional time/space bounds. (Note that a time/space bound implies a variable complexity bound, but the converse is obviously false.) The primary focus is the connection with (parallel and sequential) complexity classes.

4 Polynomially Bounded Reflective Machines

Polynomial-time reflective machines

We consider next the power of polynomial-time reflective relational machines. First, we make an observation showing that order is irrelevant for such machines. As noted earlier, a reflective machine can completely identify its input using $O(n)$ variables. In fact, this can be done in polynomial time.

Lemma 4.1 There exists a polynomial-time reflective relational machine which, for each input relation R_0, produces on the tape a standard encoding of R_0.

Proof: For simplicity, assume R_0 is binary. The reflective machine begins by determining the number n of elements occurring in R_0 (this takes $O(n)$ steps). Next the machine starts building up on the tape the relation R_0, using n variables $x_1, \ldots, x_n$ to refer to the elements in R_0. The machine starts with the empty binary relation R_0'. Then, it iterates over $i, j, 1 \leq i, j \leq n$, adding to R_0' the edge $\langle x_i, x_j \rangle$, if the result is isomorphic to a subgraph of R_0 (which can be tested by one first-order query using $x_1, \ldots, x_n$). An easy induction shows that, at the end of the iteration, R_0' is isomorphic to R_0. □

We now state the result relating polynomial-time reflective machines to PSPACE. *A priori*, the fact that polynomial-time reflective machines can be simulated within PSPACE is counterintuitive, since polynomial-time reflective machines can build relations of polynomial arity, and so of *exponential* size. Conversely, PSPACE computations can run for exponential time, so the fact that they can be simulated by a polynomial time reflective machine may appear surprising. However, as discussed in the introduction, the result appears less surprising when viewed from the point of view of *parallel* complexity, where results of this nature exist, but are formulated in

terms of *Boolean circuits*. Since the proof of our result makes explicit the connection to parallel complexity, via circuits of polynomial depth and exponential size, we give below the basic definitions concerning Boolean circuits. See, e.g., [Pa93] for a more comprehensive introduction to circuits and their place in complexity theory.

Concretely and briefly, a *Boolean circuit* is a directed acyclic graph, with nodes called *gates*. Each gate can be of one of five kinds: AND gate, OR gate, NOT gate, TRUE gate, and FALSE gate —TRUE and FALSE gates are also called *input* gates. The in-degree of a NOT gate is one, that of an input gate is zero (that is, input gates are the sources of the directed acyclic graph), and the indegree of AND and OR gates is at least two. If the indegree of OR and AND gates can be greater than two, then we have an *unbounded fan-in circuit*, otherwise a *binary circuit*. The size of a circuit is the number of gates in it, and the depth of a circuit is the length of the longest path in it.

The semantics of a Boolean circuit is the intuitive one: Let us topologically sort the gates of the circuit, that is, fix a linear order of the gates so that edges go from lower to higher-ordered gates. Since the Boolean values of the input gates are known, the values of all other gates can be computed in the topological order, by applying the Boolean functions AND, OR, (possibly of large arity) and NOT to the Boolean values of previously computed gates. Certain gates in the circuit have out-degree zero, and are considered the *outputs* of the circuit; in the end of the computation described they hold the values computed by the circuit with the specified inputs.

Circuits are an interesting model of computation. To use a circuit C *with unspecified input gates* (that is, input gates that are just designated "INPUT" and can be made either TRUE or FALSE) for deciding whether a string $x \in \{0,1\}^*$ is in a language L, we just set the first input of C to the first bit of x, and so on. However, this way circuit C specializes in solving instances of a specific length, namely, those the number of input gates. In order to render circuits a general model of computation, *families of circuits* are needed. A family of circuits $\mathcal{C} = \{C_1, C_2, \ldots\}$ is a countably infinite sequence of circuits, such that the ith circuit in the family has i input gates. The size of a family is the size of the nth circuit in it, as a function of n, and similarly for depth. Finally, we must somehow bound the complexity of the circuit itself —otherwise even undecidable problems could be decided by (non-recursive) families of circuits. A family of circuits is said to be *polynomial-time uniform* if there is an algorithm which, given n in unary, constructs the nth circuit in polynomial time; similarly for polynomial-space uniformity. Uniform families of circuits capture complexity classes: P is precisely the class of languages decided by polynomial size, polynomial-time uniform families of circuits; and PSPACE is precisely the class of languages decided by polynomial depth, polynomial-space uniform families of circuits, see [Pa93].

Theorem 4.2 The class of properties decidable by reflective relational machines in polynomial time is precisely the class of properties decidable in PSPACE.

Proof: For one direction, suppose that $\mathcal{S}$ is a property decidable in PSPACE. That is, there is a polynomial-time alternating Turing machine A [CKS81] which accepts input x if and only if it encodes a relation in $\mathcal{S}$. We can assume that all computations of A on input x have length $p(n)$, where $n = |x|$, and that A always alternates between existential and universal states; say, starting from an existential and ending up with a universal state. In other words, the computation of the alternating machine on input x can be seen as a *circuit* $A(x)$ that forms a full binary tree of depth $p(n)$, with the gates strictly alternating between AND and OR gates, such that there is a polynomial-time algorithm Π for deciding, for each input x and each leaf address $\ell \in \{0,1\}^{p(n)}$, whether the input gate at leaf ℓ of $A(x)$ is TRUE, equivalently, whether the nondeterministic choices in ℓ lead A to acceptance. Alternating Turing machine A accepts input x if and only if the circuit $A(x)$ evaluates to TRUE.

We shall construct a reflective relational machine M that decides $\mathcal{S}$. M starts like the machine that decides evenness in the previous section, by determining the number n of constants in the structure R_0. Then it goes on to determine the value of $p(n)$, i.e., the depth of the alternating circuit $A(x)$. M then encodes its input R_0 as a binary input x of A. This can be done in polynomial time by Lemma 4.1. From the fixed polynomial-time algorithm Π which decides whether a leaf is TRUE or not, the value of n, and the bit-string x, M next constructs on its tape a circuit C_x, of size polynomial in n, with inputs $y_1, \ldots, y_{p(n)}$, standing for the bits in the address of a leaf, with the property that C_x outputs TRUE if and only if with input x to A, the leaf $\ell = y_1, \ldots, y_{p(n)}$ is a TRUE input gate of $A(x)$. Let $z_1, \ldots, z_q$ be the gates of C_x other than its input gates $y_1, \ldots, y_{p(n)}$. There are polynomially many of them. Let $\varphi[C_x](y_1, \ldots, y_{p(n)})$ be a first-order formula with free variables $y_1, \ldots, y_{p(n)}$ expressing the circuit C_x, i.e. $\varphi[C_x](y_1, \ldots, y_{p(n)})$ holds iff C_x outputs TRUE on input $y_1, \ldots, y_{p(n)}$. Clearly, $\varphi[C_x](y_1, \ldots, y_{p(n)})$ can be expressed using existentially quantified variables $z_1, \ldots, z_q$ corresponding to the internal gates of C_x. For example, if z_5 is the AND of y_2 and z_4, $\varphi[C_x]$ contains the clause

$$(z_5 = 1) \leftrightarrow [(y_2 = 1) \wedge (z_4 = 1)].$$

If z_q is the output of C_x, $\varphi[C_x]$ also contains the clause $(z_q = 1)$. Finally, M asks the following query:

$$\exists y_1 \forall y_2 \ldots \forall y_{p(n)} \varphi[C_x](y_1, \ldots, y_{p(n)})?$$

The answer to this query is M's final acceptance/rejection answer. In the above, $\exists y \psi$ really means $\exists y[(y = 0) \vee (y = 1) \wedge \psi]$, and$\forall y \psi$ really means $\forall y[[(y = 0) \vee (y = 1)] \rightarrow \psi]$, where 0 and 1 are constants.

We claim that M accepts input R_0 if and only if $R_0 \in \mathcal{S}$. M accepts its input if and only if its final query is a valid first-order expression, which holds if and only if the alternating Boolean circuit that captures the computation of A evaluates to TRUE, which is equivalent to saying that A accepts the encoding of R_0, or $R_0 \in \mathcal{S}$.

For the other direction, suppose that $\mathcal{S}$ is a property decided by a reflective relational machine M in time $p(n)$, where n is the number of constants in the input. We

shall show that $\mathcal{S}$ is in PSPACE. To this end, we shall describe a polynomial-space uniform, polynomial-depth family of unbounded fan-in circuits $\mathcal{C}_M = (C_1, C_2, \ldots)$, that accepts precisely the encodings of structures in $\mathcal{S}$ (in particular, circuit C_n accepts the encodings of structures in $\mathcal{S}$ that have n constants). Such families of circuits are known to capture PSPACE.

Let us fix n, the number of constants in the input. We shall assume that all relations are of arity $p(n)$, the largest possible (we can pad relations of smaller arity with an "undefined" symbol, if needed). We next describe the circuit C_n, the nth circuit in the family $\mathcal{C}_M$, which computes the answer to inputs of length n. It consists of the cascading of $p(n)$ copies of the same circuit D_n, which simulates a step of M. D_n has a huge number of inputs, and the same number of outputs; in particular, it has $p(n) \cdot (n^{p(n)} + 1)$ inputs and outputs. For each of the $p(n)$ relations handled by M, each relation being of arity $p(n)$, and for each of the possible $n^{p(n)}$ tuples in this relation, we have a binary input (and a binary output) of D_n denoting whether or not the tuple is present before (respectively, after) the step simulated by this copy of D_n. These $p(n) \cdot n^{p(n)}$ bits constitute the *relational part* of the input and output of D_n. We also have $p(n)$ more bits in both the input and output, describing the tapes and state of M, again before and after the current step.

As we mentioned before, D_n simulates a step of M. Suppose that it is a query step, which is, naturally enough, the difficult case. The query being asked is a first-order expression of length at most $p(n)$, and so it can be expressed as an unbounded fan-in circuit with at most $p(n)$ alternating levels. The problem is, of course, that the query is not known *a priori*, but it is encoded in the bits of the non-relational part.

To overcome this problem, we build a *general-purpose circuit*, capable of answering all quesries of this form. We assume without loss of generality that all queries asked during the computation are of the form

$$R_j(y_1, \ldots, y_{p(n)}) \leftrightarrow \exists x_1 \forall x_2 \ldots \forall x_{p(n)} [C_1 \wedge C_2 \wedge \ldots \wedge C_{p(n)}].$$

This is the most general form of a query that can be formulated and submitted by M in time $p(n)$, since it has $p(n)$ alternating first-order quantifiers and $p(n)$ clauses with literals of the form, say, $(\neg)R_8(x_2, y_5, \ldots)$; to represent any query in this form, we may have to pad it with null extra variables and clauses. This query is simulated by a tree of $p(n)$ alternating levels of OR and AND gates, each with fan-in n (reflecting the n possible values for each x_i), followed by a level of AND gates with fan-in $p(n)$ (corresponding to the conjunction of the clauses). There are $n^{p(n)}$ copies of this tree, corresponding to the different values of the y_i's. Each leaf of each tree corresponds to a clause C_k of the query, and to specific values for the x and y variables.

Each output gate of D_n of the form $R_j(a_1, \ldots, a_{p(n)})$ is an OR gate with two *potential connections:* to the root of the tree that corresponds to the values $(a_1, \ldots, a_{p(n)})$ for the y_i's; and the input gate $R_j(a_1, \ldots, a_{p(n)})$. Also, each leaf of each tree is an OR gate with potential connections to all inputs of D_n and their negations.

So far this circuit computes nonsense; *we must describe how to transform this generic "programmable logic array" into a circuit that implements the present query.* The transformation must connect each leaf to the correct inputs, and each root to the correct output. Consider a root, call it r, corresponding to the values $(a_1, \ldots, a_{p(n)})$ for the y_i's. It must be connected to the output, call it s, of D_n that corresponds to $R_j(a_1, \ldots, a_{p(n)})$ if and only if the present query defines the relation R_j. This information, however, can be easily obtained from the *non-relational part* of the input to D_n. That is, we can build a small circuit that has the non-relational part as input, and outputs 1 if and only if R_j is indeed being defined; this output is then ANDed with the root R before it is ORed into the output S. This way output S will be correct.

A similar strategy works for the inputs of D_n. Consider an leaf, call it ℓ, which corresponds to a clause C_k and values $(a_1 \ldots, b_n)$ for the y_i's and the x_j's. We can build a small circuit that takes as inputs the non-relational inputs of D_n, and tells us if ℓ should take as one of its inputs a particular input gate m of D_n (or its negation). Again, m (or its negation) is ANDed with this output before it is ORed into ℓ.

If, finally, the currently simulated step is *not* a query step, the correct output of D_n is the same as the input, except that the non-relational part is modified in a simple, polynomial way that is also built into D_n. It is easy to see that the resulting circuit D_n has polynomial depth, exponential size, and can be constructed in polynomial space. Finally, C_n is a $p(n)$-fold cascading of D_n, simulating all $p(n)$ steps of M. The inputs of C_n are fixed so as to reflect the value of R_0. (All other relations are initially empty.) It is easy to see that, so initialized, C_n computes TRUE if and only if M accepts R_0, and this concludes the proof. □

Note that, in the proof of the theorem, the reflective machine used in the simulation of PSPACE has the property that the relations in the store have constant arity (since only yes/no queries, and no updates, are required). Also, note that the inclusion of the reflective machine in PSPACE is obvious if the size of the store, as well as the tape, is bounded by a polynomial. Thus, we have:

Corollary 4.3 The following are equivalent and express PSPACE:

(i) polynomial-time reflective machines;

(ii) polynomial-time reflective machine with (a fixed number of) constant-arity relations;

(iii) polynomial-space reflective machines with (a fixed number of) constant-arity relations; and,

(iv) polynomial-space reflective machines with polynomially-space bounded relational store.

Theorem 4.2 suggests that reflection allows for more "intense" data parallelism than the classical, non-reflective query languages. Indeed, consider the standard (non-reflective) relational machine, which, as discussed earlier, subsumes all common query languages.

Theorem 4.4 Assuming P $\neq$ PSPACE, there is a property that can be checked in polynomial-time by a reflective relational machine, but cannot be checked in polynomial time by any non-reflective relational machine (although it is definable by such machines).

Proof: Let $\mathcal{S}$ be a property on ordered structures which is in PSPACE but not in P (it is easily seen that such a property exists if P $\neq$ PSPACE). By Theorem 4.2, there exists a reflective relational machine accepting $\mathcal{S}$ in polynomial time. There is also a non-reflective relational machines defining $\mathcal{S}$, since such machines define all recursive properties on ordered inputs. Suppose there is a non-reflective relational machines that accepts $\mathcal{S}$ in polynomial time. Since all properties defined by polynomial-time non-reflective machines are in P, $\mathcal{S}$ is in P, a contradiction. □

Thus, reflection allows in principle to express more parallelism in a natural and easily detectable way.

Polynomial-space reflective machines

Recall that a polynomially-space bounded reflective machine is restricted to a polynomial amount of tape. What restriction does this implies on the store? A query may have a polynomial number of variables. Thus the relational store may have polynomial arity, so exponential size. Interestingly, the complexity class captured by such machines is precisely EXPSPACE. The reason is that, because of the extensive relational storage, computations that require only polynomial-space on the tapes need not terminate within exponential time.

Theorem 4.5 Polynomial-space bounded reflective relational machines decide precisely the properties decidable in EXPSPACE.

Proof: It is clear that any such machine can be simulated in exponential space. Conversely, consider a property $\mathcal{S}$ decidable in EXPSPACE, and let M be a $2^{p(n)}$ space-bounded Turing machine accepting standard encodings of relations in $\mathcal{S}$ (where $p(n)$ is some polynomial). We assume that the machine always halts, and that it has a single *cyclic* tape; that is, the tape head always proceeds to the right, and, when it is at the $2^{p(n)}$th tape square, it next materializes to the first tape square. We shall sketch the construction of a polynomial space-bounded relational machine R that simulates M.

R starts by producing on its tape a representation x of its input relation R_0; this can be accomplished in polynomial time. It then proceeds, in one instruction, to initialize $p(n)+1$-ary relation T (intuitively, encoding the tape of M). After this initialization, T contains all tuples of the form $(a_1, a_2, \ldots, a_{p(n)}, 0)$, where the a_i's are in $\{0,1\}$. Intuitively, tuple $T(a_1, a_2, \ldots, a_{p(n)}, s)$ means that the tape square of M whose position is spelled by the a's in binary contains the state-symbol combination s; 0 stands for blank. Next, R performs polynomially many updates to T, so that its contents reflect the situation in which M starts with x on its tape, with all other squares blank.

The main computation of R starts next. It consists of a loop of two query steps, repeated indefinitely. In the first query step, R checks to see if M has halted and accepted, or halted and rejected, in which case it does the same. This can be obviously accomplished by a simple relational query. In the second query step, R performs an update of T which reflects a single step of M. It is a rather standard exercise, which we omit, to see that this too can be accomplished as a relational query. □

Remark 4.6 It is worth noting that the power of the reflective machine is very sensitive to the query language used on the query tape. Indeed, suppose that we use a more concise untyped relational algebra instead of FO. Then repeated execution of the update $R_1 := R_1 \times R_1$ allows to create a relation of unbounded arity (so unbounded size) with a constant-space query tape. □

Lower complexities

The reflective relational machine is evidently a machine of the "second class" [vE90], a parallel model of computation. It is interesting that it can be used to capture feasible parallel computation (notice that, as is typical below NP, we now need order):

Theorem 4.7 On ordered structures, reflective relational machines operating within polylogarithmic time, and with a bounded number of variables per query, decide precisely the properties decidable in NC.

The proof is similar to that of Theorem 4.2. Notice that, as anticipated in the work of Immerman [Im89], hardware is exponential in the number of variables.

Furthermore, the power of polynomial time reflective machines with $O(1)$ variables can now be expressed as follows:

Theorem 4.8 On ordered structures, reflective relational machines operating within polynomial time, and with a bounded number of variables per query, decide precisely the properties decidable in P.

Note that the above theorems require $O(1)$ variable complexity of the reflective machines. And, we know that reflective machines with $O(1)$ variable complexity can

be simulated by non-reflective machines. In fact, the simulation of each dynamically generated query in the reflective machine can be done by a non-reflective machine in polynomially many steps in the size of the query [AV95]. Since P and NC are polynomially closed, the above results hold for the non-reflective relational machine as well. However, reflective machines with $O(1)$ variables may in principle still have an advantage over non-reflective machines for finer bounds where the polynomial simulation makes a difference. This possibility remains to be investigated.

5 Sublinear variable complexity

We now consider the question of the variable complexity of reflective machines. Recall that machines with $O(n)$ variable complexity are able to completely identify their input, and can therefore define any recursive property. At the other extreme, the reflective machine with $O(1)$ variable complexity is equivalent to the non-reflective machine, and has limited expressive power even with no time/space complexity bound. Indeed, such machines are subsumed by the infinitary logic $L^{\omega}_{\infty\omega}$, so they have a 0-1 law and cannot compute simple properties like evenness.

How about reflective machine with variable complexity in between $O(1)$ and $O(n)$? We next compare such machines to the machines with variable complexity $O(1)$ and $O(n)$. A function f from the non-negative integers to the positive integers is a *sublinear bounding function* if it is a nondecreasing, computable function such that $\lim_{n\to\infty} f(n) = \infty$ and $\lim_{n\to\infty} f(n)/n = 0$. We say that a reflective relational machine has *sublinear* variable complexity if it has variable complexity $f(n)$ where f is a sublinear bounding function.

First, note the following useful fact:

Lemma 5.1 On unary inputs, reflective machines with sublinear variable complexity are equivalent to machines with $O(1)$ variable complexity, and express the FO properties.

The proof uses Algorithm **Bootstrap** and Theorem 5.6 below, and is therefore provided at the end of the section. In particular, the lemma shows that reflective machines with sublinear variable complexity cannot express the evenness query on a set, so are weaker than reflective machines with $O(n)$ variables. (Note that, unlike $O(1)$ machines, sublinear machines do not have a 0-1 law, so this route cannot be used to show that evenness is not definable.) Indeed, we have:

Theorem 5.2 .

(i) There exist recursive (indeed, LOGSPACE) properties not definable by any reflective machine with sublinear variable complexity.

(ii) For every sublinear bounding function f, the reflective machines with $O(f(n))$ variable complexity are strictly more powerful than the reflective relational machines with $O(1)$ variable complexity.

Proof: Part (i) follows immediately from Lemma 5.1, in particular from the fact that sublinear relational machines cannot express evenness. For part (ii), let f be a sublinear bounding function. Consider the property of a unary relation S and a binary relation *succ* stating that S and *succ* are over disjoint domains, *succ* is a successor relation, and $|S| = f(|succ|)$. It is easily seen that this is definable by a reflective machine using $O(f(n))$ variables: first check that *succ* is a successor relation, using $O(1)$ variables; next, compute the size of *succ*, again using $O(1)$ variables. Then compute $f(|succ|)$ on the tape, and finally generate the query with $f(|succ|) + 1$ (therefore $O(f(n))$) variables checking that the size of S is $f(|succ|)$. On the other hand, a standard pebble game argument shows that the above property is not definable in $L^{\omega}_{\infty\omega}$, so it is not definable by a reflective machine with $O(1)$ variable complexity. □

Thus, sublinear machines lie strictly between $O(1)$ and $O(n)$ machines. Additional examples of graph properties checkable by sublinear machines but not by relational machines include:

- the property of being a balanced d-ary tree where d is the depth of the tree;
- the property of having a clique of size $log(d)$ where d is the diameter of the graph.

We will show that there is a hierarchy within sublinear machines. Before proving this, we note the following obvious but useful fact relating pebble games with reflective machines.

Fact: Let f be a sublinear bounding function. Let $\mathbf{I}$ and $\mathbf{J}$ be structures over the same schema, and $k = max\{f(|\mathbf{I}|), f(|\mathbf{J}|)\}$. If $\mathbf{I}$ and $\mathbf{J}$ are equivalent with respect to the k-pebble game, then $\mathbf{I}$ and $\mathbf{J}$ cannot be distinguished by any reflective machine with variable complexity f.

As we shall see, the converse of the above is false.

We are now ready to prove the following hierarchy theorem.

Theorem 5.3 Let f, g be sublinear bounding functions with $\lim_{n\to\infty} f(n)/g(n) = 0$. Then there exists a property definable by a reflective machine with $O(g(n))$ variable complexity but not by one with $O(f(n))$ variable complexity.

Proof: Similarly to the proof of Theorem 5.2, consider the property (†) of a unary relation S and a binary relation *succ* stating that:

1. S and *succ* have no common elements,

2. *succ* is a successor relation, and

3. for some n, $|S| = g(n)$ and $|succ| = n - g(n)$.

This property is definable by a reflective machine with $O(g(n))$ variable complexity as follows. First, the machine computes $m = |succ|$ using 3 variables. Next, the machine checks increasing values of n starting from m. For a given n, the machine computes $g(n)$ and tests whether $g(n) \geq |S|$. Testing the latter requires $g(n)$ variables. This continues until such an n is found. Such n must exist since $lim_{n\to\infty} g(n) = \infty$. Finally, the machine accepts if $m = n - g(n)$ and $|S| = g(n)$ (testing the latter requires $g(n)+1$, so $O(g(n))$ variables) and rejects otherwise.

To show that property (†) is not definable by a reflective machine with $O(f(n))$ variable complexity, we use a straightforward pebble game. Suppose towards a contradiction that there is some reflective machine M with variable complexity bounded by $cf(n)$ for some constant c, such that M defines the above property. Since the limit of $f(n)/g(n)$ is 0, there exists N such that, for all $n \geq N$, $cf(n) < g(n) - 1$. For $n \geq N$, let $\mathbf{I}$ be the structure over *succ* and S satisfying (†) such that $|succ| = n - g(n)$ and $|S| = g(n)$, and $\mathbf{J}$ be like $\mathbf{I}$ except that $|S| = g(n) - 1$ (thus, $\mathbf{J}$ does not satisfy (†)). Let $k = max\{cf(|\mathbf{I}|), cf(|\mathbf{J}|)\} = max\{cf(n), cf(n-1)\} = cf(n)$. Recall that $k < g(n) - 1$. Clearly, $\mathbf{I}$ and $\mathbf{J}$ are equivalent with respect to the k-pebble game (the Duplicator mimics precisely the Spoiler's moves on elements of *succ*, and picks arbitrary elements of S when the Spoiler moves on such elements). By the earlier Fact, since $k = max\{cf(|\mathbf{I}|), cf(|\mathbf{J}|)\}$, $\mathbf{I} \equiv_{cf(n)} \mathbf{J}$, so $\mathbf{I}$ and $\mathbf{J}$ are undistinguishable by M. This contradicts the assumption that M defines property (†). □

So far, we considered the power of reflective machines with sublinear variable complexity but with no bound on the time or space complexity. Let us finally look at what happens if one places such complexity bounds on these machines. Recall that, in the case of non-reflective relational machines, there is mismatch between complexity classes and classes definable using the machines; indeed, this gave rise to the notion of "relational complexity", based on the discerning power of each machine rather than the input size. On the other hand, this problem does not occur with reflective machines with variable complexity over $O(n)$, due to their ability to completely identify their input. How about the sublinear case? As shown next, the mismatch persists for these classes. More precisely, *no complexity class above P* can be captured (on unordered inputs) by sublinear machines subject to a time or space bound:

Theorem 5.4 Let C be a time or space complexity class including P and polynomially closed. There is no time/space complexity class C' such that the properties in C' are precisely those definable by the reflective relational machines with time/space bound C and variable complexity $O(f(n))$, where f is a sublinear bounding function computable in C.

Proof: Suppose, towards a contradiction, that the properties definable by reflective machines with variable complexity $O(f(n))$ and time/space C are exactly those of complexity C', for some C'.

First, note that C' must contain C as usual complexity classes. Indeed, let L be a language in C. A word w can be encoded as a relation $enc(w)$ over some ordered domain. A reflective machine with time/space complexity C and $O(1)$ variables can test whether a relation encodes a word in L as follows: first decode the word from the input relation and place it on the Turing tape, using $O(1)$ variables and polynomial time. Then, perform a Turing computation of complexity C to decide if the word is in L. Thus, $\{enc(w) \mid w \in L\}$ is a property definable by a reflective machine of time/space complexity C and variable complexity $O(1)$ so $O(f(n))$. By our assumption, this property is also in C'. To see that L is in C', consider the following decision procedure for L: given a word w, construct $enc(w)$, then decide in C' if $enc(w) \in \{enc(w) \mid w \in L\}$. If the answer is *yes* accept w, otherwise reject. Clearly, this decision procedure has complexity C', so C is contained in C' as usual complexity classes.

Consider the property of two unary relations A, B stating that $A \subseteq B$ and $|A| = f(|B|)$. This property is in C' (this uses the fact that P $\subseteq C \subseteq C'$). However, it is not definable by any reflective machine with variable complexity $O(f(n))$. Intuitively, the difficulty is that such a machine cannot compute $f(|B|)$ without computing $|B|$, which cannot be done with $O(f(n))$ variables once B is past a certain size. Formally, suppose towards a contradiction that the property is definable by some reflective machine M with variable complexity bounded by $cf(n)$ for some constant c. Let n be such that $cf(n+1) < n$ and $f(n) < f(n+1)$ (such n exists because $\lim_{n\to\infty} f(n)/n = 0$ and $\lim_{n\to\infty} f(n) = \infty$). Now let $\mathbf{I}$ be a structure over A, B such that $A \subseteq B$, $|B| = n$ and $|A| = f(n)$ (so $\mathbf{I}$ satisfies the above property) and $\mathbf{J}$ be the same except that $|B| = n+1$ (so $\mathbf{J}$ does not satisfy the property, since $f(n) < f(n+1)$ and $|A| \neq f(|B|)$). Let $k = max\{cf(|\mathbf{I}|), cf(|\mathbf{J}|)\} = cf(n+1)$. Clearly, $\mathbf{I}$ and $\mathbf{J}$ are equivalent with respect to the k-pebble game. By the earlier Fact, $\mathbf{I}$ and $\mathbf{J}$ are undistinguishable by reflective machines of variable complexity $cf(n)$. Thus, M cannot distinguish between $\mathbf{I}$ and $\mathbf{J}$ and so cannot define the above property. □

Discerning power of sublinear machines

There is a strong analogy between the behavior of sublinear reflective machines, with respect to expressive power and complexity, and that of relational machines (which are non-reflective). They both display a puzzling range of expressive power, collapsing to FO on sets but expressing all properties on ordered inputs. The source of the behavior is, in both cases, the limitation on the number of variables used in queries, which in turn yields limits on the *discerning power* of the machines. Indeed, a relational machine using some constant k number of variables has limited discerning power: given an input, different tuples may not be distinguishable using k variables. Thus,

the machine really manipulates classes of the equivalence relation on k-tuples: $u \equiv_k v$ on input $\mathbf{I}$ iff u and v cannot be distinguished by any relational machine of arity k running on input $\mathbf{I}$. See Section 2 for a review of the relevant results on relational machines.

For sublinear reflective machines, the situation is similar in that the limit on number of variables results, again, in limited distinguishing power. We wish to generalize $\equiv_k$ and the normal form to reflective machines. The definition of $\equiv_k$ is naturally extended as follows.

Definition 5.5 Let f be a sublinear bounding function. For structures $\mathbf{I}$, $\mathbf{J}$ over the same schema, $\mathbf{I} \equiv_f \mathbf{J}$ iff $\mathbf{I}$ and $\mathbf{J}$ cannot be distinguished by any reflective machine with variable complexity bounded by f. Similarly, if $\mathbf{I}$ is a structure and u, v are tuples of the same arity over elements of $\mathbf{I}$, $u \equiv_f v$ iff no reflective machine of variable complexity bounded by f distinguishes between u and v when run on input $\mathbf{I}$. More precisely, for every reflective machine M of variable complexity bounded by f and every configuration of the computation of M on input $\mathbf{I}$, $u \in R$ iff $v \in R$ for each relation R of the relational store.

Note that for every $\mathbf{I}$ and tuples u, v of arity larger than $f(|\mathbf{I}|)$, $u \equiv_f v$.

One might be tempted to think that for a structure $\mathbf{I}$, the relation $\equiv_f$ on tuples is the same as $\equiv_{f(|\mathbf{I}|)}$. Clearly, $u \equiv_{f(|\mathbf{I}|)} v$ implies that $u \equiv_f v$. However, the converse is false. This is due to the fact that the number of variables that a reflective machine can use is determined dynamically, and involves the following subtlety. A reflective machine of variable complexity $f(n)$ must use variables conservatively, since it must ensure at each step, *using the information currently available about the input*, that the $f(n)$ bound is not violated. Consequently, a reflective machine with variable complexity $f(n)$ cannot necessarily *attain* the number of variables $f(n)$ on an input of size n. This happens, for example, on unary inputs, where any sublinear machine can in fact only use $O(1)$ variables (see Lemma 5.1). Let $k(f, \mathbf{I})$ be the maximum number of variables *actually used* by some reflective machine with variable complexity $f(n)$ on input $\mathbf{I}$. We claim that $k(f, \mathbf{I})$ is computed by the following "bootstrapping" algorithm.

ALGORITHM **Bootstrap**
1. $k := f(0)$;
2. repeat until no further change:
    3. compute $summary_k(\mathbf{I})$;
    4. find $\mathbf{I}_0$ of minimum size such that
        $summary_k(\mathbf{I}_0) = summary_k(\mathbf{I})$;
    5. $k := f(|\mathbf{I}_0|)$

We first note that Algorithm **Bootstrap** is terminating. To this end, we will prove that the consecutive values of k computed by the algorithm are nondecreasing. Termination follows from this and the fact that $k \leq f(|\mathbf{I}|)$. Let $\mathbf{I}_{0,0} = \emptyset$, $k_0 = 0$, and $\mathbf{I}_{0,i}$ and k_i be the values of $\mathbf{I}_0$ and k computed at iteration i, $i \geq 1$. We show by induction that:

$$(*) \quad |\mathbf{I}_{0,m-1}| \leq |\mathbf{I}_{0,m}| \quad \text{and} \quad k_{m-1} \leq k_m$$

for each $m \geq 1$. The base case is obvious. Suppose that $(*)$ holds for some m. By the induction hypothesis,

$$|\mathbf{I}_{0,m-1}| \leq |\mathbf{I}_{0,m}| \quad \text{and} \quad k_{m-1} \leq k_m.$$

Since $k_{m-1} \leq k_m$, $\equiv_{k_m}$ refines $\equiv_{k_{m-1}}$. Recall that $\mathbf{J} \equiv_k \mathbf{I}$ iff $summary_k(\mathbf{J}) = summary_k(\mathbf{I})$. Hence,

$$\{\mathbf{J} \mid summary_{k_{m-1}}(\mathbf{J}) = summary_{k_{m-1}}(\mathbf{I})\} \supseteq \{\mathbf{J} \mid summary_{k_m}(\mathbf{J}) = summary_{k_m}(\mathbf{I})\},$$

and $|\mathbf{I}_{0,m}| \leq |\mathbf{I}_{0,m+1}|$. Since f is nondecreasing, $k_m = f(|\mathbf{I}_{0,m}|) \leq f(|\mathbf{I}_{0,m+1}|) = k_{m+1}$, and the induction is complete. Thus, Algorithm **Bootstrap** always terminates.

We now show:

Theorem 5.6 Algorithm **Bootstrap** computes $k(f, \mathbf{I})$.

Proof: Let k be the number computed by Algorithm Bootstrap on input $\mathbf{I}$. Clearly, there is a reflective machine of variable complexity f that computes k on input $\mathbf{I}$ following Algorithm Bootstrap, and using k variables, so $k \leq k(f, \mathbf{I})$.

We next show that $k(f, \mathbf{I}) \leq k$. Let M be a reflective machine of variable complexity bounded by f, and $\mathbf{I}$ an input to M. A (finite) computation of M on $\mathbf{I}$ is a (finite) sequence of consecutive configurations of M run on $\mathbf{I}$, starting from the start configuration. For each finite computation c of M, let $query(c) = q_1 \ldots q_n$ be the successive queries/updates present on the query tape of M when M reaches a query state in the computation c. For each i, let v_i be the number of variables used by q_i, and $v(c) = max\{v_i \mid 1 \leq i \leq n\}$. It is sufficient to prove that for each finite computation c of M, $v(c) \leq k$. To show this, we need an auxiliary result stating that inputs yielding a given computation c can be described by an FO sentence using $v(c)$ variables. More precisely, we will prove:

> (†) For each finite computation c of M there exists an FO sentence σ_c with $v(c)$ variables such that for each input $\mathbf{I}$, c is a computation of M on $\mathbf{I}$ iff $\mathbf{I}$ satisfies σ_c.

Suppose that (†) is proven. Then we can easily prove that $v(c) \leq k$. Indeed, suppose towards a contradiction that there exists an input $\mathbf{I}$ and a computation c of M on $\mathbf{I}$ such that $v(c) > k$. Let $query(c) = q_1 \ldots q_n$ and suppose (without loss

of generality) that q_n is the first query in the sequence using more than k variables. Let c' be the prefix of c obtained by deleting from c the suffix starting with the configuration corresponding to q_n. Thus, $query(c') = q_1, \ldots, q_{n-1}$ (empty if $n = 1$). By (†), there exists an FO sentence $\sigma_{c'}$ with $v(c')$ variables such that an input $\mathbf{J}$ yields c' iff $\mathbf{J}$ satisfies $\sigma_{c'}$. By our assumption, $v(c') \leq k$ and $v(q_n) > k$. Let $\mathbf{I}_0$ be an instance of minimum size such that $\mathbf{I}_0 \equiv_k \mathbf{I}$. By the construction of k, $k = f(|\mathbf{I}_0|)$. Since the computation of M on $\mathbf{I}$ is c', $\mathbf{I}$ satisfies $\sigma_{c'}$ by (†). Since $\mathbf{I}_0 \equiv_k \mathbf{I}$ and $\sigma_{c'}$ uses $v(c') \leq k$ variables, $\mathbf{I}_0$ also satisfies $\sigma_{c'}$, so by (†) again, $\mathbf{I}_0$ also yields the computation c'. Since there is no test on the store between the end of c' and the application of query q_n in c, $\mathbf{I}_0$ also yields the computation c, and in particular produces query q_n whose number of variables is larger than $k = f(|\mathbf{I}_0|)$. This violates the bound on the number of variables, which is a contradiction.

It remains to prove (†). More specifically, we prove that for each computation c such that $query(c) = q_1 \ldots q_n$:

1. there exists an FO sentence σ_c using $v(c)$ variables $x_1, \ldots, x_{v(c)}$ such that for each structure $\mathbf{J}$, $\mathbf{J}$ yields c iff $\mathbf{J}$ satisfies σ_c, and

2. at the end of c, each relation in the relational store is either the same as at the start of c, or it is definable from $\mathbf{J}$ by a formula with $v(c)$ variables $x_1, \ldots, x_{v(c)}$.

The proof is by induction on n. Suppose $n = 1$. If q_1 is an update, $\sigma_c = \mathbf{true}$ has the property (1); since q_1 uses $v(c)$ variables, (2) also holds (the variables of q_1 may have to be renamed to $x_1, \ldots, x_{v(c)}$). If q_1 is a query, $\sigma_c = q_1$ satisfies (1) (modulo renaming). Since no update takes place, the relational store remains unchanged and (2) is also satisfied. For the induction step, suppose (1) and (2) hold for n and let c be a computation such that $query(c) = q_1 \ldots q_n q_{n+1}$, where the $v(c)$ variables occurring in $query(c)$ are among $x_1, \ldots, x_{v(c)}$. Let c' be the computation obtained by truncating c after q_n, so that $query(c') = q_1 \ldots q_n$. By the induction hypothesis, there exists $\sigma_{c'}$ satisfying (1) and all relations in the relational store before the application of q_{n+1} either unchanged from the initial configuration or are definable by FO formulas with at most $v(c')$ variables. Consider q_{n+1}. There are two cases:

(i) $q_{n+1}(R_{i_1}, \ldots, R_{i_n})$ is a query using relations $R_{i_1}, \ldots, R_{i_n}$. By the induction hypothesis, each R_{i_j} is either unchanged from the beginning of c, or it is definable from $\mathbf{J}$ using an FO formula φ_{i_j} with at most $v(c)$ variables. Without loss of generality, assume $R_{i_1}, \ldots, R_{i_m}$ are the changed relations and $R_{i_{m+1}}, \ldots, R_{i_n}$ are the unchanged ones. Note that, since the R_{i_j}, $1 \leq j \leq m$ are obtained by updates, $v(c)$ must be at least as large as the arities of these relations. Now let $q_{n+1}(\varphi_{i_1}, \ldots, \varphi_{i_m}, R_{i_{m+1}}, \ldots, R_{i_n})$ be the formula obtained by substituting φ_{i_j} for R_{i_j} for each j, $1 \leq j \leq m$. More precisely, the substitution is obtained as follows. First, we rewrite q_{n+1} so that no literal $R_{i_j}(\vec{x})$ uses repeated variables for $1 \leq j \leq m$ (e.g., $R(z, z)$ is replaced by $\exists x(R(x, z) \wedge x = z)$). Since $v(c) \geq arity(R_{i_j})$, the resulting formula still has at most $v(c)$ variables.

Next, consider a literal $R_{i_j}(z_1, \ldots, z_l)$ occurring in the rewritten formula, and $\varphi_{i_j}(y_1, \ldots, y_l)$ be the FO formula with at most $v(c)$ variables defining R_{i_j}. Let f be a bijection on the variables of φ_{i_j} such that $f(y_i) = z_i$, $1 \leq i \leq l$, and $f(\varphi_{i_j})$ be obtained by replacing in φ_{i_j} each occurrence of a variable y by $f(y)$. Clearly, $f(\varphi_{i_j})$ and φ_{i_j} are equivalent and have the same number of variables. Finally, $q_{n+1}(\varphi_{i_1}, \ldots, \varphi_{i_m}, R_{i_{m+1}}, \ldots, R_{i_n})$ is obtained by replacing each literal $R_{i_j}(z_1, \ldots, z_l)$ by $f(\varphi_{i_j})(z_1, \ldots, z_l)$. The sentence

$$\sigma_c = \sigma_{c'} \wedge q_{n+1}(\varphi_{i_1}, \ldots, \varphi_{i_m}, R_{i_{m+1}}, \ldots, R_{i_n})$$

satisfies (1). Since q_{n+1} does not change any relation in the store, (2) is satisfied as well.

(ii) q_{n+1} is an update of a relation R_h. The proof is almost identical to (i); since no test is involved, $\sigma_c = \sigma_{c'}$ satisfies (1). Since R_h is updated, all relations except R_h are as defined by c', so by the induction hypothesis (2) is satisfied for them. Relation R_h is defined by a formula with no more than $v(c)$ variables, as seen by an argument similar to the above.

□

Consider the complexity of Algorithm **Bootstrap**. Ignoring the complexity of computing f, which can be arbitrary, steps 3 and 4 take time exponential in $|\mathbf{I}|$. It remains open whether this can be improved.

As a consequence of Theorem 5.6, it is clear that the distinguishing power of sublinear reflective machines is captured by the equivalence relation $\equiv_{k(f,\mathbf{I})}$ on $k(f, \mathbf{I})$-tuples. Indeed, we have:

Corollary 5.7 Let f be a sublinear bounding function and $\mathbf{I}$ a structure. The equivalence relations $\equiv_f$ and $\equiv_{k(f,\mathbf{I})}$ on tuples of the same arity are identical.

We can now obtain the following normal form for sublinear reflective machines:

Theorem 5.8 Let f be a sublinear bounding function computable in EXPTIME. For each reflective machine with variable complexity bounded by f there exists an equivalent reflective machine with variable complexity bounded by f whose computation consists of two phases:

1. A computation of complexity EXPTIME producing on the tape a standard encoding of an ordered structure, followed by
2. A computation involving only the tape.

Proof: Let M be a reflective machine with variable complexity $f(n)$. On input $\mathbf{I}$, the first phase computes $k(f, \mathbf{I})$ as in Algorithm **Bootstrap**. (This is in EXPTIME since f itself is in EXPTIME and a brute-force search for $\mathbf{I}_0$ at each iteration can be performed in EXPTIME.) The remainder of the construction mimics the normal form for relational machines as described in [AV91, AV95], except that the summary tables are not explicitly computed (that would require more than $k(f, \mathbf{I})$ variables). More precisely, the classes of $\equiv_{k(f,\mathbf{I})}$ are first computed in some order, and each is stored in a separate relation of the store. This can be done using $k(f, \mathbf{I})$ variables. Next, an integer encoding of $summary_{k(f,\mathbf{I})}(\mathbf{I})$ is generated directly on the tape (the integers are the classes of $\equiv_{k(f,\mathbf{I})}$), using again $k(f, \mathbf{I})$ variables. This concludes the first phase. The second phase simulates the computation of M using exclusively the tape; queries are answered and updates simulated using the information provided by the encoding of $summary_{k(f,\mathbf{I})}(\mathbf{I})$ on the tape. □

The above normal form provides a bridge between computation without order and computation with order for sublinear reflective machines. It shows that a computation of a reflective machine with sublinear variable complexity over an unordered input can be reduced to a computation over an ordered input in exponential time (modulo the complexity of computing the sublinear bounding function). It remains open whether the complexity can be improved. Recall that for relational machines, the first phase of the normal form takes only polynomial time (see Section 2).

As a consequence of the normal form provided by Theorem 5.8, equivalence of structures with respect to reflective machines of sublinear variable complexity can be decided in EXPTIME. More precisely, we have:

Theorem 5.9 Let $\mathbf{I}$, $\mathbf{J}$ be structures over the same schema and f a sublinear bounding function. $\mathbf{I} \equiv_f \mathbf{J}$ iff the following hold:

(i) $k(f, \mathbf{I}) = k(f, \mathbf{J})$ and

(ii) $summary_{k(f,\mathbf{I})}(\mathbf{I}) = summary_{k(f,\mathbf{J})}(\mathbf{J})$.

Proof: Suppose (i) and (ii) are satisfied by $\mathbf{I}$ and $\mathbf{J}$, and consider a reflective machine of sublinear variable complexity. We can assume the machine is in the normal form provided by Theorem 5.8. From (i) and (ii) it follows that $\mathbf{I}$ and $\mathbf{J}$ yield the same result following the first phase of the normal form. Since the second phase does not examine the relational store, $\mathbf{I}$ and $\mathbf{J}$ are either both accepted or both rejected by M. Thus, $\mathbf{I} \equiv_f \mathbf{J}$.

Conversely, suppose $\mathbf{I} \equiv_f \mathbf{J}$. Let $M_{\mathbf{I}}$ be the reflective machine depending on $\mathbf{I}$ which, on input $\mathbf{J}$, accepts $\mathbf{J}$ iff

$$k(f, \mathbf{I}) = k(f, \mathbf{J}) \text{ and } summary_{k(f,\mathbf{I})}(\mathbf{I}) = summary_{k(f,\mathbf{J})}(\mathbf{J}).$$

Clearly, this can be done with variable complexity bounded by $f(|\mathbf{J}|)$. Since $\mathbf{I}$ is accepted by M and $\mathbf{I} \equiv_f \mathbf{J}$, $\mathbf{J}$ is also accepted by M so (i) and (ii) are satisfied. □

It follows that equivalence of structures with respect to machines of sublinear variable complexity can be decided in EXPTIME (again, modulo the complexity of computing f). Theorem 5.9 additionally yields an immediate characterization of $\equiv_f$ by pebble games:

Corollary 5.10 Let f be a sublinear bounding function and $\mathbf{I}$ and $\mathbf{J}$ be structures over the same schema. Then $\mathbf{I} \equiv_f \mathbf{J}$ iff $k(f, \mathbf{I}) = k(f, \mathbf{J}) = k$ and $\mathbf{I}$ and $\mathbf{J}$ are equivalent with respect to the k-pebble game.

Note that the above corollary refines the Fact of Section 5.

We conclude by providing the delayed proof of Lemma 5.1, which makes use of Algorithm **Bootstrap**.

Proof of Lemma 5.1 We show that every sublinear reflective machine with unary input has $O(1)$ variable complexity. This is sufficient because (i) reflective machines with $O(1)$ variable complexity are equivalent to relational machines by [AV95], and (ii) relational machines are equivalent to FO on unary inputs, also by [AV95].

Consider thus a reflective machine with sublinear variable complexity bound f and unary input. Consider first the sequence $\{f^i(0)\}_{i\leq 1}$. Since f is nondecreasing and and $f(0) > 0$, the sequence $\{f^i(0)\}_{i\leq 1}$ is nondecreasing. On the other hand, since $\lim_{n\to\infty} f(n)/n = 0$, there exists N such that $f(n) < n$ for every $n \geq N$. It easily follows that the sequence $\{f^i(0)\}_{i\leq 1}$ eventually becomes constant, that is there exists i such that $f^j(0) = f^i(0)$ for every $j \geq i$. Let $f^*(0)$ denote the limit $f^i(0)$.

Now consider a run of Algorithm **Bootstrap** on unary input $\mathbf{I}$. Clearly, $k(f, \mathbf{I}) = f^*(0)$ for every $\mathbf{I}$ such that $|\mathbf{I}| \geq f^*(0)$. (This uses the fact that for unary input $\mathbf{I}$ and $k \leq |\mathbf{I}|$, the minimum instance $\mathbf{I}_0$ such that $summary_k(\mathbf{I}_0) = summary_k(\mathbf{I})$ has size k.) Thus, the number of variables used by the reflective machine is bounded by a constant, so the machine has $O(1)$ variable complexity. □

Acknowledgement: The authors wish to thank Moshe Vardi for useful discussions on this paper.

References

[AV89] S. Abiteboul and V. Vianu. Fixpoint extensions of first-order logic and Datalog-like languages. In *Proc. Fourth Annual Symposium on Logic in Computer Science*, Asilomar, California, pages 71-79, 1989.

[AV91] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 209–219, 1991.

[AV95] S. Abiteboul and V. Vianu. Computing with first-order logic. *JCSS* 50(2), pages 309–335, 1995.

[AVV97] S. Abiteboul, M. Vardi, and V. Vianu. Fixpoint Logics, Relational Machines, and Computational Complexity. *Journal of the ACM*, **44(1)**: 30-56, 1997.

[AVV95] S. Abiteboul, M. Vardi, and V. Vianu. Computing with Infinitary Logic. *Theoretical Computer Science*, 149:1, pp.101-128, 1995.

[ACY91] Afrati, F., S.S. Cosmadakis, M. Yannakakis, On Datalog vs. polynomial time. In *Proc. 10th ACM Symp. on Principles of Database Systems*, pages 13–25, 1991.

[BDK92] *Building an object-oriented database system, the story of O_2*, eds. F. Bancilhon, C. Delobel and P. Kannelakis, Morgan Kaufmann, 1992.

[Ba77] J. Barwise. On Moschovakis closure ordinals. In *J. of Symbolic Logic*, 42, pages 292-296, 1977.

[Ch81] A.K. Chandra. Programming primitives for database languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 50–62, 1981.

[CH82] Chandra, A.K., D. Harel, Structure and complexity of relational queries, Journal of Computer and System Sciences 25:1 (1982), pages 99-128.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. In *JACM*, pages 114-133, 1981.

[Cl93] S. Cluet. Personal communication, 1993.

[DG92] D.J. Dewitt and J. Gray, Parallel Database Systems: the Future of High Performance Database Systems, Comm. ACM, Vol. 35, no 6, 1992.

[Fri71] H. Friedman. Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory. In R.O.Gangy and C.M.E.Yates, editors, *Logic Colloquium '69*, pages 361–389. North Holland, 1971.

[Gu84] Y. Gurevich. Toward logic tailored for computational complexity In *Computation and Proof Theory*, ed. M.M. Richter et.al, pages 175–216, Springer-Verlag LNM 1104, 1984.

[vE90] P. van Emde Boas. Machine models and simulations. In *Handbook of Theoretical Computer Science*, vol. A, ed. J. van Leeuwen, Elsevier Science Publishers B.V., 1990.

[Fa74] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Computation*, ed. R.Karp, SIAM-AMS Proc. 7, pages 43-73, 1974.

[Im82] N. Immerman. Upper and lower bounds for first-order expressibility. In *JCSS*, 25, pages 76-98, 1982.

[Im86] N. Immerman. Relational queries computable in polynomial time. *Inf. and Control*, 68:86–104, 1986.

[Im87] N. Immerman. Languages which capture complexity classes In *SIAM J. on Computing*, 16(4), pages 760-778, 1987.

[Im89] N. Immerman. Expressibility and parallel complexity. In *SIAM J. on Computing*, 18(3), pages 625-638, 1989.

[IL90] N. Immerman and E.S. Lander. Describing graphs: a first-order approach to graph cannonization. In *Complexity theory retrospective*, ed. A. Selman, pages 59-81, Springer-Verlag, 1990.

[KV87] Kolaitis, P., M.Y. Vardi, The decision problem for the probabilities of higher-order properties, Proc. 19th ACM Symp. on Theory of Computing, 1987, pages 425-435.

[KV90] P.G. Kolaitis and M.Y. Vardi. 0-1 laws for infinitary logic. In *Proc. Symp. on Logic in Computer Science*, pages 156-167, 1990.

[KV92] P.G. Kolaitis and M.Y. Vardi. Fixpoint logic vs. infinitary logic in finite-model theory. In *Proc. Symp. on Logic in Computer Science*, pages 46-57, 1992.

[Lei89a] D. Leivant. Descriptive characterization of computational complexity. *Journal of Computer and System Sciences*, 39:51–83, 1989.

[Lei89b] D. Leivant. Monotonic use of space and computational complexity over abstract structures. Technical Report CMU-CS-89-212, Computer Science Dept., Carnegie-Mellon, 1989.

[O2] *O2* user manual.

[Pa93] C.H. Papadimitriou. *Computational Complexity,* Addison-Wesley, 1993.

[Pa85] C.H. Papadimitriou. A note on the expressive power of Prolog. In *Bulletin of the EATCS*, 26, pages 21-23, 1985.

[Po82] B. Poisat. Deux ou trois choses que je sais de L_n. In *J. of Symbolic Logic*, 47(3), pages 641-658, 1982.

[PS76] V.R. Pratt and L. Stockmeyer. A characterization of the power of vector machines. In *JCSS* 12, pages 198-221, 1976.

[R92] K. Ross. Relations with relation names as arguments: algebra and calculus. In *Proc. 11th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 346-353, 1992.

[Val93] P. Valduriez. Parallel Database Systems: Open Problems and New Issues, Distributed and Parallel Databases 1, 137–165, Kluwer Academic Publishers, Boston, 1993

[VVV93] J. Van den Bussche, D. Van Gucht and G. Vossen. Reflective programming in the relational algebra. In *Proc. 12th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 17-25, 1993.

[Var82] M.Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.