The Power of Reflective Relational Machines (Extended Abstract)

Serge Abiteboul*, Christos H. Papadimitriou† and Victor Vianu‡

March 2, 1995

Abstract

A model of database programming with reflection, called *reflective relational machine* is introduced and studied. The reflection consists here of dynamic generation of queries in a host programming language. The main results characterize the power of the machine in terms of known complexity classes. In particular, the PTIME restriction of the 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, more convenient for problems naturally formulated in logic terms. Since time in the polynomially-bounded machine coincides with time in the uniform circuit model, this also shows that reflection allows for more "intense" parallelism, not attainable otherwise. Other results concern the power of the reflective relational machine subject to restrictions on the number of variables used.

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 in **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 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

*INRIA, B.P. 105, 78153 Le Chesnay CEDEX, France. Serge.Abiteboul@inria.fr.

†‡CSE 0114, UC San Diego, La Jolla, CA 92093-0114. Work performed while visiting INRIA. V.Vianu supported in part by the NSF under grant IRI-9221268.

to circuits, which does not hold for non-reflective machines, suggests that reflection allows to express more "intense" parallelism than non-reflective programming. The study of the reflective relational machine from a parallel computation viewpoint contributes to understanding the computational paradigm underlying the increasingly widespread client/server architecture for database applications, where a client workstation interacts with a database server running on a highly parallel machine [DG92, Val93].

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]). 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*. The results also yield a relation between the relative power of languages with iteration or fixpoint (in particular, inflationary vs. noninflationary fixpoint) and important complexity questions (the P vs. PSPACE problem) [AV91].

Relational machines capture a mode of computation *and programming* that is commonplace in database systems: Relational query languages such as SQL are typically *embedded* in a more general programming environment, such as C or C++. The C/C++program interacts with the database via the SQL queries, and performs a general computation based on the answers to the queries; the queries can thus be iterated in an arbitrarily complex way. It is important to note that the two components of the computation are expected to execute on machines with dramatically different power (say, a workstation and a database server, respectively). 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 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,

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

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

O2]) do away with this limitation, and allow *programmable queries* constructed by the program, and not directly by the programmer. This *reflective* style of database programming ("reflective" in the sense of "program-modifying") seems to be useful and popular among users, for instance in programming user interfaces [Cl93]. Relational machines as restricted 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. The 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. Application-specific software, in particular interface software, is run on the clients, which are typically workstations or personal computers. The database system resides on a parallel machine. Thus, the core of the parallelization consists in the use of a parallel 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 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 2).

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 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 PSPACE 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.

A review of some query languages and of the relational machine, as well as proof sketches for some of the results, are included in an appendix.

2 The Model

A *reflective relational machine* M is a Turing machine with a special *query tape* and a separate *relational store*, capable of storing arbitrary relations $R_0, R_1, R_2, \ldots$. Initially, the relational store contains only the *input relation*[3] R_0 whose arity r_0 depends only on M; all tapes are empty and the initial state prevails. M then computes as an ordinary 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 a FO query. We allow both relation definitions and updates of the form

$$R_j \leftrightarrow \phi(R_{i_1}, \ldots, R_{i_k}),$$

and yes-no queries of the form

$$\phi(R_{i_1}, \ldots, R_{i_k})?$$

where $j, i_1, \ldots, i_k$ are natural numbers, and ϕ is a FO expression over the vocabulary $\{R_{i_1}, \ldots, R_{i_k}\}$. Actually, we allow general updates in addition to the yes/no queries only in

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

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. In response to the query state q, the relational store *in a single step* executes the query on the query tape, suitably updating (or creating) the stored relation R_j —or, in the case of a yes-no query, communicating the yes-no answer on the query tape. If undefined relations are mentioned in the query, or if the query tape contains any other syntax error, the machine halts and rejects.

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 —if it does; the corresponding amount of *space* is the maximum number of tape squares occupied 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 *permutation-invariant;* that is, $S \in \mathcal{S}$ if and only if $S_\pi \in \mathcal{S}$, where S_π is S with its constants permuted by an arbitrary permutation π. Generic sets of structures are called *properties*.

We now make some simple remarks on the power of reflection. The *relational machine* model defined in [AV91] (actually called "loosely coupled generic machine" in that paper; the term "relational machine" was first used in [AVV92]) 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 [AV92] 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

$$\exists x_1 \exists x_2 \ldots \exists x_m [(x_1 \neq x_2) \wedge (x_1 \neq x_3) \wedge \ldots (x_{m-1} \neq x_m)]?$$

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 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 4.

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.

3 Polynomial-time/space Reflective Machines

Polynomial-time reflective machines

We consider next the power of polynomial-time reflective relational machine. 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 3.1 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: See Appendix. □

We now state the result relating polynomial-time reflective machines to PSPACE. *A priori*, the fact that polynomial-time reflective machines stay within PSPACE is counter-intuitive, 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 circuits. Indeed, the proof of our result makes explicit the connection to parallel complexity, via circuits of polynomial depth and exponential size.

Theorem 3.2 The class of properties decidable by reflective relational machines in polynomial time is precisely the class of properties decidable in PSPACE.

Proof: See Appendix. □

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 3.3 The following are equivalent and express PSPACE:

(i) polynomial-time reflective machines;

(ii) polynomial-time reflective machine with constant-arity relations;

(iii) polynomial-space reflective machines with constant-arity relations; and,

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

Theorem 3.2 suggests that reflectiveness allows for a more "intense" data parallelism than in classical, non-reflective query languages. Indeed, consider the standard (non-reflective) relational machine, which, as discussed earlier, subsumes all common query languages.

Theorem 3.4 Assuming P $\neq$ PSPACE, there is a property that can be checked in polynomial-time by a reflective relational machine, but which requires superpolynomial time in any non-reflective relational machine.

Proof: See Appendix. $\square$

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, but there is no restriction on the size of the relational store (which may in fact grow to exponential arity and doubly 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 3.5 Polynomial-space bounded reflective relational machines decide precisely the properties decidable in EXPSPACE.

The nontrivial direction is to simulate exponential space with a polynomial-space bounded reflective relational machine. This is done by first defining an exponentially large relation, whose tuples correspond to the *tape squares* of the simulated machine. The queries then repeatedly update the tape according to the moves of the simulated machine. The details are omitted.

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 3.6 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 3.2. 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 3.7 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 [AV92]. 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 still have an advantage over non-reflective machines for finer bounds where the polynomial simulation makes a difference.

4 Sublinear variable complexity

We now briefly return to 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 monotonically increasing, computable function from positive integers to positive integers, 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 4.1 On unary inputs, reflective machine with sublinear variable complexity are equivalent to machines with $O(1)$ variable complexity, and express the FO properties.

The proof is similar to the argument in [AV91, AV92] showing that relational machines on sets collapse to FO. 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 4.2 .

(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: See Appendix. □

Thus, sublinear machines lie strictly between $O(1)$ and $O(n)$ machines. Moreover, there is a hierarchy within sublinear machines:

Theorem 4.3 Let f and g be sublinear bounding functions such that $\lim_{n\to\infty} f(n)/g(n) = 0$. 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: See Appendix. □

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 over P* can be captured (on unordered inputs) by sublinear machines subject to a time or space bound:

Theorem 4.4 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: See Appendix. □

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 I iff u and v cannot be distinguished by any relational machine with k variables running on input I. See the Appendix 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. One might be tempted to think that the discerning power of a sublinear machine with variable complexity $f(n)$ on input I is captured by $\equiv_{f(|I|)}$. However, the situation is made more complicated by the fact that the number of variables used is determined dynamically. This 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 4.1). Let $k(f,I)$ be the maximum number of variables *actually used* by some reflective machine with variable complexity $f(n)$ on input I. We claim that $k(f,I)$ is computed by the following "bootstrapping" algorithm.

ALGORITHM **Bootstrap**
1. $k := f(0)$;
2. repeat until no further change:
    3. compute $summary_k(I)$;
    4. find I_0 of minimum size such that $summary_k(I_0) = summary_k(I)$;
    5. $k := f(|I_0|)$
6. $k(f,I) := k$.

Note that, in Algorithm **Bootstrap**, the successive values of k computed by the algorithm are nondecreasing. This is because the successive $summary_k(I)$ provide increasingly accurate information about I, so for $k \leq k'$,

$$\{J \mid summary_k(J) = summary_k(I)\} \supseteq \{J \mid summary_{k'}(J) = summary_{k'}(I)\}.$$

We can show:

Theorem 4.5 Algorithm **Bootstrap** computes $k(f,I)$.

Consider the complexity of Algorithm **Bootstrap**. Ignoring the complexity of computing f, which can be arbitrary, step 3. takes polynomial time in $|I|$, but step 4. takes polynomial *space* by a brute-force, exhaustive search for I_0. It remains open whether this can be improved.

It is now apparent that the distinguishing power of sublinear reflective machines is captured by the equivalence relation $\equiv_{k(f,I)}$ on $k(f,I)$-tuples. We can now obtain the following normal form for sublinear reflective machines:

Theorem 4.6 Let f be a sublinear bounding function computable in PSPACE. For each reflective machine with variable complexity $O(f(n))$ there exists an equivalent reflective machine with variable complexity $O(f(n))$ whose computation consists of two phases:

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

Proof: See Appendix. $\Box$

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 sublinear $O(f(n))$ reflective machine over an unordered input can be reduced to a computation over an ordered input in polynomial space (modulo the complexity of f). It remains open whether the complexity can be improved. Recall that for relational machines, the first phase of the normal form takes polynomial time (see Appendix).

Lastly, note that two structures I, J are equivalent with respect to reflective machines of variable complexity $f(n)$ iff $summary_{k(f,I)}(I) = summary_{k(f,J)}(I)$. Thus, equivalence of structures with respect to such machines can be decided in PSPACE (again, modulo the complexity of computing f). Also, it follows that the $k(f,I)$-pebble game for $L_{\infty\omega}^{k(f,I)}$ [Im82, Po82] characterizes equivalence of structures with respect to reflective machines with $f(n)$ variable complexity.

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.

[AV92] S. Abiteboul and V. Vianu. Computing with first-order logic. To appear in *JCSS*.

[AVV92] S. Abiteboul, M. Vardi, and V. Vianu. Fixpoint Logics, Relational Machines, and Computational Complexity. In *Proc. Conf. on Structure in Complexity Theory*, 1992.

[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), pp. 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.

[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 canonization. In *Complexity theory retrospective*, ed. A. Selman, pages 59-81, Springer-Verlag, 1990.

[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.

[O2] *O2* user manual.

[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.

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

5 Appendix

In this section, we provide a brief review of standard notation and previous results, followed by sketches of some of the proofs of results in the paper.

Review

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. In this section we briefly review several important extensions of *FO*: *fixpoint*, *while*, and relational machines.

Fixpoint and While

Most of the extensions of FO with recursion that have been proposed converge towards two classes of queries, *fixpoint* and *while* [CH82]. Although widely varying paradigms can be used to extend FO (e.g. procedural, logic, or logic programming), most languages based on the various paradigms are equivalent to either the *fixpoint* or the *while* queries.

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.

Relational Machines

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".

A *relational machine* is a Turing Machine (TM) augmented with a finite set of fixed-arity relations forming a *relational store*. The relational store can be viewed as associative access storage supporting the generic portion of the computation, while standard computation is carried out on the tape. Designated relations contain initially the input. In a transition, the relational store can be modified through a first-order (FO) query. The input is accepted iff the machine halts in an accepting state.

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 TM component of relational machines provides full computational power, relational machines do *not* express all computable queries. Indeed, 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.

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, AV92], this behavior is due to the fact that relational machines 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 I iff u and v cannot be distinguished by any relational machine with k variables running on input I. As implicit in [IL90, AV91] and shown in [AV92, 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 I, let $summary_k(I)$ be the set of summary tables obtained for 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 $summary_k(I)$. 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 measure is not the most natural one 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 I, called the *k-size* of I. To see the difference between k-size and regular size, note that the k-size of sets is constant, that of binary trees is 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 new measure 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 P_r = *fixpoint* and $PSPACE_r$ = *while*, generalizing the results of [Im86, Var82] that P = *fixpoint* and PSPACE = *while*, *on ordered inputs*.

Proof sketches

Proof of Lemma 3.1: 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 $< x_i, x_j >$, 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.

Proof of Theorem 3.2: 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 $\mathcal{S}$. In fact, we shall look at the computation of the alternating machine on input x as a *circuit* that forms a full binary tree of depth $p(n)$ (where n is the number of constants in the input relation encoded by x, and the gates strictly alternate between AND and OR gates), such that there is a polynomial-time algorithm for deciding, for each input x and leaf ℓ, whether input gate ℓ is TRUE or not in the computation on input x.

We shall exhibit 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 $p(n)$ —the depth of the alternating tree of A. M then encodes its input R_0 as a binary input x of A — this can be done in polynomial time by Lemma 3.1. From the fixed polynomial-time algorithm which decides whether a leaf is TRUE or not, the value of n, and the bitstring x, M next constructs on the tape a circuit C_x of polynomial size, with inputs $y_1, \ldots, y_{p(n)}$, standing for the address of a leaf, with the property that C_x outputs TRUE if and only if with input x to A, the leaf $y_1, \ldots, y_{p(n)}$ is an accepting one. Let $z_1, \ldots, z_q$ be the gates of C_x other than its input gates $y_1, \ldots, y_{p(n)}$ —polynomially many of them. Let $\varphi[C_x](y_1, \ldots, y_{p(n)})$ be a first-order forumula 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 z_2 and z_4, $\varphi[C_x]$ contains the clause

$$(z_5 = 1) \leftrightarrow [(z_2 = 1) \wedge (z_4 = 1)].$$

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 tree 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 circuits $\mathcal{C}_M = (C_1, C_2, \ldots)$ that accepts precisely the encodings of structures in $\mathcal{S}$.

Let us fix n, the number of constants in the input. The circuit C_n consists of the cascading of $p(n)$ copies of the same circuit D_n, which simulates a step of M. D_n has a large number of inputs, and the same number of outputs; in particular, it has $p(n) \cdot (n^{p(n)} + 2)$ inputs and outputs. For each of the possibly up to $p(n)$ relations defined by M (we can assume without loss of generality that M never refers to a relation R_j with $j \geq p(n)$), each relation being of arity up to $p(n)$, 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 and 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 $p(n)$ bits denoting whether each relation has been defined, plus $p(n)$ more bits describing the tapes and state of M.

D_n performs a step of M; naturally, the difficult case is when it is a query step, that is, the state is q. Each query is a first-order expression of length at most $p(n)$, and so it can be expressed as an unbounded fan-in circuit with $p(n)$ 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, where we have $p(n)$ alternating levels, each level with $p(n) \cdot n^{p(n)}$ gates, and each gate having as inputs all gates at the previous level and their negations (naturally, since our gates are of fan-in two, we must simulate this by many full binary trees of gates feeding the next level). So far this circuit computes non-sense; *we must describe how to transform this generic "programmable logic array" into a circuit that implements the present query.* Consider a gate g_{ij}, where $1 \leq i \leq p(n)$ is the level and $1 \leq i \leq n^{p(n)}$ is the index of the gate at the ith level. In the correct circuit that implements the present query, g_{ij} may or may not be an input of $g_{i+1,k}$. We claim that this is a bit, denote it d_{ijk}, *which can be computed in polynomial time from the non-relational part of the input.* Similarly for the negation of g_{ij} being an input of

$g_{i+1,k}$. It follows that D_n can be built in polynomial space, and has depth $O(np^2(n))$. This concludes the proof. □

Proof of Theorem 3.4: Consider a set $\mathcal{S}'$ of ordered structures which is in PSPACE but not in P. Let $\mathcal{S}$ be the property saying that the input is ordered and has property $\mathcal{S}'$. By Theorem 3.2, there exists a reflective relational machine accepting $\mathcal{S}$ in polynomial time. There are also standard relational machines that accept $\mathcal{S}$, since the input is ordered. Suppose this can be done in polynomial time. Then it would follow that $\mathcal{S}$ is in P, contradiction. □

Proof of Theorem 4.2: Part (i) follows immediately from Lemma 4.1, in particular from the fact that sublinear relational machines cannot express evenness. For part (ii), consider a sublinear machine with variable complexity $O(f(n))$. Consider the property of a unary relation S_0 and a binary relation *succ* stating that *succ* is a successor relation, and $|S_0| = f(|succ|)$. 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_0 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. □

Proof of Theorem 4.3: Similarly to the proof of Theorem 4.2, consider the property of a unary relation S_0 and a binary relation *succ* stating that *succ* is a successor relation, and $|S_0| = g(|succ|)$. An argument similar to that in Theorem 4.2 shows that this property is definable by a reflective machine with $O(g(n))$ variable complexity, but not by one with $O(f(n))$ variable complexity. □

Proof of Theorem 4.4: 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'. Clearly C' must contain C. 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 $\mathrm{P} \subseteq C \subseteq C'$). 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. □

Proof of Theorem 4.6: Let M be a reflective machine with variable complexity $cf(n)$ for some constant c. On input I, the first phase computes $k(cf, I)$ as in Algorithm **Bootstrap**, then $summary_{k(cf,I)}(I)$, which is isomorphic to an ordered structure whose elements are the classes of $\equiv_{k(cf,I)}$ (this uses $3k(cf, I)$ variables, i.e. $O(f(n))$). To end the first phase, $summary_{k(f,I)}(I)$ is encoded on the tape. The second phase simulates the computation of M using exclusively the tape; queries are answered using the information provided by $summary_{k(f,I)}(I)$. □