Computing on Structures

Serge Abiteboul[1] and Victor Vianu[2]

[1] INRIA, BP 105, 78153 Le Chesnay CEDEX, France
email: abitebou@inria.inria.fr
[2] CSE C-0114, UC San Diego, La Jolla, CA 92093, USA
email: vianu@cs.ucsd.edu

1 Introduction

One of the exciting developments in complexity theory is the discovery of a very intimate connection between computational complexity and logic. This intimate connection was first discovered by Fagin, who showed that the complexity class NP coincides with the class of properties expressible in existential 2nd-order logic [Fag74] (cf. [JS74]). Another demonstration of this connection was shown by Immerman and Vardi, who discovered tight relationships between the complexity class P and inflationary fixpoint logic [Imm86, Var82] and between the class PSPACE and noninflationary fixpoint logic. This initiated, during the 1980s, a new branch of complexity theory, which focuses on the *descriptive complexity* of problems, i.e. the complexity of describing problems in some logical formalism [Imm87b].

The tight connection between descriptive and computational complexity, typically referred to as the connection between "logic and complexity", was then proclaimed by Immerman [Imm87a], and studied by many researchers[3]. Although the relationship between descriptive and computational complexity is intimate, it is not without its problems, and the partners do have some irreconcilable differences. While computational devices work on *encodings* of problems, logic is applied directly to the underlying mathematical structures. As a result, machines are able to enumerate objects that are logically unordered. For example, while we typically think of the set of nodes in a graph as unordered, it does become ordered when it is encoded on the tape of a Turing machine. This "impedance mismatch" does not pose any difficulty in the identification of NP with existential 2nd-order logic (in [Fag74]), since the logic can simply assert the existence of the desired order. The mismatch with logic becomes apparent, however, at complexity classes below NP. For example, the relationship between the class P and inflationary fixpoint logic is more complicated as a result of the mismatch. Although inflationary fixpoint logic can describe P-complete problems, there are some very easy problems in P that are not expressible in inflationary fixpoint logic (e.g., checking whether the cardinality of the structure is even [CH82]). It is only when we assume built-in order that we get that P coincides with the class of properties expressible in inflationary fixpoint logic [Imm86, Var82]. Similarly, it is only when we assume a built-in order that we get that PSPACE coincides with the class of properties expressible in noninflationary fixpoint logic [Var82].

[3] The focus here is on the connection between finite-model theory and complexity. The connection between logic and complexity has also a proof-theoretic aspect; see [Bus86, GSS90, Lei91]. See [Imm89] for a survey.

We shall present in this paper various devices operating directly on structures, without encoding. The motivation and benefits for doing this are manyfold. On a fundamental level, encodings of structures seem to be a technical device rather than an intrinsic feature. This point has already been made by several mathematicians such as Tarski [Tar86], and Harvey Friedman [Fri71] (see Section 5). It has come up more recently in the context of databases, where devices computing on structures model more acurately database computation carried out against an abstract interface hiding the internal representation of data. Thus, the primary benefit of studying devices and languages computing on structures is that they clarify issues which are obscured in classical devices such as Turing machines. For example, they yield new notions of complexity, quite different from classical computational complexity. They reflect more acurately the actual complexity of computation, which, like database computation, cannot take advantage of encodings of structures. An example is provided by the query *even* on a set[4]: it is in Turing LOGSPACE, but requires EXPSPACE in some of the devices we will consider. We shall examine several notions of complexity, proper to computation on structures.

A related reason for studying devices operating on structures is that this seems a natural way to eliminate the "impedance mismatch" between logic, operating on structures, and devices operating on encodings of structures. Indeed, the investigation of devices computing on structures yields an order-free correspondence between some *classical* questions in computational and descriptive complexity. This order-free correspondence is obtained via the new devices, which act as a mediator between logic and classical devices. A typical result of this type is that *fixpoint* = *while* iff P = PSPACE. Similarly, we present logical analogs of all inclusions between the complexity classes ¿from P to EXPTIME, in terms of various fixpoint logics. Such results are consequences of a powerful normal form for several languages and devices, which provides a bridge between computation without order and computation with order. The removal of the order assumption is technically significant because questions about logical expressiveness over ordered structures are typically much harder than their unordered counterparts; see for example [dR84].

We focus in this paper on the *relational machine* of [AV91, AV91c, AVV92a, AVV92b], and the *generic machine* of [AV91], Section 5 discusses briefly various ancestors and cousins of these devices, from [CH80, Cha81, CH82, Fri71, Lei89a].

2 FO, fixpoint, and while

We review informally three languages which play a central role in our discussion: *FO*, *fixpoint*, and *while*. To begin, we recall some terminology and notation of relational databases [Ull88, Kan91]. This is interchangeable with the closely analogous terminology of logic and finite model theory.

A *database schema* S is a finite set of predicates of fixed arity (denoted $P, Q, R, \ldots$). A *relational database* over S is a finite structure over the predicates in S. A *query* φ is a mapping from finite structures over an *input* schema to finite structures of the same domain (i.e., the same set of elements as the input) over an *output* schema. The mapping must be computable and *generic*: for each automorphism ν of an input

[4] *even*(p)= *true* iff $|p|$ is even.

I, $\varphi(\nu(I)) = \nu(\varphi(I))$. A query language or computing device is called *complete* if it expresses exactly all queries. Complexity classes of queries are defined based on Turing complexity in the classical fashion. For example, the P class consists of all queries φ such that there is a TM which, given on the tape a standard encoding $enc(I)$ of an input I, produces a standard encoding of $\varphi(I)$, in time polynomial in $|enc(I)|$.

Most database query languages are based on extensions of first-order logic without function symbols (FO). Many of these extensions converge around two central classes of queries: the *fixpoint* queries and the *while* queries. We briefly present the languages *FO*, *fixpoint*, and *while*.

The *FO* formulas over predicate symbols $\{R_1, ..., R_n\}$ are built from atomic formulas $R_i(x_1, ..., x_m)$ (R_i of arity m) and equality $x = y$ using the standard connectives $\vee, \wedge, \neg$ and quantifiers $\exists, \forall$. The semantics is also standard. Codd introduced a many-sorted algebraïzation of *FO* called relational algebra that we denote here $\mathcal{A}$ (see [Ull88]) which involves π (projection), $\times$ (cross product), $\cup$ (set union), $-$ (set difference), and $\sigma_{i=j}$ (select from a relation the tuples where the i-th and j-th co-ordinates are equal). It is also important to note that *FO* is in (uniform) AC_0 [Imm87a], so *FO* queries can be evaluated in constant parallel time with polynomial resources.

The *fixpoint queries* (*fixpoint*, also FP) [CH82] are constructed using the first-order constructors as in *FO* together with a fixpoint operator (μ). The fixpoint operator binds a predicate symbol R 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.

The *while* language [Cha81, CH82] also provides an inductive definition of relations up to a fixpoint. $\mathcal{A}$ 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 first-order condition (e.g., $X = \emptyset$) holds[5].

We will see that the same expressive power as FP can be achieved using an *inflationary fixpoint logic* (*IFP*) [GS86], where the iteration of the formula is cumulative, so converges even if the formula is not monotonic in the bound predicate. Also, *while* is equivalent to *noninflationary fixpoint logic* (*NFP*) [AV89]. The operator NFP iterates an arbitrary FO formula up to a fixpoint, which may or may not exist. Therefore, it is only partially defined.

We are concerned with the expressive power and complexity of languages like *fixpoint* and *while*. These turn out to depend in an essential way on assumptions on the database, such as the presence or absence of a total order on the domain. A database is said to be *ordered* if one particular binary relation *succ* provides a successor relation on the elements of the domain. With such an order, *fixpoint* expresses exactly P [Imm86, Var82] and *while* expresses precisely PSPACE [Var82]. On the other hand, both languages collapse to FO on sets, i.e. the set of queries they express on inputs consisting of unordered sets coincides with the FO queries on such inputs. This shows, in particular, that *while* cannot compute many "simple" queries such as the *even* query on a set.

[5] One could define a *cumulative while* using cumulative assignment. The language thereby obtained has exactly the expressive power of *fixpoint*.

A normal form for *while* provides a bridge between computation without order and computation with order. It says, intuitively, that each *while* computation over an unordered domain can be reduced to a *while* computation over an *ordered* domain via a *fixpoint* query. More precisely, a *while* program in the normal form consists of two phases. The first is a *fixpoint* query which performs an analysis of the input. It computes an equivalence relation on tuples which is a congruence with respect to the rest of the computation, in that equivalent tuples are treated identically throughout the computation. Thus, each equivalence class is treated as an indivisible "block" of tuples, which is never split later in the computation. The *fixpoint* query outputs these equivalence classes in some order, so that each class can then be thought of abstractly as an integer. The second phase consists of a *while* query which can be viewed as computing on an *ordered* database obtained by replacing each equivalence class produced in the analysis phase by its corresponding integer. The result that *fixpoint* = *while* iff P = PSPACE is a direct consequence of this normal form. We will come back to this and similar results in Section 3.3.

The *while* queries, and indeed most query languages, are characterized by the fact that they manipulate the input using some finite set of FO queries. The first device (computing on structures) that we consider, called "relational machine", models precisely this type of computation.

3 Relational Machines

The *relational machine (RM)* [AV91, AV91c, AVV92a] extends the *while* queries in the following way. The *while* queries can be viewed as the result of coupling a finite set of FO queries with a finite-state control. In relational machines, the finite-state control of *while* is extended to a computationally complete device. It consists of a Turing Machine (TM) augmented with a finite set of fixed-arity relations forming a *relational store*. We describe it next.

3.1 Description and normal form

The machine works as follows. The input is given in designated relations of the store, and the result is computed in designated 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 the store satisfies some FO condition, and assign to a relation the result of an FO query on the store. An RM has *arity* k if all relations in the store have arity at most k and the maximum number of variables in its FO queries is at most k.

Thus, relational machine provides arbitrary computation interacting with the database by FO queries posed against an abstract interface (the relational store). This models accurately database application programs, which often use a FO relational language (say, SQL) embedded in a full programming language (say, C). The TM part of our machine models the use of C, whereas the FO queries on the relational store correspond to the SQL queries.

Since a relational machine has a TM component, it can perform arbitrarily complex computations on the tape. Thus the relational machines are complete for ordered inputs. However, it is not complete in general. Indeed, it cannot compute *even*, and in fact collapses to FO on sets. This wide variation in expressive power generalizes the situation for the *while* language and has similar causes. Like *while*, the relational machine interacts with the databases via a finite set of FO queries, which limits its ability to distinguish among tuples in the input. Essentially, relational machine can perform *any* computation on equivalence classes of tuples it can distinguish. As for *while*, the information about these equivalence classes can be computed by a *fixpoint* query. This yields a generalization of the normal form for *while* to relational machine, which says essentially that any relational machine computation can be reduced to a relational machine computation over an ordered input via a *fixpoint* query. More precisely, for each relational machine M, there exists an equivalent machine which works in four phases. The first phase is equivalent to a *fixpoint* query which performs the analysis of the input exactly like for *while*. In the second phase, the result of the analysis is coded on the tape using the integers representing the equivalence classes. (This is done in P.) Next, the computation is carried out exclusively on the tape. The content of a relation is represented at all times by a sequence of integers representing the equivalence classes it contains. Lastly, if the simulation halts, the representation of the result (a sequence of integers) is decoded from the tape into the output relations. This last phase is not needed if the machine is an acceptor.

3.2 Robustness

A particularly appealing feature of relational machines is their robustness. First, RM is equivalent to an extension of the while language with integer arithmetic. Let $while_N$ [Cha81, CH82] be the *while* language augmented with (i) integer variables $i, j, ...$, initialized to zero; (ii) $increment(i)$ and $decrement(i)$ statements; and (iii) tests of the form $i = 0$ in termination conditions of loops. From the equivalence between TM's and counter machines, it is is immediate that RM is equivalent to $while_N$.

It is also possible to extend relational machines with more flexible interfaces to the relational store without change in expressive power. One such extension allows dynamic generation of queries. That is, one can add a "query tape" to the machine. The machine can generate dynamically (encodings of) FO queries using up to k variables on the query tape. The machine can request at any time the application of the query on the tape against the relational store, and use it either as a test in the control or to assign the result to a relation in the store. An orthogonal extension allows addressable relations, i.e. the relational store has a variable, unbounded number of relations of bounded arity. Neither extension affects expressive power.

Finally, there is a natural correspondence between RM and infinitary logic. Infinitary logic with finitely many variables (denoted $L^{\omega}_{\infty\omega}$) (see [Ba77]) is first-order logic (FO) extended by allowing disjunctions and conjunctions of infinite sets of formulas. Although it generally has non-effective syntax, and can define non-computable queries, $L^{\omega}_{\infty\omega}$ subsumes most query languages and thus provides an elegant unifying formalism for a wide variety of languages [ACY91, KV90b]. It is easy to see that RM is also subsumed by $L^{\omega}_{\infty\omega}$. Moreover, RM corresponds *precisely* to the natural effective fragment of $L^{\omega}_{\infty\omega}$, which we describe next. We define a semantic notion of recursively enumerable (r.e.) formula. The notion of r.e. requires the set of models of the sentence to be r.e. More precisely, a set S of finite structures over some schema σ is *recursively enumerable* (r.e.) if there exists a recursive enumeration $I_1, ..., I_i, ...$ of structures over σ such that I is in S iff some structure isomorphic to I belongs to the enumeration.

The main result on the connection between RM and $L^{\omega}_{\infty\omega}$-formulas whose sets of models are r.e. is the following (note that this also yields a syntactic normal form for such formulas). Let S be a set of finite structures over some fixed schema. The following are equivalent:

(i) S is accepted by some relational machine;
(ii) $S = models(\varphi)$ for some $\varphi \in L^{\omega}_{\infty\omega}$ and S is r.e.;
(iii) For some k, $S = models(\bigvee A)$, for some recursive set A of FO sentences using k variables[6].

The connection between relational machines and $L^{\omega}_{\infty\omega}$ is developed in [AVV92b]. It is also important to note that it yields a characterization of the discerning power of relational machines in terms of certain infinite 2-player pebble games, which we outline next.

The *k-pebble game* between the Spoiler and the Duplicator on the l-tuples ($l \leq k$) $\mathbf{u}$ and $\mathbf{v}$ of a structure I has the following rules. Each of the players has k pebbles, say $p_1, \ldots, p_k$ and $q_1, \ldots, q_k$, respectively. The game starts with the Spoiler choosing one of the tuples $\mathbf{u}$ or $\mathbf{v}$, placing $p_1, \ldots, p_l$ on the elements of the chosen tuple (i.e., on $u_1, \ldots, u_l$ or $v_1, \ldots, v_l$), and placing the pebbles $p_{l+1}, \ldots, p_k$ on some elements of I. The Duplicator responds by placing the pebbles $q_1, \ldots, q_l$ on the elements of the other tuple, and placing the pebbles $q_{l+1}, \ldots, q_k$ on some elements of I. In each following round of the game, the Spoiler moves some pebble p_i (or q_i) to another element of I, and the Duplicator responds by moving the corresponding pebble q_i (or p_i).

Let a_i (resp., b_i), $1 \leq i \leq k$, be the elements of I under the pebbles p_i (resp., q_i), $1 \leq i \leq k$, at the end of a round of the game. If the mapping h with $h(a_i) = b_i$, $1 \leq i \leq k$, is not an isomorphism between the substructures of I 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 the tuple $\mathbf{u}$ and $\mathbf{v}$ of a structure I, then we say that $\mathbf{u}$ and $\mathbf{v}$ are k-equivalent, denoted $\mathbf{u} \equiv_k \mathbf{v}$, over I. The relation $\equiv_k$ characterizes the discerning power of k-ary relational machines: The tuples $\mathbf{u}$ and $\mathbf{v}$ are k-equivalent over a structure I if and only if no k-ary relational machine M can discern between $\mathbf{u}$ and $\mathbf{v}$ over I.

The results on the robustness of RM allow to identify the factors that have essential impact on its expressive power. Thus, it is apparent that the limit on the arity of the relations in the store, and on the number of variables in FO formulas used to manipulate it, are determining factors. In contrast, expressive power is insensitive

[6] We note that the normal form provided by (iii) can be viewed as the effective counterpart of a similar result shown for the full $L^{k}_{\infty\omega}$ in [KV92]. It is also shown in [KV92] that every formula of $L^{k}_{\infty\omega}$ is equivalent to a countable disjunction of FO^l-formulas, for some $l \geq k$. Furthermore, it follows from [DLW91] that this holds also for $l = k$.

to variations such as whether the number of relations in the store is fixed or variable, and whether the set of FO queries used is fixed or generated dynamically.

3.3 Relational Complexity

As seen above, relational machines subsume a wide variety of query languages. We would like to use RM to gain better understanding of the complexity of queries expressed in such languages. This can be done by studying complexity classes for RM, defined by the amount of resources (space, time) required in RM computations on given inputs. However, the definition poses an interesting problem. Recall that for classical devices, like TM, the measure of the input is its size. For relational machines, the size of the input (as for TM) is not an appropriate measure for relational machines. This is because, unlike Turing machines, relational machines have limited access to their input, and in particular cannot generally compute its size. Indeed, relational machines are less "discerning" than Turing machines. The discerning power of k-ary RM was captured by the equivalence relation $\equiv_k$. Indeed, recall that $u \equiv_k v$ iff u and v are not distinguished by any RM of arity k.

Since k-ary relational machines cannot discern among k-equivalent tuples, they cannot measure the size of their input. Instead, computations of k-ary relational machines are determined by the k-equivalence classes. Intuitively, relational machines are complete on the equivalence classes of tuples they can distinguish. This is best understood by looking at the extremes. As shown in [AV91], relational machines are complete on ordered inputs (where all distinct tuples can be distinguished from each other), but they collapse to first-order logic on unordered sets (where the number of k-equivalence classes is bounded by a constant independent on the size of the input). Thus, it seems natural to measure the input size with respect to $\equiv_k$. Let the *k-size* of a structure I, denoted $size_k(I)$, be the number of $\equiv_k$-classes of k-tuples over I. As in [AV91, AV91c], we propose to measure the time or space complexity of k-ary relational machines as a function of the k-size of their input. This measure, however, can reasonably serve as a basis for measuring complexity only if it can be calculated by relational machines. The techniques of [AV91, AV91c] can now be used to show that relational machines *can* measure[7] k-size.

From now on we measure the complexity of k-ary relational machines in terms of the k-size of their input. For instance, we can define the relational complexity class P_r (relational polynomial time), $\mathrm{NPSPACE}_r$, etc. We already know the effective enumerability of these classes. In contrast, it is not clear how to get effective enumerability if we define relational complexity in terms of the actual size of the input.

Let C denote a Turing complexity classes and C_r denotes the analog relational complexity class. All complexity classes considered here include P. It is easily seen that, for C at least P, $C_r \subset C$. Note that, a priori, known relationships between deterministic, nondeterministic, and alternating complexity classes, such as PSPACE=NPSPACE=APTIME, need not hold for relational complexity classes, since these relationships are typically the result of simulations that use order. However, the normal form for RM allows to prove the following.

[7] An alternate proof of the result, which makes an explicit connection with the pebble games, was recently given in [DLW91].

Let C, C' be complexity classes. Then $C = C'$ iff $C_r = C'_r$.

3.4 Relational complexity and fixpoint logics

A measure of the naturaleness of relational complexity is that, as will be shown in this section, there is a seemless match between relational complexity and various fixpoint logics. We begin by defining several such logics.

Noninflationary Fixpoint Logic We already considered inflationary fixpoint logic IFP. A more powerful logic results if one iterates general 1st-order operators, until a fixpoint is reached (which may never happen). In this case we may have non-terminating computations, unlike inflationary fixpoint logic, where the iteration was guaranteed to converge. Noninflationary fixpoint logic (NFP) coincides with the query language *while* and thus captures the complexity class PSPACE.

IFP and NFP are obtained by iterating inflationary and noninflationary 1st-order operators, respectively. In both cases, the iteration is sequential and deterministic. Certain problems, however, seem to defy description by such iterations (e.g., nonuniversality of finite automata over a binary alphabet). One can consider a *nondeterministic iteration* of 1st-order operators. *Nondeterministic fixpoint logics* are obtained by augmenting 1st-order logic with the nondeterministic (inflationary and noninflationary) fixpoint formation rules, under the restriction that negation cannot be applied to nondeterministic fixpoints[8].

Another type of fixpoint logic is obtained by allowing alternation in the control of the iteration. This yields alternating fixpoint logics. Let Φ and Ψ be 1st-order operators. This pair of operators generates *convergent trees* of stages that are obtained by successively applying, till convergence is reached, either one of Φ and Ψ or both of Φ and Ψ. See [AVV92a] for more motivation on the fixpoint logics.

The discussion so far shows that fixpoint logics can be parameterized along two dimensions: the power of their iteration construct, *d*eterministic vs. *n*ondeterministic vs. *a*lternating, and the power of their 1st-order operators, *i*nflationary vs. *n*oninflationary. This gives rise to six fixpoint logics. We use the notation $FP(\alpha, \beta)$ to refer to a fixpoint logic with iteration of type α and operators of type β. Thus, the logics IFP and NFP will be denoted $FP(D, i)$ and $FP(D, n)$, respectively, and $FP(A, n)$ denotes alternating noninflationary fixpoint logic.

The link between relational complexity and the fixpoint logics is provided by the following:

$$FP(D,i) = \mathrm{P}_r; \quad FP(D,n) = FP(N,n) = FP(A,i) = \mathrm{PSPACE}_r;$$
$$FP(N,i) = \mathrm{NP}_r; \; FP(A,n) = \mathrm{EXPTIME}_r.$$

An immediate consequence is that $FP(A, i)$, $FP(D, n)$ $FP(N, n)$ all have the same expressive power.

[8] The nondeterminism described above resides in the *control*, i.e. the choice of the formula to be applied at each stage. This should be contrasted with the *data nondeterminism* of [AV91a], which allows to nondeterministically choose an arbitrary tuple from a relation. Data nondeterminism is strictly stronger than control nondeterminism. For instance, with data nondeterminism, one can (nondeterministically) order the domain elements and use the constructed order to compute the parity of the set of domain elements.

These results also demonstrate how complexity theory can yield *unconditional* results about expressive power of logics. (An example where complexity theory yields *conditional* results about expressive power of logics appeared in [Fag74]. Fagin showed that existential and universal second-order logics coincide iff Hamiltonicity is expressible in universal second-order logic.)

3.5 Relational complexity as mediator

A consequence of Fagin's results is that NP=co-NP if and only if existential and universal 2nd-order logic have the same expressive power. This equivalence of questions in computational and descriptive complexity is one of the major features of the connection between the two branches of complexity theory. It holds the promise that techniques from one domain could be brought to bear on questions in the other domain.

Unfortunately, the order issue complicates matters. The results by Immerman and Vardi show that P=PSPACE if and only if inflationary and noninflationary fixpoint logics have the same expressive power over *ordered* structures. Because of this restriction, the above equivalence provides a translation of the P vs. PSPACE question to a descriptive complexity question, but does not provide a translation of the inflationary vs. noninflationary question to a computational complexity question. Thus, it is desirable to obtain an order-free correspondence between questions in computational and descriptive complexity. (See also [IL90] for a discussion of the order issue from another perspective.)

We can now provide a partial solution to the order issue, by using relational machines as a mediator between classical complexity and fixpoint logics. On one hand, we showed that questions about containments among standard complexity classes can be translated to questions about containments among relational complexity classes. On the other hand, the expressive power of fixpoint logic can be *precisely* characterized in terms of relational complexity classes. This tight three-way relationship among fixpoint logics, relational complexity and standard complexity yields in a uniform way logical analogs to all containments among the complexity classes P, NP, PSPACE, and EXPTIME. For example[9], P = PSPACE iff $FP(D,i) = FP(D,n)$, P = NP iff $FP(D,i) = FP(N,i)$, NP = PSPACE iff $FP(N,i) = FP(N,n)$, and PSPACE = EXPTIME iff $FP(A,i) = FP(A,n)$. This also enables us to translate known relationships among complexity classes, such as the equality of PSPACE, NPSPACE, and APSPACE, into results about the expressive power of fixpoint logics. This fulfills the promise of applying results from one domain to another domain and shows that some of the most tantalizing questions in complexity theory – P. vs. PSPACE, NP vs. PSPACE, and PSPACE vs. EXPTIME – boil down to *one fundamental issue*: the relative power of inflationary vs. noninflationary 1st-order operators.

4 The Generic Machine

In previous sections, we focused on the relational machine, a very powerful device which nonetheless is not complete. In this section we consider a complete extension

[9] The first equivalence was shown in [AV91] by a direct proof using the normal form for *while*.

of the relational machine, called *generic machine*. This yields yet a new kind of complexity, *generic complexity*.

The *generic machine* (GM) of [AV91] differs ¿from the relational machine in the interaction between the tape and the relational store. GM allows loading the content of relations on the tape, and storing tuples back into relations. To load a relation in a deterministic way, tuples cannot be put on the tape in some arbitrary sequence. A natural solution is to introduce parallelism. A load operation therefore involves spawning a new copy of the machine for each tuple loaded. All copies then compute synchronously in parallel. There is a mechanism for merging parallel machines. The output is only obtained after all machines are merged into a single one, which ensures genericity of the global computation. It was shown in [AV91] that GM can express all queries.

There are many alternative ways of defining complete devices operating directly on structures. Recall that *while* cannot go beyond PSPACE because (i) it uses, throughout the computation, only values from the input, and (ii) it uses relations of fixed arity. The addition of integers as in $while_N$ is one way to break the space barrier. The parallelism of GM is another. Yet another is to relax (i) or (ii). Relaxing (i) is done by allowing the creation of new values, not present in the input. Relaxing (ii) yields an extension of *while* with *untyped algebra*, i.e. an algebra of relations with variable arities. We next describe two complete languages obtained by relaxing (i) and (ii).

The first is an extension of *while* allowing the creation of new values throughout the computation called $while_{new}$. The language *while* is modified as follows: (i) there is a new instruction $R := new(S)$, where R and S are relational variables and $arity(R) = arity(S) + 1$; and (ii) the looping construct is of the form *while* R *do* s, where R is a relational variable. The semantics of (i) is the following. Relation R is obtained by extending each tuple of S by one distinct new value from **dom**, not occurring in the input, the current state, or in the program. The semantics of *while* R *do* s is that statement s is executed while R is nonempty.

Note that the *new* construct is, strictly speaking, nondeterministic. Indeed, the new values are arbitrary, so several possible outcomes are possible depending on the choice of values. However, the different outcomes differ *only* in the choice of new values. Therefore, we must consider only *well-behaved* $while_{new}$ programs whose answer never contains values introduced by the new statements. It is possible to give a syntactic restriction on $while_{new}$ programs which guarantees good-behavior, can be checked, and yields a class of programs equivalent to all well-behaved $while_{new}$ programs.

One can show that the language of well-behaved $while_{new}$ programs is complete[10]. This is proved using new values to denote enumerations of domain elements.

A second extension of *while* (historically, the first language shown to be complete [CH80]) that we call $while_{uty}$ is obtained by relaxing the fixed-arity requirement present in the languages encountered so far. This is done by using an *untyped* version

[10] One may also consider queries containing new values in the answer. Such queries arise naturally in the context of object-oriented databases where new object identifiers appear in query results. It is interesting to note that $while_{new}$ is not complete with respect to this extended notion of query which leads to introduce other constructs [AK89].

of relational algebra instead of the familiar typed version.

Based on GM, we can define complexity classes proper to generic computation, focusing on generic analogs of PTIME (GEN-PTIME) and PSPACE (GEN-PSPACE). These classes are robust. Indeed, the natural polynomial time and space restrictions of $while_{uty}$ and $while_{new}$ coincide respectively with GEN-PTIME and GEN-PSPACE. Similar results are obtained for other computation models in [AK89, DV91].

The classes GEN-PTIME and GEN-PSPACE are studied in [AV91]. In particular, it is shown that, in agreement with several known nonexpressibility results, *even* is a hard query relative to the notion of generic complexity provided by GM. Indeed, it is in GEN-EXPSPACE but not in GEN-PSPACE. Such results point to a trade-off between complexity and computing with an abstract interface.

5 Other work on computing on structures

A number of mathematicians have been interested in computation carried out directly on structures. Several recursion theorists and logicians have adressed the issue, motivated by the question of how tied recursion theory is to the particular structure of integers, or to representations by ordered strings. Typical in that respect is the work of Harvey Friedman on *formal algorithmic procedures (fap)* [Fri71]. The different motivation generally renders this and similar models incomparable to models directly motivated by database queries, such as relational machines. Nonetheless, we briefly describe next some of his and other work that appears related, at least in spirit, to our discussion.

Let σ be a first-order signature (language) with predicate and function symbols. Equality is assumed present and always interpreted as equality. A *formal algorithmic procedure (fap)* over σ is a program using finitely many variables $x_1, \ldots, x_k$ (representing domain elements), the predicate and function symbols of σ, and consisting of a finite concatenation of labeled instructions of the form:

- if $R(x_{i_1}, \ldots, x_{i_r})$ then i else j,
  where R is a predicate symbol of σ of arity r, x_{i_j} are variables, and i, j are instruction labels;
- $x_{i_0} = f(x_{i_1}, \ldots, x_{i_r})$,
  where f is a function symbol of σ of arity r and x_{i_j} are variables; and,
- stop.

Suppose a structure over σ is given, as well as an initialization for the first n ($n < k$) variables. The semantics of the instructions is the obvious one. Note that, at each point in the computation, each variable either holds a value or is undefined if no value has been assigned to it. Thus, a fap can be viewed as an acceptor of n-tuples over the domain of σ: $< a_1, \ldots a_n >$ is accepted iff the program stops and x_k is defined; otherwise (x_k is underfined or the computation does not stop) $< a_1, \ldots, a_n >$ is rejected. In particular, each fap p over σ defines a partial mapping ¿from structures $\mathbf{I}$ over σ to n-ary relations over the same domain as σ (denoted also by p), as follows:

$$p(\mathbf{I}) = \begin{cases} \text{undefined, if } p \text{ does not terminate on some } n\text{-tuple of elements in } \mathbf{I} \\ \{< a_1, \ldots, a_n > \mid p \text{ accepts } < a_1, \ldots, a_n >\} \text{ otherwise} \end{cases}$$

Clearly, such a mapping is a query (by the definition of Chandra and Harel). It is easily verified that any query defined by a fap is in LOGSPACE. Furthermore, Hartmanis [Har72] and later Gurevich [Gur88], showed that the set of queries computed by fap on ordered structures is *exactly* LOGSPACE. Leivant [Lei89a] reviews these results in the context of a syntactic variant of the fap called *sequential on-site acceptor*. This also provides insight into the connection with FO; since FO is less than LOGSPACE even on ordered structures, fap compute non-FO queries. On the other hand, there are simple FO queries that fap obviously cannot compute, such as the projection of a relation.

There are several ways to add power to fap. One way, considered by Friedman, is to add counting capability to fap, similarly to $while^+$. This is done by allowing integer variables, increment and decrement instructions, and an *if i=0 then i else j* instruction. This yields the *fap with counting (fapc)*. This is strictly more powerful than fap, and complete on ordered structures. Although similar in spirit to $while^+$ the fapc is nonetheless weaker since, like the fap, it cannot compute the projection of a relation. On the other hand, it is easily seen that every fapc can be simulated by a $while^+$ program.

Another way to augment the power of the fap is proposed by Leivant in [Lei89a]. This consists in allowing (second order) typed relational variables $X, Y, \ldots$ in addition to domain variables, and instructions of the form:

- $\epsilon X(x_{i_1}, \ldots, x_{i_r}$, where $\epsilon \in \{+, -\}$, r is the arity of X, and x_{i_j} are domain variables;
- if $X(x_{i_1}, \ldots, x_{i_r})$ then i else j.

The first instruction provides tuple insertion and deletion into variable relations, and the second extends the *if* instruction to such variables. Let us call such programs *second order fap (SO fap)*. Clearly, SO fap can be evaluated in PSPACE. On ordered structures, they express precisely PSPACE [Lei89a]. Without order, they are strictly included in *while*.

As an analog of Turing machine to arbitrary structures, Friedman proposes *generalized Turing algorithms (gTa)* [Fri71]. In addition to usual tape symbols called *auxialliary*, a gTa can place on the tape elements from the structure. The machine can tell if it scans an auxiliary symbol or a data element, but can only read auxiliary symbols. The manipulation of data elements on the tape remains generic. More precisely, a gTa starts with an n-tuple of domain elements in adjacent squares on the tape, the head on the leftmost square, and the other squares blank. Besides usual moves, it can switch the content of adjacent squares, test membership of an r-tuple of consecutive square values into a relation and replace a value by the result of applying a function in the structure to a tuple of consecutive values. Acceptance is defined in the natural way. Clearly, a gTa defines a query from structures to relations consisting of the n-tuples accepted by the machine. Friedman shows that fapc are strictly weaker than gTa. Interestingly, fap augmented with *stacks* of data elements are equivalent to gTa.

Friedman intention for gTa to compute all "reasonable" functions on structures. Clearly, this does not fit the database domain, since gTa cannot compute many queries (like the projection of a relation) which are considered essential. The emphasis in Friedman's and other similar investigations is on lifting classical theorems of recursion theory to models of computation such as gTa. Leivant uses his version of fap, on-site acceptors, to obtain descriptive complexity characterizations of complexity classes, but his results assume ordered structures, so the focus is again different from our study of relational machines. Yet another related line of investigation concerns *program schemata* [LP89, LPP70], which are essentially the same as fap. The emphasis here is on program verification. See the survey [She85] for an excellent overview of Friedman's work on fap, and related work.

The notion of computation or algorithm which use only properties of given structures, has been of interest to other logicians and computer scientists from various points of view. Gurevich points out the naturaleness of such algorithms [Gur88]. Tarski discusses in [Tar86] *logical notions*, i.e. notions which are invariant under isomorphisms of the domain of discourse. Thus a generic query is a logical notion; and, a computational device becomes a logical notion when its definition depends only on the information present in the structure it operates on. Turing machines are not logical notions with respect to the an arbitrary structure, but are so relative to its encoding on the tape; relational machines, generic machines, fap and gTa are logical notions with respect to the structure.

References

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

[AK89] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 159–173, 1989. to appear in *J. ACM*.

[AV89] S. Abiteboul and V. Vianu. Fixpoint extensions of first-order logic and Datalog-like languages. In *Proc. 4th IEEE Symp. on Logic in Computer Science*, pages 71–79, 1989.

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41:181–229, 1990.

[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.

[AV91a] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.

[AV91c] S. Abiteboul and V. Vianu. Computing with first-order logic. To appear in *Journal of Computer and System Sciences*.

[AVV92a] S. Abiteboul, Moshe Y. Vardi, and V. Vianu. Fixpoint logics, relational machines, and computational complexity. In *Proc. Conf. on Structure in Complexity Theory*, Boston, 1992.

[AVV92b] S. Abiteboul, Moshe Y. Vardi, and V. Vianu. Computing with infinitary logic. To appear in *Int'l. Conf. on Database Theory*, Berlin, 1992.

[Ba75] Barwise, J., *Admissible Sets and Structures*, Springer-Verlag, 1975.

[Ba77] Barwise, J., On Moschovakis closure ordinals, *J. Symbolic Logic*, 42 (1977), pp. 292–296.

[BF85] Barwise, J., S. Feferman (eds.), *Model-Theoretic Logics*, Springer-Verlag, 1985.

[Bus86] S. R. Buss. *Bounded Arithmetics*. Bibliopolis, 1986.

[CH80] A.K. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

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

[Cha81] A.K. Chandra. Programming primitives for database languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 50–62, 1981.

[CKS81] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[Com88] K. J. Compton. An algebra and a logic for NC^1. In *Proc. 3rd IEEE Symp. on Logic in Computer Science*, pages 12–21, 1988.

[DLW91] A. Dawar, S. Lindell, and S. Weinstein. Infinitary logic and inductive definability over finite structures. Research report, Univ. of Pennsylvania, 1991.

[DV91] K. Denninghoff and V. Vianu, The Power of Methods with Parallel Semantics, In Proc. Intern. Conf. on Very Large Data Bases, pages 221-232, 1991.

[Fag74] R. Fagin. Generalized first–order spectra and polynomial–time recognizable sets. In R. M. Karp, editor, *Complexity of Computation, SIAM-AMS Proceedings, Vol. 7*, pages 43–73, 1974.

[Fag75] R. Fagin. Monadic generalized spectra. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:89–96, 1975.

[F90] Fagin R., Finite-Model Theory—a Personal Perspective, Proc. 3rd Int'l. Conf. on Database Theory, Springer-Verlag, Lecture Notes in Computer Science 470, 1990, pp. 3–24, to appear in *Theoretical Computer Science.*

[Fed81] T. Feder. *Stable networks and product graphs.* PhD thesis, Stanford University, 1981.

[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.

[Goe89] A. Goerdt. Characterizing complexity classesby higher-type primitive-recursive definitions. In *Proc. 4th IEEE Symp. on Logic in Computer Science*, pages 364–374, 1989.

[Gra84] E. Grandjean. The spectra of first-order sentences and computational complexity. *SIAM Journal on Computing*, 13:356–373, 1984.

[Gra85] E. Grandjean. Universal quantifiers and time complexity of random access machines. *Mathematical System Theory*, 13:171–187, 1985.

[Gur83] Y. Gurevich. Algebras of feasible functions. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 210–214, 1983.

[Gur84] Y. Gurevich. Toward logic tailored for computational complexity. In M. M. Ricther et al., editor, *Computation and Proof Theory, Lecture Notes in Mathematics 1104*, pages 175–216. Springer-Verlag, 1984.

[Gur88] Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current trends in theoretical computer science*, pages 1–57. Computer Science Press, 1988.

[GS86] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.

[GSS90] J-Y. Girard, A. Scedrov, and P. SCott. Bounded linear logic: a modular approach to polynomial time computability. In S. R. Buss and P. Scott, editors, *Feasible Mathematics*, pages 195–207. Birkhauser, 1990.

[Har72] J. Hartmanis. On nondeterminacy in simple computing devices. *Acta Informatica*, 1:336–344, 1972.

[IL90] N. Immerman and E. S. Lander. Describing graphs: a first-order approach to graph canonization. In A. Selman, editor, *Complexity Theory Retrospective*, pages 59–81. Springer-Verlag, 1990.

[Imm82] N. Immerman. Upper and lower bounds for first-order expresibility. *Journal of Computer and System Sciences*, 25:76–98, 1982.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[Imm87a] N. Immerman. Expressibility as a complexity measure: results and directions. In *Second Structure in Complexity Conference*, pages 194–202, 1987.

[Imm87b] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[Imm89] N. Immerman. Descriptive and computational complexity. In J. Hartmanis, editor, *Computational Complexity Theory, Proc. Symp. Applied Math., Vol. 38*, pages 75–91. American Mathematical Society, 1989.

[JS74] N. G. Jones and A.L. Selman. Turing machines and the spectra of first-order formulas. *Journal of Symbolic Logic*, 39:139–150, 1974.

[Kan91] P. C. Kanellakis. Elements of relational database theory. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1074–1156. North Holland, 1991.

[KV90a] Ph. G. Kolaitis and M. Y. Vardi. 0-1 laws for infinitary logics. In *Proc. 5th IEEE Symp. on Logic in Computer Science*, pages 156–167, 1990.

[KV90b] Ph. G. Kolaitis and M. Y. Vardi. On the expressive power of Datalog: tools and a case study. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pages 61–71, 1990. Full version appeared in IBM Research Report RJ8010, March 1991.

[KV92] Kolaitis, P., M.Y. Vardi, Fixpoint vs. infinitary logic in finite-model theory, to appear in *Proc. 7th IEEE Symp. on Logic in Computer Science*, 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. Unpublished manuscript, 1989.

[Lei90] D. Leivant. Inductive definitions over finite structures. *Information and Computation*, 89:95–108, 1990.

[Lei91] D. Leivant. A foundational delineation of computational feasibility. In *Proc. 6th IEEE Symp. on Logic in Computer Science*, pages 2–11, 1991.

[Lin91] S. Lindell. An analysis of fixed point queries on binary trees. *Theoretical Computer Science*, 85:75–95, 1991.

[LP89] D. Luckham and D. Park. The undecidability of the equivalence problem for program schemata. Technical Report UCSC-CRL-89-40, UCSC, December 1989. to appear in *ACM Transactions on Database Systems*.

[LPP70] D. Luckham, D. Park, and M. Paterson. Formalized computer programs. *Journal of Computer and System Sciences*, 4:220–249, 1970.

[Mos74] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North Holland, 1974.

[dR84] M. de Rougemont. Uniform definability on finite structures with successor. In *Proc. 16th ACM Symp. on Theory of Computing*, pages 409–417, 1984.

[Saz80b] V. Yu. Sazonov. Polynomial computability and recursivity in finite domains. *Elektronische Informationverarbeitung und Kybernetik*, 16:319–323, 1980.

[She85] J. C. Shepherdson. Algorithmic procedures, generalized turing algorithms, and elementary recursion theory. In L.A.Harrington et al., editor, *Harvey Friedman's research on the foundations of mathematics*, pages 285–308. North Holland, 1985.

[Sto77] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[Tar86] A. Tarski. What are logical notions? *History and Philosophy of Logic*, 7:143–154, 1986. (Ed. J.Corcoran).

[TU88] J. Tiuryn and P. Urzyczyn. Some relationships between logic of programs and complexity theory. *Theoretical Computer Science*, 60:83–108, 1988.

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

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

This article was processed using the LaTeX macro package with LLNCS style