Generic Computation and Its Complexity*

Serge Abiteboul
I.N.R.I.A.
Rocquencourt BP 105
78153 Le Chesnay, France
abitebou@inria.inria.fr

Victor Vianu
CSE C-0114
UC San Diego
La Jolla, CA 92093, USA
vianu@cs.ucsd.edu

Abstract

There is a fundamental mismatch between the hardness of database queries and their Turing complexity. For example, the *even* query on a set has low Turing complexity but is by all accounts a hard query. The mismatch is due to the abstract, *generic* nature of database computation: data items which are indistinguishable by logical properties are treated uniformly. The issues specific to generic computation are obscured in the Turing model. Two models of generic computation are proposed. They are extensions of Turing Machines with a *relational store*. The machines differ in the interaction between the Turing component and the relational store. The first, simpler machine, allows limited communication with the relational store. However, it is not complete. Nonetheless, it subsumes many query languages which have emerged as central, including the *fixpoint* and *while* queries. We prove a normal form for the machine, which provides a key technical tool. The normal form specialized to the *while* queries allows resolving the open problem of the relation of *fixpoint* and *while*: they are equivalent iff PTIME = PSPACE. The second machine allows extended communication between the relational store and the Turing component, and is complete. The machine involves parallelism. Based on it, we define complexity measures for generic mappings. We focus on notions of polynomial time and space complexity for generic mappings, and show their robustness. The results point to a trade-off between complexity and computing with an abstract interface.

*Work supported in part by an INRIA-NSF cooperation grant, by the French Ministry of Research under grant PRC-BD3 and the National Science Foundation under grants IRI-8816078 and INT-8817874. Work performed in part while the second author was visiting INRIA. o

1 Introduction

Database queries have traditionally been studied in the framework of classical Complexity Theory. However, there is now ample evidence of a fundamental mismatch between the Turing complexity of queries and their practical hardness. We argue in this paper that this mismatch occurs because database computation is *not* adequately captured by Turing Machines (TM's). We propose as an alternative the *Generic Machine* (GM), and define robust complexity classes of queries using the machine. We study in depth GM^{loose}, a restricted machine which naturally generalizes many important query languages. The results on GM^{loose} provide insight into computational properties of languages such as *fixpoint* and *while*. For instance, they are used to solve the open problem of the relationship of *fixpoint* and *while*.

Databases provide a logical, abstract view of data, purposedly independent of the internal representation. Queries use only information provided by the abstract interface to the database. This leads to the notion of "generic" computation, which treats uniformly data with identical logical properties, and therefore is insensitive to automorphisms of the input. To illustrate this, consider a database consisting of a binary relation specifying the edges of a directed graph. Consider a query which returns as answer a set of vertices in the graph. The desired set of vertices is specified using solely the graph and not its internal representation. For instance, the query might extract all vertices with positive in-degree, or all vertices belonging to some cycle. The vertices in the answer are *all* those satisfying the specified property. Hence, vertices which cannot be distinguished using the graph are always treated uniformly. More formally, generic computation is insensitive to automorphisms of the input.

Generic computation does not assume an ordered domain; this leads to a mismatch between the complexity of generic computation and standard Turing complexity. A typical example is the *even* query on a set[1]. Clearly, *even* has low Turing complexity. However, it is hard to compute generically because all elements of the set must be treated uniformly throughout the computation. This rules out the straightforward solution of repeatedly extracting one arbitrary element from the set until the set is empty, while keeping a binary counter. Indeed, there is no way to generically extract an arbitrary element from the set. Of course, the problem disappears if the database provides an ordering of the domain, since then every element in the set can be distinguished from every other. In the absence of the order, to compute *even* generically one has to construct *all* orderings of elements in the set, then check the parity by traversing the orderings. This requires exponential space, and indeed cannot be achieved in the usual query languages. In a Turing Machine the issue becomes moot, since the input is presented in a sequential fashion to begin with. Thus, TM's cannot capture the generic nature of database computation.

We introduce two models of generic computation, GM and GM^{loose}. Both are TM's 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, and others hold the output at the end of the computation. In a transition, the relational store can be modified through a first-order (FO) query. The use of first-order operations as primitive steps is justified by the fact that they are in (uniform) AC_0 [I87b], and thus, in a reasonable sense, take constant parallel time. Moreover, FO has a very simple algebraïzation involving basic operations on relations (see Preliminaries). This also 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.

GM and GM^{loose} differ in the interaction between the tape and the relational store. In GM^{loose}, the connection is loose: the TM can only ask *yes/no* FO queries on the relational store, and can request FO transformations of the store. Although the TM component provides full computational power, we show that GM^{loose} is *not* computationally complete, and that for instance, it cannot compute the *even* query. On the other hand, GM is complete. It 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.

Based on GM, we define complexity classes proper to generic computation, focusing on generic analogs of PTIME (GEN-PTIME) and PSPACE (GEN-PSPACE). One of the main results is the robustness of these classes. Indeed, we show that the natural polynomial time and space restrictions of two previously defined (computationally complete) query languages from [CH80, AV90] coincide respectively with GEN-PTIME and GEN-PSPACE. We also show that, in agreement with our intuition, *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.

GM^{loose} is of special interest because it subsumes many query languages which have emerged as central. In particular, GM^{loose} generalizes the *fixpoint* and *while* queries, which augment FO with recursion, and stay respectively within PTIME and PSPACE. The connection with *while* is emphasized by a result showing that GM^{loose} is equivalent to *while* extended with integer variables and arithmetic. One may wonder whether GM^{loose} can express all GEN-PTIME queries. The answer is negative; thus, the parallelism provided by GM is necessary even if one is interested solely in GEN-PTIME queries.

We study GM^{loose} in depth. We prove a normal form which provides a key technical tool. In the normal form, GM^{loose} computations are carried out in two phases: the first involves a *fixpoint* computation on the relational store alone, and the second concerns exclusively the tape, ignoring relations. The *fixpoint* query analyzes the input, and computes certain equivalence classes of tuples. The second phase can be viewed as an abstract computation on an *ordered* database whose constants represent equivalence classes obtained in the first phase. As a consequence, GM^{loose} computations (and also computations in restricted languages such as *while*) are determined by the equivalence classes, and are basically insensitive to the size of these classes. By applying the normal form to the *while* queries, we show that *fixpoint* = *while* iff PTIME = PSPACE. We thereby answer an open question of [C88]. Surprisingly, the result also applies to the PTIME fragment of *while* ($while|_{PTIME}$): we prove that PTIME = PSPACE iff *fixpoint* = $while|_{PTIME}$. Note that this reduces the separation of PTIME and PSPACE to the separation of two classes of queries *within* PTIME.

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

The problem of finding query languages expressing exactly low complexity classes of queries, e.g., the PTIME queries, is central and largely unresolved. Unfortunately, the problem persists with the new notion of generic complexity. Although we now exclude generically hard queries such as *even* from low complexity classes, we still have difficulties in characterizing such classes. We show that the problem comes from defining complexity measures w.r.t. the *size* of the input, since determining that size in a generic manner is itself hard. We propose an alternative measure based on the number of equivalence classes with respect to automorphisms, called *type index*[2]. The type index captures the amount of generic information contained in the input. This new measure is motivated by the fact that GM^{loose} computations depend on the type index of the input rather than its size. This can be best understood by looking at the extremes. If the constants of the input are ordered, the type index and the size of the input roughly coincide; on such inputs GM^{loose} is computationally complete, *fixpoint* expresses PTIME [I86, V82], and *while* yields PSPACE [V82]. At the other extreme, if the input is an unordered set, the type index is a constant independent of the input size; on such inputs, GM^{loose} computations take constant time and collapse to FO, as do *fixpoint* and *while*. We show that, with the type index as a measure of the input, *fixpoint* and *while* are precisely the polynomial time and space restrictions of GM^{loose}. This matches the intuition that, in some natural but restricted sense, *fixpoint* and *while* represent the polynomial time and space generic computations.

There is a large body of work on finite relational structures (see the surveys [G88, F90]). The study of computable queries originated in the work of Chandra and Harel [CH80, C81, CH82]. Since then, the complexity and expressiveness of query languages, and the relationship with logic, have been widely investigated, e.g. [V82, CH85, GS85, I86, I87a, I87b, C88, G88, KP88, AV88, AV89]. Below NP, expressiveness results usually assume an ordered input. Without order, languages typically express queries complete within a class, but are unable to express *simple* queries. It remains open whether there is a language expressing exactly PTIME .

2 Preliminaries

We review informally some terminology and notation of relational databases and query languages. A general presentation of the database field is given in [U88], and of database theory in [K90].

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 over an *output* schema. The mapping must be computable and *generic*: for each automorphism ν of an input I, $\phi(\nu(I)) = \nu(\phi(I))$. A query language or computing device is called *complete* if it expresses all queries. Complexity classes of queries are defined based on Turing complexity in the natural fashion. For example, the PTIME class consists of all generic mappings 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 $\phi(I)$, in time polynomial in $|enc(I)|$.

We briefly present the languages FO, *fixpoint*, *while*, $while^{unsorted}$, and $while^{invent}$ that are used in the paper. A database is said to be *ordered* if one particular binary relation *succ* provides a successor relation on the constants occurring in the database.

Most traditional query languages are based on first-order logic without function symbols (FO). The FO formulas over predicate symbols $\{p_1, \ldots, p_n\}$ are built from atomic formulas $p_i(x_1, \ldots, x_m)$ (p_i of arity m) and $x = y$ 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 [U88]). It involves the following simple operations on relations: π (projection on some co-ordinates), $\times$ (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). For instance, the FO query:

$$\{x, y \mid \phi\} \quad \text{where} \quad \phi = q(x,y) \vee \exists z(p(x,z) \wedge p(z,y))$$

can be algebraïcally computed by:

$$q \cup \pi_{1,4}(\sigma_{2=3}(p \times p)).$$

The simplicity of these operations justify our use of FO queries as primitive steps in computations. It is also important to note that FO is in (uniform) AC_0 [I87b], so FO queries can be evaluated in constant parallel time with polynomial resources.

There are many useful queries that FO cannot express, such as the transitive closure of a graph. Numerous extensions of FO with recursion have been proposed. Most of them converge towards two central classes of queries: *fixpoint* and *while* [CH80]. Both languages allow inductive definitions of relations up to a fixpoint, by iterating a FO formula. For the *fixpoint* queries, results of iterations are cumulated, so convergence is guaranteed in polynomial time. For *while*, each iteration is destructive, so *while* $\subseteq$ PSPACE. In particular, *while* queries may not converge. The *fixpoint*

[2] Each equivalence class can be identified with a *type* in the logic sense.

queries (*fixpoint*) [CH82] are constructed using the *FO* constructors together with a fixpoint operator (μ). The fixpoint operator binds a predicate symbol p 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. For instance, the transitive closure of relation q is given by the *fixpoint* query: $\mu p \phi(x, y, p)$ for ϕ as above. *Fixpoint* expresses exactly PTIME on ordered databases [I86, V82]. However, it cannot express the query *even*.

The *while* language also provides an inductive definition of relations up to a fixpoint. It was originally introduced in [CH82] in a procedural form. $\mathcal{A}$ is extended with (i) sorted relational variables ($X, Y, \ldots$), (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[3]. For instance, consider a graph g. The following program removes from g all edges (a, b) if there is a path of length 2 from a to b, and inserts an edge (a, b) if there is a vertex not directly connected to a and b; this is iterated while some change occurs:

$$\begin{array}{l}
X := \emptyset; \\
\textit{while } X \neq g \textit{ do} \\
\qquad \textit{begin} \\
\qquad X_1 := \{x, y \mid \exists z (g(x, z) \wedge g(z, y))\}; \\
\qquad X_2 := \{x, y \mid \exists z (\neg g(x, z) \wedge \neg g(z, x) \\
\qquad\qquad\qquad \wedge \neg g(y, z) \wedge \neg g(z, y))\}; \\
\qquad X := g; \quad g := (g - X_1) \cup X_2; \\
\qquad \textit{end.}
\end{array}$$

(In the example, we freely mix relational algebra and calculus using the fact that *FO* and $\mathcal{A}$ are equivalent. Note also that variables in small letters denote constants whereas variables in capital letters denote whole relations.) It is clear that *while* programs may not terminate. The *while* language is equivalent to an extension of *FO* with a *partial* fixpoint operator (*FO+PFP*) [AV89]. The partial fixpoint of $\phi(p)$ is computed by iterating ϕ destructively, so convergence is no longer guaranteed, and the fixpoint operator is only partially defined. For instance, the *while* program above is equivalent to the partial fixpoint $\mu g \psi(x, y, g)$ where:

$$\begin{aligned}
\psi \; &= (g(x, y) \wedge \neg \exists z\, (g(x, z) \wedge g(z, y))) \\
&\vee \exists z\, (\neg g(x, z) \wedge \neg g(z, x) \wedge \neg g(y, z) \wedge \neg g(z, y))\,.
\end{aligned}$$

It is easy to see that *while* programs stay within PSPACE. In fact, *while* expresses exactly PSPACE on ordered databases [V82]. Like *fixpoint*, *while* does not express the query *even*.

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

The languages above are not complete. Indeed, any language using (i) a finite set of fixed-arity relations and (ii) only constants from the input, stays within PSPACE. To break the space barrier, (i) or (ii) must be relaxed. The first complete language, $\textit{while}^{unsorted}$, was introduced in [CH80], and is based on relaxing (i). It is obtained by using an unsorted algebra, and allowing in *while*, unsorted variables, i.e. variables standing for relations whose arity is not fixed. The possibility to count using the arity of intermediary results yields the desired computational power. Another complete language, $\textit{while}^{invent}$, is based on relaxing (ii), and was presented in [AV90]. The expressive power is obtained by having a constructor (*new*) allowing the introduction of new constants in the database. For instance, the following program creates a new constant for each pair of tuples in p using two variables X (4-ary) and Y (5-ary):

$$X := p \times p; \quad Y := new(X).$$

3 The Loosely-Coupled Generic Machine

3.1 Description of the Machine

We introduce here our first model of generic computation, called *loosely coupled Generic Machine*, and denoted GM^{loose}.

GM^{loose} consists of a TM interacting with a database (*relational store*) through an abstract interface. (See Figure 1.) The relational store holds relations over a finite set of fixed-arity predicates, some of which are designated as input or output predicates. A configuration of GM^{loose} consists of a configuration of the TM component, together with the current content of the relational store. Transitions are based on the following factors: (i) the current state, (ii) the content of the current tape cell, and (iii) the answer to a *yes/no FO* query on the relational store. Transitions involve the following actions: (i) move right or left on the tape, (ii) overwrite current tape cell with some tape symbol, (iii) change state, and (iv) assign to some predicate the result of a *FO* query on the store. Note that the description of the transitions is finite. In particular, it involves a finite number of *FO* formulas.

The machine starts in the start state, with the input in the designated predicates of the relational store, and an empty tape. It halts when a halting state is reached. The output is then found in the designated output predicates of the store. Thus, each machine defines a generic mapping from input to output relations. When there is no confusion, we also denote by

Figure 1: A GM^{loose} configuration

GM^{loose} the set of mappings computed by GM^{loose} machines. GM$^{loose}\downarrow$ denotes the total mappings computed by GM^{loose}. Note that GM^{loose} can also be viewed as an acceptor, either by having some of the halting states be accepting, or by taking as output a 0-ary predicate.

It is easy to see that *while* programs can be viewed as GM^{loose} machines where the tape is never used. Conversely, GM^{loose} is strictly more powerful than *while*. For instance, GM^{loose} can compute queries such as "does the diameter of graph G satisfy property P ?", where P is an arbitrary recursive property of the integers. It turns out that the full GM^{loose} is equivalent to *while* augmented with integer arithmetic. Let $while^{+}$ be the *while* language augmented with integer variables $i, j, \ldots$, $increment(i)$ and $decrement(i)$ statements, and use of $i = 0$ tests in looping conditions. From the equivalence between TM's and counter machines, it easily follows that GM^{loose} expresses the same queries as $while^{+}$. Interestingly, the equivalence still holds for $while^{+}$ extended with assignments $p := \{i\}$, where p is a unary predicate and i an integer variable (as long as integers are not allowed in the final result). Thus, mixing integers with the data throughout the computation does not yield more power.

Remark (due to Moshe Vardi): It turns out that *while* has a 0/1 law, i.e. the asymptotic probability of each property it expresses always exists and is 0 or 1. This result still holds for a powerful (non-recursive) extension of *while*, called infinitary logic[4] with finitely many variables (FO^{∞}) [KV90]. It can be shown that FO^{∞} also subsumes GM$^{loose}\downarrow$. Thus, each property expressible by GM$^{loose}\downarrow$ also has a 0/1 law.

GM^{loose} captures an important class of generic computations. We shall study it in detail.

3.2 Expressiveness and Normal Form

Since GM^{loose} has a TM component, it can perform arbitrarily complex computations on the tape. However, the following shows that GM^{loose} is not complete w.r.t. generic computations, and in particular, that it cannot compute *even*[5].

Theorem 3.1 (i) $GM^{loose}\downarrow$ is equivalent to FO on unary input predicates. (ii) GM^{loose} is complete on ordered inputs.

The above exhibits a rather puzzling range of expressive power. Why such different behaviors on ordered inputs vs. unordered sets ? This has to do with the generic treatment of the data by GM^{loose}. Due to genericity, GM^{loose} computations are essentially determined by the equivalence classes of tuples w.r.t. automorphisms of the input. Therefore, GM^{loose} computations on sets are insensitive to the size of the sets, so they take constant time and collapse to *FO*. At the other extreme, for ordered inputs, the information on equivalence classes determines completely the input. Basically, GM^{loose} can perform any computation involving the equivalence classes it can distinguish.

Let us now elaborate on the equivalence classes determining GM^{loose} computations. For simplicity, suppose that the relational store consists of a single predicate p (if not, a simple FO encoding can be performed) of arity k. Let I be a finite structure. The automorphisms of I induce an equivalence relation $\equiv_I$ over k-tuples of constants from I, as follows: $u \equiv_I f(u)$ for each k-tuple u and each automorphism f (extended to tuples). The equivalence relation $\equiv_I$ can be alternatively defined by: $u \equiv_I v$ iff for *every* FO formula ϕ using predicate p, with k free variables, $u \in \phi(I)$ iff $v \in \phi(I)$. (Thus, an equivalence class is identified by its *type*, i.e. the set of FO formulas satisfied by the tuples in the class.) Unfortunately, the ability of a GM^{loose} M to distinguish equivalence classes is limited by the fact that the data is "polled" using just the finite set of FO formulas in its finite-state control (denoted $FO(\mathrm{M})$). Thus, each machine can only compute a coarser equivalence relation. In this spirit, let $\equiv_{I,M}$ be the equivalence on

[4] Infinitary logic allows disjunctions and conjunctions of infinite sets of formulas.

[5] Note that this also follows from the above remark on 0/1 laws, since *even* has no asymptotic probability.

k-tuples defined as follows: $u \equiv_{I,M} v$ iff for every sequence $\phi_1(p), \phi_2(p), ..., \phi_n(p)$ of formulas in $FO(\mathrm{M})$,

$$u \in \phi_n(...(\phi_1(I)...) \text{ iff } v \in \phi_n(...(\phi_1(I)...).$$

Clearly, each equivalence class of $\equiv_{I,M}$ is a union of equivalence classes of $\equiv_I$.

Now consider a given GM^{loose} M on some input I. If two tuples u, v are equivalent w.r.t. $\equiv_{I,M}$, they both belong to the result, or none does; the machine cannot distinguish between tuples that are $\equiv_{I,M}$-equivalent. This suggests that the equivalence classes of $\equiv_{I,M}$ can be treated in the computation as abstract entities. Indeed, the store transitions can be described in terms of the equivalence classes. Surprisingly, it turns out that the entire information determining the computation of GM^{loose} on a given input can be computed by a *fixpoint* query. Indeed, for a given M, there exists a *fixpoint* query which on input I computes the following:

- the equivalence classes of $\equiv_{I,M}$, in some total order[6]. These are obtained by recursively splitting the space of k-tuples. Initially, the split is in two classes: I and its complement. This is refined by applying repeatedly all FO formulas in $FO(\mathrm{M})$, until no refinement occurs. The ordered equivalence classes are represented by a $2k$-ary relation storing a forest of balanced trees of k-tuples. The tuples at each level in the forest constitute an equivalence class.

- a description of the action of the FO transformations occurring in the machine on the equivalence classes of $\equiv_{I,M}$.

At the end of the fixpoint computation, each class of $\equiv_{I,M}$ can be identified with an integer (the level of the class in the forest). Thus, the entire information about the input can be encoded on the tape using integers representing equivalence classes. Then the remainder of the computation can be carried out solely on the tape, except for a last part which places the final result in the output predicate of the store.

This yields the normal form. GM^{loose} can be decomposed in three parts. The first is a *fixpoint* query which provides the information described above. This is followed by a computation on the tape, on integers representing the equivalence classes obtained by the *fixpoint* formula. Lastly, the final result is decoded from the tape into the output relations.

The normal form provides a key technical tool. In the next section, we use the normal form to understand the relationship between *fixpoint* and *while*.

[6]It is unlikely that a *fixpoint* query could define exactly $\equiv_I$, since this involves checking graph isomorphism, and *fixpoint* is in PTIME.

3.3 While versus Fixpoint

Since *fixpoint* and *while* express PTIME and PSPACE, respectively, on ordered inputs[7], *fixpoint* $\subset$ *while* if PTIME $\subset$ PSPACE. However, the converse has been an open question [C88]. In this section, we show that the separation of *fixpoint* and *while* implies PTIME $\subset$ PSPACE.

The proof uses the normal form for GM^{loose}. This specializes to *while* as follows. Consider a *while* query q defined by iterating an FO formula ϕ. The query q can be written as fw, where f is a *fixpoint* query, and w a *while* query. As for GM^{loose}, f computes on an input I the equivalence classes of $\equiv_{I,q}$, in some total order, and a description of the action of ϕ on the equivalence classes. The query w works on the ordered equivalence classes provided by f. Its computation is isomorphic to that of a *while* query on a totally ordered input whose constants represent the equivalence classes of $\equiv_{I,q}$. Using the normal form, we have:

Theorem 3.2 *Fixpoint* = *while* iff PTIME = PSPACE.

Proof sketch: The "only if" part is known. Consider the "if" part. Suppose PTIME = PSPACE. Consider a *while* query q in the normal form fw as above. The computation of w is isomorphic to that of a *while* query w' on the totally ordered input where each equivalence class of $\equiv_{I,q}$ is replaced by a distinct constant. Now w' is in PSPACE, therefore in PTIME (w.r.t. the number of ordered constants representing equivalence classes). Since *fixpoint* = PTIME on ordered inputs, there is a *fixpoint* query f' equivalent to w' on the ordered input. A *fixpoint* query f'' equivalent to w is then obtained from f' by replacing the abstract ordered constants by tuples from the corresponding equivalence classes, and equality by equivalence. Thus, q is equivalent to ff'', so is a *fixpoint* query. □

The above shows that *fixpoint* and *while* are most likely distinct. There is an intuitive feeling that *fixpoint* and *while* capture, in some natural sense, the polynomial time and space generic computations, respectively. We shall see in the next section that *fixpoint* and *while* are *not* the PTIME and PSPACE restrictions of GM^{loose}. Is there *some* generic computing device such that *fixpoint* and *while* are its PTIME and PSPACE restrictions ? If so, it would have to be that the PTIME fragment of *while* is *fixpoint*. It turns out that this is highly unlikely. Indeed, it is equivalent, once more, to PTIME = PSPACE. This is surprising, and contrary to the "running conjecture" in the field. For example, Kolaitis conjectured that the PTIME fragment of FO^{∞} is *fixpoint*, which would imply that $while|_{PTIME}$ = *fixpoint* [Ko90].

[7]Here and below, $\subset$ denotes proper inclusion.

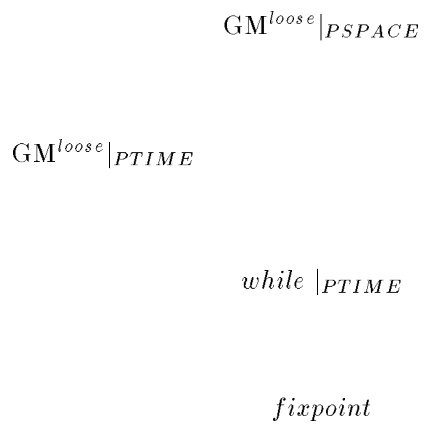

Figure 2: PTIME and PSPACE restrictions of GM^{loose}

Theorem 3.3 *fixpoint* $=$ *while*$|_{PTIME}$ iff PTIME $=$ PSPACE.

Proof sketch: The "if" part follows from Theorem 3.2. The "only if" part is based on a padding argument. Suppose *fixpoint* $=$ *while*$|_{PTIME}$. Let L be a language in PSPACE. A word of length k can be encoded in a relation using k ordered constants. Since *while* is PSPACE on ordered inputs, there is a *while* query ϕ accepting precisely encodings of words w in L, in space $|w|^d$ (and time $2^{|w|^d}$) for some d. Consider now an encoding of words of length k using complete binary trees I_{k^d} of depth k^d; a word is encoded using the order provided by the first k levels in the tree. One can check that the input has this form in time polynomial in k. Now ϕ still takes 2^{k^d} time, which however is only polynomial in $|I_{k^d}|$. Thus, ϕ is in *while*$|_{PTIME}$. Hence, ϕ is in *fixpoint*. Next, *fixpoint* queries on I_{k^d} converge in a number of stages polynomial in k. However, each stage may not be polynomial in k since it involves a FO query on the entire tree. Using a technique of Lindell [L90], it can be shown that nonetheless the *fixpoint* query can be computed in PTIME with respect to k, using a concise representation of I_{k^d} in terms of its equivalence classes w.r.t. $\equiv_{I_{k^d}}$. Thus, $L \in$ PTIME, and PSPACE $=$ PTIME. $\square$

Note that the above reduces the separation of PTIME and PSPACE to the separation of two classes of queries *within* PTIME.

3.4 Polynomial Time and Space Restrictions of GM^{loose}

We consider here polynomial time and space restrictions of GM^{loose}. Let $GM^{loose}|_{PTIME}$ denote the generic mappings computable by a GM^{loose} in a number of steps polynomial in the input size. $GM^{loose}|_{PSPACE}$ denotes the mappings computable by a GM^{loose} using an amount of tape polynomial in the input size. Note that $GM^{loose}|_{PTIME} \subset$ PTIME and $GM^{loose}|_{PSPACE} \subset$ PSPACE. It is easily seen that $GM^{loose}|_{PTIME}$ and $GM^{loose}|_{PSPACE}$ coincide with PTIME and PSPACE, respectively, on ordered inputs. Thus PTIME $\subset$ PSPACE implies $GM^{loose}|_{PTIME} \subset GM^{loose}|_{PSPACE}$. The converse is open. A natural question concerns the connection of $GM^{loose}|_{PTIME}$ and $GM^{loose}|_{PSPACE}$ with *fixpoint* and *while*. Using a technique similar to the proof of Theorem 3.3, and the fact that PSPACE $\neq$ EXPTIME and PSPACE $\neq$ EXPSPACE, we can show:

Theorem 3.4 (i) *while*$|_{PTIME} \subset GM^{loose}|_{PTIME}$, and (ii) *while* $\subset GM^{loose}|_{PSPACE}$.

Note that *fixpoint* $\subset GM^{loose}|_{PTIME}$ is a consequence of (i). Not surprisingly, *while* and $GM^{loose}|_{PTIME}$ are incomparable. These results are summarized in Figure 2. In the figure, a line indicates proper inclusion, whereas a dashed line indicates proper inclusion if PTIME $\neq$ PSPACE.

3.5 Complexity Measures Revisited

As customary, our complexity measures were based on functions of the input size. However, we have seen that size is not the determining factor in GM^{loose} computations. Instead, these computations are sensitive to the number of equivalence classes w.r.t. automorphisms of the input. This leads to the notion of "type index". Let I be a k-ary relation. The *type index* of I, denoted $\#(I)$, is the number of equivalence classes in $\equiv_I$, i.e., the number of distinct types in I. Since the type index is not accessible by a GM^{loose} using FO formulas of bounded length, we use a coarser measure, depending on the machine. Let M be a GM^{loose}. The *M-type index* of I, denoted $\#_M(I)$, is the number of equivalence classes of $\equiv_{I,M}$. The type index is an appealing notion because it captures the characteristics of the input which determine GM^{loose} computations, and factors out the irrelevant ones.

Now consider time and space restrictions of GM^{loose} based on $\#_M(I)$. A GM^{loose} machine M can distinguish at most $\#_M(I)$ non-equivalent tuples. Therefore, our measure of the input is not absolute, but relative to each machine. This is natural, since different machines view the input with different degrees of accuracy, depending on the formulas in FO(M). Let GM$^{loose}|_{PTIME(\#)}$ (resp., GM$^{loose}|_{PSPACE(\#)}$) be the mappings computable by some GM^{loose} M which on input I stops in a number of steps (resp. using an amount of tape) polynomial in $\#_M(I)$. (Note that FO queries are counted as single steps.) The following confirms the intuition that, in some natural but restricted sense, *fixpoint* and *while* capture the polynomial time and space generic computations.

Theorem 3.5 GM$^{loose}|_{PTIME(\#)}$ = *fixpoint*, and GM$^{loose}|_{PSPACE(\#)}$ = *while*.

Thus, measures based on type index yield complexity classes which can be captured by languages, whereas this remains an elusive goal for measures based on the input size.

4 The Generic Machine

The generic machine GM^{loose} discussed so far is not complete, since it cannot compute *even*. In this section, we consider a complete generic machine, denoted GM, and use it to define complexity measures for generic mappings.

Recall that in GM^{loose}, communication with the relational store was limited to FO queries on the store. GM allows extended communication, by transferring tuples between the relational store and the tape. However, this raises the following basic problem: in a deterministic device, the content of a relation cannot be loaded on the tape by writing the tuples on the tape in some sequential order. Instead, loading a relation with n tuples results in spawning n copies of the machine, with one tuple appended to the tape of each copy. Each individual copy is called a *unit-GM*.

A *Generic Machine* consists of several unit-GM's computing synchronously in parallel. Each unit-GM has its own finite state control, tape and relational store. Initially, GM consists of a single unit-GM in the start state, empty tape, and input held in the relational store. Unit-GM's proliferate as a result of load operations (and also imports, see below). If several unit-GM's reach simultaneously the same state and tape content, they collapse automatically to a single unit-GM whose relational store is the union of their relational stores. No explicit *merge* instruction is necessary. At the end of the computation there must be a single machine in a halting state with an empty tape. This ensures genericity of the mapping computed on the store.

One difference with GM^{loose} concerns the tape alphabet. Since tuples can be loaded on the tape, the tape cells can hold constants from the input. Thus, the tape alphabet is infinite. However, the finite state control can refer explicitly to only a fixed, finite set of constants. To deal with the infinite alphabet, we allow testing equality of the content of the current tape cell with the one to its right. We note that Hull and Su have also discussed handling an infinite tape alphabet with a finite control, motivated by database computation. They defined an extension of TM's with infinite alphabet, called *Generic Turing Machine* (GTM) [HS89]. However, the input of a GTM is presented on the tape in some order which may affect the result of the computation. Thus, GTM's do not typically define generic mappings, and the results have no bearing on ours.

In each unit-GM, transitions depend on the same factors as in GM^{loose}, and the test described above. In a transition, three actions in addition to those of GM^{loose} can be taken: (i) *store* the current k-tuple[8] in a relation, (ii) *load* a relation p on the tape, and (iii) *import*.

The *store* instruction is clear. The *load*(p) instruction executed by some unit-GM M works as follows. Suppose p contains n k-tuples. The machine M is replaced by n new unit-GM's whose configurations are the same as that of M, except that in each unit-GM, one of the n tuples is copied in the k cells to the right of the head. *Import* allows the unit-GM's working in parallel to communicate, while generating additional parallelism. Suppose that at step i, there are n unit-GM's. An *import* instruction executed by machine k results in replacing machine k by n new unit-GM's whose state and relational store are those of k, and whose tapes contain the tape of k to which is appended the (non-blank portion of the) tape of one of the unit-GM's at step i. (In particular, if the n unit-GM's are all performing an *import*, the result consists of n^2 unit-GM's.) We now have:

Theorem 4.1 GM is complete.

Proof sketch: Clearly, GM computes only generic mappings. Let ϕ be a generic mapping. By Theorem 3.1, there is a GM^{loose} machine M which computes ϕ

[8] The current k-tuple consists of the k constants in the k cells immediately to the right of the head.

if an order is provided. On an input with n constants, GM can perform repeated *load* operations resulting in $n!$ unit-GM's, each holding an ordering of the n constants. Then M can be simulated by each unit-GM. By genericity, the result is independent of the ordering and all unit-GM's compute identical results. □

The completeness of GM is essentially due to parallelism which allows the temporary suspension of the generic character of the computation. Indeed, in each unit-GM, the computation is generic only relative to the tuples previously loaded on its tape. The genericity is recovered at the global level via merging of the unit-GM's.

Remark: (i) The following simplified version of GM remains complete: there are no FO transitions of the relational store, and no *yes/no* queries are used in the finite control. Thus, the only operations involving the store are *load*, *store*, and unioning of stores in automatic collapses of unit-GM's. All results remain valid for the simplified machine. (ii) GM without the *import* operation remains complete. However, computations without *import* are more expensive. There is a trade-off between communication and complexity. We chose to include *import* because this models more accurately the computation in existing complete query languages. Lastly, the simplified GM of (i) is no longer complete without *import*.

We next use polynomial time and space restrictions of GM to define complexity classes of generic mappings. A mapping is in GEN-PTIME iff it is computed by some GM M such that on each input I: (i) the number of transitions of M is polynomial in the size of I, and (ii) the number of unit-GM's in each configuration of M in the computation on I is polynomial in the size of I. A generic mapping is in GEN-PSPACE iff it is computed by some GM M such that on each input I: (i) the amount of tape used by each unit-GM is polynomial in the size of I, and (ii) as above.

The robustness of GEN-PTIME and GEN-PSPACE is particularly important. We consider two existing query languages which are complete: $while^{unsorted}$ of [CH80] and $while^{invent}$ of [AV90] (see Preliminaries). We define their natural PTIME and PSPACE restrictions based on the number of steps and the size of intermediate results. We are able to show:

Theorem 4.2 (i) GEN-PTIME $= while^{unsorted}|_{PTIME} = while^{invent}|_{PTIME}$;
(ii) GEN-PSPACE $= while^{unsorted}|_{PSPACE} = while^{invent}|_{PSPACE}$.

We are also interested in the connection of these classes with GM^{loose}. Could it be that the parallelism of GM is not needed if one is solely interested in GEN-PTIME ? The answer is negative: GEN-PTIME $\not\subseteq \mathrm{GM}^{loose}$.

The following shows that, as expected, *even* is a hard query w.r.t. generic complexity.

Theorem 4.3 *even* $\notin$ GEN-PSPACE.

Proof sketch : This is a consequence of the following stronger result:

> (*) every GEN-PSPACE mapping on unary predicate input is in FO.

By Theorem 3.1, it suffices to show that each GEN-PSPACE GM on unary predicate input can be simulated by a GM^{loose}. Indeed, consider such a GM M, and a computation of M on input I. By induction on the length of the computation, one can show that, for each configuration in the computation:

(†) all the unit-GM's are in the same state and their tapes and relational stores are isomorphic; and
(‡) the set of unit-GM configurations is closed under permutations of constants in I.

We wish to simulate M with a GM^{loose}. To do so, we must represent configurations of M using a single tape and a single relational store. To see how this can be achieved, consider the store of one of the unit-GM's. Let $a_1, ..., a_m$ be the sequence of distinct constants occurring on the tape, in the order of occurrence. The relational store is invariant under permutations of I fixing $a_1, ..., a_m$. Thus, each unit-GM is completely identified by its sequence $a_1, ..., a_m$. Due to (‡), and the fact that the number of unit-GM's in the configuration is bounded by a polynomial in $|I|$, m is bounded by some integer b. This allows representing the GM configuration as a GM^{loose} configuration. The isomorphic tapes of the unit-GM's are represented on the GM^{loose} tape using b fixed constants $c_1, ..., c_b$. The relational store of the GM^{loose} uses the relations in the store of M, extended with b columns. A relation p in the store of the unit-GM identified by $a_1, ..., a_m$, $m \leq b$, is represented in the GM^{loose} store as $p \times \{[a_1, ..., a_m, ..., a_m]\}$. The computation of M can be carried out in the GM^{loose} using this representation, and a last step produces the final result by projecting out the b extra columns. □

As a consequence of Theorem 4.3, GEN-PTIME $\subset$ PTIME and GEN-PSPACE $\subset$ PSPACE.

5 Conclusion

The machines described here model computations where a structure is accessed through an abstract interface. They were used to define generic complexity classes. As seen, the generic complexity of a query is at least as high as its Turing complexity. We argued that the generic complexity is a more realistic complexity measure for languages using solely the abstract interface to the database. Indeed, the possibly lower Turing complexity is generally not realizable by uniform compilation of queries expressed in such a language. The results point to a trade-off between complexity and computation with an abstract interface.

Our models are realistic for languages where each individual statement is at most FO. One can add to any purely generic language, any finite number of **oracle** queries with high generic complexity but low Turing complexity, such as *even*, or various forms of *counting* [I87b, I89]. An important, open problem is whether there is a choice of such oracles which yields a language expressing exactly PTIME. We believe that our investigation provides useful new tools for approaching this question.

An unresolved problem concerns the existence of query languages expressing exactly GEN-PTIME and GEN-PSPACE. For the case of GM^{loose}, we proposed a complexity measure that allowed to characterize *fixpoint* and *while* in complexity terms. This suggests that languages expressing GEN-PTIME and GEN-PSPACE might be obtained by extending *fixpoint* and *while* with constructs allowing **parallelism**.

As illustrated in recent work such as [AK89], object-based languages allowing the creation of **objects** that belong to the output raise new issues of expressive power. In [DV91], an object-oriented model of database parallel computation using **methods** is investigated. It is shown that the PTIME and PSPACE complexity classes defined using that model essentially coincide with GEN-PTIME and GEN-PSPACE (with minor differences due to the presence of invented object identifiers in the result). Thus, the model in [DV91] provides a natural and practically motivated alternative to GM. The coincidence of complexity classes based on the two models speaks once more to the naturalness and robustness of GEN-PTIME and GEN-PSPACE.

Finally, the *load* operation provides a deterministic solution to communication between the GM tape and the relational store, based on parallelism. An alternative is to load the tuples of a relation on the store, in some arbitrarily chosen order. Let NGM be the device allowing such form of **non-determinism**. Acceptance occurs if at least one of the possible computations accepts. Let GEN-NP be the set of mappings computed by NGM in a number of transitions polynomial in the input size. It can be shown that GEN-NP = NP. Clearly, *even* $\in$ GEN-NP. Since *even* $\notin$ GEN-PTIME, it follows that **GEN-P $\neq$ GEN-NP**.

Acknowledgement. We wish to thank Christos Papadimitriou and Moshe Vardi for their comments on a draft of the paper. Thanks also to Dino Karabeg for useful discussions related to this material.

References

[AK89] S. Abiteboul and P. Kanellakis, Object Identity as a Query Language Primitive. In Proc. ACM SIGMOD Int'l. Conf. on Management of Data (1989), full version to appear in JACM.

[AV88] Abiteboul, S., V. Vianu, Datalog Extensions for Database Updates and Queries, I.N.R.I.A. Technical Report No.715 (1988). To appear in JCSS.

[AV89] Abiteboul, S., V. Vianu, Fixpoint Extensions of First-Order Logic and Datalog-like Languages, Proc. Fourth Annual Symposium on Logic in Computer Science, Asilomar, California (1989) pp. 71-79.

[AV90] Abiteboul, S., V. Vianu, Procedural Languages for Database Queries and Updates, JCSS, 41,2 (1990) pp. 181-229.

[BR86] Bancilhon R., R. Ramakrishnan, an Amateur's Introduction to Recursive Query Processing, Proc. ACM SIGMOD Int'l. Conf. on Management of Data (1986) pp. 16-52.

[C81] Chandra, A.K., Programming Primitives for Database Languages, Proc. ACM Symposium on Principles of Programming Languages, Williamsburg (1981), pp. 50-62.

[C88] Chandra, A.K., Theory of Database Queries, Proc. 7th ACM Symp. on Principles of Database Systems (1988), pp. 1-9.

[CH80] Chandra, A.K., D. Harel, Computable Queries for Relational Databases, Journal of Computer and System Sciences 21:2 (1980), 156-178.

[CH82] Chandra, A.K., D. Harel, Structure and Complexity of Relational Queries, Journal of Computer and System Sciences 25:1 (1982), 99-128.

[CH85] Chandra, A.K., D. Harel, Horn Clause Queries and Generalizations, J. Logic Programming 2,1 (1985), pp. 1-15.

[DV91] Denninghof, K., V. Vianu, The Power of Methods With Parallel Semantics, UCSD Technical Report CS91-184 (preliminary version, 1991).

[F90] Fagin R., Finite-Model Theory—a Personal Perspective, Proc. Int'l. Conf. on Database Theory (1990).

[G88] Gurevich, Y., Logic and the Challenge of Computer Science, Trends in Theoretical Computer Science, (E.Borger), Computer Science Press (1988). pp.1-57.

[GS85] Gurevich Y., S. Shelah, Fixed-Point Extensions of First-Order Logic, 26th IEEE FOCS (1985), pp. 346-353.

[HS89] Hull R, J. Su, Untyped Sets, Invention and Computable Queries, Proc. ACM Symp. on Principles of Database Systems (1989), pp. 347-359.

[I86] Immerman N., Relational Queries Computable in Polynomial Time, Information and Control 68 (1986), pp. 86-104.

[I87a] Immerman N., Languages which Capture Complexity Classes, SIAM J. Comp., 16,4 (1987) pp. 760-778.

[I87b] Immerman N., Expressibility as a Complexity Measure: Results and Directions, Yale Univ. Res. Rep. DCS-TR-538, (1987).

[I89] Cai J.-W., M. Fürer, N. Immerman, An Optimal Lower Bound on the Number of Variables for Graph Identification, Proc. IEEE FOCS (1989), pp. 612-617.

[Ko90] Kolaitis, P., Personal communication, 1990.

[K90] Kanellakis, P.C., Elements of Relational Database Theory, to appear as a chapter in Handbook of Theoretical Computer Science, North-Holland.

[KP88] Kolaitis, P., C. Papadimitriou, Why not negation by fixpoint? Proc. ACM SIGACT-SIGART-SIGMOD Symp. on Principles of Database Systems, (1988), pp. 231-239.

[KV90] Kolaitis, P., M.Y. Vardi, 0/1 Laws for Infinitary Logic, IEEE Symp. on Logic in Computer Science (1990), pp. 156-167.

[L90] S. Lindell, an Analysis of Fixed-Point Queries on Binary Trees, PhD thesis, UCLA. To appear in *Theoretical Computer Science.*

[U88] Ullman, J.D., Principles of Database and Knowledge Base Systems, Computer Science Press (1988).

[V82] Vardi, M.Y., The Complexity of Relational Query Languages, Proc. 14th ACM STOC (1982), pp. 137-146.