Computing With First-Order Logic*

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 92093-0114, USA
vianu@cs.ucsd.edu

Abstract

We study two important extensions of first-order logic (FO) with iteration, the *fixpoint* and *while* queries. The main result of the paper concerns the open problem of the relationship between *fixpoint* and *while*: they are the same iff PTIME = PSPACE. These and other expressibility results are obtained using a powerful normal form for *while* which shows that each *while* computation over an unordered domain can be reduced to a *while* computation over an *ordered* domain via a *fixpoint* query. The *fixpoint* query computes an equivalence relation on tuples which is a congruence with respect to the rest of the computation. The same technique is used to show that equivalence of tuples and structures with respect to FO formulas with bounded number of variables is definable in *fixpoint*.

Generalizing *fixpoint* and *while*, we consider more powerful languages which model arbitrary computation interacting with a database using a finite set of FO queries. Such computation is modeled by a relational machine called *Loosely Coupled Generic Machine*, GM^{loose}. GM^{loose} consists of a Turing Machine augmented with a finite set of fixed-arity relations forming a *relational store*. The connection with *while* is emphasized by a result showing that GM^{loose} is equivalent to *while* augmented with integer variables and arithmetic. The normal form for *while* extends to these languages. We study the expressive power of GM^{loose} and its PTIME and PSPACE restrictions. We argue that complexity measures based on the size of the input may not be best suited for database computation. Instead, we suggest an alternative measure based on the discerning power of the machine, i.e. its ability to distinguish between tuples given its

*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. A preliminary version of this paper was presented at the 1991 *ACM Symposium on Theory of Computing*, under the title *Generic Computation and Its Complexity.*

input. With this measure of the input, it is shown that *fixpoint* and *while* are the PTIME and PSPACE restrictions of GM^{loose}.

1 Introduction

Most database query languages are based on extensions of first-order logic (FO). Many of these extensions converge around two central classes of queries: the *fixpoint* queries and the *while* queries. This paper focuses on *fixpoint*, *while*, and other languages which interact with the database by means of a finite set of FO queries. The main result of the paper concerns the open problem of the relationship of *fixpoint* and *while*: they are the same iff PTIME = PSPACE. This explains the difficulty of separating *fixpoint* and *while*. It also provides a new characterization of the relationship of PTIME and PSPACE in purely logical terms, without usual assumptions such as an ordered domain [Imm86, Var82]. The result is obtained using a powerful normal form for *while*, which we also extend to other languages.

The *fixpoint* and *while* queries were first defined by Chandra and Harel [Cha81, CH82]. They have emerged as central, since most other extensions of FO with recursion which have been proposed, using various formalisms, turn out to be equivalent to either *fixpoint* or *while*. Informally, *fixpoint* provides recursive computation which is *monotonic*, and guarantees termination in PTIME. The *while* language can yield non-monotonic computations, and the complexity is PSPACE.

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. Indeed, with such an order, *fixpoint* expresses exactly PTIME [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[1].

The need to consider computations on unordered inputs arises from the fact that databases provide a logical, abstract view of data, purposedly independent of the internal representation. Queries posed against the database use only information provided by the abstract interface, which may not include an ordering of the domain. This leads to the notion of "generic" computation, which treats uniformly data with identical logical properties. 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,

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

vertices which cannot be distinguished using the graph are always treated uniformly. More formally, generic computation is insensitive to automorphisms of the input.

The normal form we prove 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.

By applying the normal form to the *while* queries, we show that *fixpoint* = *while* iff PTIME = PSPACE. We thereby answer an open question of [Cha88]. Surprisingly, the result also applies to the PTIME fragment of *while* (i.e., $while|_{PTIME}$): 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.

The *while* queries are characterized by the fact that they manipulate the input using some finite set of FO queries. More generally, we would like to understand the properties and limitations of this type of generic computation. To this end, we introduce a relational computing device called[2] *Loosely Coupled Generic Machine*, GM^{loose}. This 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. GM^{loose} extends the finite-state control of *while* 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*. 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. Thus, GM^{loose} 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. We note that the use of first-order operations in practical languages is widespread, and is motivated by the fact that FO has a very simple algebraïzation involving basic operations on relations (see Preliminaries), which can be implemented very efficiently. In particular, note that FO queries are in (uniform) AC_0 [Imm87b], and thus, in a

[2]In several follow-up papers, such as [AVV92b, AVV92a], the machine GM^{loose} is simply called *relational machine*.

reasonable sense, take constant parallel time.

GM^{loose} is of special interest because it subsumes most query languages which have emerged as central, including the *fixpoint* and *while* queries. The connection with *while* is emphasized by a result showing that GM^{loose} is equivalent to *while* extended with integer variables and arithmetic. Although GM^{loose} is arbitrarily powerful, it shares with *while* the characteristic that it manipulates the databases using just some finite set of FO queries. This fact is crucial, and allows to extend the normal form for *while* to the full GM^{loose}. It is also a source of limitation in expressive power, which varies widely on ordered vs. unordered inputs. Indeed, GM^{loose} is complete on ordered inputs but, like *while*, collapses to FO on unordered sets.

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 for complexity classes defined based on GM^{loose} and other generic computational devices. We suggest that the problem comes from defining complexity measures with respect to the *size* of the input, since determining that size in a generic manner is itself hard, or impossible, without a counting construct. Indeed, GM^{loose} has limited discerning power, i.e. limited ability to distinguish among tuples based on its input. Consequently, we propose an alternative measure based on the number of equivalence classes of tuples which are undistinguishable by the first-order formulas used by a machine M. This is called the *M-size* of the input. The M-size captures the amount of information that M can extract from the input. This new measure is justified by the fact that the computation of a GM^{loose} M depends on the M-size of the input rather than its size. This can be best understood by looking at the extremes. If the elements of the domain are ordered, the M-size and the size of the input roughly coincide; on such inputs GM^{loose} is computationally complete, *fixpoint* expresses PTIME [Imm86, Var82], and *while* yields PSPACE [Var82]. At the other extreme, if the input is an unordered set, the M-size 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 call complexity classes obtained by bounding the resources of GM^{loose} by functions in the M-size of the input, *relational complexity classes*. We show that *fixpoint* and *while* are precisely relational polynomial time and relational polynomial space, respectively. This matches the intuition that, in some natural but restricted sense, *fixpoint* and *while* represent the polynomial time and space generic computations.

In [AV91b] we argue that there is a mismatch between Turing complexity and the actual hardness of database queries. A typical example is the query *even* on a set, which has low Turing complexity but is not expressible in powerful query languages, and indeed may be viewed as hard when the domain is not ordered. In [AV91b] we define complexity classes proper to generic computation, using a model of generic computation which captures more accurately its true complexity. The GM^{loose} model is not well-suited to serve this purpose. Indeed, although the TM component of GM^{loose} provides full computational power, GM^{loose} is *not* computationally complete, and for instance, it cannot compute the *even* query. In [AV91b] we consider a second model of generic computation, the *Generic Machine (GM)*, which is complete. GM and GM^{loose} differ in the interaction between the tape and the relational store. Recall that 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. On the other hand, GM allows loading the content of relations on the tape, and storing tuples back into relations. Based on GM, we defined in [AV91b] robust complexity classes proper to generic computation. GM is only briefly discussed in Section 6, and will be presented in detail in a companion paper.

There is a large body of work on finite relational structures (see the surveys [Gur88, Fag90]). The study of computable queries originated in the work of Chandra and Harel [CH80, Cha81, CH82]. Since then, the complexity and expressiveness of query languages, and the relationship with logic, have been widely investigated, e.g. [Var82, CH85, GS86, Imm86, Imm87a, Imm87b, Cha88, Gur88, KV90, Lei89, AV91a]. 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.

The relational machine GM^{loose} is similar in spirit to several devices previously proposed, which operate directly on structures rather than on encodings of structures. Such machines have been introduced by H. Friedman [Fri71] and later by D. Leivant [Lei89]. The device introduced in [Fri71], the FAP, is reminiscent of program schemes. The device of [Lei89] is a syntactic variant of the FAP. The emphasis in [Fri71] is on extending recursion theory to FAPs. In [Lei89], ordered structures are emphasized. These results have a different focus than those presented here.

The paper is organized as follows. Section 2 consists of preliminaries on the *fixpoint* and *while* queries, and other languages used in the paper. In Section 3, we discuss the normal form for the *while* queries. Section 4 contains the result that *fixpoint* = *while* iff PTIME = PSPACE, and related results. Section 5 discusses GM^{loose}, the extension of the normal form to GM^{loose}, its equivalence to extensions of *while* with arithmetic, and relational complexity. Section 6 briefly outlines the results on GM obtained in [AV91a]. Finally, Section 7 provides a review of the results and pointers to follow-up papers.

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 [Ull88], and of database theory in [Kan91].

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 over an *output* schema. The mapping must be computable and *generic*:

for each automorphism ν of an input I, $\varphi(\nu(I)) = \nu(\varphi(I))$. 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 $\varphi(I)$, in time polynomial in $|enc(I)|$.

We briefly present the languages *FO*, *fixpoint*, and *while*, 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 elements of the domain. For an instance I, the set of domain elements of I is denoted $elem(I)$.

The First-Order Queries

Most traditional query languages are based on first-order logic without function symbols (*FO*). 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]). 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:

$$\{x, y \mid \varphi\} \quad \text{where} \quad \varphi = Q(x, y) \vee \exists z(R(x, z) \wedge R(z, y))$$

can be algebraïcally computed by: $Q \cup (\pi_{1,4}(\sigma_{2=3}(R \times R)))$. 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 [Imm87b], so *FO* queries can be evaluated in constant parallel time with polynomial resources.

We assume in general some default order on the free variables of a formula and often omit the list of variables in a *FO* query. For instance, we will consider query φ (for φ as above) assuming that x comes first in the default order.

Conjunctive Queries

We will also use an important subset of *FO*. The *conjunctive queries* [CM77] are *FO* queries whose formulas are in prenex form[3] and (a) use only existential quantifiers, and (b) use only the connective $\wedge$ (i.e., do not use $\vee$, $\neg$ or $\rightarrow$). For example,

$$\exists z(R(x, z) \wedge R(z, y)),$$

$$\exists y(P(x, y) \wedge Q(y, t) \wedge x = z),$$

[3]Some definitions of the conjunctive queries do not require prenex normal form.

$$x = x \wedge y = y$$

are conjunctive queries (the last query defines the cross-product of the domain with itself). The query

$$\exists z(R(z, z)) \wedge z = z$$

is not in prenex form and therefore not a conjunctive query, but is equivalent to the conjunctive query

$$\exists y(R(y, y) \wedge z = z).$$

The Fixpoint Queries

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* [Cha81, CH82]. 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. We describe *fixpoint* and *while* next.

The *fixpoint queries* (*fixpoint*) [CH82] are constructed using the first-order constructors as in FO together with a fixpoint operator (μ). The fixpoint operator binds a predicate symbol 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. For instance, the transitive closure of relation Q is given by the *fixpoint* query: $\mu_R \varphi(x, y, R)$ for φ as above.

The same expressive power can be achieved using an *inflationary fixpoint logic* (*FO+IFP*) [GS86]. The inflationary fixpoint operator (*IFP*) no longer requires that R occur positively in $\varphi(R)$, and convergence is instead guaranteed in polynomial time by cumulating iterations of $\varphi(R)$ up to a fixpoint. Inflationary fixpoint formulas are defined next.

Inflationary fixpoint formulas are obtained by repeated applications of first-order operators ($\neg, \vee, \wedge, \exists, \forall$) and the inductive fixpoint operator starting from atoms. The inductive fixpoint operator is defined as follows. Let $\varphi(\mathrm{T})$ be an *FO+IFP* formula with n free variables, where T is an n-ary predicate occurring in φ. Then $IFP(\varphi(T), T)$ denotes the n-ary predicate which is the limit of the sequence defined by: $J_0 = T$ and for each $i > 0$, $J_i = \varphi(J_{i-1}) \cup J_{i-1}$. If $\vec{t}$ is a sequence of n variables or constants, $IFP(\varphi(T), T)(\vec{t})$ is a formula.

The While Queries

The *while* language also provides an inductive definition of relations up to a fixpoint. It was originally introduced in a procedural form, under the names LE in [Cha81], and RQ in [CH82]. $\mathcal{A}$ 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[4]. 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 or b; this is iterated while some change occurs:

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

(In the example, and later in the paper, 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 domain elements whereas variables in capital letters denote whole relations.) It is clear that *while* programs may not terminate.

The *while* language can also be seen as first-order logic extended with a partial fixpoint operator [AV91a]. We briefly consider this aspect next.

Partial fixpoint formulas are obtained by repeated applications of first-order operators $(\neg, \vee, \wedge, \exists, \forall)$ and the partial fixpoint operator starting from atoms. The partial fixpoint operator is defined as follows. Let $\varphi(T)$ be an $FO{+}PFP$ formula with n free variables, where T is an n-ary predicate occurring in φ. Then $PFP(\varphi(T), T)(\vec{t})$ is a formula, where $\vec{t}$ is a sequence of n variables or constants. The interpretation of $PFP(\varphi(T), T)$ is the following: $PFP(\varphi(T), T)$ denotes the n-ary predicate which is the limit, **if it exists**, of the sequence defined by: $J_0 = T$ and for each $i > 0$, $J_i = \varphi(J_{i-1})$ (if φ is undefined on J_{i-1}, then J_i and the interpretation of $PFP(\varphi(T), T)$ are both undefined).

The partial fixpoint of $\varphi(R)$ is computed by iterating φ destructively. For instance, the *while* program above is equivalent to the partial fixpoint $PFP(\psi(G), G)$ 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 shown in [AV91a] that *partial fixpoint logic* and *while* have the same expressive power. It is important to note that, unlike traditional fixpoint extensions of first-order logic ($FO{+}FP$ and $FO{+}IFP$), sentences in $FO{+}PFP$ do not generally have interpretations for all structures (instances). Hence, the transformations defined by $FO{+}PFP$ formulas are partial mappings.

The semantics used here is closely related in spirit to program semantics. From a more logical viewpoint, the fact that truth values need not be defined may be unusual.

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

One may prefer to define the interpretation of $PFP(\varphi(T), T)$ as empty when the limit does not exist. Note that convergence can be tested in time exponential in the number of elements of the input structure. The difference between the two approaches is not essential.

Expressiveness of Fixpoint and While

The expressive power of *fixpoint* and *while* is well understood with the assumption that an order is available. In this case, *fixpoint* expresses PTIME [Imm86, Var82], and *while* expresses PSPACE [Var82]. However, in the absence of order neither can express the query *even* on a set [CH82].

Due to the expressiveness with order, *fixpoint* and *while* are distinct if PTIME $\neq$ PSPACE. However, the converse has been an open question [Cha88]. We answer it affirmatively in this paper.

Complete Languages

The languages above are not complete. Indeed, any language using (i) a finite set of fixed-arity relations and (ii) only elements from the input, stays within PSPACE. We next consider briefly complete languages. To break the space barrier, (i) or (ii) must be relaxed. The first complete language, $while^{unsorted}$, 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, $while^{invent}$, 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 domain elements in the database. For instance, for a binary relation R, the following program creates a new domain element for each pair of tuples in R using two variables X (4-ary) and Y (5-ary):

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

3 A Normal Form for the While Queries

The *while* queries display a puzzling range of expressive power. On ordered sets, *while* yields exactly the PSPACE queries (the maximum that could be expected), whereas on unordered sets, it collapses to FO and thus yields a very small subset of PTIME. Why such different behaviors on ordered inputs vs. unordered sets? This has to do with the generic treatment of data by *while* programs. Due to genericity, *while* computations are essentially determined by the equivalence classes of tuples with respect to automorphisms of the input. Therefore, *while* computations on sets are insensitive to the size of the sets. In particular, *while* programs on sets terminate after a constant number of iterations, so they collapse to *FO*. At the other extreme, for ordered inputs, the information on equivalence classes determines completely the input. Basically, *while* programs can perform any PSPACE computation involving the equivalence classes they can distinguish.

In this section we provide a normal form for *while* which helps understand its expressive power on ordered and unordered inputs. It shows that each *while* computation over an unordered input can be reduced to a *while* computation over an ordered input via a *fixpoint* query.

The normal form provides a key technical tool. Indeed, in the next section, we use the normal form to show that the separation of *fixpoint* and *while* is equivalent to the separation of PTIME and PSPACE (thus solving an open problem posed in [Cha88]). We will later generalize the normal form to $while^+$, a powerful extension of the *while* queries. As a side-effect, the techniques developed in conjunction with the normal form allow to prove some additional useful results. For instance, we show that the property of tuples or structures of being distinguishable by some first-order formula with a bounded number of variables (say k), is definable by a *fixpoint* query that only depends on k.

We next discuss in detail the normal form for *while* programs. We first describe the normal form intuitively. The normal form states that each *while* query w can be written as[5] $f \circ w'$ where f is a *fixpoint* query and w' is another *while* query. The query f 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 query f outputs these equivalence classes in some order, so that each class can then be thought of abstractly as an integer. In the second phase, the query w' can be viewed as computing on an *ordered* database obtained by replacing each equivalence class produced by f by its corresponding integer.

The key to the normal form is the ability to compute equivalence classes of tuples by a *fixpoint* query and order them. The finest equivalence relation that we may want to obtain is that induced by the automorphisms of the input. It is defined as follows. Let I be a finite structure over the input predicates and $\Delta_k(I)$ be the set of k-tuples that can be constructed with elements from I.

Definition 3.1 For each u, v in $\Delta_k(I)$, $u \equiv_I^k v$ iff there exists an automorphism f of I (extended to k-tuples), such that $v = f(u)$.

This equivalence relation would serve our purpose if it were computable by *fixpoint*. However, this is unlikely, since computing $\equiv_I^k$ involves checking graph isomorphism, and *fixpoint* is in PTIME. To understand what we could use in place of $\equiv_I^k$, let us now elaborate on the equivalence classes determining *while* program computations.

[5] We use $\circ$ to denote composition. In the present paper, $f \circ g(x)$ is $g(f(x))$, i.e., f is applied first.

Consider a *while* program w. Assume for the moment that w uses only predicates of some arity k, $k > 0$, and no constants. We are searching for an easily computable equivalence relation of tuples, whose equivalence classes are never split in the course of the computation. We are naturally lead to the following. We say that a program w *distinguishes* among tuples u and v in $\Delta_k(I)$ in the course of its computation on I if, at some point in the computation, u is in some predicate of the program and v is not, or conversely. Let $\equiv^k_{I,w}$ be the equivalence relation thus defined on $\Delta_k(I)$. Observe that due to genericity, $\equiv^k_I$ refines $\equiv^k_{I,w}$, i.e. if $u \equiv^k_I v$ then $u \equiv^k_{I,w} v$.

The equivalence relation $\equiv^k_{I,w}$ would also be a good candidate for use in the normal form since its equivalence classes are treated as indivisible blocks in the computation of w. Unfortunately, $\equiv^k_{I,w}$ cannot in general be computed except by going through the entire computation of the *while* program on I. Instead, we will consider a third equivalence relation which lies between $\equiv^k_I$ and $\equiv^k_{I,w}$. We will show that it is computable by *fixpoint*. This third equivalence relation will thus have the desired property since it also refines $\equiv^k_{I,w}$.

Note that the computation of a *while* program w can be viewed as performing compositions of the FO formulas in w, subject to the control flow imposed by the program[6]. The equivalence relation which we introduce next, and which we use for the normal form, is obtained by ignoring the runtime restrictions arising from the control. Thus, two tuples are equivalent if they cannot be distinguished by *any* composition of FO formulas occurring in w. Let $FO(w)$ be the set of FO formulas occurring in w. We will show how to compute the equivalence relation characterizing the ability to distinguish among tuples using compositions of formulas in $FO(w)$, denoted by $\equiv^k_{I,FO(w)}$. The ability to distinguish tuples using compositions of FO formulas from some given finite set is of interest not only in the case of the *while* language, but also for other languages whose computations are based on some finite set of first-order formulas. Therefore, we will consider in more detail this aspect in the conclusion of the section. We next develop the tools that we need for the normal form.

Given a finite set F of FO formulas, F^* denotes the closure of F under composition of formulas. Let us be more precise. To simplify, we assume that all predicates have the same arity k and each formula in F has k free variables. We also use the following notation. For each formula $\{y_1, ..., y_k \mid \psi\}$ and vector $x_1, ..., x_k$, $\psi(x_1, ..., x_k)$ is obtained by substituting each x_i for y_i in ψ. (It is assumed that bound variables are distinct from $x_1, ..., x_k$; otherwise some appropriate renaming is performed.) Now consider a formula $\varphi(R_{i_1}, ..., R_{i_m})$ with m occurrences of predicates; and let $\varphi_1, ..., \varphi_m$ be formulas with k free variables. Then $\varphi(\varphi_1, ..., \varphi_m)$ is the formula obtained by replacing the j-th occurrence of a predicate, $R_{i_j}(\vec{x})$ by $\varphi_j(\vec{x})$.

Using this notation, we now have:

Definition 3.2 Let F be a set of FO formulas with k free variables, over schema S with k-ary predicates. The set F^* is the smallest set of formulas such that:

[6]In this section, we use the procedural *while* programs described in the preliminaries.

1. for each predicate R occurring in S, and distinct variables $x_1, ..., x_k$, $R(x_1, ..., x_k)$ is in F^*.

2. for each $\varphi(R_{i_1}, ..., R_{i_m})$ in F and $\varphi_1, ..., \varphi_m$ in F^*, $\varphi(\varphi_1, ..., \varphi_m)$ is in F^*.

Observe that, for a *while* program w, the set $(FO(w))^*$ of formulas can be viewed as a rough approximation of the relation computations of w. In particular, the content of some relation is always the result of some FO query in $(FO(w))^*$. On the other hand, it may be the case that the result of some query in $(FO(w))^*$ will never be constructed during the *while* computation. This is due to the brute force construction of $(FO(w))^*$ which ignores the control of w and also does not distinguish between the various predicates. For instance, suppose that $k = 2$ and $FO(w)$ contains the formulas

$$\neg R_1(x, y), \quad \text{and} \quad R_2(x, y) \wedge R_2(y, x).$$

Then, $FO(w)$ contains (among other formulas):

$$R_1(x, y), R_2(x, y), \neg R_1(x, y), \neg R_2(x, y),$$

$$R_1(x, y) \wedge R_1(x, y),$$

$$\neg R_2(x, y) \wedge \neg R_2(x, y),$$

etc.

The following equivalence relation will play a crucial role:

Definition 3.3 Let k be an integer and S an input schema with predicates of arity k. Let F be a finite set of first-order formulas with k free variables, over S. For each finite structure I over S, let $\equiv^k_{I,F}$ be the equivalence relation over $\Delta_k(I)$ defined by: $u \equiv^k_{I,F} v$ iff for every formula τ in F^*, $u \in \tau(I)$ iff $v \in \tau(I)$.

The crux of the normal form and the other results in the section is that we can compute in *fixpoint* a refinement of $\equiv^k_{I,F}$ (so of $\equiv^k_{I,w}$) and, furthermore, order its equivalence classes.

Normal Form for Simple While Programs

To simplify the presentation, we first consider a restricted class of *while* programs, which we call "simple".

Definition 3.4 A *while* program w is *simple* if there exists some k, $k > 1$, such that w uses only predicates of arity k and FO formulas which are conjunctive queries and negations[7] with k free variables.

[7] Negations are formulas of the form $\neg P(x_1, ..., x_k)$, where the x_i are all distinct.

a	b	b	a
a	c	b	a
a	b	c	a
a	c	c	a
b	a	b	c
c	a	b	c
b	a	c	b
c	a	c	b
b	c	a	a
c	b	a	a
a	a	b	b
a	a	c	c

Figure 1: Representation of an Ordered Partition

Thus we consider first programs involving predicates of some fixed arity and formulas consisting of conjunctive queries and negations. Let G be the set of formulas occurring in such a program. We will show that we can compute with *fixpoint* a refinement of $\equiv^k_{I,G}$, and output the corresponding equivalence classes in some order. This will provide the main technical tool relating to the normal form.

We first make more precise the above statement.

Definition 3.5 An *ordered partition for* $\Delta_k(I)$ is a sequence $\langle \delta_1, ..., \delta_n \rangle$ of distinct subsets of $\Delta_k(I)$ such that $\{\delta_1, ..., \delta_n\}$ is a partition of $\Delta_k(I)$.

We will represent an ordered partition $\langle \delta_1, ..., \delta_n \rangle$ using a $2k$-ary predicate $R^k_\prec$ holding the relation $(\delta_1 \times \delta_2) \cup ... \cup (\delta_{n-1} \times \delta_n)$, i.e.,

$$R^k_\prec = \{u \times v \mid \exists i, u \in \delta_i, v \in \delta_{i+1}\}.$$

For example, let I be the instance $\{\langle a, b\rangle, \langle a, c\rangle\}$. Let χ be the ordered partition $\langle \delta_1, \delta_2, \delta_3, \delta_4, \delta_5 \rangle$ of $\Delta_k(I)$, where:

$$\begin{aligned}
\delta_1 &= \{\langle a, b\rangle, \langle a, c\rangle\}, \\
\delta_2 &= \{\langle b, a\rangle, \langle c, a\rangle\}, \\
\delta_3 &= \{\langle b, c\rangle, \langle c, b\rangle\}, \\
\delta_4 &= \{\langle a, a\rangle\}, \\
\delta_5 &= \{\langle b, b\rangle, \langle c, c\rangle\}.
\end{aligned}$$

(Note that χ happens to consist of the equivalence classes induced by the automorphisms of I.) The ordered partition χ is represented by the relation in Figure 1. For each ordered partition χ, the corresponding relation over $R^k_\prec$ is called the *representation* of χ.

We will say that a program computes an ordered partition $\langle \delta_1, ..., \delta_n \rangle$ if it produces a relation $R^k_\prec$ as above.

Lemma 3.6 Let k be an integer and S an input schema with predicates of arity k. Let G be a finite set of conjunctive queries and negations over S with k free variables. There exists a *fixpoint* query which computes, for each input I, an ordered partition $\langle\delta_1, ..., \delta_n\rangle$ of $\Delta_k(I)$ such that $\{\delta_1, ..., \delta_n\}$ refines $\equiv^k_{I,G}$.

Proof: Let $S = \{R_1, ..., R_s\}$. For each $0 \leq i_1, ..., i_s \leq 1$, let

$$\alpha_{i_1 \ldots i_s} = \{u \mid u \in \Delta_k(I) \text{ and } u \in I(R_{i_j}) \text{ iff } i_j = 1\}.$$

We start from the ordered partition

$$\chi = \langle\alpha_0, \alpha_1, ..., \alpha_{2^s-1}\rangle.$$

This initial ordered partition is refined until the result is obtained. We next describe one refinement step. Let $\langle\delta_1, ..., \delta_n\rangle$ be the current ordered partition. Let $\varphi_1, ..., \varphi_p$ be the conjunctive queries in G. If $\varphi_i(\vec{x}) = \exists\vec{z}(R_{i_1}(\vec{z}_{i_1}) \wedge ... \wedge R_{i_l}(\vec{z}_{i_l}))$ has free variables $\vec{x}$ and $\delta_{i_1}, ..., \delta_{i_l}$ is a sequence of blocks, we use the notation:

$$\varphi_i[\delta_{i_1}, ..., \delta_{i_l}] = \{v(\vec{x}) \mid v \text{ is a valuation of the variables in } \varphi_i, \text{ and for each } j, v(\vec{z}_{i_j}) \in \delta_{i_j}\}.$$

To perform a refinement using φ_i, we consider all sequences $\delta_{i_1}, ..., \delta_{i_l}, \delta_{i_{l+1}}$ of blocks in the current partition, in some order, until we find one such that $\delta_{i_{l+1}}$ is neither included in $\varphi_i[\delta_{i_1}, ..., \delta_{i_l}]$, nor in its complement. If such $\varphi_i, \delta_{i_1}, ..., \delta_{i_{l+1}}$ are found, one can refine the partition by splitting $\delta_{i_{l+1}}$ into two blocks:

(†) $\delta' = \delta_{i_{l+1}} \cap \varphi_i[\delta_{i_1}, ..., \delta_{i_l}]$ and
$\delta'' = \delta_{i_{l+1}} - \varphi_i[\delta_{i_1}, ..., \delta_{i_l}]$.

The new ordered partition is obtained by replacing $\delta_{i_{l+1}}$ by the sequence $\delta'\delta''$.

This is iterated until no refinement can be performed with any of the φ_i. Let $\chi = \langle\delta_1, ..., \delta_n\rangle$ be the ordered partition that is obtained. Note that, since no further refinement is possible using the conjunctive queries in G, χ is a congruence with respect to these queries i.e.,

(‡) for each $\varphi_i(R_{i_1}, ..., R_{i_l})$ and sequence of blocks $\delta_{i_1}, ..., \delta_{i_l}$, $\varphi_i[\delta_{i_1}, ..., \delta_{i_l}]$ is a union of blocks of $\langle\delta_1, ..., \delta_n\rangle$.

To conclude the proof, we show that:

(i) $\{\delta_1, ..., \delta_n\}$ refines $\equiv^k_{I,G}$, and

(ii) $\langle\delta_1, ..., \delta_n\rangle$ can be computed by a *fixpoint* query.

To prove (i) we show that, for each σ in G^* and input I, $\sigma(I)$ is a union of blocks in $\langle\delta_1, ..., \delta_n\rangle$. The proof is by structural induction on σ. For the base case, we consider the values of I for the input predicates. By construction of χ, each such value is a union of blocks.

For the induction step, suppose that $\sigma = \varphi(\varphi_{i_1}, ..., \varphi_{i_l})$, where $\varphi(R_{i_1}, ..., R_{i_l})$ is in G and for each $i_j, 1 \leq j \leq l$, $\varphi_{i_j}(I)$ is a union of blocks of $\langle\delta_1, ..., \delta_n\rangle$. Two cases occur:

1. φ is a conjunctive query. Let

$$\varphi(R_{i_1}, ..., R_{i_l}) = \exists \vec{z}(R_{i_1}(\vec{z}_{i_1}) \wedge ... \wedge R_{i_l}(\vec{z}_{i_l})).$$

   Then

$$\begin{array}{rcl} \sigma(I) & = & \varphi[\varphi_{i_1}(I), ..., \varphi_{i_l}(I)] \\ & = & \bigcup\{\varphi[\delta_{i_1}, ..., \delta_{i_l}] \mid \delta_{i_j} \in \varphi_{i_j}(I), 1 \leq j \leq l\} \end{array}$$

   By (‡), each $\varphi[\delta_{i_1}, ..., \delta_{i_l}]$ is a union of blocks in $\langle\delta_1, .., \delta_n\rangle$. It follows that $\sigma(I)$ is also a union of such blocks.

2. φ is a negation, i.e. $\sigma = \neg\varphi_1$.
   Since $\varphi_1(I)$ is a union of blocks of $\langle\delta_1, ..., \delta_n\rangle$, its complement with respect to $\Delta_k(I)$, i.e. $\sigma(I)$, also has this property.

To show (ii), we demonstrate that the $2k$-ary predicate $R^k_{\prec}$ representing $\langle\delta_1, ..., \delta_n\rangle$ can be computed in *fixpoint*, more precisely *inflationary fixpoint* logic $(FO + IFP)$. The crucial point is to ensure that the computation can be performed in an inflationary manner. There are four aspects to the computation:

1. *initialization:*
   We have to compute a representation of the initial χ above. This can be done in FO, so in fixpoint.

2. *refinement of the partition:*
   During the computation of the ordered partition, we will have to split a block into two new blocks. Suppose that $R^k_{\prec}$ contains the representation of the current ordered partition $\langle\delta_1, ..., \delta_n\rangle$ and that some δ_i must be split into δ_i', δ_i''. Suppose $1 < i < n$ (the cases $i = 1$ and $i = n$ require only minor modifications). Then in the current $R^k_{\prec}$,

$$(\delta_{i-1} \times \delta_i) \cup (\delta_i \times \delta_{i+1})$$

   must be replaced by

$$(\delta_{i-1} \times \delta_i') \cup (\delta_i' \times \delta_i'') \cup (\delta_i'' \times \delta_{i+1}).$$

   This would require removing from $R^k_{\prec}$:

$$(\delta_{i-1} \times \delta_i'') \cup (\delta_i' \times \delta_{i+1}).$$

   However, this cannot be done directly in an inflationary manner. To overcome this difficulty, we will represent $R^k_{\prec}$ using two predicates s_{pos}, s_{neg}. The current ordered partition is represented by $s_{pos} - s_{neg}$. Then removal of tuples from $R^k_{\prec}$ is done by inserting the tuples in s_{neg}, which can be done in an inflationary manner. (A *FO* query is needed at the end to construct $R^k_{\prec}$ from s_{pos} and s_{neg}.) Thus, the split of δ_i is performed by:

$$\begin{array}{lcl} \text{adding} & \delta_{i-1} \times \delta''_i & \text{to} \quad s_{neg} \\ \text{adding} & \delta'_i \times \delta_{i+1} & \text{to} \quad s_{neg} \\ \text{adding} & \delta'_i \times \delta''_i & \text{to} \quad s_{pos}. \end{array}$$

Note that there is no need to remove a tuple from s_{neg} once inserted. Indeed, a tuple $[t, t']$ in s_{neg} indicates that t and t' belonged to blocks which were adjacent at some point, and now belong to blocks which are no longer adjacent. Further splits can only increase the distance between the blocks of t and t', so they will never again belong to adjacent blocks. Therefore, they need never be removed from s_{neg}, whence the inflationary nature of the computation.

3. *detecting the need for another refinement:*
   We outline the construction of a *FO+IFP* query which detects the need for a refinement given the current $R^k_{\prec}$. We show how this is done for a given conjunctive query $\varphi(R_{i_1}, ..., R_{i_l})$. Since the fixpoint queries are closed under composition, we can just compose the fixpoint queries corresponding to each of the conjunctive queries, in some arbitrary fixed order.

   Let $\varphi(R_{i_1}, ..., R_{i_l})$ be a conjunctive query. Given the current $R^k_{\prec}$ (or, equivalently, the current s_{neg} and s_{pos}) representing an ordered partition $\langle \delta_1, ..., \delta_n \rangle$, we construct a *fixpoint* query that finds a sequence of blocks $\langle \delta_{i_1}, ..., \delta_{i_l}, \delta_{i_{l+1}} \rangle$ such that $\delta_{i_{l+1}} \cap \varphi[\delta_{i_1}, ..., \delta_{i_l}] \neq \emptyset$ and $\delta_{i_{l+1}} - \varphi[\delta_{i_1}, ..., \delta_{i_l}] \neq \emptyset$, if such a sequence exists.

   The idea of the construction is to step through all candidate sequences $\langle \delta_{i_1}, ..., \delta_{i_l}, \delta_{i_{l+1}} \rangle$ in lexicographic order with respect to the order on blocks defined by $R^k_{\prec}$. For each such sequence, one can test if $\varphi[\delta_{i_1}, ..., \delta_{i_l}]$ splits $\delta_{i_{l+1}}$ using just a *FO* query. To step through the sequences in lexicographic order, one first constructs a $2k(l+1)$-ary relation $next^{l+1}_k$ which defines a successor relation among the sequences. Thus,

$$\begin{array}{rcl} next^{l+1}_k & = & \{\langle u^1_{i_1}, ..., u^1_{i_{l+1}}, u^2_{i_1}, ..., u^2_{i_{l+1}} \mid u^m_{i_j} \in \delta^m_{i_j}, 1 \leq m \leq 2, 1 \leq j \leq l+1, \text{ and} \\ & & \langle \delta^2_{i_1}, ..., \delta^2_{i_{l+1}} \rangle \text{ is the successor of} \\ & & \langle \delta^1_{i_1}, ..., \delta^1_{i_{l+1}} \rangle \text{ in the current lexicographic order} \\ & & \text{ on blocks induced by } R^k_{\prec}.\} \end{array}$$

   The relation $next^{l+1}_k$ can be constructed straightforwardly by a *fixpoint* query starting from $R^k_{\prec}$.

   Putting (2) and (3) together, we conclude that there exists a *fixpoint* query which, given a current ordered partition, detects the need for a refinement of the partition and performs it.

4. *iterating the refinements*
The refinement of the partition must be iterated an unbounded number of times until there is no need for another refinement. This is easily done in *while*. It is slightly more subtle in $FO+IFP$, since the computation has to be inflationary and thus disallows deletions when iterating the refinements.

To iterate the refinements in an inflationary manner, we use timestamping, a technique first used in [AV91a]. The computation of (2) and (3) is "timestamped" to distinguish successive iterations. More precisely, for each relation R of arity m used in (2) and (3), we use a timestamped version $\widehat{R}$ of arity $m+2k$. A tuple u is represented in the timestamped simulation using $u \times t$ where t is a $2k$-ary tuple used as the current timestamp. For timestamps, we use in the i-th iterations ($i \geq 1$) the following:

- The tuples from the initial ordered partition (χ), if $i = 1$.
- If $i > 1$ we use the tuples newly inserted in s_{pos} in the $(i-1)$-th iteration.

Timestamps are held in a $2k$-ary predicate *Timestamp*. Observe that at each iteration, we have many current timestamps. Another $2k$-ary predicate *OldTimestamp* is used to record outdated timestamps. Tuples with outdated timestamps are ignored. More precisely, during an iteration of (2) and (3), each literal $R(\vec{x})$ used in (2) or (3) is replaced by

$$\exists \vec{z}[\widehat{R}(\vec{x}, \vec{z}) \wedge Timestamp(\vec{z}) \wedge \neg OldTimestamp(\vec{z})].$$

were $\vec{z}$ is a vector of $2k$ new distinct variables. At the end of each iteration, *Timestamp, OldTimestamp* are updated. Observe that the computation is inflationary and can be done in *FO+IFP*. Therefore, the whole refinement process can be expressed in *fixpoint*. □

We proved that the partition $\delta = \{\delta_1, ..., \delta_n\}$ computed from G by the procedure described in Lemma 3.1, is a refinement of $\equiv^k_{I,G}$. In general, $\equiv^k_{I,G}$ is coarser than δ. The following provides a simple condition on G under which δ and $\equiv^k_{I,G}$ coincide. The condition requires that G contain queries which can perform boolean operations on relations (complement and intersection). More precisely, if G contains at least one negation and a conjunctive query of the form $P(x_1, ..., x_k) \wedge Q(x_1, ..., x_k)$, the two partitions coincide.

Corollary 3.7 Let k be an integer and S an input schema with predicates of arity k. Let G be a finite set of conjunctive queries and negations over S with k free variables, such that

(a) G contains at least one negation, and

(b) G contains a conjunctive query of the form[8] $P(x_1, ..., x_k) \wedge Q(x_1, ..., x_k)$.

[8] P and Q need not be distinct.

Then the equivalence relation $\equiv^k_{I,G}$ is computable by a *fixpoint* query.

Proof: Consider the construction of the ordered partition $\delta = \langle\delta_1, ..., \delta_n\rangle$ in the proof of Lemma 3.1. In order for δ and $\equiv^k_{I,G}$ to coincide, each block in δ must be definable by a formula in G^*. Consider such a block δ. There are two possibilities: δ is in the initial partition, or δ was obtained in the course of the refinement process by (†) in the proof of Lemma 3.1. Suppose first that δ is in the initial partition. Let $S = \{R_1, ..., R_s\}$. By construction, there exists a sequence $i_1, ..., i_s$, $0 \leq i_1, ..., i_s \leq 1$, such that δ is of the form

$$\{u \mid u \in \Delta_k(I) \text{ and } u \in I(R_{i_j}) \text{ iff } i_j = 1\}$$

Then δ is defined by the formula:

$$\bigwedge\{R_j(x_1, ..., x_k) \mid i_j = 1\} \wedge \bigwedge\{\neg R_j(x_1, ..., x_k) \mid i_j = 0\}.$$

Clearly, the above formula is in G^*, since G^* contains all literals $R_j(x_1, ..., x_k)$ and G contains a negation and a conjunction.

Next, suppose δ is obtained in the refinement process by (†). Thus,

$$\delta = \delta_{i_{l+1}} \cap \varphi_i[\delta_{i_1}, ..., \delta_{i_l}] \quad \text{or} \quad \delta = \delta_{i_{l+1}} - \varphi_i[\delta_{i_1}, ..., \delta_{i_l}].$$

Suppose $\delta_{i_1}, ..., \delta_{i_{l+1}}$ are definable by formulas $\varphi_{i_1}, ..., \varphi_{i_{l+1}}$ in G^*. In the first case, δ is definable by the formula

$$\varphi_{i_{l+1}}(x_1, ..., x_k) \wedge \varphi_i(\varphi_{i_1}, ..., \varphi_{i_l})(x_1, ..., x_k)$$

which is clearly in G^*, since G contains the query required by (b). In the second case, δ is definable by the formula

$$\varphi_{i_{l+1}}(x_1, ..., x_k) \wedge \neg\varphi_i(\varphi_{i_1}, ..., \varphi_{i_l})(x_1, ..., x_k),$$

which is in G^* since G contains the queries required by (a),(b). $\Box$

Given a simple *while* program w, and G its set of queries, the result of the *fixpoint* query described in Lemma 3.1 is a $2k$-ary predicate $R^k_{\prec}$ representing an ordered partition of $\Delta_k(I)$ refining $\equiv^k_{I,G}$. Thus, we succeeded in computing economically an equivalence relation whose classes are treated as indivisible blocks throughout the computation of w. Indeed, the normal forms we will prove for *while* and other languages require that, once the blocks are computed, they can be treated as abstract entities (integers). To this end, we also need a description of the action of the formulas in G on the equivalence classes. Since complement does not modify equivalence classes, we are concerned only with the conjunctive queries in G. Let $Conj(G)$ denote the conjunctive queries in G. A description of the effect of the formulas of $Conj(G)$ in terms of the blocks of $\delta = \langle\delta_1, ..., \delta_n\rangle$ is easily produced using a *fixpoint* query. More precisely, let

$$\varphi = \{\vec{u} \mid \exists\vec{z}(R_{i_1}(\vec{z}_{i_1}) \wedge ... \wedge R_{i_l}(\vec{z}_{i_l}))\}$$

be a conjunctive query in $Conj(G)$. We can construct a predicate R_φ of arity $(k \times (l+1))$ that describes the effect of φ on the equivalence classes. Predicate R_φ is a union of cross products of partition blocks, defined as follows: for each $\delta_{i_1}, ..., \delta_{i_l}, \delta_{i_{l+1}}$,

$$\delta_{i_1} \times ... \times \delta_{i_l} \times \delta_{i_{l+1}} \subseteq R_\varphi \text{ iff } \delta_{i_{l+1}} \subseteq \varphi[\delta_{i_1}, ..., \delta_{i_l}].$$

We will refer to the above R_φ as the *action table* of φ on δ.

Lemma 3.8 For each conjunctive query φ in $Conj(G)$, R_φ can be computed by a *fixpoint* query.

Proof: In fact, R_φ can be computed by a FO query. Informally, it suffices to compute:

$$\{\vec{z}'_{i_1}, ..., \vec{z}'_{i_l}, \vec{u}' \mid \exists x_1, ..., x_q(R_{i_1}(\vec{z}_{i_1}) \wedge ... \wedge R_{i_l}(\vec{z}_{i_l}) \wedge \vec{z}_{i_1} \equiv \vec{z}'_{i_1} \wedge ... \vec{z}_{i_l} \equiv \vec{z}'_{i_l} \wedge \vec{u} \equiv \vec{u}')\}$$

where $x_1, ..., x_q$ is the set of variables in φ and $\equiv$ indicates that the two k vectors are in the same partition block. Note that $\equiv$ is first-order definable from $R^k_\prec$. Indeed, $\vec{x} \equiv \vec{y}$ iff

$$\exists \vec{w}(R^k_\prec(\vec{x}, \vec{w}) \wedge R^k_\prec(\vec{y}, \vec{w})) \vee (\neg \exists \vec{w} R^k_\prec(\vec{x}, \vec{w}) \wedge \neg \exists \vec{w} R^k_\prec(\vec{y}, \vec{w})).$$

□

Lemmas 3.1 and 3.3 combined show that, given G, a *fixpoint* query can compute the ordered partition $R^k_\prec$ and predicates $R_{\varphi_1}, ..., R_{\varphi_n}$ describing the effect of the conjunctive queries in G in terms of the blocks of the partition. At the end of the *fixpoint* computation, each block δ_i of the ordered partition $\langle \delta_1, ..., \delta_n \rangle$ can be identified with the integer i. Thus, the entire information about the input and the computation can be represented by relations between the integers corresponding to these blocks. Subsequently, any computation which transforms the input using just formulas in G can be seen abstractly as a computation on an ordered domain. This yields normal forms for *while* and other extensions, which provide a bridge between computation without order and computation with order.

We next describe the normal form for simple *while* programs. Each simple *while* program can be decomposed in two phases. The first is a *fixpoint* query which provides the information described above. This is followed by a *while* computation which can be viewed as a computation on an ordered domain where each equivalence class is represented abstractly as an integer. Before stating the normal form precisely, we formalize the "abstraction" of tuples as integers.

Definition 3.9 Given $R^k_\prec$ representing an ordered partition $\langle \delta_1, ..., \delta_n \rangle$ of $\Delta_k(I)$, let $abs_k(\vec{x})$ denote, for each tuple $\vec{x} \in \Delta_k(I)$, the integer i such that $\vec{x} \in \delta_i$. For each mk-ary predicate R, let $abs_k(R)$ be an m-ary predicate. For each instance J over R, let $abs_k(J)$ be the instance over $abs_k(R)$ defined by

$$abs_k(J) = \{ \langle abs_k(\vec{x}_1), ..., abs_k(\vec{x}_m) \rangle \mid \langle \vec{x}_1, ..., \vec{x}_m \rangle \in J \}.$$

(Each $\vec{x}_i$ above is a k-tuple, so that $\langle \vec{x}_1, ..., \vec{x}_m \rangle$ is an mk-tuple.) A *while* program is *k-abstractable* if the following hold:

- $sch(w)$ consists of predicates of arities mk, $m \geq 1$;
- for each literal $P(\vec{x})$ occurring in a FO formula of w, $\vec{x}$ consists of distinct variables;
- all quantifications are of the form $\forall\vec{x}$ or $\exists\vec{x}$ where $\vec{x}$ is a vector of k distinct variables.

Then, $abs_k(w)$ is defined by applying the abs_k mapping to the predicates of $sch(w)$ and replacing each vector $\vec{x}$ of k distinct variables by one distinct variable uniquely determined by $\vec{x}$.

We can now specify the *while* program w' of the second phase of the normal form. The program w' is a k-abstractable program. We will have that

$$w' = abs_k \circ abs_k(w') \circ abs_k^{-1},$$

and predicate $abs_k(R_{\prec}^k)$ will provide a total order for the elements used in $abs_k(w')$. Thus, the second phase can be viewed as a computation on an ordered database.

We are now ready to state the normal form for simple programs.

Lemma 3.10 Let w be a simple *while* program over schema S. Then, $w = f \circ w'$ for some *fixpoint* query f and some k-abstractable *while* query w' such that:

(i) f is over output schema $S \cup \{R_{\prec}^k\} \cup \{R_\varphi \mid \varphi \in Conj(FO(w))\}$, and on each input I, $f(I)(S) = I(S)$, $f(I)(R_{\prec}^k)$ is an ordered partition of $\Delta_k(I)$, and each relations $f(I)(R_\varphi)$ is the action table of φ on the ordered partition in $f(I)(R_{\prec}^k)$.

(ii) the input schema of w is the output schema of f and on each input I, $f \circ w'(I) = f \circ abs_k \circ abs_k(w') \circ abs_k^{-1}(I)$.

Proof: By Lemmas 3.1 and 3.3, there exists a *fixpoint* query f which computes the $R_{\prec}^k$ and R_φ, for φ in $Conj(FO(w))$. The ordered partition computed in $R_{\prec}^k$ refines $\equiv_{I,FO(w)}^k$ and therefore $\equiv_{I,w}^k$. Let w' be the *while* query obtained from w as follows:

- $sch(w') = sch(w) \cup \{R_{\prec}^k, R_\varphi \mid \varphi \in Conj(FO(w))\}$;
- each statement of the form
$$R := \neg\, P(\vec{x})$$
in w remains unchanged in w';
- each termination condition
$$while\ R \neq \emptyset\ do$$
in w remains unchanged;

- each statement in w of the form

$$R := \varphi(R_1, ..., R_k),$$

where φ is a conjunctive query in $FO(w)$, is replaced by

$$R := \exists\vec{x}_1, ..., \exists\vec{x}_k(R_1(\vec{x}_1) \wedge ... \wedge R_k(\vec{x}_k) \wedge R_\varphi(\vec{x}_1, ..., \vec{x}_k, \vec{x})),$$

where each $\vec{x}_i$ and $\vec{x}$ are k-tuples, and there are no repeated variables.

It is now easily seen by construction that w' is k-abstractable, and (i,ii) hold. $\Box$

We next consider arbitrary *while* programs.

Normal Form for Arbitrary While Programs

We next extend the normal form for simple *while* programs to arbitrary *while* programs. This involves removing the restrictions that (i) only conjunctive queries and negations are used, and (ii) all predicates have the same arity k. To this end, we show that an arbitrary *while* program can be written as $pad_k \circ w_k \circ \pi$, where pad_k is FO, π is a sequence of projections, and w_k is a simple *while* program. Then we can use the normal form already shown for simple *while* programs, to obtain a slightly different normal form for arbitrary *while* programs. We will use two lemmas.

Lemma 3.11 For each *while* program w there exists an equivalent *while* program w' (over the same input and output schema but with additional intermediate predicates) such that all queries occurring in w' are conjunctive queries or negations.

Proof (sketch): Let w be a *while* program and $R := \varphi$ an assignment statement in w. We will replace the statement by an equivalent sequence of assignment statements that use only conjunctive queries and negations.

Without loss of generality, we can assume that φ has no variable occurring free and bound, or bound to different quantifiers. Let $\varphi^\exists$ be the FO formula obtained by rewriting φ using just $\exists, \neg, \wedge$. Consider the syntax tree of $\varphi^\exists$, denoted by $tree(\varphi^\exists)$. We will label each node n of $tree(\varphi^\exists)$ by an assignment statement denoted *label(n)*. For each n, let $R_n = R$ if n is the root; otherwise, let R_n be a new predicate whose arity is the number of free variables in the subtree whose root is n. Next, label the nodes n as follows:

- if n is a positive literal ψ, then *label(n)* is $R_n := \psi$;

- if n is $\exists x$ and its child is m, then *label(n)* is $R_n := \exists x R_m(\vec{y})$, where $\vec{y}$ are the free variables of the subtree whose root is m;

- if n is $\wedge$ and n has children m, l then *label(n)* is $R_n := R_m(\vec{x}) \wedge R_l(\vec{y})$, where $\vec{x}$ and $\vec{y}$ are the free variables of the subtrees whose roots are m and l, respectively;

- if n is $\neg$, and its child is m, then *label(n)* is $R_n := \neg R_m(\vec{x})$, where $\vec{x}$ are the free variables of the subtree whose root is m.

Finally, $R := \varphi$ is replaced by the sequence of assignment statements *label(n_1);...;label(n_t)*, where $n_1, ..., n_t$ is a topological sort of $tree(\varphi^\exists)$. $\Box$

To remove the arity restriction, we will use the following.

Notation: Let S be a schema and k some integer larger than the arity of the predicates in S. For each R in S, R_k is a predicate of arity k. The schema S_k consists of all the R_k for R in S. The *padding* query pad_k maps instances over S to instances over S_k. For each R of arity l, pad_k constructs R_k by "padding" R, i.e., R_k is the result of the query

$$\{x_1, ..., x_k \mid R(x_1, ..., x_l) \wedge x_{l+1} = x_{l+1} ... \wedge x_k = x_k\}.$$

The *projection* query π_l over R_k is the query:

$$\{x_1, ..., x_l \mid \exists x_{l+1}, ..., x_k(R_k(x_1, ..., x_k))\}.$$

Lemma 3.12 Let w be a *while* program over some schema S, using just conjunctive queries and negations, and k the maximum number of variables used in any formula occurring in w. There exists a simple *while* program w_k such that w is equivalent to $pad_k \circ w_k \circ \pi$, where π is a sequence of projections.

Proof (sketch): The program w_k is obtained from w as follows:

- each test $R = \emptyset$ becomes $R_k = \emptyset$;
- $P := \neg R(\vec{x})$ is replaced by $P_k := \neg R_k(\vec{x}, \vec{z})$, where $|\vec{z}| = k - |\vec{x}|$ and the $\vec{z}$ are new distinct variables;
- $P := \exists \vec{u}(R^1(\vec{x}_1) \wedge ... \wedge R^h(\vec{x}_h))$; let $x_{i_1}, ..., x_{i_m}$, $i_1 < i_2 < \ldots < i_m$ be the free variables in the above query, and $M = max\{i_1, ..., i_m\}$. The statement is replaced by $P_k := \exists \vec{u}\vec{z}_1 \ldots \vec{z}_h(R_k^1(\vec{x}_1\vec{z}_1) \wedge \ldots \wedge R_k^h(\vec{x}_h\vec{z}_h)) \wedge x_{M+1} = x_{M+1} \wedge \ldots \wedge x_{M+(k-m)} = x_{M+(k-m)}$ where $|\vec{z}_j| = k - |\vec{x}_j|$, and the $\vec{z}_j$ contain new distinct variables.

Clearly, w_k is a simple *while* program and w is equivalent to $pad_k \circ w_k \circ \pi$ for some sequence π of projections. $\Box$

Putting together Lemmas 3.5 and 3.6 and the normal form for simple *while* programs, we obtain the following normal form for arbitrary *while* programs.

Theorem 3.13 Let w be a *while* program over input schema S. Then, $w = f \circ w' \circ \pi$ for some *fixpoint* query f, some sequence of projections π, some integer k and some k-abstractable *while* query w' such that:

(i) f has output schema $S_k \cup \{R^k_\prec\} \cup \{R_\varphi \mid \varphi \in \Phi\}$ for some set Φ of conjunctive queries, and for each input I, $f(I)(R^k_\prec)$ is an ordered partition of $\Delta_k(I)$, and $f(I)(R_\varphi)$ is the action table of φ on the ordered partition in $f(I)(R^k_\prec)$.

(ii) the input schema of w' is the output schema of f and, for each input I, $w'(f(I)) = abs_k \circ abs_k(w') \circ abs_k^{-1}(f(I))$.

Proof: By Lemmas 3.5 and 3.6, w is equivalent to $pad_k \circ w_k \circ \pi$, where w_k is a simple *while* program and π is a sequence of projections. By the normal form for simple *while* programs (Lemma 3.4), $w_k = f_1 \circ w'$ for some *fixpoint* query f_1 and k-abstractable *while* program w'. Finally, let $f = pad_k \circ f_1$. We have that $w = f \circ w' \circ \pi$, with the various programs satisfying the conditions of the theorem. $\Box$

Remark (Monadic Predicates): Note that the *while* program $abs_k(w')$ in the normal form uses only monadic predicates besides the input ones ($R^k_\prec$ and R_φ) which are polyadic. Thus, each *while* program can be reduced to an essentially monadic one via a *fixpoint* query.

The following long example illustrates the main techniques developed in this section in conjunction with the normal form.

Example. Let φ be the first-order query

$$\forall x_1[G(x_1, x_2) \rightarrow T(x_1)].$$

and w the *while* program:

$T := \emptyset$;
$newT := \varphi$;
$D := newT(x_1) \wedge \neg T(x_1)$;
while $D \neq \emptyset$ do
  begin
  $T := newT$;
  $newT := \varphi$;
  $D := newT(x_1) \wedge \neg T(x_1)$;
  end

The input predicate of w is G, and the output T. If G represents the edges of a graph, w outputs in T the set of nodes in G which are *well-founded*, i.e. are not reachable from a cycle of G. For instance, if G is the relation represented in Figure 2 then the output of w on G is $T = \{c, d, e\}$.

Following the construction in the proof of Lemma 3.5, the program w can be rewritten as a *while* program w' which uses only conjunctive queries and negations. To this end, the assignment statements are rewritten as follows. The statement

$$D := newT(x_1) \wedge \neg T(x_1);$$

a	b
b	a
c	d
e	d

Figure 2: A set of edges G

becomes:

$$\begin{aligned}
&R^1 := T(x_1);\\
&R^2 := newT(x_1);\\
&R^3 := \neg R^1(x_1);\\
&R^4 := R^2(x_1) \wedge R^3(x_1);\\
&D := R^4(x_1);
\end{aligned}$$

Next, note that $\varphi^{\exists}$ is

$$\neg\exists x_1[G(x_1, x_2) \wedge \neg T(x_1)].$$

Then the statement

$$newT := \varphi;$$

is replaced by:

$$\begin{aligned}
&R^5 := T(x_1);\\
&R^6 := \neg R^5(x_1);\\
&R^7 := G(x_1, x_2);\\
&R^8 := R^7(x_1, x_2) \wedge R^6(x_1);\\
&R^9 := \exists x_1 R^8(x_1, x_2);\\
&R^{10} := \neg R^9(x_2);\\
&newT := R^{10}(x_2);
\end{aligned}$$

Next, we wish to write w' as $pad_k \circ w_k \circ \pi$, as in Lemma 3.6. Since the maximum number of variables in any formula in w' is 2, $k = 2$. Since G is binary, pad_2 is the identity on G. The unary relations $T, newT, D, R^1, R^2, R^3, R^4, R^5, R^6, R^9, R^{10}$ are replaced by binary relations $T_2, newT_2, D_2, R_2^1, R_2^3, R_2^4, R_2^5, R_2^6, R_2^9, R_2^{10}$, and the above assignment statements become[9]:

1. $R_2^1 := \exists x_2(T_2(x_1, x_2)) \wedge x_2 = x_2;$
2. $R_2^2 := \exists x_2(newT_2(x_1, x_2)) \wedge x_2 = x_2;$
3. $R_2^3 := \neg R_2^1(x_1, x_2);$
4. $R_2^4 := \exists x_2(R_2^2(x_1, x_2)) \wedge \exists x_2(R_2^3(x_1, x_2)) \wedge x_2 = x_2;$
5. $D_2 := \exists x_2(R_2^4(x_1, x_2)) \wedge x_2 = x_2;$

[9]For better readability, some of the formulas are not put in prenex normal form, required by the definition of conjunctive queries.

6. $R_2^5 := \exists x_2(T_2(x_1, x_2)) \wedge x_2 = x_2$;

7. $R_2^6 := \neg R_2^5(x_1, x_2)$;

8. $R_2^7 := G(x_1, x_2)$;

9. $R_2^8 := R_2^7(x_1, x_2) \wedge \exists x_2(R_2^6(x_1, x_2))$;

10. $R_2^9 := \exists x_1(R_2^8(x_1, x_2)) \wedge x_3 = x_3$;

11. $R_2^{10} := \neg R_2^9(x_1, x_2)$;

12. $newT_2 := \exists x_2(R^{10}(x_1, x_2)) \wedge x_2 = x_2$;

The sequence π of projections consists of a single projection

$$\exists x_2(T(x_1, x_2)).$$

Consider now the simple *while* program w_2 obtained as above. By the normal form for simple *while* programs, w_2 can be evaluated in two phases: the first is a *fixpoint* query which produces on input G an ordered partition χ of $\Delta_2(G)$ together with relations describing the action of the conjunctive queries in w_2 on the blocks of $\Delta_2(G)$. The blocks are computed by iterative refinements, as in the proof of Lemma 3.1, by successively applying the conjunctive queries in w_2 to the blocks in the current partition. First, note that several of the conjunctive queries above have the same structure. Since in the refinement process relation names are irrelevant, we can use the following "reduced" set of conjunctive queries:

$\psi_1 = \exists x_2(R(x_1, x_2)) \wedge x_2 = x_2$	corresponding to (1,2,5,6,12)
$\psi_2 = \exists x_2(R(x_1, x_2)) \wedge \exists x_2(R(x_1, x_2)) \wedge x_2 = x_2$	corresponding to (4)
$\psi_3 = R(x_1, x_2) \wedge \exists x_2(R(x_1, x_2))$	corresponding to (9), and
$\psi_4 = \exists x_1(R(x_1, x_2)) \wedge x_3 = x_3$	corresponding to (10).

Consider again the G in Figure 2. The initial partition of $\Delta_2(G)$ is $\chi = [G, \Delta_2(G) - G]$. If we follow the refinement procedure in Lemma 3.1, for the above conjunctive queries (in the order in which they are written), the partition obtained is that in Figure 4. For example, the first refinement of the initial partition is due to applying ψ_1 to G and results in the partition represented in Figure 3. The conjunctive queries ψ_2 and ψ_3 do not yield any refinement.

The normal form produces the ordered partition $R^2_{\prec}$ and, for each conjunctive query $\psi_i, 1 \leq i \leq 4$, a relation R_{ψ_i} describing the effect of ψ_i on the blocks of the partition. The image of this under abs_2 is the ordered instance represented in Figure 5. The second phase of the normal form is a 2-abstractable *while* program, constructed as in the proof of Lemma 3.4. Its image under abs_2 is (for brevity, we denote $abs_2(R)$ by $\overline{R}$):

$$
\begin{array}{cc}
a & b \\
b & a \\
c & d \\
e & d \\
\hline
a & d \\
a & a \\
b & b \\
b & d \\
a & c \\
a & e \\
b & c \\
b & e \\
c & a \\
c & b \\
e & a \\
e & b \\
c & c \\
c & e \\
e & e \\
e & c \\
\hline
d & d \\
d & a \\
d & b \\
d & c \\
d & e
\end{array}
$$

Figure 3: Ordered partition after first refinement

$$
\begin{array}{cc}
a & b \\
b & a \\
\hline
c & d \\
e & d \\
\hline
a & d \\
a & a \\
b & b \\
b & d \\
a & c \\
a & e \\
b & c \\
b & e \\
\hline
c & a \\
c & b \\
e & a \\
e & b \\
c & c \\
c & e \\
e & e \\
e & c \\
\hline
d & d \\
d & a \\
d & b \\
d & c \\
d & e
\end{array}
$$

Figure 4: Final ordered partition

$\overline{T}_2 := \emptyset;$
$\overline{R}_2^5 := \exists z_1(\overline{T}_2(z_1) \wedge \overline{R}_{\psi_1}(z_1, z_2));$
$\overline{R}_2^6 := \neg \overline{R}_2^5(z_1);$
$\overline{R}_2^7 := \overline{G}_2(z_1);$
$\overline{R}_2^8 := \exists z_1 z_2(\overline{R}_2^7(z_1) \wedge \overline{R}_2^6(z_2) \wedge \overline{R}_{\psi_3}(z_1, z_2, z_3));$
$\overline{R}_2^9 := \exists z_1(\overline{R}_2^8(z_1) \wedge \overline{R}_{\psi_4}(z_1, z_2));$
$\overline{R}_2^{10} := \neg \overline{R}_2^9(z_1);$
$\overline{newT}_2 := \exists z_1(\overline{R}_2^{10}(z_1) \wedge \overline{R}_{\psi_1}(z_1, z_2))$
$\overline{R}_2^1 := \overline{T}_2;$
$\overline{R}_2^2 := \overline{newT}_2;$
$\overline{R}_2^3 := \neg \overline{R}_2^1(z_1)$;
$\overline{R}_2^4 := \exists z_1 z_2(\overline{R}_2^2(z_1) \wedge \overline{R}_2^3(z_2) \wedge \overline{R}_{\psi_2}(z_1, z_2, z_3))$;
$\overline{D}_2 := \overline{R}_2^4$;
while $\overline{D}_2 \neq \emptyset$ do
begin
$\overline{T}_2 := \overline{newT}_2;$
$\overline{R}_2^5 := \exists z_1(\overline{T}_2(z_1) \wedge \overline{R}_{\psi_1}(z_1, z_2));$
$\overline{R}_2^6 := \neg \overline{R}_2^5(z_1);$
$\overline{R}_2^7 := \overline{G}_2(z_1);$
$\overline{R}_2^8 := \exists z_1 z_2(\overline{R}_2^7(z_1) \wedge \overline{R}_2^6(z_2) \wedge \overline{R}_{\psi_3}(z_1, z_2, z_3));$
$\overline{R}_2^9 := \exists z_1(\overline{R}_2^8(z_1) \wedge \overline{R}_{\psi_4}(z_1, z_2));$
$\overline{R}_2^{10} := \neg \overline{R}_2^9(z_1);$
$\overline{newT}_2 := \exists z_1(\overline{R}_2^{10}(z_1) \wedge \overline{R}_{\psi_1}(z_1, z_2))$
$\overline{R}_2^1 := \overline{T}_2;$
$\overline{R}_2^2 := \overline{newT}_2;$
$\overline{R}_2^3 := \neg \overline{R}_2^1(z_1)$;
$\overline{R}_2^4 := \exists z_1 z_2(\overline{R}_2^2(z_1) \wedge \overline{R}_2^3(z_2) \wedge \overline{R}_{\psi_2}(z_1, z_2, z_3))$;
$\overline{D}_2 := \overline{R}_2^4$;
end

Running the above program on the input in Figure 5 yields the value $\{2, 4, 5\}$ for $\overline{T}_2$. The final result is obtained by projecting on the first coordinate the blocks corresponding to these integers. This yields $\{c, d, e\}$, the answer of the original program w. $\Box$

Finally, we note that the normal form provides additional insight into the complexity of *while* queries. For an instance I, consider the equivalence relations $\equiv_I^k$ (equivalence on k-tuples determined by automorphisms of I) and $\equiv_{I,F}^k$ where F is a set of FO^k formulas with k free variables. Let the size of $\equiv_I^k$ (the number of its equivalence classes) be denoted $\#_k(I)$. We refer to this number as the *k-type index* of I. (The name comes from the fact that each equivalence class of $\equiv_I^k$ is characterized

$abs_2(R^2_{\prec})$

1	2
2	3
3	4
4	5

$abs_2(G)$

1
2

$abs_2(R_{\psi_1})$

1	1
1	2
2	2
2	4
3	1
3	3
4	2
4	4
5	5

$abs_2(R_{\psi_2})$

1	1	1
1	1	3
1	3	1
1	3	3
2	2	2
2	2	4
2	4	2
2	4	4
3	3	1
3	3	3
4	2	2
4	2	4
4	4	2
4	4	4
5	5	5

$abs_2(R_{\psi_3})$

1	1	1
1	3	1
2	2	2
2	4	2
3	1	3
3	3	3
4	2	4
4	4	4
5	5	5

$abs_2(R_{\psi_4})$

1	1
1	3
2	5
3	1
3	2
3	3
3	4
3	5
4	1
4	2
4	3
4	4
5	1
5	2
5	3
5	4
5	5

Figure 5: Integer representation of analysis output

by its *type*, i.e. the set of all FO formulas satisfied by *all* tuples in the class.) Next, let $size_{k,F}(I)$ be the size of $\equiv^k_{I,F}$. Recall that $\equiv^k_I$ refines $\equiv^k_{I,F}$, so $size_{k,F}(I) \leq \#_k(I)$.

In the normal form, a *while* query is a *fixpoint* query followed by a *while* query $abs_k(w')$ over an ordered domain of size $size_{k,F}(I)$, for some k. Thus, each *while* computation on input I is decomposed into a PTIME computation in the size of I, followed by a PSPACE computation with respect to $size_{k,F}(I)$ (and therefore with respect to $\#_k(I)$). Note that $\#_k(I)$ can be much smaller than the size of I. For example, if I is a set (unary relation), $\#_k(I)$ is constant (independent of the size of I). If the input is a complete binary tree, $\#_k(I)$ is logarithmic in the size of I. For such inputs, where $\#_k(I)$ (and therefore $size_{k,F}(I)$) is much smaller than the size of I, the abstraction in the normal form can be viewed as a form of optimization. To evaluate the potential for optimization, it is important to have information on the likelihood that $size_{k,F}(I)$ is much smaller than $|I|$. This issue is studied in [ACV92], where it is shown that for each k there exists a constant M such that $size_{k,F}(I) \leq M$ with asymptotic probability 1. This indicates that the potential for optimization using the normal form is high. Practical optimization algorithms based on the normal form are developed and evaluated in [AG92].

The results of [ACV92] also shed additional light on the connection between the type index $\#_k(I)$ and $size_{k,F}(I)$. Indeed, it is known that finite structures are rigid (they have no nontrivial automorphisms) with asymptotic probability 1 [P3́7, Fag77]. It follows that $\#_k(I)$ equals a polynomial in I with asymptotic probability 1. On the other hand, $size_{k,F}(I)$ is bounded by a constant with asymptotic probability 1 [ACV92]. It follows that for "most" instances I, $size_{k,F}(I)$ is much smaller than $\#_k(I)$. It also follows that an order cannot be extracted from rigid structures by any FO^k (or, for that matter, $L^\omega_{\infty\omega}$) formula.

The $size_{k,F}(I)$ of a given input is an important measure, which, intuitively, quantifies the amount of information which can be extracted from I using formulas in F. We will argue later that, in the context of computation based on first-order queries, $size_{k,F}(I)$ is more appropriate as a measure of inputs than their size.

Distinguishing Tuples and Structures by FO

In the analysis phase of the normal form, the *fixpoint* query that we use depends on the formulas of the *while* program. It turns out that a unique *fixpoint* query can be used for the analysis of *all* programs using first-order formulas with a bounded number of variables. This is what we consider next. We are lead to consider the general problem of distinguishing tuples or structures using first-order formulas with a bounded number of variables.

We denote by FO^k the set of first-order formulas where at most k variables $\{x_1, ..., x_k\}$ occur. We first present a lemma, which states that a finite set F of conjunctive queries and negations in FO^k is sufficient to express all FO^k formulas by composition.

Lemma 3.14 Let k be an integer and S an input schema with predicates of arity k. There exists a finite set F of conjunctive queries and negations such that:

- F uses only the predicates in S;
- each formula in F has k free variables $x_1, ..., x_k$;
- each formula in F^* is expressible in FO^k; and
- each FO^k formula over S, with k free variables, is in F^*.

Proof: Let $x_1, ..., x_k$ be distinct variables. Suppose P is in S. Let F consist of the following formulas:

(i) $x_1 = x_1 \wedge ... \wedge x_k = x_k \wedge x_i = x_j$ with $1 \leq i, j \leq k$.

(ii) $P(\vec{x}) \wedge x_1 = x_1 \wedge ... \wedge x_k = x_k$ for each k-vector $\vec{x}$ of (not necessarily distinct) variables among $x_1, ..., x_k$;

(iii) $\neg P(x_1, ..., x_k)$;

(iv) $P(x_1, ..., x_k) \wedge P(x_1, ..., x_k)$;

(v) $\exists x_i(P(x_1, ..., x_k)) \wedge x_i = x_i$;

Note that, strictly speaking, the query in (v) is not syntactically a conjunctive query, but it is a FO^k formula defining a conjunctive query. However, the query requires more than k variables if written as a conjunctive query, because then the variable x_i cannot be re-used in the formula. We next prove the statement of the lemma. We first show that each query in F^* is expressible in FO^k. Note that each query in F is expressible in FO^k. Therefore, it is sufficient to prove the following more general statement:

(†) each formula in $(FO^k)^*$ over S is expressible in FO^k.

The proof is by structural induction on formulas in $(FO^k)^*$ over S. For the base case, all formulas of the form $R(x_1, ..., x_k), R \in S$, are in FO^k. For the induction, let $\psi = \varphi(\varphi_1, ..., \varphi_m)$, where $\varphi(R_{i_1}, ..., R_{i_m}) \in FO^k$ and $\varphi_j \in FO^k$. Clearly, it is sufficient to prove that, for each vector $\vec{x}$ of variables among $x_1, ..., x_k$, not necessarily distinct, $\varphi_j(\vec{x}) \in FO^k$. Recall that the definition of substitution requires in general introducing new variables. We have to show that this can be avoided. To this end, it is sufficient to show that all substitutions which are permutations of $x_1, ..., x_k$ can be obtained without increasing the number of variables. Repeating variables can be easily simulated by equalities among variables. Thus, consider a FO^k formula $\varphi(x_1, ..., x_k) \in FO^k$ and let σ be a permutation of $\{1, ..., k\}$. Then $\varphi(x_{\sigma(1)}, ..., x_{\sigma(k)})$ is equivalent to the formula obtained by syntactically replacing in $\varphi(x_1, ..., x_k)$ each occurrence of a variable x_i by $x_{\sigma(i)}$.

Consider now the converse, i.e. $FO^k \subseteq F^*$. Since formulas in FO^k may have subformulas with fewer than k free variables, we have to deal with the lower arity as well. We show that, for each formula φ over S in FO^k, with m free variables, $m \leq k$, there exists a formula ψ_φ in F^* such that, for each instance I over S,

$$(\star) \quad (\varphi \circ pad_k)(I) = \psi_\varphi(I).$$

Note that, in particular, $(\star)$ proves the lemma. We prove $(\star)$ by induction on the depth of φ. Consider first the base cases:

1. for $x_i = x_j$, we use (i);
2. for $R(\vec{x})$, we use (ii).

For the induction, there are three cases:

1. $\varphi = \neg\varphi_1(\vec{x})$; let $\psi_\varphi = \psi(\psi_{\varphi_1})$ where ψ is formula (iii);
2. $\varphi = \varphi_1(\vec{x}) \wedge \varphi_2(\vec{y})$; let $\psi_\varphi = \psi(\psi_{\varphi_1}, \psi_{\varphi_2})$, where ψ is the formula (iv); and
3. $\varphi = \exists x_i \varphi_1(\vec{x})$; let $\psi_\varphi = \psi(\psi_{\varphi_1})$ where ψ is formula (v).

It is easily seen that $(\star)$ holds for the ψ_φ constructed in the three cases. This completes the induction. $\Box$

We next consider the capability of FO formulas with bounded number of variables to distinguish between tuples or structures. Let S be a schema and k be some integer. Let I be a finite structure over S and u, v be two k-tuples with entries from $elem(I)$. Then u, v are *FO^k-equivalent*, denoted $u \equiv^k_{I,FO^k} v$, iff for each $\varphi \in FO^k$ with k free variables, $u \in \varphi(I)$ iff $v \in \varphi(I)$. We will show that $\equiv^k_{I,FO^k}$ is computable by a *fixpoint* query. We first prove the result under the restrictions in Lemma 3.8, then remove the restrictions.

Proposition 3.15 Let k be an integer and S an input schema with predicates of arity k. Then $\equiv^k_{I,FO^k}$ is computable by a *fixpoint* query.

Proof: Consider the set F of conjunctive queries and negations in Lemma 3.8 such that F^* coincides with the FO^k formulas over S with k free variables. Note that F satisfies the conditions of Corollary 3.2. Hence, there is a *fixpoint* query which computes precisely $\equiv^k_{I,F}$, which by Lemma 3.8 is the same as $\equiv^k_{I,FO^k}$. $\Box$

We next eliminate the restrictions present in the statement of Proposition 3.9. To this end, we extend the notion of FO^k-equivalence to m-tuples, $m \leq k$, as follows. Two m-tuples u, v are FO^k-equivalent, denoted $u \equiv^m_{I,FO^k} v$, iff for each FO^k formula φ with m free variables, $u \in \varphi(I)$ iff $v \in \varphi(I)$.

Theorem 3.16 Let S be an input schema, k a positive integer, and $m \leq k$. There exists a *fixpoint* query computing $\equiv^m_{I,FO^k}$.

Proof: We reduce the problem to the restricted case of Proposition 3.9. For each m-tuple $u = \langle a_1, ..., a_m \rangle$, let $ext_k(u)$ be the k-tuple $\langle a_1, ..., a_m, a_m, ..., a_m \rangle$, obtained by padding u with $(k-m)$ a_m's to the right. Let S_k be obtained by extending each predicate in S to arity k. For each I over S, let ext_k be a FO^k query defining an instance over S_k extending the relations in S by the mapping ext_k applied to each tuple. By Proposition 3.9, there is a *fixpoint* query f defining FO^k-equivalence of k-tuples with respect to instances over S_k. Then it is easily shown that $ext_k \circ f \circ \pi_m$ defines FO^k-equivalence of m-tuples, with respect to instances over S. Indeed, this follows from the fact that u and v are FO^k-equivalent with respect to I iff $ext_k(u)$ and $ext_k(v)$ are FO^k-equivalent with respect to $ext_k(I)$. □

Remark. We note that the above results can easily be generalized to a powerful extension of FO^k, called *infinitary* logic with k variables, denoted $L^k_{\infty\omega}$. The logic allows infinite conjunctions and disjunctions, as long as just k variables are used. It has been pointed out by P. Kolaitis and M. Vardi [KV90] that each formula in that logic can be expressed using a (possibly infinite) disjunction of (possibly infinite) conjunctions of formulas in FO^k. Thus, two tuples are equivalent with respect to $L^k_{\infty\omega}$ iff they are equivalent with respect to FO^k. From Theorem 3.10 it then follows that equivalence of m-tuples ($m \leq k$) with respect to $L^k_{\infty\omega}$ is definable in *fixpoint*. We will discuss again infinitary logic later.

We finally consider the problem of equivalence of finite *structures* with respect to first-order formulas with a bounded number of variables. This is similar to the problem of FO^k-equivalence of tuples. Let I, J be two structures over S. Then I, J are *FO^k-equivalent* iff for each FO^k sentence φ, I satisfies φ iff J satisfies φ.

Theorem 3.17 Let k be a positive integer and S, S' input schemas such that S' consists of the same predicates as S, only renamed. Then there exists a *fixpoint* query $f_{k,S,S'}$ with input schema $S \cup S'$ such that for each $\langle I, J \rangle$ over $\langle S, S' \rangle$, $f_{k,S,S'}(I, J)$ returns *true* if I, J are FO^k-equivalent, and *false* otherwise.

Proof (sketch): As in the case of FO^k-equivalence of tuples, we can consider the restricted case where S and S' have just predicates of arity k. The restriction is easily removed similarly to the tuple case. The query $f_{k,S,S'}$ (for the restricted case) works as follows. It performs precisely the same analysis for the two structures over S and S', using the set F of conjunctive queries in Lemma 3.8. The results of the analysis are, for each structure, $R^k_{\prec}$ and the R_φ's for φ in F. Next, the query checks whether the results are isomorphic for the two structures, i.e. the $abs_k(R^k_{\prec})$ and $abs_k(R_\varphi)$'s are the same, as well as $abs_k(S)$ and $abs_k(S')$. This is a PTIME computation on an ordered database and thus, can be performed in *fixpoint* by [Imm86, Var82]. The query returns *true* if no difference is detected, and *false* otherwise. We next show that the query returns *true* if and only if I and J are FO^k-equivalent.

Suppose that the query returns *true*, i.e., there is no difference in the outcomes of the analysis on I and J. Let σ be an FO^k sentence over[10] S (or S'). Then there exists another FO^k formula ψ with k free variables, such that on each input over S (or S'), ψ is non-empty iff the input satisfies σ (just use existential quantification). By Lemma 3.8, ψ is in F^*. Since $abs_k(R^k_{\prec})$, $abs_k(R_\varphi)$, and $abs_k(S)$ and $abs_k(S')$ are the same for I and J, it easily follows that $abs_k(\psi(I)) = abs_k(\psi(J))$, so $\psi(I) = \emptyset$ iff $\psi(J) = \emptyset$. Thus, I and J are FO^k-equivalent.

Conversely, suppose there is a difference in the results of the analysis. There are two possibilities:

1. the number of blocks of the final partition is different for I and J, and

2. the number of blocks is the same, but the $abs_k(S)$ or $abs_k(R_\varphi)$ are different in I and J.

Suppose (1) holds. Consider the refinement process described in Lemma 3.1. Note first that

($\star$) Each block of the current ordered partition during the computation is the result of an FO^k query.

To see ($\star$), recall that F satisfies the conditions of Corollary 3.2. As shown in the proof to that corollary, each block of the current partition is definable by a query in F^*. By Lemma 3.8, each query in F^* is definable in FO^k, so ($\star$) holds.

Since (1) holds, there exists some point where the computations of the ordered partitions for I and J differ. The first point where this occurs, the ordered partitions computed so far for I and J have the same number of blocks, and the need for a refinement arises in the computation for I but not in the computation for J (or conversely, a case which is treated symmetrically). Let $\langle \delta_1, .., \delta_n \rangle$ and $\langle \delta'_1, .., \delta'_n \rangle$ be the ordered partitions computed so far for I and J. There exists a sequence $i_1, ..., i_l, i_{l+1}$ and a conjunctive query φ such that, in the computation on I,

($\star\star$) $\varphi[\delta_{i_1}, ..., \delta_{i_l}] \cap \delta_{i_{l+1}} \neq \emptyset$ and $\delta_{i_{l+1}} - \varphi[\delta_{i_1}, ..., \delta_{i_l}] \neq \emptyset$,

but ($\star\star$) does not hold in the computation on J. Since by ($\star$) each δ_{i_j} is definable by a FO^k formula, it is easily seen that ($\star\star$) is also expressible by some FO^k sentence, which is satisfied by I but not by J. Thus, I and J are not FO^k-equivalent.

Next, suppose that (2) holds. Suppose the final ordered partitions for I and J has n blocks $\delta_1, ..., \delta_n$. By ($\star$), the i-th block δ_i is defined by some FO^k formula, say ψ_i. Suppose $abs_k(R_\varphi)$ is different for I and J. Then there exists a sequence $i_1, ..., i_l, i_{l+1}$ such that $\langle i_1, ..., i_l, i_{l+1} \rangle \in abs_k(R_\varphi)$ holds for I but not J (or conversely). Thus, $\delta_{i_{l+1}} \subseteq \varphi[\delta_{i_1}, ..., \delta_{i_l}]$ holds in I but not J. Since each δ_{i_j} is defined by an FO^k formula, the above is expressible by an FO^k sentence which is satisfied by I but not J. $\square$

[10] The difference in the names of predicates in S and S' is ignored, so that they are viewed as the same schema, and the same query can be applied to both S and S'. More formally, one should considered a query φ and the query obtained from it by renaming its predicates.

Remark. The above result extends to equivalence of finite structures with respect to sentences in the infinitary logic with k variables, $L^k_{\infty\omega}$. Once again, this is due to the fact (pointed out by P. Kolaitis and M. Vardi [KV90]) that each $L^k_{\infty\omega}$ sentence is equivalent on finite structures to an infinite disjunction of infinite conjunctions of FO^k sentences. Thus, two instances are equivalent with respect to $L^k_{\infty\omega}$ iff they are equivalent with respect to FO^k. It follows from Theorem 3.11 that equivalence of structures with respect to $L^k_{\infty\omega}$ sentences is definable in *fixpoint*. □

4 While versus Fixpoint

Since *fixpoint* and *while* express PTIME and PSPACE, respectively, on ordered inputs[11], *fixpoint* $\subset$ *while* if PTIME $\subset$ PSPACE. However, the converse has been an open question [Cha88]. In this section, using the normal form, we show that the separation of *fixpoint* and *while* implies PTIME $\subset$ PSPACE.

Theorem 4.1 *Fixpoint* = *while* iff PTIME = PSPACE.

Proof (sketch): The "only if" part follows from the fact that, on ordered databases, *fixpoint* expresses PTIME [Imm86, Var82], and *while* expresses PSPACE [Var82]. Indeed, suppose that PTIME $\neq$ PSPACE. Then there exists a PSPACE function f which is not computable in PTIME. We can assume without loss of generality that f is a boolean function on the integers. Now consider the query q which takes as input a binary relation. The answer is yes iff the binary relation encodes a straightline graph of length n and $f(n)$ is true. Then this query is in PSPACE, so in *while* since the database is ordered; and since f is not in PTIME, the query is not in PTIME, so not in *fixpoint*. Hence, *fixpoint* $\neq$ *while*.

Consider the "if" part. Suppose PTIME = PSPACE. Consider a *while* query w in the normal form of Theorem 3.7. Thus, $w = f \circ w' \circ \pi$ where $w'(f(I)) = abs_k \circ abs_k(w') \circ abs_k^{-1}(f(I))$ for each input I. The query $abs_k(w')$ is a *while* query on the ordered input $abs_k(f(I))$ (with respect to the ordering in $abs_k(R^k_{\prec})$). Now $abs_k(w')$ is in PSPACE with respect to the size of $abs_k(f(I))$, thus in PTIME. Since *fixpoint* = PTIME on ordered inputs, there is a *fixpoint* query f' equivalent to $abs_k(w')$ on ordered databases. It is easily seen that there exists a k-abstractable *fixpoint* query f'' such that $f' = abs_k(f'')$. (Intuitively, one can think of f'' as $abs_k^{-1}(f')$.) Indeed, f'' is obtained by replacing variables standing for elements from the ordered domain, by k-tuples of variables representing tuples from the corresponding equivalence classes, and tests of equality of elements by tests of equivalence of k-tuples. Then $w = f \circ f'' \circ \pi$, so w is a *fixpoint* query. □

Theorem 4.1 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

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

time and space generic computations, respectively. In the next section, we will consider GM^{loose}, a powerful extension of the *while* language. We shall see 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, we will show next that the PTIME fragment of *while* is *fixpoint* iff PTIME = PSPACE.

To prove the result, we will use the fact that certain fixpoint computations on inputs with a high degree of regularity can be achieved very efficiently. This uses a technique of Lindell [Lin90] and is based on encoding words in a relational structure.

Suppose we wish to encode a word w over $\{0,1\}$ in a relational structure. This can be easily done as follows. Suppose $w = x_1 ... x_d$. We use a binary relation $\prec$ which provides a total order for d elements $\{a_1, ..., a_d\}$. We also use a unary relation $L = \{a_i \mid 1 \leq i \leq d, x_i = 1\}$ which marks the positions in the word where 1 occurs. For each $w \in \{0,1\}^*$, let I_w be the instance over $\prec, L$ encoding w.

To obtain our result, we will need to use a padding technique which relies on an encoding of words $w = x_1 ... x_d$ using a labeled complete binary tree of size exponential in d rather than a total order of size d. For this encoding we use again a binary relation T and a unary relation L. Here T represents the tree and L a subset of the nodes of T (the "labeled" ones). An instance $\overline{I}$ over $\{T, L\}$ is a labeled complete binary tree if $\overline{I}(T)$ is a complete binary tree and $\overline{I}(L)$ is a subset of the vertices of the tree such that if one vertex is in $\overline{I}(L)$, then all vertices at the same depth are in $\overline{I}(L)$ as well. Observe that (as noted in [Lin90]), testing whether an instance $\overline{I}$ is a labeled complete binary tree can be done in *fixpoint*. Next, note that a labeled complete binary tree of depth n can be viewed as the encoding of a word $w = x_1 ... x_n$ over $\{0,1\}$ defined by: $x_i = 1$ iff the nodes at the i-th level of the tree are labeled. Let $\overline{I}_w$ denote the labeled complete binary tree over $\{T, L\}$ which encodes w. Consider now a boolean query f over instances of $\{T, L\}$ which are labeled complete binary trees. Let $L_f = \{w \mid w \in \{0,1\}^*, f(\overline{I}_w) = \ true\}$. We will use the following result of Lindell.

Lemma 4.2 [Lin90] Let f be a boolean *fixpoint* query on instances over $\{T, L\}$ which are labeled complete binary trees. Then L_f is in PTIME.

For the sake of completeness, we briefly explain Lindell's technique. The equivalence classes under automorphisms of labeled complete binary trees are used. It is shown that for such $\overline{I}$, $\equiv^k_{\overline{I}}$ is a polynomial in the depth of the tree. Next, one can compute a concise representations of the equivalence classes of $\equiv^k_{\overline{I}}$, and the effect of an FO formula on the tree can be described by another FO formula in terms of the representations of the equivalence classes alone. This allows the efficient simulation of the *fixpoint* computation on this concise representation. The *fixpoint* query on the concise representation runs in PTIME in the depth of the tree. This yields a program for recognizing L_f in PTIME with respect to the word length. For more details, see [Lin90].

Using this result, we show:

Theorem 4.3 *fixpoint* $=$ *while*$|_{PTIME}$ iff PTIME $=$ PSPACE.

Proof sketch: The "if" part follows from Theorem 4.1. The "only if" part is based on a padding argument. Suppose *fixpoint* $=$ *while*$|_{PTIME}$. Let L be a language over $\{0,1\}$ in PSPACE. We show that L is in PTIME in three phases.

Phase 1

For the first phase, we use the encoding of words as instances over $\{\prec, L\}$. Since *while* expresses all of PSPACE on ordered inputs, there is a *while* query q accepting precisely the set of instances $\{I_w \mid w \in L\}$. Note that q takes space $|w|^k$ and time $2^{|w|^k}$ for some k.

Phase 2

In the second phase, we use a padding technique to obtain from q a *while* query running in time polynomial in the size of the input, and which by the hypothesis is equivalent to a *fixpoint* query. An input is an encoding of a word w, but the encoding is exponential in $|w|$. We encode a word of length d as a labeled complete binary tree of depth polynomial in d. More precisely, if w has length d, let $\overline{w} = w0^{(d^k-d)}$ (w is extended to a word of length d^k by padding it with zeroes to the right). Let $\overline{L} = \{\overline{w} \mid w \in L\}$. The encoding of w is $\overline{I}_{\overline{w}}$. Observe that a *fixpoint* query can be used to verify that an input over $\{T, L\}$ is the encoding $\overline{I}_{\overline{w}}$ of some word w.

Consider the *while* query q' over $\{T, L\}$ obtained from q by using, instead of $a_1, ..., a_d$, the elements in the d first levels of the tree $\overline{I}(T)$. Whenever q tests for equality between two constants, q' tests whether they are at the same level in the tree. Note that this can be done in *fixpoint*. Thus, each step of q is simulated by at most a polynomial number of steps in q'. Thus q' takes time at most $2^{d^k} \times poly(\overline{I}_{\overline{w}})$ for some polynomial *poly*, i.e., q' takes only polynomial time in the size of $\overline{I}_{\overline{w}}$. Thus, q' is in *while*$|_{PTIME}$. Hence, q' is equivalent to some f in *fixpoint*.

Phase 3

At this point, we have a *fixpoint* query f which accepts precisely inputs encoding words in L. However, while f is polynomial in the size of $\overline{I}_{\overline{w}}$, this does not show that L is in PTIME, since constructing $\overline{I}_{\overline{w}}$ would require exponential time. However, Lemma 4.2 states that $\overline{L}$ is indeed in PTIME, since $\overline{L} = L_f$ for the *fixpoint* query f. It easily follows that L is in PTIME. Thus PSPACE $=$ PTIME. □

Note that Theorem 4.3 reduces the separation of PTIME and PSPACE to the separation of two classes of queries *within* PTIME.

5 The Loosely-Coupled Generic Machine

In this section, we present several powerful extensions of the *while* queries. Essentially, the extensions augment the *while* queries with the ability to perform arbitrary computations which interact with the database by means of FO queries. We consider several equivalent extensions. The first is GM^{loose}, a relational machine consisting of a Turing Machine (TM) component loosely coupled with a relational store. The machine is called *loosely coupled Generic Machine* - GM^{loose} (in follow-up papers such as [AVV92b], the machine is simply called *relational machine*). The other extensions augment *while* with integer variables and the ability to perform arithmetic. This yields two languages, $while^{+}$ and $while^{++}$, which differ in the way the integer computation interacts with the database. However, $while^{+}$ and $while^{++}$ turn out to be equivalent. The main result for GM^{loose}, $while^{+}$, and $while^{++}$ is an extension of the normal form for *while* to these languages. Using the normal form, we study the expressive power of the languages. We also consider the PTIME and PSPACE restrictions of the machine with respect to (i) the size of the input and (ii) an alternative measure of the input based on the discerning power of the machine.

Description of GM^{loose}

We introduce here our machine model of generic computation, called *loosely coupled Generic Machine*, and denoted GM^{loose}. GM^{loose} consists of a TM augmented with a *relational store*. This models general-purpose computation interacting with a relational database through an abstract interface. 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 in the relational store 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. We omit the formal definition of GM^{loose}, which is straightforward but lengthy.

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. It is easily seen that such machines define only generic mappings from input to output relations. When there is no confusion, we also denote by 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.

5.1 Normal Form

In this section, we extend the normal form of *while* to GM^{loose}. Results on the expressive power of the machine are derived in the next section.

Since GM^{loose} has a TM component, it can perform arbitrarily complex computations on the tape. We will see that GM^{loose} is 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*, GM^{loose} interacts with the databases via a finite set of FO queries, which limits its ability to distinguish among tuples in the input. Essentially, GM^{loose} 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 GM^{loose}, which says essentially that any GM^{loose} computation can be reduced to a GM^{loose} computation over an ordered input via a *fixpoint* query.

We next present the normal form for GM^{loose}. (For a given GM^{loose} M, let $FO(M)$ denote the set of FO formulas used in its finite control.)

For each GM^{loose} M, there exists an equivalent machine which works in four phases:

1. The first phase is equivalent to a *fixpoint* query which performs the analysis of the input and produces (i) the computation of an ordered partition refining $\equiv_{I,FO(M)}$ and (ii) action tables for the conjunctive queries in $FO(M)$ on the blocks of the partition.

2. 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 PTIME.)

3. 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; complement queries are simulated directly; conjunctive queries are simulated using the description of their effect, which is loaded on the tape.

4. 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.

We now state the normal form for GM^{loose} more precisely. We use informally the notion of GM^{loose} program (finite control) and concatenation of programs, with the obvious meaning.

Theorem 5.1 For each GM^{loose} program M, there is an equivalent GM^{loose} program of the form:

$$M_{fixpoint}; M_{load-tape}; M_{compute-tape}; M_{store-result},$$

where:

1. $M_{fixpoint}$ computes a *fixpoint* query on the relational store.
2. $M_{load-tape}$ is a PTIME computation involving both the tape and the store.
3. $M_{compute-tape}$ uses exclusively the tape.
4. $M_{store-result}$ is a PTIME computation involving both the tape and the store.

Proof (sketch): Consider a given GM^{loose} M. It is easy to see that there exist GM^{loose} M_1, M_2, M_3 such that $M = M_1; M_2; M_3$ where M_1 and M_3 compute first-order transformations, and M_2 has the property that all relations in the store have the same arity k, and that its FO formulas consist just of conjunctive queries and negations. Thus, we can assume without loss of generality that the given GM^{loose} M has this property. One can define the equivalence relation $\equiv^k_{I,M}$ analogously to $\equiv^k_{I,w}$ for a *while* query w: for k-tuples u, v, $u \equiv^k_{I,M} v$ iff at any point in the computation of M on I, u and v belong to exactly the same relations in the relational store. In other words, $u \equiv^k_{I,M} v$ iff M cannot distinguish between u and v on input I. Let $FO(M)$ denote the FO queries used by M. Clearly, $\equiv^k_{I,FO(M)}$ refines $\equiv^k_{I,M}$. By Theorem 3.1, there exists a *fixpoint* query f which computes in a predicate $R^k_{\prec}$ an ordered partition refining $\equiv^k_{I,FO(M)}$, so $\equiv^k_{I,M}$. Also, f computes, for each conjunctive query φ in $FO(M)$, a relation R_φ describing the effect of φ in terms of the blocks of the partition. This concludes the first phase, which is the same as that in the normal form for *while*.

In the second phase, the entire information about the input, $R^k_{\prec}$, and the R_φ, is reconstructed on the tape. This is possible because the description can be done using just integers corresponding to the blocks in the ordered partition. The procedure involves computing the rank of tuples according to $R^k_{\prec}$. The rank of a tuple is computed by stepping through the consecutive levels of $R^k_{\prec}$ and counting the number of steps it takes to reach the level of the tuple. This phase takes PTIME in the size of the input. The third phase consists of a simulation of M using the tape alone. This is possible because the tape contains all necessary information about the input and the action of the FO formulas on it. Lastly, the fourth phase decodes the representation of the result on the tape into the actual output relations. This is the converse of the second phase: the integers on the tape, representing equivalence classes of tuples, are replaced by the actual tuples, using again $R^k_{\prec}$. Like phase 2, this also takes PTIME in the size of the input. □

5.2 Expressiveness

We first consider computations on ordered inputs.

Theorem 5.2 GM^{loose} is complete on ordered inputs.

Proof: For an ordered input I, the entire information about the input can be reconstructed on the tape, since each element can be identified with an integer. Then an arbitrary computation can be performed on the tape, and the final result is decoded into the output relations of the store. $\Box$

We next consider computations on unary relations.

Theorem 5.3 $GM^{loose}\!\downarrow$ is equivalent to FO for inputs consisting only of unary predicates. (In particular, GM^{loose} cannot compute *even*.)

Proof (Sketch) : Let M be a GM$^{loose}\!\downarrow$ machine with unary input predicates. Suppose M is in the normal form

$$M_{fixpoint}; M_{load-tape}; M_{compute-tape}; M_{store-result}.$$

Suppose that the input I consists of m unary relations. Consider first the classes of $\equiv_I^1$ on singletons. There are at most 2^m such classes, since a class consists of the intersection of some input relations. Now for each k, it is easily seen that $\equiv_I^k$ consists of at most $(2^m)^k$ classes. Thus the number of equivalence classes of $\equiv_I^k$ is bounded by a constant independent of I. Due to the inflationary nature of *fixpoint* computations, and the fact that equivalence classes of $\equiv_I^k$ are treated as indivisible blocks of tuples, the number of steps of the computation of $M_{fixpoint}$ is bounded by a constant. Thus, the *fixpoint* query can be replaced by a first-order query φ returning the result of the analysis. Next, note that $\equiv_I^k$ refines the ordered partition produced by $M_{fixpoint}$ (for an appropriate k). Thus, the size of that partition is bounded by a constant. It follows that there is only a bounded number of possible outcomes of $M_{load-tape}$, which describes on the tape the result of the analysis of the input in terms of integers representing the equivalence classes. Let $\{\xi_1, ..., \xi_p\}$ be the set of possible outcomes. For each ξ_i, let J_i be the corresponding final result in the relational store. Clearly, for each ξ_i, there is a first-order sentence φ_{ξ_i} which allows to check whether φ yields description ξ_i on the input. For each output predicate R, the result $J_i(R)$ is also definable by a first-order query on the relational store, say $\psi_{J_i(R)}$. Then, the first-order query computing the same mapping as M is:

	the result of the analysis is pattern ξ_1 and $\psi_{J_1(R)}$ defines the result on R
or	the result of the analysis is pattern ξ_2 and $\psi_{J_2(R)}$ defines the result on R
...	

which is formally:

$$(\bigvee_{i=1}^{p}(\varphi_{\xi_i} \wedge \psi_{J_i(R)})).\Box$$

Remark (Infinitary Logic): Moshe Vardi noted that GM^{loose} is subsumed by the infinitary logic with finitely many variables ($L^{\omega}_{\infty\omega}$), studied in [KV90]. Recall that $L^{\omega}_{\infty\omega}$ is first-order logic extended by allowing disjunctions and conjunctions of infinite sets of formulas, but using only a bounded number of variables. Intuitively, the restriction on the number variables in the formulas corresponds in GM^{loose} to the fact that relations in the relational store have fixed arities. The connection with infinitary logic provides an alternative proof of the fact that *even* cannot be computed in GM^{loose}. Indeed, $L^{\omega}_{\infty\omega}$ has a 0/1 law, i.e., the asymptotic probability of each property it expresses always exists and is 0 or 1 [KV90]. As a consequence, GM^{loose} also has a 0/1 law, so it cannot compute *even*.

The connection between GM^{loose} and infinitary logic is further studied in [AVV92a]; it is shown there that GM^{loose} corresponds precisely to a natural effective fragment of $L^{\omega}_{\infty\omega}$. Specifically, GM^{loose} expresses exactly the properties definable by sentences in $L^{\omega}_{\infty\omega}$ whose sets of models are r.e.

5.3 The while language with integer arithmetic

In this section, we show that GM^{loose} has the same expressive power as two extensions of the *while* language with integer arithmetic.

First, let *while*$^+$ be the *while* language augmented with

1. integer variables $i, j, ...$, initialized to zero,
2. $increment(i)$ and $decrement(i)$ statements, and
3. tests of the form $i = 0$ in termination conditions of loops.

To see an example of a *while*$^+$ program, consider the following program which computes the diameter of a graph G:

$$
\begin{array}{l}
R := \emptyset; \\
S := G; \\
while\ R \neq S\ do \\
\quad begin \\
\quad R := S; \\
\quad S := \{x, y \mid (S(x, y) \vee \exists z(G(x, z) \wedge S(z, y)))\}; \\
\quad increment(i) \\
\quad end;
\end{array}
$$

From the equivalence between TM's and counter machines, the following is immediate.

Fact: GM^{loose} is equivalent to *while*$^+$. □

The language *while*$^+$ was first studied in [Cha81], under the name LEC. It was also shown there that *even* cannot be computed in that language.

In the first extension of *while* above, there is a clear separation between the integer and the first-order computations. In some sense, the coupling is loose like in the machine. We next consider a tighter coupling where it is possible to "move" integers from the integer variables to the relational store. We show that, surprisingly, this does not yield additional expressive power.

Let $while^{++}$ be the language extending $while^{+}$ with statements of the form $R := \{\langle i_1, ..., i_k \rangle\}$ where R is a k-ary relation and the i_j are integer variables. (We assume here that the set of domain elements and the set of integers are disjoint.) Thus, $while^{++}$ allows mixing integers with the data.

A subtlety in the semantics of $while^{++}$ is the meaning of negation (or complement). We assume that the domain of the database consists of the original domain extended with the integers that have been introduced so far in relations. Thus these integers are now viewed as standard elements. In particular, complement is performed with respect to the extended set of elements.

In $while^{++}$, the result of a program may contain integers. We are not concerned by this aspect here, so we consider *pure* $while^{++}$ programs, i.e., $while^{++}$ programs where the outputs never contain integers. We also assume without loss of generality that the $while^{++}$ programs are simple, so in particular, all predicates have the same arity, say k. Like in the previous sections, $\Delta_k(I)$ is the set of k-tuples formed with elements from the input I (and no integers).

We next show that the tighter coupling does not yield additional expressive power.

Theorem 5.4 GM^{loose} is equivalent to pure $while^{++}$.

Proof: Since $\mathrm{GM}^{loose} = while^{+}$ and $while^{+} \subseteq$ pure $while^{++}$, it follows that $\mathrm{GM}^{loose} \subseteq while^{++}$.

Conversely, let w be a simple, pure $while^{++}$ program whose relations have arity k, and I an input instance. Consider a relation R of arity k obtained in the computation of w on input I. The relation R contains elements from I ($elem(I)$) mixed with integers. One can view R as consisting of the union of several relations over $elem(I)$ as follows. Let $Index_k(Z) = (Z \cup \{@\})^k$ where Z is a set of integers and the symbol @ is meant as a place holder for elements from I. For each $f \in Index_k(Z)$, let $slice_f$ be the mapping on finite relations of arity k which first selects all tuples u satisfying the "pattern" described by f (i.e. $u(i) \in elem(I)$ if $f(i) = @$, and $u(i) = f(i)$ otherwise), then projects out the integer components. To see an example, if $k = 3$ and $R = \{[1, a, b], [1, b, c], [2, c, 3]\}$, $slice_{1@@}(R) = \{[a, b], [b, c]\}$, and $slice_{2@3}(R) = \{[c]\}$.

We simulate the computation of the $while^{++}$ program on the tape as follows. We store on the tape two lists: the list Z_1 of all the integers that have been introduced in the relational storage so far; and the list Z_2 of all indices of length k that can be constructed with values in $Z_1 \cup \{@\}$. For each integer variable of the program, we keep its current value on the tape. To represent relation R in GM^{loose}, we would like to store the slices of R. However, their number is unbounded, so we cannot store them in the relational store. Instead, for each relation R and each $i_1 ... i_k$ in $Index_k(Z_2)$, we keep on the tape an abstract representation of $slice_{i_1 \ldots i_k}(R)$, described next. Generally, a slice is a set of tuples whose arity is possibly less than k. However, we can pad these tuples by cross product with the set of elements in the input, in order to only deal with k-tuples. For a slice $slice_{i_1 \ldots i_k}(R)$, denote by $\overline{slice}_{i_1 \ldots i_k}(R)$ the padded slice of arity k. We first use a *fixpoint* query to obtain an ordered partition of the set $\Delta_k(I)$, which depends on the first-order queries $FO(w)$ occurring in w. We will show that each slice (padded if necessary) is the union of a set of equivalence classes in the partition. The representation of $slice_{i_1,\ldots,i_k}(R)$ then consists of the list of integers corresponding to the ranks of those classes in the ordered partition.

We wish to obtain a partition which refines every possible padded slice which can be obtained in the computation. To this end we have to construct, from the conjunctive queries of w, a modified set C of conjunctive queries which describe the manipulation of padded slices by the original conjunctive queries in $FO(w)$. Each padded slice obtained in the computation of w on I will then be a union of equivalences classes of $\equiv^k_{I,C}$.

We first illustrate with an example how the queries in C are obtained. Consider the conjunctive query:

$$\varphi = \exists y_5, y_6(R(y_1, y_6, y_3, y_3) \wedge R(y_2, y_4, y_5, y_1)).$$

The query is evaluated by considering all possible valuations of the variables $y_1, \ldots, y_6$ and extracting the values of $y_1, \ldots, y_4$ for the valuations satisfying the matrix of the formula. Consider one such valuation v. For each y_i, $v(y_i)$ is an integer or a domain element. Consider, for example, valuations v such that $v(y_1) = 2$, $v(y_5) = v(y_4) = 3$ and $v(y_2), v(y_3), v(y_6)$ are domain elements. The noninteger portions of R to which these valuations apply are $slice_{2@@@}(R)$ and $slice_{@332}(R)$. The noninteger part of the result is included in $slice_{2@@3}(\varphi(R))$. We wish to express the answer in terms of the padded slices alone. For the above valuations v, we construct from φ a query $\overline{\varphi}$ such that the answer (padded to arity k) is

$$\overline{\varphi}[\overline{slice}_{2@@@}(R), \overline{slice}_{@332}(R)].$$

The query $\overline{\varphi}$ is:

$$(y_1 = y_1) \wedge (y_4 = y_4) \wedge$$
$$\exists y_5, y_6, y_1, y_4(R(y_1, y_6, y_3, y_3) \wedge R(y_2, y_4, y_5, y_1)).$$

This is generalized next.

Let

$$\varphi = \exists y_1 \ldots y_n(R_{i_1}(\vec{z}_1) \wedge \ldots \wedge R_{i_m}(\vec{z}_m))$$

be a conjunctive query in $FO(w)$. Let v be a valuation of the variables in φ into $\{@, 1\}$. We would like to obtain conjunctive queries representing the action of φ on the (padded) slices for which integers occur in positions x where $v(x) = 1$, and elements occur where $v(x) = @$. For each such v, create a new conjunctive query by modifying φ as follows:

- First, each variable x for which $v(x) = 1$ becomes existentially quantified, yielding query φ'. Note that if the variable was already existentially quantified, its status does not change (this is the case of y_5 in the example).
- Let $x_1, ..., x_p$ be the variables free in φ, for which $v(x) = 1$. The final query $\overline{\varphi}$ is

$$\bigwedge_{i=1}^{p} (x_p = x_p) \wedge \varphi'.$$

  Note that the variables x_i are used to pad representations of slices to k-tuples with cross-products of the elements.

By Theorem 3.1, there exists a *fixpoint* query f computing an ordered partition δ which refines $\equiv^k_{I,C}$.

The simulation of w by a GM^{loose} machine proceeds in four phases, much like the normal form:

$$M_f; M_{load-tape}; M_{simulate-on-tape}; M_{store-result},$$

where

(i) M_f computes the *fixpoint* query f yielding the ordered partition δ and the action tables of queries in C on δ.

(ii) $M_{load-tape}$ stores on the tape all information produced by M_f, using just integers corresponding to equivalence classes of δ.

(iii) $M_{simulate-on-tape}$ simulates w using exclusively the tape (we elaborate below on the simulation).

(iv) $M_{store-result}$ decodes the result into output relations of the relational store.

To see that (iii) can indeed be done, suppose phase (ii) in the simulation has been completed. We show by induction on the number of steps in the computation so far that, at any point in the computation of w:

1. each (padded) slice is a union of equivalence classes of δ,
2. each step of *while*$^{++}$ can be simulated on the tape.

Basis: The input relations do not contain integers, so by construction of δ, each input relation is a union of equivalence classes of δ. Thus, since Z_1 is empty, (1) holds. The indices contain only @'s and we can compute on the tape the representation of $slice_{@...@}(R)$ for each R.
Induction: Suppose now that after m steps in the computation of w, each slice consists of a union of some classes of δ, and that we have Z_1, Z_2 and δ stored on the tape. Consider the next step of the program. Four cases arise:

1. *branching condition*
   To test whether a relation R is empty, it suffices to check (on the tape) whether all slices of R are empty,

2. *complement*
   Let D_1 be the set of original elements, D_2 the set of integers introduced so far and R a relation of arity k. Then, it is easy to verify that: For each $s \in Index_k$ with integers in D_2,

   $$\overline{slice}_s((D_1 \cup D_2)^k - R) = D_1^k - \overline{slice}_s(R).$$

   Thus, we need to complement each slice with respect to the original domain. This can also clearly be done on the tape.

3. *conjunctive query* $R := \varphi(R_1, ..., R_m)$
   We have to simulate on the tape the computation of the conjunctive query. Let $x_1, ..., x_p$ be the variables occurring in the query. We have to consider all valuations of $x_1, ..., x_p$ to elements from the input (symbolically @) or to integers in Z_1. Consider first all valuations of $x_1, ..., x_p$ in $Z_1 \cup \{@\}$. For each such valuation v, the conjunctive query $R := v\varphi(R_1, ..., R_m)$ can be viewed as a conjunctive query over the slices of $R, R_1, ..., R_m$ induced by v, which results in one or more slices. It is easily seen that the set C constructed above contains all conjunctive queries necessary to define any resulting slice representation in terms of previous slice representations. By construction, any resulting slice representation remains a union of equivalence classes of δ.

4. *assignment* $R := \{\langle j_1, ..., j_k \rangle\}$
   We may have to modify Z_1 and Z_2. This can easily be done on the tape. The new slices are empty except for $slice_{i_1,...,i_k}(R)$ where $i_i, ..., i_k$ are the current values of variables $j_1, ..., j_k$. The slice $slice_{i_1,...,i_k}(R)$ contains the empty tuple $\langle\ \rangle$.

By induction, the simulation can be carried out on the tape. Since the integers do not occur in the result, the construction of the output in the store (phase (iv)) can be done as in the normal form for GM^{loose}. □

5.4 Robustness of GM^{loose}

The normal form shown for GM^{loose}, and the above results on the equivalence of GM^{loose} and *while*$^{++}$, allow proving interesting robustness properties of GM^{loose}, which we briefly discuss next.

Intuitively, the equivalence of GM^{loose} and *while*$^{+}$ with *while*$^{++}$ points to a rather remarkable fact. The integers present inside *while*$^{++}$ computations can be viewed as indices allowing to use an unbounded number of relations. This suggests that the fundamental limitation of GM^{loose} is not the finite relational store, but rather the bound on the arity of relations used in the computation. If so wished, one can define computational devices which use an unbounded number of relations explicitly, but remain equivalent to GM^{loose}. We briefly sketch such a device, which we call a *pointer machine (PM)*. A PM has an infinite tape where each cell is a pointer to a relation of some fixed arity k. There is a finite-state control. Transitions are determined by the current state and an FO query on the relation pointed to by the head. In a transition, the head can move left or right, the state can change, and the current relation can be overwritten using a FO query involving it and the last current relation. The input is the content of the current relation at the start of the computation, and the output the content of the current relation at the end of the computation. It can be seen that PM and GM^{loose} are equivalent, using the equivalence of GM^{loose} with $while^{++}$.

In the same spirit as pointer machines, one could consider relational machines extended as follows. The relational store uses an unbounded number of relations R_i of fixed arity, identified by integers i. It is easily seen that this extension with addressable relations remains equivalent to GM^{loose}.

Finally, we mention another extension of GM^{loose} which allows the dynamic generation of queries. Suppose that the GM^{loose} is extended with a query tape over alphabet $\{\exists, \forall, \wedge, \vee, \neg, x_1, ..., x_k\} \cup \{R \mid R \text{ is a predicate in the relational store}\}$, for some k. The machine can write a query on the query tape and use it to query or modify the relational store. Once again, it is easily seen that this does not increase the expressive power of GM^{loose}.

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

We consider here polynomial time and space restrictions of GM^{loose}. We consider the relation of *fixpoint* and *while* to the PTIME and PSPACE restrictions of the machine. In particular, we show that *fixpoint* and *while* do not coincide with these restrictions, despite the intuition that these languages correspond, in some natural sense, to PTIME and PSPACE generic computations. However, in the next section we show a result of this nature when an alternative measure of the input is considered instead of its size.

Let $GM^{loose}|_{PTIME}$ denote the generic mappings computable by a GM^{loose} in a number of steps polynomial in the input size. Observe that each step is computable in polynomial time (each step is constant time except FO queries which are polynomial time in the domain, therefore in the original input instance). Thus $GM^{loose}|_{PTIME} \subset$ PTIME (the query *even* separates the two classes).

Let $\mathrm{GM}^{loose}|_{PSPACE}$ denote the mappings computable by a GM^{loose} using an amount of tape polynomial in the input size. Since the relational store is also polynomial in the size of the input, $\mathrm{GM}^{loose}|_{PSPACE} \subset$ PSPACE. (The inclusion is again strict because of the *even* query.)

It is easily seen that $GM^{loose}|_{PTIME}$ and $\mathrm{GM}^{loose}|_{PSPACE}$ 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.

We are particularly interested in the connection of $GM^{loose}|_{PTIME}$ and GM$^{loose}|_{PSPACE}$ with *fixpoint* and *while*. This and related questions are answered next.

Theorem 5.5 (i) $while \subset$ GM$^{loose}|_{PSPACE}$,
(ii) $fixpoint \subset$ GM$^{loose}|_{PTIME}$,
(iii) GM$^{loose}|_{PTIME} \not\subseteq while$ (if PSPACE $\neq$ EXPTIME), and
(iv) $while \not\subseteq$ GM$^{loose}|_{PTIME}$ (if PTIME $\neq$ PSPACE).

Proof: We first prove (iii). Suppose that PSPACE $\neq$ EXPTIME. We use a padding technique similar to that in the proof of Theorem 4.3.

Let L be a language in EXPTIME and not PSPACE. Say L is in TIME(2^{n^k}) for some k. As in the proof of Theorem 4.3, we consider words $w = x_1...x_d$ encoded as labeled complete binary trees $\overline{I_{\overline{w}}}$ of depth d^k. Again, only the first d levels of the tree are used as indices for encoding the word of length d, and the remainder are used just for padding.

Consider the query q which accepts exactly the encodings $\overline{I_{\overline{w}}}$ of words w in L. We prove that q is in GM$^{loose}|_{PTIME}$ - *while*.

The query q is computed by a GM^{loose} machine in PTIME steps in the size of the input $\overline{I_{\overline{w}}}$ as follows. The word is decoded and its representation is constructed on the tape. This is in *fixpoint* so in polynomial time in the input. One can then decide membership in L in time 2^{d^k}, which is polynomial in the input. Thus q is in GM$^{loose}|_{PTIME}$.

Now suppose there is a *while* program h computing q. Thus, h accepts encodings of words w in L using labeled complete binary trees $\overline{I_{\overline{w}}}$ of depth d^k. From the normal form for *while* it follows that $h = fh'$, where f is a *fixpoint* query and h' is a *while* query on an ordered input whose number of elements is bounded by the number of equivalence classes of $\equiv^k_{\overline{I_{\overline{w}}}}$, which is polynomial in d. Hence, h' is in PSPACE(d). Using Lemma 4.2, it is easily seen that the mapping $abs_k(f(\overline{I_{\overline{w}}}))$ on words $w = x_1...x_d$ is in PTIME(d). It follows that $\{w \mid q(\overline{I_{\overline{w}}}) = \ true\}$ is in PSPACE, i.e. L is in PSPACE, which is a contradiction. Hence q is in GM$^{loose}|_{PTIME}$ $-$ *while*.

Consider (i). Clearly, $while \subseteq$ GM$^{loose}|_{PSPACE}$. Consider a language L in EXPSPACE and not in PSPACE. Such a language exists by the space hierarchy theorem [HU79]. We use the same technique as in (iii) to obtain a query in GM$^{loose}|_{PSPACE}$ $-$ *while*. Part (ii) is proven in the same way, using a language L in EXPTIME and not in PTIME, which exists by the time hierarchy theorem [HU79].

For (iv), it suffices to consider ordered inputs. Indeed, on ordered inputs *while* is PSPACE, and GM$^{loose}|_{PTIME}$ is PTIME. $\Box$

Some of these results are summarized in Figure 6. In the figure, a line indicates proper inclusion, whereas a dashed line indicates proper inclusion subject to PTIME $\neq$ PSPACE or PSPACE $\neq$ EXPTIME.

$\mathrm{GM}^{loose}|_{PSPACE}$

$\mathrm{GM}^{loose}|_{PTIME}$ $\qquad$ $while$

$while\ |_{PTIME}$

$fixpoint$

Figure 6: PTIME and PSPACE restrictions of GM^{loose}

5.6 Relational Complexity

The complexity measures considered so far were standard measures based on the input size. However, we have seen that size is not the determining factor in GM^{loose} computations. Indeed, the relational machines GM^{loose}, unlike Turing Machines, have limited access to their input, due to their limited ability to distinguish among tuples for a given input. No generic computing device can distinguish among tuples which are equivalent with respect to automorphisms of the input. For GM^{loose} machines, their discerning power is further limited by the fact that they access the relational store using just a finite set of FO queries. Thus, given an input I, a GM^{loose} M cannot distinguish between k-tuples u, v such that $u \equiv^k_{I,FO(M)} v$. Let k be the maximum number of variables used in $FO(M)$. The number of equivalence classes in $\equiv^k_{I,FO(M)}$, denoted by $size_M(I)$, is an appealing measure of the input reflecting the discerning power of M. This measure is machine dependent, which is natural since different relational machines have different degrees of accuracy, depending on their FO formulas.

We now define *relational complexity classes* based on the measure $size_M(I)$ of the input. For each Turing complexity class C, one can define a relational analog, C_r, based on the new measure of the input. For example, suppose C is PSPACE. Then PSPACE_r is the class of queries computable by some GM^{loose} M using an amount of tape space polynomial in $size_M(I)$. Similarly, PTIME_r is the set of queries computable by some GM^{loose} M in a number of steps polynomial in $size_M(I)$. (Note that FO queries are counted as single steps.)

The result of this section shows a tight correspondence between relational complexity and the *fixpoint* and *while* queries. It confirms the intuition that *fixpoint* and *while* represent, in some natural but limited sense, the PTIME and PSPACE queries.

Theorem 5.6 (i) PTIME$_r$ = *fixpoint*, and
(ii) PSPACE$_r$ = *while*.

Proof (sketch): We prove (i). The proof of (ii) is similar.

Consider first *fixpoint* $\subseteq$ PTIME$_r$. Let f be a *fixpoint* query. It is easily seen that there is a GM^{loose} M$_f$ such that the maximum arity of a relations in f and M$_f$ is the same (say, k) and for each input I:

- the number of steps in the computations of f and M$_f$ on I differ by a constant; and
- $\equiv^k_{I,f}$ and $\equiv^k_{I,M_f}$ are the same.

Since the number of steps in the computation of f on I is bounded by a polynomial in the size of $\equiv^k_{I,f}$, it follows that the number of steps in the computation of M$_f$ on I is bounded by a polynomial in the size of $\equiv^k_{I,M_f}$. Thus, f is in PTIME$_r$.

To see the converse, let q be an arbitrary PTIME$_r$ query. Thus, there is a GM^{loose} machine M^q computing q, such that, on input I, M^q computes for $(size_M(I))^m$, steps, for some fixed m. By the normal form, M^q is equivalent to

$$M^q_{fixpoint}; M^q_{load-tape}; M^q_{compute-tape}; M^q_{store-result},$$

where $M^q_{computetape}$ is essentially a Turing Machine running in time polynomial in its input. The fact that the computation of $M^q_{compute-tape}$ can be done in *fixpoint* follows from the fact that *fixpoint* on ordered inputs is PTIME [Imm86, Var82], and the fact that $M^q_{compute-tape}$ runs in time polynomial in the size of its input ($(size_M(I))^m$ steps). It follows easily that $M^q_{load-tape}; M^q_{compute-tape}; M^q_{store-result}$ can be simulated by *fixpoint*. Finally, $M^q_{fixpoint}$ can be expressed in *fixpoint* by definition. □

Thus, relational complexity classes can be captured by languages, whereas this remains an elusive goal for most Turing complexity classes of interest.

Remark. The tight dependence of $size_M(I)$ on the particular machine M can be slightly relaxed as follows. The discerning ability of GM^{loose} machines depends essentially on the number of variables used in their FO formulas. Call a GM^{loose} M *k-ary* if the maximum number of variables used in $FO(M)$ is k. Let $size_k(I)$ be the number of equivalence classes of k-tuples which cannot be distinguished by *any* GM^{loose} of arity k (this turns out to be the same as equivalence with respect to all FO^k formulas). One can define relational complexity classes based on $size_k(I)$. The relational complexity classes obtained are the same as above, and so Theorem 5.6 continues to hold. The measure $size_k(I)$ is adopted in the follow-up paper [AVV92b].

6 The Generic Machine

In [AV91b], we considered in detail complexity issues raised by generic computation without order. We briefly recall here the main ideas. This portion of [AV91b] will be presented in detail in a future paper.

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. This mismatch occurs because database computation is *not* adequately captured by Turing Machines (TM's). In [AV91b], we propose as an alternative an extension of GM^{loose}, the *Generic Machine* (GM). We define robust complexity classes of queries using the machine.

As discussed earlier, databases provide a logical, abstract view of data, purposedly independent of the internal representation. 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. A typical example of the mismatch between complexity of generic computation and standard Turing complexity, is the *even* query on a set, which has low Turing complexity but is a hard query in the usual query languages. In particular, it cannot be computed by GM^{loose}. The source of the difficulty is that 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.

To understand the complexity of queries like *even*, we need to provide a complete and robust model of generic computation, which can be used as a basis for defining complexity classes proper to generic computation. In particular, such a device must overcome the limitations of GM^{loose}. The machine GM defined in [AV91b] is a complete extension of GM^{loose}. The machines differ 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.

Based on GM, we can 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. Similar results can be obtained for the object-oriented language IQL of [AK89]. Indeed, more recently, 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 once again with GEN-PTIME and GEN-PSPACE. (There are minor differences due to the presence of invented object identifiers in the result).

We obtain in [AV91b] several results concerning GEN-PTIME and GEN-PSPACE. In particular, we 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. These results point to a trade-off between complexity and computing with an abstract interface.

7 Conclusion

The results obtained in the paper provide insight into the nature of various forms of general-purpose computation interacting with a database by means of a finite set of FO queries. The normal form provides a bridge between computing without order on the database domain and computing with order. This helps understand the impact of order, and the cost of generic computation without order.

The main result obtained using the normal form is that *fixpoint* = *while* iff PTIME = PSPACE. This is significant from the point of view of the theory of query languages, since it shows that, most likely, *fixpoint* is weaker than *while*. However, the result is also significant from the point of view of the connection of logic and complexity. Indeed, it characterizes the relationship of PTIME and PSPACE in purely logical terms. These results are extended in [AVV92b], where logical analogs of the containments of all complexity classes between PTIME and EXPTIME are provided. The logics used are variations of *fixpoint* logic.

We argued that there is a fundamental mismatch between Turing complexity classes and logic, due to the fact that Turing machines work on encodings of structures, while logic works on the structures themselves. To overcome this mismatch, we proposed a model of computation consisting of a relational machine GM^{loose}, which operates directly on relations. These model naturally computation where an application program interacts with a database by means of an abstract interface. We defined *relational* complexity classes C_r based on GM^{loose}, using a measure of the input capturing the discerning power of the machine. We showed that PTIME_r = *fixpoint* and PSPACE_r = *while*. Thus, there a precise match between *fixpoint* and *while*, and relational complexity classes. These results are extended in [AVV92b] to other logics and other relational complexity classes. Moreover, it is argued that relational complexity provides a natural bridge between logic and Turing complexity.

The relational machine GM^{loose} was shown to be subsumed by the infinitary logic $L^{\omega}_{\infty\omega}$. It is of interest to understand the precise connection between the two. This issue is explored in [AVV92a], where it is shown that GM^{loose} corresponds to the natural effective fragment of $L^{\omega}_{\infty\omega}$.

As pointed out in an earlier remark, the normal form can be used to optimize relational computations. Indeed, the normal form reduces computation over the database to a computation over an ordered domain which could be much smaller than the original one. This is investigated in [ACV92] where the potential of the normal form as an optimization tool is demonstrated by showing that the expected benefit of such optimization is high on the average. In [AG92], practical heuristics based on the normal form are developed and evaluated experimentally.

Acknowledgement. We wish to thank Moshe Vardi and Phokion Kolaitis for useful discussions on the normal form and for pointing out the connection with first-order logic with a bounded number of variables, and with infinitary logic. Christos Papadimitriou and Moshe Vardi provided valuable comments on an early draft of this paper. Thanks also to Dino Karabeg for discussions related to this material.

References

[ACV92] S. Abiteboul, K. Compton, and V. Vianu. Queries are easier than you thought (probably). In *Proc. ACM Symp. on Principles of Database Systems*, pages 23–32, 1992.

[AG92] S. Abiteboul and A. Van Gelder. Optimizing active databases using the split technique. In *Proc. of Intl. Conf. on Database Theory*, 1992. To appear.

[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*.

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

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

[AV91b] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 209–219, 1991.

[AVV92a] S. Abiteboul, M.Y. Vardi, and V. Vianu. Computing with infinitary logic. In *Proc. of Intl. Conf. on Database Theory*, 1992. To appear.

[AVV92b] S. Abiteboul, M.Y. Vardi, and V. Vianu. Fixpoint logics, relational machines, and computational complexity. In *Proc. Conference on Structure in Complexity Theory*, pages 156–168, 1992. To appear in *JCSS*.

[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.

[CH85] A.K. Chandra and D. Harel. Horn clause queries and generalizations. *J. Logic Programming*, 2(1):1–15, 1985.

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

[Cha88] A.K. Chandra. Theory of database queries. In *Proc. ACM Symp. on Principles of Database Systems*, pages 1–9, 1988.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation on conjunctive queries in relational data bases. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 77–90, 1977.

[DV91] K. Denninghoff and V. Vianu. The power of methods with parallel semantics. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 221–232, 1991.

[Fag77] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. on Database Systems*, 2:262–278, 1977.

[Fag90] R. Fagin. Finite-model theory - a personal perspective. In *Third Int'l. Conf. on Database Theory*, pages 3–24, 1990.

[Fri71] H. Friedman. Axiomatic recursive function theory. In R.O. Gandy and C.E.M. Yates, editors, *Logic Colloquium '69*, pages 113–137. North Holland, Amsterdam, 1971.

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

[Gur88] Y. Gurevich. Logic and the challenge of computer science. In E. Borger, editor, *Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory.* Addison-Wesley, 1979.

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

[Imm87a] N. Immerman. Expressibility as a complexity measure: Results and directions. Technical Report DCS-TR-538, Yale Univ., 1987.

[Imm87b] N. Immerman. Languages which capture complexity classes. *SIAM J. on Computing*, 16(4):760–778, 1987.

[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.

[KV90] P.G. Kolaitis and M.Y. Vardi. 0-1 laws and decision problems for fragments of second-order logic. *Information and Computation*, 87:302–338, 1990.

[Lei89] D. Leivant. Descriptive characterizations of computational complexity. *Journal of Computer and System Sciences*, 39(1):51–83, 1989.

[Lin90] S. Lindell. *An analysis of fixed-point queries on binary trees.* PhD thesis, UCLA, 1990.

[P3́7] G. Pólya. Kombinatorische anzahlbestimmungen fur gruppen, graphen und chemische verbindungen. *Acta Math.*, 68:145–254, 1937.

[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.