Expressive Power of Query Languages*

Serge Abiteboul   Victor Vianu

Abstract

Recent research on query languages and their expressive power is discussed. Several query languages are described, emphasizing recursive extensions of the first-order queries based on three paradigms: logic, algebraic, and logic programming. Many of these languages converge around two central classes of queries, the *fixpoint* queries and the *while* queries. The relative expressive power of these languages is examined, as well as their connection to complexity classes of queries. The focus is on the inability of languages to express low complexity classes of queries, PTIME and below. We consider several ways to circumvent this difficulty. The first is to introduce an ordering of the domain which leads to a trade-off between complexity and the data independence principle. The cost of computing without order is formalized by defining non-standard complexity classes based on a model of database computation called *Generic Machine*. As an alternative to order, it is shown that the difficulty of expressing low complexity classes can be circumvented by introducing non-deterministic constructs in query languages. Expressiveness above NP is also discussed.

*Authors addresses: Serge Abiteboul, INRIA, B.P. 105, 78153 Le Chesnay CEDEX, France (abitebou@inria.inria.fr); Victor Vianu, CSE 0114, U.C. San Diego, La Jolla, CA 92093-0114, USA (vianu@cs.ucsd.edu).
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.

1 Introduction

Query languages constitute a central aspect of practical and theoretical database research. While closely related to finite model theory and descriptive complexity, research on query languages for databases raises important specific issues. This paper provides a bird's eye view of some of the main ideas and recent developments in research on query languages, in a concise and mostly informal manner.

The paper has two main aspects. First, a review of relational query languages is provided, emphasizing the recursive extensions of the first-order queries. Along the way, results on the relative expressive power of these languages are mentioned. The second aspect is the expressive power of the languages w.r.t. complexity classes of queries, with an eye to complexity-taylored query language design.

First-order logic without function symbols (FO) provided the basis for query languages for the early commercial relational database systems. Its appeal lies in its simplicity, clear semantics, and dual declarative and procedural incarnations. Indeed, FO has a simple algebraization which is particularly amenable to optimization. While FO has many appealing features, it has limited expressive power. For instance, it cannot compute the transitive closure of a graph. (We provide a sketch of this result using Ehrenfeucht-Fraissé games.) Therefore, many recursive extensions of FO have been proposed. We review several such extensions, based on three paradigms: logic, algebraic, and logic programming. Many of these extensions converge around two central classes of queries: the *fixpoint* queries and the *while* queries. We examine various aspects of *fixpoint* and *while* in detail throughout the paper.

We present various ways of "sizing up" query languages w.r.t. complexity. We are particularly interested in the ability of languages to capture complexity classes of queries. In practice, the problem of capturing *low* complexity classes, below PTIME, is of most interest. Unfortunately, this problem remains largely unresolved. Indeed, the existence of a language expressing exactly PTIME is one of the main open problems in the theory of query languages. On the other hand, such classes can be captured under the assumption that an order on the domain is available.

The impact of order is a running thread throughout the discussion of expressive power. The need to consider computation without order is a consequence of *data independence*, a basic principle in databases. Data independence arises from the assumption that the database provides an abstract interface which hides the internal representation of data. The abstract interface does not necessarily provide an order on the domain. However, such an order could be obtained by accessing the internal representation of data. Therefore, we view computation without order as a metaphor for the data independence principle in databases.

While there are no expressiveness results known for classes below PTIME in the absence of order, such results abound above NP. We suggest an explanation for this situation which is again based on the impact of order. We claim that the ability to capture these high complexity classes is due to the fact that order can be defined *within* these classes.

Low complexity classes of queries are most important from a practical point of view. We discuss various trade-offs which allow circumventing the inability to capture these classes by languages. Specifically, we point out trade-offs between (i) expressive power and data independence, concretized by results in the presence of order, and (ii) expressive power and determinism. The latter involves relaxing the determinism of query languages by introducing non-deterministic constructs. We show that this allows expressing "nice" classes of queries, including PTIME. However, determinism of the programs in the language can no longer be guaranteed.

The impact of order is also emphasized by a normal form which we state for the *while* queries. The normal form provides a bridge between computation without order and computation with order. It says that every *while* computation on an unordered domain can be reduced to a computation on an *ordered* domain via a *fixpoint* query. In particular, this allows to resolve the problem of the relationship of *fixpoint* and *while*: they are the same iff PTIME = PSPACE.

Intuitively, the lack of order resulting from computation with an abstract interface exacts a price in terms of complexity. We use as a running example the query *even*, which asks if a set has even or odd cardinality. We argue that queries like *even* are hard to compute without order, but become easy in the presence of order. This suggests a trade-off between abstraction and complexity, which cannot be captured by classical models such as Turing Machines. Instead, we describe a device called *Generic Machine*, which models computation with an abstract interface. Using the machine, we define robust complexity classes of queries, which differ from those of classical Complexity Theory. We outline results which formalize the trade-off between abstraction and complexity. For instance, we prove that *even* is in EXPSPACE but not in PSPACE w.r.t. the new complexity classes.

The paper provides a personal view of various issues of expressiveness of query languages, and does not constitute a comprehensive survey. There is a large body of work in the area (see the surveys [G88, F90]). The study of computable queries originated in the work of Chandra and Harel [CH80, C81, CH82]. Since then, the complexity and expressiveness of query languages, and the relationship with logic, have been widely investigated, e.g. [V82, CH85, GS85, I86, I87, C88, G88, KP88, AV88, AV89]. While we discuss many results by other authors, we focus primarily on our results from [AV88, AV89, ASV90, AV90, AV91a, AV91b].

The outline of the presentation is broadly based on [Vi89]. The paper consists of six sections. Relational languages and their relative expressiveness are discussed in Section 2. We present FO and recursive extensions based on the algebraic, logic, and logic programming paradigms. In particular, we introduce Ehrenfeucht-Fraissé games as a tool for proving non-expressibility results for FO (this portion is independent of the rest of the development). Section 3 contains a discussion on how to evaluate the complexity of query languages, and high-level remarks on trade-offs involving order and non-determinism. The impact of order is discussed in detail in Section 4, and non-deterministic languages and expressiveness results are presented in Section 5. Expressiveness above NP is discussed in Section 6, including second-order logic and the use of complex objects.

2 Relational Query Languages and Their Relative Expressiveness

In this section we present some of the better known relational query languages. We focus on relational calculus and algebra, and several recursive extensions. We consider the relative expressiveness of these languages.

2.1 Preliminaries

We review informally some terminology and notation of relational databases and query languages. A general presentation of the database field is given in [U88], and of database theory in [K91]. We assume the existence of three infinite and pairwise disjoint sets of symbols: the set **att** of *attributes*, the set **dom** of *constants*, and the set **var** of *variables*. A *relational schema* is a finite set of attributes. A *tuple* over a relational schema R is a mapping from R into **dom** $\cup$ **var**. A *constant tuple* over a relational schema R is a mapping from R into **dom**. An *instance* over a relation schema R is a *finite* set of constant tuples over R. A *database schema* is a finite set of relational schemas. An *instance* $\mathbf{I}$ over a database schema $\mathbf{R}$ is a mapping from $\mathbf{R}$ such that for each R in $\mathbf{R}$, $\mathbf{I}(R)$ is an instance over R. The set of constants occurring in an instance is called the *active domain*. The set of all instances over a schema $\mathbf{R}$ is denoted by $inst(\mathbf{R})$.

Note that, in logic terms, a database schema supplies a language consisting of a finite set of predicates, and a database instance provides an interpretation of the predicates as *finite* structures. Indeed, only finite structures are considered in this paper.

2.2 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 algebraization of FO called relational algebra that we denote here by $\mathcal{A}$ (see [U88]). It involves the following simple operations on relations: π (projection on some co-ordinates), $\times$ (cross product), $\cup$ (set union), $-$ (set difference), and $\sigma_{i=j}$ (select from a relation the tuples where the i-th and j-th co-ordinates are equal). For instance, the FO query:

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

can be algebraically computed by: $Q \cup (\pi_{1,4}(\sigma_{2=3}(R \times R)))$.

In the paper we use interchangeably FO formulas or expressions in $\mathcal{A}$, as convenient.

Expressiveness of FO: Ehrenfeucht-Fraissé Games. We briefly describe a characterization of first-order logic with finite semantics, using Ehrenfeucht-Fraissé games [F54, E61]. The games are used to prove that FO cannot express connectivity of graphs (and therefore transitive closure). This portion is independent of the rest of the development in the paper.

We first describe the game. Suppose that L is a first-order language with finitely many relation and constant symbols but no function symbols. Let A and B be two L-structures[1] with disjoint universes $\mid A \mid$ and $\mid B \mid$. Let r be a positive integer. The *game of length r associated with A and B* is played by two players, I and II, making r moves each. Player I starts by picking an element in $\mid A \mid$ or $\mid B \mid$ and player II picks an element in the opposite structure. This is repeated r times. At each move, player I has the choice of the structure, and player II must respond in the opposite structure. Let a_i (b_i) be the i^{th} element picked in $\mid A \mid$ ($\mid B \mid$). Player II wins the round $\{(a_1, b_1), ..., (a_r, b_r)\}$ iff the mapping $a_i \rightarrow b_i$ is an isomorphism of the substructures of A and B generated by $\{a_1, ..., a_r\}$ and $\{b_1, ..., b_r\}$, respectively, $A/\{a_1, ..., a_r\}$ and $B/\{b_1, ..., b_r\}$.

Player II *wins the game of length r* associated with A and B if (s)he has a winning strategy, i.e., player II can always win any game of length r on A and B, no matter how player I plays. This is denoted by $A \equiv_r B$. Note that the relation $\equiv_r$ is an equivalence relation on structures.

Intuitively, the equivalence $A \equiv_r B$ says that A and B cannot be distinguished by looking at just r constants at a time in the two structures. The main result concerning Ehrenfeucht-Fraissé games states that the ability to distinguish among structures using games of length r is equivalent to the ability to distinguish among structures using some first-order sentence of "quantifier depth" r. Quantifier depth is defined next.

[1] In database terms, A and B are instances over the relations of L.

Definition 2.1 The *quantifier depth* of a first-order sentence is the maximum number of quantifiers in a path from the root to a leaf in the representation of the sentence as a tree.

In particular, note that the quantifier depth of a sentence in prenex normal form is simply the number of quantifiers in the sentence. The following result is due to Ehrenfeucht-Fraissé [F54].

Theorem 2.2 Let K be a class of L-structures. The following statements are equivalent.

1. There is a first-order sentence ϕ of L of quantifier depth r such that :

$$K = \{A \mid A \models \phi\},$$

2. for all L-structures A and B,

$$A \in K \text{ and } A \equiv_r B \text{ implies } B \in K.$$

Suppose that K is a class of structures having some property of interest. The first statement says that the property is definable using a first-order sentence of quantifier depth r; the second states that two structures which are undistinguishable using games of length r either both have the property or neither does. In particular, the $1 \Rightarrow 2$ part of the theorem provides a technique for proving that a property K is not definable by any first-order sentence. Indeed, it is sufficient to exhibit, for each r, two structures A_r and B_r such that A_r has the property, B_r does not, and $A_r \equiv_r B_r$. Since $1 \Rightarrow 2$ is the part of the theorem of most interest in this paper, we state it in a slightly simpler form, and sketch its proof.

Proposition 2.3 Let ϕ be a sentence of L with quantifier depth r and let A and B be two L-structures. If $A \models \phi$ and $A \equiv_r B$, then $B \models \phi$.

Proof : The proof is done by a case analysis on the sentence. We only sketch it on an example. Let ϕ be the sentence : $\forall x_1 \, \exists x_2 \, \forall x_3 \; \psi(x_1, x_2, x_3)$, and let A and B be two structures such that : $A \models \phi$ and $A \equiv_3 B$. We have to show that $B \models \phi$. Suppose not. Then $B \models \exists x_1 \, \forall x_2 \, \exists x_3 \, \neg\psi(x_1, x_2, x_3)$. We will show that player I can prevent player II from winning by forcing the choice of constants a_1, a_2, a_3 in A and b_1, b_2, b_3 in B such that $A \models \psi(a_1, a_2, a_3)$ and $B \models \neg\psi(b_1, b_2, b_3)$. Then the mapping $a_i \rightarrow b_i$ cannot be an isomorphism of the substructures of A and B restricted to $\{a_1, a_2, a_3\}$ and $\{b_1, b_2, b_3\}$ respectively, contradicting the assumption that player II has a winning strategy. To force this choice, player I always picks "witnesses" corresponding to the existential quantifiers in ψ and $\neg\psi$. Player I starts by picking an element b_1 in $\mid B \mid$ such that : $B \models \forall x_2 \, \exists x_3 \, \neg\psi(b_1, x_2, x_3)$. Player II must respond by picking an element a_1 in $\mid A \mid$. Due to the universal quantification in ϕ, $A \models \exists x_2 \, \forall x_3 \; \psi(a_1, x_2, x_3)$ regardless of which a_1 was picked.

Next, player I picks an element a_2 in $\mid A \mid$ such that : $A \models \forall x_3 \; \psi(a_1, a_2, x_3)$. Regardless of which element b_2 in $\mid B \mid$ that player II picks, $B \models \exists x_3 \neg\psi(b_1, b_2, x_3)$. Finally, player I picks b_3 in $\mid B \mid$ such that $B \models \neg\psi(b_1, b_2, b_3)$; player II picks some a_3 in $\mid A \mid$, and $A \models \psi(a_1, a_2, a_3)$. □

The next example shows how Proposition 2.3 can be used to prove that graph connectivity, and therefore transitive closure, are not first-order definable.

Example 2.4 For each r, we exhibit a connected graph A and a disconnected graph B such that $A \equiv_r B$. For a sufficiently large n (depending only on r), A consists of a cycle of $2n$ nodes and B of two disjoint cycles B_1 and B_2 of n nodes each. We outline the winning strategy for player II. If player I picks an element a_1 in A then player II picks an arbitrary element b_1, say in B_1. Now, if player I picks an element b_2 in B_2, then player II picks an element a_2 in A far from a_1. Next, if player I picks a b_3 in B_1 close to b_1, then player II picks an element a_3 in A close to a_1. The graphs are sufficiently large that this can proceed for r moves with the resulting subgraphs isomorphic. By Proposition 2.3, we conclude that no FO sentence can define connectivity of graphs.

2.3 Recursive Extensions: Fixpoint, While, Datalog$^\neg$

We have seen that FO has limited expressive power; for example, it cannot compute the transitive closure of a graph. This is due to the lack of recursion in the language. In the next three sections, we consider several extensions of FO with recursion.

The integration of recursion and negation is very natural and yields highly expressive languages. We will see how it can be achieved in the algebraic, logical, and deductive paradigms. The algebraic language is an extension of $\mathcal{A}$ with a looping construct, in the style of traditional imperative programming languages. The logic language is an extension of the calculus where recursion is provided by a fixpoint operator. The deductive language consists of an extension of "Datalog" with negation.

As we consider more and more powerful languages, the complexity of evaluating the queries is of increasing concern. We will consider two flavors of the languages in each paradigm: one that guarantees termination in time polynomial in the size of the database, and a second which only guarantees that a polynomial amount of space is used. We will also show that the polynomial-time bounded languages defined in the different paradigms are equivalent. The set of queries they define is called the *fixpoint queries*. The polynomial-space bounded languages are also equivalent, and the corresponding set of queries is called the *while* queries. In Section 3, we will examine in more detail the expressiveness and complexity of the *fixpoint* and *while* queries.

Before we describe the specific languages, it is useful to understand the idea underlying the two flavors of the languages, "inflationary" and "non-inflationary". All languages we consider use a fixed set of relation schemas throughout the computation. At any point in the computation, intermediate results contain only constants from the input database or specified in the query. Suppose that the relations used in the computation have arities $r_1, ..., r_k$, the input database contains n constants, and the query refers to c constants. Then the number of tuples in any intermediate result is bounded by $\sum_{i=1}^{k}(n+c)^{r_i}$, which is a polynomial in n. Thus, such queries can be evaluated in polynomial space. Suppose we wish to force termination in polynomial time. The standard way to achieve this in the languages we consider is the following. The semantics of the language is such that a tuple cannot be deleted from a relation once it is inserted. Thus, space usage is *increasing* throughout the computation. Furthermore, the semantics ensures that programs do not work vacuously without adding tuples. Since there are only polynomially many tuples, the program terminates in polynomial time. This type of semantics is referred to as *inflationary* (there is an inflation of tuples!). In contrast, the unrestricted semantics, which does not ensure the continuous growth of the set of tuples, is called *non-inflationary*.

In summary, inflationary languages in the three paradigms express the *fixpoint* queries and terminate in polynomial time. Non-inflationary languages express the *while* queries and use a polynomial amount of space.

2.4 Algebra + *while*

Relational algebra is essentially a procedural language. Of the query languages, it is the closest to traditional imperative programming languages. The extensions of the algebra with recursion also follow the procedural, imperative paradigm. They provide: (i) relational variables (P, Q, R...) which can hold relations of specified sorts, (ii) assignment of relational algebra expressions to relational variables, and (iii) a *while* construct allowing to iterate a program while some condition holds.

The resulting language comes in two flavors: inflationary and non-inflationary. The two versions of the language differ in the semantics of the assignment statement and the termination condition for loops. The non-inflationary version was the one first defined historically, and we discuss it next. The resulting language is called the *while* language.

We next discuss the assignment statement and while construct. The assignment statement is of the form

$$P := e$$

where e is an algebra expression and P a relational variable of the same sort as the result of e. In the *while* language, the semantics of an assignment statement is: the value of P becomes the result of evaluating the algebra expression e on the current state of the database. This is consistent with the standard "destructive" assignment in imperative programming languages, where the old value of a variable is overwritten in an assignment statement.

While loops are of the form

$$\begin{array}{l} \textit{while}\ \langle \textit{condition} \rangle\ \textit{do} \\ \qquad \textit{begin}\ \ \langle \textit{loop body} \rangle\ \ \textit{end.} \end{array}$$

Termination conditions for *while* loops are tests of emptiness of the form $e = \emptyset$ or $e \neq \emptyset$, where e is a relational algebra expression. The body of the loop is executed as long as the condition is satisfied. A *while* program is a finite sequence of assignment or while statements. The program uses a finite set of relational variables of specified sorts, including the names of relations in the input database. A designated relational variable holds the output to the query at the end of the computation. One can view a *while* program as defining one query for each of its relational variables.

Example 2.5 *Transitive Closure.* Consider a binary relation G, specifying the edges of a graph. The following *while* program computes in T the transitive closure of G. Here, T and *oldT* are also binary.

$$\begin{array}{l} \textit{oldT} := \emptyset; \\ T := G \\ \textit{while}\ T\ - \textit{oldT}\ \neq\ \emptyset\ \textit{do} \\ \qquad \textit{begin} \\ \qquad \textit{oldT}\ :=\ T; \\ \qquad T\ :=\ T \cup \pi_{1,4}(\sigma_{2=3}(T \times G)); \\ \qquad \textit{end.} \end{array}$$

In the program, *oldT* keeps track of the value of T resulting from the previous iteration of the loop. The computation ends when *oldT* and T coincide, which means that no new edges were added in the current iteration, so T holds the complete transitive closure.

Example 2.6 *Add-Remove.* Consider again a binary relation G specifying the edges of a graph. The following program removes from G all edges (a,b) if there is a path of length 2 from a to b, and inserts an edge (a,b) if there is a vertex not directly connected to a and b; this is iterated while some change occurs. The result is in the binary relation T. In addition, the binary relation variables P, Q, $oldG$ are also used. For the sake of readability, we use the calculus whenever this is easier to understand than the corresponding algebra expression. The semantics in the calculus is the active domain semantics (with respect to the input).

$$
\begin{array}{l}
T := G; oldT := \emptyset; \\
while\ (oldT - T) \cup (T - oldT) \neq \emptyset\ do \\
\qquad begin \\
\qquad P := \{\langle x,y\rangle \mid \exists z(T(x,z) \wedge T(z,y))\}; \\
\qquad Q := \{\langle x,y\rangle \mid \exists z(\neg T(x,z) \wedge \neg T(z,x) \wedge \neg T(y,z) \wedge \neg T(z,y))\}; \\
\qquad oldT := T; \quad T := (T \cup Q) - P; \\
\qquad end.
\end{array}
$$

Like in the previous example, $oldT$ is used to detect when no change occurs as a result of executing the loop. The test $(oldT - T) \cup (T - oldT) \neq \emptyset$ could have been replaced with the comparison $oldT \neq T$, if such comparisons were allowed in the language.

In the example, the transitive closure query always terminates. This is not the case for the *Add-Remove* query. Indeed, *while* programs may not terminate. This is typically the case for non-inflationary languages. We define next an inflationary version of the *while* language, which we denote by $while^{inf}$. The $while^{inf}$ language differs from *while* in the semantics of the assignment statement and the termination conditions for loops. The assignment statement in $while^{inf}$ is non-destructive. Consider an assignment of an algebra expression e to variable P. The semantics is now cumulative; the value of P after the assignment is obtained by *adding* to the old value of P the result of e. Thus, no tuple is removed from any relation throughout the execution of the program. To distinguish the cumulative semantics from the destructive one, we use the notation $P \mathrel{+}= e$ for the cumulative semantics.

While loops are modified as follows. There is no explicit termination condition. Instead, a loop runs as long as the execution of the body causes some change to some relation. For readability, the syntax of loops becomes *while change do ...* . As seen from the previous examples, the "no change" semantics is often natural. Clearly, both the cumulative assignment and the no change semantics for while loops can be simulated in *while*. Thus, the set of queries expressible in $while^{inf}$ is a subset of those expressible in *while*.

Example 2.7 *Transitive Closure Revisited.* Following is a $while^{inf}$ program computing the transitive closure of a graph represented by a binary relation G. The result is obtained in the variable T.

$$
\begin{array}{l}
T \mathrel{+}= G; \\
while\ change\ do \\
\qquad begin \\
\qquad T \mathrel{+}= \pi_{1,4}(\sigma_{2=3}(T \times G)); \\
\qquad end.
\end{array}
$$

Note that the variable *oldT* used in Example 2.5 is no longer needed. Also, it is no longer necessary to explicitly add to T its old value, since this is taken care of by the cumulative semantics of the assignment statement.

2.5 Calculus+Fixpoint

As for the algebra, we provide inflationary and non-inflationary extensions of the calculus with recursion. This could be done using assignment statements and while loops, as for the algebra. Indeed, we used calculus notation in Example 2.6 (*Add-Remove*). However, an equivalent but more logic-oriented construct is used to augment the calculus. The construct, called a *fixpoint operator*, allows the iteration of calculus formulas up to a fixpoint. This in effect allows defining relations inductively, using calculus formulas. To illustrate this, consider again the transitive closure of a graph G. The relations T_n holding pairs of nodes at distance at most n can be defined inductively using a single formula

$$\phi(T) = G(x,y) \ \vee \ T(x,y) \ \vee \ \exists \, z(T(x,z) \wedge G(z,y))$$

as follows:

$$\begin{aligned} T_0 &= \emptyset; \\ T_n &= \phi(T_{n-1}), \ n > 0. \end{aligned}$$

Here $\phi(T_{n-1})$ denotes the result of evaluating $\phi(T)$ when the value of T is T_{n-1}. Note that, for any given G, the sequence $\{T_n\}_{n\geq 0}$ converges, i.e. there exists some k for which $T_k = T_j$ for every $j > k$ (indeed, k is the diameter of the graph). Clearly, T_k holds the transitive closure of the graph. Thus, the transitive closure T of G can be defined as the limit of the above sequence. Note that $T_k = \phi(T_k)$, so T_k is also a *fixpoint* of $\phi(T)$. The relation T_k is denoted by $\mu_T(\phi(T))$. Then the transitive closure of G is defined by

$$\mu_T(\ G(x,y) \ \vee T(x,y) \ \vee \ \exists \, z(T(x,z) \wedge G(z,y))).$$

Thus, μ_T is an operator which produces a new relation (the fixpoint T_k) when applied to $\phi(T)$. Note that, although T is used in $\phi(T)$, T is not a database relation, but rather a relation used to define inductively $\mu_T(\phi(T))$ from the database, starting with $T = \emptyset$. T is said to be *bound* to μ_T. Indeed, μ_T is somewhat similar to a quantifier over relations.

In the above example, the limit of the sequence $\{T_n\}_{n\geq 0}$ happens to exist. This is not always the case. Indeed, for

$$\phi(T) = (x = 0 \wedge \neg T(0) \wedge \neg T(1)) \vee (x = 0 \wedge T(1)) \vee (x = 1 \wedge T(0))$$

the sequence $\{T_n\}_{n\geq 0}$ is $\{\langle 0\rangle\}, \{\langle 1\rangle\}, \{\langle 0\rangle\}, \ldots$, i.e. T flip-flops between zero and one. The sequence does not converge, so $\mu_T(\phi(T))$ is not defined. Thus, the operator μ is defined for some formulas and databases, and undefined for others. Situations where μ is undefined correspond to non-terminating computations in the *while* language. Following is a non-terminating *while* program corresponding to $\mu_T(\phi(T))$ above:

```
oldT := ∅;
T := {⟨0⟩};
while (oldT − T) ∪ (T − oldT) ≠ ∅ do
        begin
        oldT := T;
        T := {⟨0⟩, ⟨1⟩} − T;
        end.
```

Since μ is only partially defined, it is called a *partial fixpoint operator.* We define its syntax and semantics in more detail next.

Partial Fixpoint Operator. Let $\mathbf{R}$ be a database schema, and T a relation schema not in $\mathbf{R}$, of arity m. Let $\mathbf{S}$ denote the schema $\mathbf{R} \cup \{T\}$. Let $\phi(T)$ be a relational calculus formula using T and relations in $\mathbf{R}$, with m free variables. Given an instance $\mathbf{I}$ over $\mathbf{R}$, $\mu_T(\phi(T))$ denotes the relation which is the limit, *if it exists*, of the sequence $\{T_n\}_{n\geq 0}$ defined by:

$$\begin{aligned} T_0 &= \emptyset; \\ T_n &= \phi(T_{n-1}),\ n > 0, \end{aligned}$$

where $\phi(T_{n-1})$ denotes the result of evaluating ϕ on the instance $\mathbf{J}$ over $\mathbf{S}$ whose restriction to $\mathbf{R}$ is $\mathbf{I}$ and $\mathbf{J}(T) = T_{n-1}$.

Thus, $\mu_T(\phi(T))$ denotes a new relation (if it is defined). In turn, it can be used in more complex formulas like any other relation. For example, $\mu_T(\phi(T))(y, z)$ states that $\langle y, z\rangle$ is in $\mu_T(\phi(T))$. If $\mu_T(\phi(T))$ defines the transitive closure of G, the complement of the transitive closure is defined by

$$\{\langle x, y\rangle \mid \neg\ \mu_T(\phi(T))(x, y)\}.$$

The extension of the calculus with μ is called *partial fixpoint logic*, denoted $FO+\mu$.

Partial Fixpoint Logic. *$FO+\mu$ formulas* are obtained by repeated applications of FO operators $(\exists, \forall, \vee, \wedge, \neg)$ and the partial fixpoint operator, starting from atoms. In particular, $\mu_T(\phi(T))(x_1, ..., x_n)$, where T has arity n and $\phi(T)$ has n free variables, and the x_i are variables or constants, is a formula. Its free variables are the variables in the set $\{x_1, ..., x_n\}$ (thus, the variables occurring inside $\phi(T)$ are not free in this formula). Partial fixpoint operators can be nested. *$FO+\mu$ queries* over a database schema $\mathbf{R}$ are expressions of the form

$$\{\langle x_1, ..., x_n\rangle \mid \xi(x_1, ..., x_n)\},$$

where $\xi(x_1, ..., x_n)$ is a $FO+\mu$ formula with free variables $x_1, ..., x_n$. The formula ξ may use relation names in addition to those in $\mathbf{R}$; however, each occurrence P of such relation name must be bound to some partial fixpoint operator μ_P. The semantics of $FO+\mu$ queries is defined as follows. First, note that, given an instance $\mathbf{I}$ over $\mathbf{R}$ and a sentence σ in $FO+\mu$, there are three possibilities: σ is undefined on $\mathbf{I}$; σ is defined on $\mathbf{I}$ and is true; and, σ is defined on $\mathbf{I}$ and is false. Given an instance $\mathbf{I}$ over $\mathbf{R}$, the answer to the query

$$\{\langle x_1, ..., x_n\rangle \mid \xi(x_1, ..., x_n)\}$$

is the n-ary relation consisting of all valuations ν of $x_1, ..., x_n$ for which the formula $\xi(\nu(x_1), ..., \nu(x_n))$ is defined and true. The queries expressible in Partial Fixpoint Logic are called the *partial fixpoint queries*. We will see that they are equivalent to the *while* queries.

Example 2.8 *Add-Remove Revisited.* Consider again the query in Example 2.6. To express the query in $FO+\mu$, a difficulty arises; the *while* program initializes T to G before the while loop, whereas $FO+\mu$ lacks the capability to do this directly. To distinguish the initialization step from the subsequent ones, we use a ternary relation Q and two distinct constants: 0 and 1. To indicate that the first step has been performed, we insert in Q the tuple $\langle 1, 1, 1\rangle$. The presence of $\langle 1, 1, 1\rangle$ in Q inhibits the repetition of the first step. Subsequently, an edge $\langle x, y\rangle$ is encoded in Q as $\langle x, y, 0\rangle$. The *while* program in Example 2.6 is equivalent to the $FO+\mu$ query

$$\{\langle x, y\rangle \mid \mu_Q(\phi(Q))(x, y, 0)\}$$

where

$$\begin{aligned} \phi(Q) \quad = \quad & \neg Q(1,1,1) \wedge ((G(x,y) \wedge z = 0) \vee (x = 1 \wedge y = 1 \wedge z = 1)) \\ & \vee Q(1,1,1) \wedge ((x = 1 \wedge y = 1 \wedge z = 1) \\ & \quad \vee((z = 0) \wedge Q(x,y,0) \wedge \neg\exists w(Q(x,w,0) \wedge Q(w,y,0))) \\ & \quad \vee\ ((z = 0) \wedge \exists w(\neg Q(x,w,0) \wedge \neg Q(w,x,0) \\ & \qquad \wedge \neg Q(y,w,0) \wedge \neg Q(w,y,0)))). \end{aligned}$$

Clearly, this query is more awkward than its counterpart in *while*. The simulation highlights some peculiarities of computing with $FO+\mu$.

As seen, the non-convergence of the sequence $\{T_n\}_{n\geq 0}$ in some cases is similar to non-terminating computations in the *while* language with non-inflationary semantics. The semantics of the partial fixpoint operator μ can also be viewed as non-inflationary. Indeed, in the inductive definition of the T_n, each step is a destructive assignment. As for *while*, we can make the semantics inflationary by having the assignment at each step of the induction be cumulative. This will yield an inflationary version of μ which is defined for all formulas and databases to which it is applied. The inflationary version of μ is denoted by μ^{inf}. It is referred to simply as an *inflationary fixpoint operator.*

Making the fixpoint operator inflationary by definition is not the only way to guarantee polynomial-time termination of the fixpoint iteration. An alternative approach (and historically first) is to restrict the *formulas* $\phi(T)$ so that convergence of the sequence $\{T_n\}_{n\geq 0}$ associated with $\mu_T(\phi(T))$ is guaranteed. This can be done by requiring that T occur only *positively* in $\phi(T)$, i.e. under an even number of negations in the syntax tree of the formula. With this requirement, it can be easily checked that the partial fixpoint semantics yields an increasing sequence $\{T_n\}_{n\geq 0}$, and $\mu_T(\phi(T))$ is always defined. It can be shown that the two approaches are equivalent, i.e. the sets of queries expressed are identical.

We describe the syntax and semantics of μ^{inf} next.

Inflationary Fixpoint Operators. The definition of $\mu_T^{inf}(\phi(T))$ is identical to that of the partial fixpoint operator except that the sequence $\{T_n\}_{n\geq 0}$ is defined as follows:

$$\begin{array}{lcl} T_0 & = & \emptyset; \\ T_n & = & T_{n-1} \cup \phi(T_{n-1}),\ \ n > 0. \end{array}$$

This definition ensures that the sequence $\{T_n\}_{n\geq 0}$ is increasing: $T_{i-1} \subseteq T_i$ for each $i > 0$. Since for each instance there are finitely many tuples which can be added, the sequence converges in all cases.

Adding μ^{inf} instead of μ to FO yields *inflationary fixpoint logic*, denoted by $FO+\mu^{inf}$.

Inflationary Fixpoint Logic. The syntax of inflationary fixpoint logic is identical to that of the partial fixpoint logic, except that μ^{inf} is used instead of μ. Note that inflationary fixpoint queries are always defined.

For example, the transitive closure of a graph G is defined by the $FO+\mu^{inf}$ query

$$\{\langle x, y\rangle \mid \mu_T^{inf}(G(x,y) \vee \exists z(T(x,z) \wedge G(z,y))(x,y)\}.$$

The set of queries expressible by inflationary fixpoint logic is called the *fixpoint queries.* The fixpoint queries were historically defined first among the inflationary languages in the algebraic, logic, and deductive paradigms. Therefore, the class of queries expressible in inflationary languages in the three paradigms has come to be referred to as the *fixpoint* queries.

2.6 Datalog and Datalog$^{\neg}$

A third paradigm for query languages, besides the logic and algebraic, is based on logic programming. We consider first Datalog, which can be viewed as a "pure" relational version of Prolog. The syntax of Datalog is essentially that of Horn Clauses. A *Datalog rule* is an expression of the form :

$$R_1(u_1) \leftarrow R_2(u_2), ..., R_n(u_n)$$

where for some $n \geq 1$, $R_1, ..., R_n$ are relation names, $u_1, ..., u_n$ are tuples of variables or constants of appropriate arities. Furthermore, each variable occurring in the head must also occur in the body. A *Datalog program* is a finite set of Datalog rules.

In a Datalog program P, $sch(P)$ denotes the database schema consisting of all relations involved in the program P. The relations occurring in heads of rules are the *intensional* relations (idb) of P, and the others are the *extensional* relations (edb) of P.

Datalog can be given a model-based semantics, or a fixpoint semantics. The two semantics turn out to be equivalent. With the model-based semantics, the result of a Datalog program is the unique minimal model satisfying the sentences corresponding to the rules in the program, and containing the input database. The fixpoint semantics consists of firing the rules of the program with all applicable valuations until a fixpoint is reached. We illustrate this with an example.

Example 2.9 The following Datalog program computes the transitive closure of the graph G in T:

$$\begin{array}{lcl} T(x,y) & \leftarrow & G(x,y) \\ T(x,y) & \leftarrow & T(x,z), G(z,y) \end{array}$$

Datalog provides recursion, but no negation. In particular, it is incomparable to FO. Indeed, it only defines monotonic queries. The precise connection with FO is investigated in [AG89]. It is shown that a Datalog program defines a FO query iff it is bounded, i.e. it terminates after a *constant* number of iterations of firing the rules (the "only if" part is quite surprising!). The precise expressive power of Datalog is not easy to characterize. While Datalog defines only monotonic queries in PTIME, it has been shown that there are such queries that Datalog cannot express, even with order and inequality [ACY91].

Viewed from the standpoint of the deductive paradigm, Datalog provides a form of *monotonic reasoning*. Adding negation to Datalog rules allows defining non-monotonic queries, and performing *non-monotonic reasoning*.

Adding negation in Datalog rules requires defining semantics for negative facts. This can be done in many ways. The different definitions depend to some extent on whether Datalog is viewed in the deductive framework, or simply as a specification formalism like any other query language. Indeed, Datalog with negation can essentially be viewed as a subset of the *while* or *fixpoint* queries, and treated similarly. This is not necessarily appropriate in the deductive framework. For instance, the basic assumptions on the reasoning process may require that, once a fact is assumed false at some point in the inferencing process, it should not be proven true at a later point. The issue of a "natural" semantics of negation in a deductive context is not considered here.

As usual, we will consider inflationary and non-inflationary versions of Datalog with negation. We start with the inflationary version, which is most commonly used. The inflationary language allows negations in bodies of rules, and is denoted by $Datalog^{\neg}$. Like Datalog, its rules are used to infer a set of facts. Once a fact is inferred, it is never removed from the set of true facts. This yields the inflationary character of the language. To illustrate this, consider the following Datalog$^{\neg}$ program. Its input is a graph G. The program computes the relation $closer(x,y,x',y')$ defined as follows: $closer(x,y,x',y')$ means that the distance $d(x,y)$ from x to y in G is smaller than the distance $d(x', y')$ from x' to y'.

$$\begin{array}{lcl} T(x,y) & \leftarrow & G(x,y) \\ T'(x,y) & \leftarrow & T(x,y) \\ T(x,y) & \leftarrow & T(x,z), G(z,y) \\ closer(x,y,x',y') & \leftarrow & T'(x,y), T(x',y'), \neg T'(x',y'). \end{array}$$

The program is evaluated as follows. The rules are fired simultaneously with all applicable valuations. At each such firing, some facts are inferred. This is repeated until no new facts are inferred, or the rules can no longer be fired. A negative fact such as $\neg T'(x', y')$ is true if $T'(x', y')$ has not been inferred so far. This does not preclude $T'(x', y')$ from being inferred at a later firing of the rules. One firing of the rules is called a *stage* in the evaluation of the program. In the above program, T and T' hold the transitive closure of G. Consider the consecutive stages in the evaluation of the program. Note that, if the fact $T(x, y)$ is inferred at stage n, then $d(x, y) = n$. On the other hand, T' is one step behind T, i.e. $T'(x, y)$ is inferred one stage later than $T(x, y)$. Thus, if $T'(x, y), T(x', y'), \neg T'(x', y')$ holds at some stage n, then $d(x, y) < n$ and $d(x', y') = n$. By the fourth rule, $closer(x, y, x', y')$ is then inferred.

Syntax and Semantics of Datalog$^\neg$. The formal syntax and semantics of Datalog$^\neg$ are straightforward extensions of those for Datalog. A Datalog$^\neg$ rule is an expression of the form

$$H(h) \leftarrow B_1(u_1), ..., B_n(u_n),$$

where: (i) H is a relation name, (ii) B_i are relation names (in which case B_i is called *positive*) or $\neg R_i$ where R_i is a relation name (then B_i is called *negative*), and (iii) h and u_i are tuples of variables or constants of appropriate arities. A Datalog$^\neg$ program is a finite set of Datalog$^\neg$ rules.

The semantics of Datalog$^\neg$ is as follows. Let $\mathbf{K}$ be an instance over $sch(P)$. A fact $R(r)$ is an *immediate consequence* for $\mathbf{K}$ and P if $R(r) \leftarrow A_1, ..., A_n$ is an instantiation of a rule in P and: each positive A_i is a fact in $\mathbf{K}$, and (ii) if $A_i = \neg A_i'$ then $A_i' \notin \mathbf{K}$. Instantiations use valuations into the active domain with respect to the input idb relations. The *immediate consequence operator* of P, denoted Γ_P, is now defined in the same way as for Datalog. Given an instance $\mathbf{I}$ over $edb(P)$, one can compute $\Gamma_P(\mathbf{I}), \Gamma_P^2(\mathbf{I}), \Gamma_P^3(\mathbf{I})$, etc. As for Datalog, the sequence reaches a fixpoint, denoted $\Gamma_P^\infty(\mathbf{I})$, after a finite number of steps.

An important difference with Datalog is that $\Gamma_P^\infty(\mathbf{I})$ is no longer guaranteed to be a minimal model of P containing $\mathbf{I}$. This is easily seen by the following example.

Example 2.10 Let P be the program

$$\begin{array}{lcl} R(0) & \leftarrow & Q(0), \neg S(0) \\ S(0) & \leftarrow & Q(0), \neg R(0). \end{array}$$

Let $edb(P) = \{Q\}$ and $idb(P) = \{R, S\}$. Let $\mathbf{I}$ be the instance over Q: $\{\langle 0 \rangle\}$. Then $P(\mathbf{I}) = \{Q(0), R(0), S(0)\}$. While $P(\mathbf{I})$ is a model of P, it is not minimal. Indeed, the minimal models containing $\mathbf{I}$ are $\{Q(0), R(0)\}$ and $\{Q(0), S(0)\}$.

The above shows a divergence between Datalog and Datalog$^\neg$ with respect to the model theoretic semantics. This can be viewed as a drawback of the fixpoint semantics for Datalog$^\neg$. However, a natural model theoretic semantics is not easy to formulate for Datalog$^\neg$. As seen in Example 2.10, there are Datalog$^\neg$ programs which simply do not have a unique minimal model which can be taken as the semantics. In such cases, a model-based semantics requires choosing a natural model among several possible candidates. Inevitably, such choices are open to debate. We do not elaborate on the model-based semantics in this paper. However, it is important to note that some of the most widely accepted model-based semantics yield the same expressive power as the inflationary semantics described here. Such a semantics is the *well-founded* semantics of Van Gelder, Ross, and Schlipf [VRS88]. However, another popular semantics for negation, *stratified negation* [CH85, ABW86, N86, VG86], is strictly weaker than the *fixpoint* queries. This was shown by [K87] using a result by [D87]. Intuitively, it is due to the fact that the stratified semantics, unlike the inflationary semantics, bounds the number of applications of the negation by disallowing recursion through negation.

Datalog$^{\neg*}$. The language Datalog$^\neg$ has inflationary semantics. This is because the set of facts inferred through the consecutive firings of the rules is increasing. To obtain a non-inflationary variant, there are several possibilities. One could keep the syntax of Datalog$^\neg$ but make the semantics non-inflationary by retaining, at each stage, only the newly inferred facts. Another possibility is to allow explicit retraction of a previously inferred fact. Syntactically, this can be done using negations in heads of rules, interpreted as deletions of facts. We adopt this solution here, since it is more common in practical languages such as those used in production systems. The resulting language is denoted by Datalog$^{\neg*}$, to indicate that negations are allowed in both heads and bodies of rules.

Example 2.11 *Add-Remove Revisited.* The following Datalog$^{\neg*}$ program computes in T the *add-remove* query of Example 2.6, given as input a graph G.

$$\begin{array}{lcl} T(x,y) & \leftarrow & G(x,y), \neg \mathit{off}(1) \\ \mathit{off}(1) & \leftarrow & \\ \neg T(x,y) & \leftarrow & T(x,z), T(z,y), \mathit{off}(1) \\ T(x,y) & \leftarrow & \neg T(x,z) \neg T(z,x) \neg T(y,z) \neg T(z,y), \mathit{off}(1). \end{array}$$

The result of the query is produced in T. Relation *off* is used to inhibit the first rule (initializing T to G) after the first step.

The semantics of negations in heads of rules is the following. If $\neg R(x,y)$ is inferred, the fact $R(x,y)$ is removed, unless $R(x,y)$ is also inferred in the same firing of the rules. This gives priority to inference of positive over negative facts, and is somewhat arbitrary. Other possibilities are: (i) giving priority to negative facts, and (ii) interpreting the inference of $R(a,b)$ and $\neg R(a,b)$ simultaneously as a contradiction which makes the result undefined. The chosen semantics has the advantage over (ii) that the semantics is always defined. In any case, the choice of semantics is not crucial: they are essentially equivalent.

With the above semantics, termination is no longer guaranteed. For instance, the program

$$\begin{array}{lcl} T(0) & \leftarrow & T(1) \\ \neg T(1) & \leftarrow & T(1) \\ T(1) & \leftarrow & T(0) \\ \neg T(0) & \leftarrow & T(0). \end{array}$$

never terminates on input $T(0)$. Indeed, the value of T flip-flops between $\{\langle 0\rangle\}$ and $\{\langle 1\rangle\}$ so no fixpoint is reached.

2.7 Equivalence

We introduced inflationary and non-inflationary recursive languages with negation in the algebraic, logic, and deductive paradigms. We show here that the inflationary languages in the three paradigms,

$while^{inf}$, $FO+\mu^{inf}$, and Datalog$^\neg$, are equivalent. The same holds for the non-inflationary languages *while*, $FO+\mu$, and Datalog$^{\neg\star}$. This yields two classes of queries which are central in the theory of query languages: the *fixpoint* queries (expressed by the inflationary languages), and the *while* queries (expressed by the non-inflationary languages).

We begin with the equivalence of the inflationary languages.

Theorem 2.12 $FO+\mu^{inf}$, $while^{inf}$, and Datalog$^\neg$ are equivalent.

The equivalence of $FO+\mu^{inf}$ and $while^{inf}$ is easy. Indeed, the languages have similar capabilities; program composition in $while^{inf}$ corresponds closely to formula composition in $FO+\mu^{inf}$, and the "*while change*" loop of $while^{inf}$ is close to the inflationary fixpoint operator of $FO+\mu^{inf}$. More difficult is the equivalence of these languages with Datalog$^\neg$, since this much simpler language has no explicit constructs for program composition or nested recursion. Thus, the non-trivial part of the equivalence proof is showing that Datalog$^\neg$ can simulate these constructs. The following example illustrates some techniques used in the simulation.

Example 2.13 The following Datalog$^\neg$ program computes the complement of the transitive closure of a graph G. The example illustrates the technique used to delay the firing of a rule (computing the complement) until the fixpoint of a set of rules (computing the transitive closure) has been reached. The idea is that, as the transitive closure of G is computed in relation T, the result is duplicated in relations *previous* and *previous-unless-last*, but with a delay of one iteration. However, the result of the previous iteration is copied in *previous-unless-last*, only if the current iteration is not the last. Thus, *previous* and *previous-unless-last* differ only at the last iteration, which allows recognizing when the fixpoint has been reached, and firing the rule computing the complement in relation $\overline{T}$. The program is:

$$
\begin{array}{lcl}
T(x,y) & \leftarrow & G(x,y) \\
T(x,y) & \leftarrow & G(x,z), T(z,y) \\
previous(x,y) & \leftarrow & T(x,y) \\
previous\text{-}unless\text{-}last(x,y) & \leftarrow & T(x,y), G(x',z'), T(z',y'), \neg T(x',y') \\
 & & \neg T(x,y), previous(x',y'), \\
\overline{T}(x,y) & \leftarrow & \neg previous\text{-}unless\text{-}last(x',y').
\end{array}
$$

(It is assumed that G is not empty.)

An analogous equivalence result can be proven for the non-inflationary languages *while*, $FO+\mu$, and Datalog$^{\neg\star}$. The proof of the equivalence of $FO+\mu$ and Datalog$^{\neg\star}$ is easier than in the inflationary case because the ability to perform deletions in Datalog$^{\neg\star}$ facilitates the task of simulating explicit control. Thus we can prove:

Theorem 2.14 *while*, $FO+\mu$, and Datalog$^{\neg\star}$ are equivalent.

The set of queries expressible in *while*, $FO+\mu$, and Datalog$^{\neg\star}$ is called the *while queries*.

Normal Forms. The equivalence results have some interesting side-effects. Consider the inflationary languages. We showed the rather surprising fact that $FO+\mu^{inf}$ is equivalent to Datalog$^\neg$, which is essentially a fragment of $FO+\mu^{inf}$, much simpler than the full language. This yields a normal form for $FO+\mu^{inf}$ queries: each $FO+\mu^{inf}$ query can be expressed using a single application of the fixpoint operator on an existentially quantified formula (in prenex form). The same normal form holds for $FO+\mu$, and analogous normal forms can be shown for $while^{inf}$ and *while*.

2.8 While versus Fixpoint

The problem of the relationship of the *fixpoint* and *while* query is a difficult one. However, it seems very likely that *while* is strictly stronger than *fixpoint*. Indeed, the following was shown in [AV91a, AV91b].

Theorem 2.15 *fixpoint* = *while* iff PTIME = PSPACE.

The proof is based on Theorem 4.1 and on a normal form for *while*, both stated in Section 4.2.

Surprisingly, the result also applies to the PTIME fragment of *while* (i.e., $while|_{PTIME}$): it is shown in [AV91a, AV91b] 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.

2.9 Breaking the Space Barrier

The languages described so far have bounded complexity. Indeed, any language using (i) a finite set of fixed-arity relations and (ii) only constants from the input, stays within PSPACE. We next consider briefly languages with unbounded complexity, which express all queries. To break the space barrier, (i) or (ii) must be relaxed. The first such 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 arities are 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 constants in the database. For instance, for a binary relation R, the following program creates a new constant for each pair of tuples in R using two variables X (4-ary) and Y (5-ary):

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

2.10 Bibliographic Notes on Languages

The *while* language was first introduced as RQ in [CH82] and as LE in [C81]. The other non-inflationary languages, $FO{+}\mu$ and $\text{Datalog}^{\neg\star}$, were defined in [AV88, AV89]. The equivalence of the non-inflationary languages was shown in [AV88, AV90]. The inflationary languages have a long history. Logics with fixpoints have been considered by logicians in the general case where infinite structures (corresponding to infinite database instances) are allowed [M74]. In the finite case, relevant here, the fixpoint queries were first defined using the partial fixpoint operator μ_T applied only to formulas positive in T [CH82]. The language allowing applications of μ_T to formulas monotonic, but not necessarily positive, in T, was further studied in [G84]. The two languages were shown equivalent in [GS86], where it was also proven that they are equivalent to $FO{+}\mu^{inf}$. As a side-effect, it was shown in [GS86] that the nesting of μ (or μ^{inf}) provides no additional power. This fact had been proven earlier for the first language in [I86]. Of the other inflationary languages, $while^{inf}$ was defined in [AV90] and $\text{Datalog}^{\neg}$ with fixpoint semantics was first defined in [AV88, KP88]. The equivalence of $\text{Datalog}^{\neg}$ with $FO{+}\mu^{inf}$ and $while^{inf}$ was shown in [AV88]. The relationship between the *while* and *fixpoint* queries was investigated in [AV91a, AV91b], where it was shown that they are equivalent iff PTIME = PSPACE.

The normal forms discussed here can be viewed as variations of well-known "folk theorems", described in [H80]. The deletion ability in Datalog$^{\neg\star}$ can be viewed as an updating capability. Other mechanisms to perform updates using a logic programming based formalism are described in [NK88, MW88].

3 Sizing Up Languages

In the previous section we presented a variety of languages which converge around three important classes of queries: *FO*, *fixpoint*, and *while*. Clearly, many extensions of these languages can be defined, all the way to *complete* languages, which can define all queries. It is relatively easy to define very powerful languages, as seen in Section 2.9. However, the real challenge for the language designer lies not simply in defining more and more powerful languages. Instead, an important aspect of language design is to achieve a good balance between expressiveness and the complexity of evaluating queries in the language. The ideal language would allow expressing most "useful" queries, while at the same time providing reasonable complexity bounds for *all* queries expressible in the language. We will discuss the expressiveness and complexity of query languages in detail. In this section, we begin by answering the following question: how does one evaluate a query language w.r.t. expressiveness and complexity? We discuss the issue of "sizing up languages".

3.1 Queries

The expressive power of a language is measured by the set of queries it can express. We start by making more precise the notion of query. A query is a mapping from *inst(***R***)* to *inst(***S***)*, where **R** and **S** are disjoint database schemas. The input predicates are sometimes referred to as extensional database predicates, and the output predicates as intensional database predicates, following the Datalog terminology.

Database queries are usually required to obey three conditions: *well-typedness, effective computability* and *genericity* [AU79, AV88, C88]. Well-typedness is captured by requiring that instances over a *fixed* schema be related to instances over another *fixed* schema. Effective computability is self explanatory. Genericity originates from the data independence principle: a query can only use information provided at the conceptual level. In particular, distinct data values can be treated differently only if they can be distinguished using the information available at the conceptual level, or if they are named explicitly in the query. This is formalized by the notion of genericity:

> Let **R** and **S** be database schemas, and C a finite set of constants. A mapping τ from inst(**R**) to inst(**S**) is *C-generic* iff for each bijection ρ over **dom** which is the identity on C, $J = \tau(I)$ iff $\rho(J) = \phi(\rho(I))$.

In the definition of genericity, the set C specifies "exceptional" constants, which can be treated differently from other constants. This is needed because a query can explicitly name a finite set of constants. However, in this paper we ignore the issue of constants in queries, which is not central. Thus, the set C in the definition of genericity is empty.

A query language or computing device is called *complete* if it expresses all queries.

3.2 Programs and Queries

Given a program P (in a query language L), the mapping (or relation) between database instances that the program describes is called the *effect* of the program. The concepts of *input, output* and *temporary relations* are also important. When a program is applied to a given database, its effect is often interpreted by identifying some relations as input relations, and other relations as output relations. In addition to semantically significant input and output relations, the programs may use "temporary" relations. Thus, it appears useful to also define the effect of a program with respect to specified input and output database schemas. Given a program P and two schemas $\mathbf{R}$ and $\mathbf{S}$, P transforms instances over $\mathbf{R}$ into instances over $\mathbf{S}$ as follows: relations which are not in the input schema are assumed to be empty before the program is executed; after the program is run, the relations in the output schema must contain the desired result. (The content of the other relations is immaterial.)

3.3 Complexity of queries

We will refer to complexity classes of database queries. We use as complexity measures the time and space used by a Turing Machine to produce a standard encoding of the output instance starting from an encoding of the input instance. The complexity is w.r.t. the size of the encoded input. Thus, the query is viewed as constant, and the input database as variable. This choice is due to the fact that, typically, one can expect that the size of the database dominates by far the size of the query. This way of looking at complexity is called *data complexity* [V82], as opposed to *query complexity*, where the measure is the size of the query. In this paper we only consider data complexity.

For each Turing Machine complexity class C, there is a corresponding complexity class of queries, which we also denote by C. 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 $enc(\mathbf{I})$ of an input $\mathbf{I}$, produces a standard encoding of $\phi(\mathbf{I})$, in time polynomial in $|enc(\mathbf{I})|$. Note that this is slightly different from the traditional definition in terms of the associated recognition problem [V82], where the complexity of a transformation is defined as the complexity of deciding if a given tuple belongs to the result. We sometimes describe complexity in terms of the associated recognition problem. Thus, NP denotes deterministic transformations for which the recognition problem is in NP. We will specify which measure is used only when the distinction is relevant.

3.4 Expressiveness w.r.t. Complexity Classes

So far we provided a definition of the complexity of an *individual* query. In order to measure the complexity of a query language L, we would like to establish a correspondence between

- the class of queries expressible in L, and
- a complexity class C of queries.

The most straightforward connection is when L and C are precisely the same[2]. In this case, it is said that L *expresses* C. Thus, every query in L is computable with complexity C, and conversely, L can express every query of complexity C.

[2] By abuse of notation, we also denote by L the set of queries expressible in L.

Ideally, one would be able to perform "complexity taylored" language design; that is, for a desired complexity C, one would be able to design a language expressing precisely C. Unfortunately, we will see that this is not always possible. In fact, there are no such results for complexity classes of polynomial time and below, which are of most interest. We will dwell at length upon this phenomenon in Section 4. Intuitively, the "shapes" of classes of queries of low complexity simply do not match those of classes of queries defined by any known language. Therefore, we are lead to consider a less straightforward way to match languages to complexity classes.

3.5 Completeness w.r.t. Complexity Classes

Consider a language L which does not correspond precisely to any natural complexity class of queries. Nonetheless, we would like to say something about the complexity of queries in L. For instance, we would like to guarantee that all queries in L can be evaluated with some complexity C, even though L may not express *all* of C. In order for the bound to be meaningful, we would also like that C be, in some sense, a tight upper bound for the complexity of queries in L. In other words, L should be able to express at least some queries which are among the "hardest" in C. The property of a problem being "hardest" in a complexity class C is captured, in complexity theory, by the notion of *completeness* of the problem in the class[3]. This leads to the following. A language L is *complete w.r.t. a complexity class* C if:

- every query in L is also in C, and
- there exists a query in L which is complete w.r.t. the complexity class C.

In some sense, completeness says something negative about the language L. Indeed, L can express some queries which are as hard as any query in C; on the other hand, there may be *easy* queries in C which may not be expressible in L. This may at first appear contradictory, since L expresses some queries which are complete in C, and any query in C can be reduced to the complete queries. However, there is no contradiction. Indeed, the *reduction* of the "easy" query to the complete query may not be expressible in L.

3.6 Fixpoint and While vs. PTIME and PSPACE

In the discussion on inflationary and non-inflationary languages, we pointed out that *fixpoint* $\subseteq$ PTIME, and *while* $\subseteq$ PSPACE. We would like to understand the precise relationship with these complexity classes, in view of the previous discussion on expressiveness and completeness. It turns out that *fixpoint* is complete w.r.t. PTIME and *while* is complete w.r.t. PSPACE. The completeness is a consequence of the above observation, and results in Section 4. Thus, *fixpoint* can express some of the hardest queries in PTIME, and analogously for *while* and PSPACE. Nonetheless, there are "easy" queries in PTIME and PSPACE that the two languages cannot express. The typical example of such a query is the *even* query on a set:

$$\textit{even}(p) = \textbf{true} \text{ if } |p| \text{ is even, and } \textbf{false} \text{ otherwise.}$$

It is rather puzzling that a query as simple as *even* is not expressible by powerful extensions of FO such as *fixpoint* and *while*. Indeed, the *even* query illustrates well several important issues

[3] A problem is complete in C if it is in C, and any other problem in C can be reduced to it using a reduction of reasonable cost relative to C.

of expressiveness of query languages, and we will use it repeatedly in the paper. Intuitively, *even* is hard to compute by query languages because of the generic nature of the resulting computation. Indeed, if the abstract interface to the database only provides an unordered set, all elements of the set are 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 deterministically 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.

The limitation in the expressive power of *fixpoint* and *while* is emphasized by an interesting feature of properties expressible in the two languages. Consider a property of finite structures expressed by some sentence σ. Let $\mu_n(\sigma)$ be the probability that a structure of size n satisfies σ. The property σ *has a 0-1 law* if $\mu_n(\sigma)$ converges to 0 or 1. The 0-1 laws were defined by Fagin, who also proved that all properties expressible in FO have a 0-1 law [F76]. This was extended to *fixpoint* and *while* [KV89], and to more powerful languages [KV90a, KV90b]. Note that the non-expressibility of *even* follows directly from the fact that *while* has a 0-1 law, since the probability of evenness flip-flops between 0 and 1 and does not converge. A comprehensive survey of 0-1 laws is provided in [Co88].

The difficulty of expressing PTIME with *fixpoint* extends to other languages. Indeed, no language is known that expresses precisely PTIME. The existence of such a language is a major open problem in the theory of query languages. The "running conjecture" is that no such language exists. The same remarks apply to other complexity classes below PTIME, for which there are no expressiveness results.

3.7 Trade-offs: Order and Non-determinism

We saw that the query *even* becomes easy if an order on the domain is available. This is not accidental, and in fact extends to all queries in PTIME and PSPACE. More precisely, with the order assumption, all queries in PTIME and PSPACE can be expressed by *fixpoint*, respectively *while*. Thus, a precise match of these languages with complexity classes occurs. We present these results in more detail in the next section.

Results such as the above show that the presence of order can solve some of the problems of expressiveness of query languages. This can be interpreted as a trade-off between expressiveness and the data independence provided by the abstract interface. Indeed, the internal representation, hidden by the abstract interface, could be used to extract an ordering of the constants used in the database. Thus, giving up the data independence principle allows access to an order, and results in improved expressive power. The tension between data independence and efficient computing of database queries is further examined and formalized in Section 4.3.

Consider again the query *even*. Another way to circumvent the difficulty of computing it generically, is to relax the *determinism* of the query language. Indeed, if one could choose, whenever desired, an *arbitrary* element from the set, this would provide another way of enumerating the elements of the set and computing *even*. The drawback is that, with such a non-deterministic construct in the language, determinism of queries can no longer be guaranteed. We elaborate on the power of non-determinism in query languages in Section 5.

The trade-offs based on order and non-determinism are not unrelated, as it may seem at first. Indeed, suppose that an order is obtained by suspending the data independence principle and accessing the internal representation. In general, the computation may depend on the particular order accessed. Then at the conceptual level, where the order is not visible, the mapping defined by the query appears as non-deterministic. Indeed, different outcomes are possible for the same conceptual-level view of the input. Thus, the trade-offs based on order and on relaxing determinism are intimately connected. The trade-off between determinism and expressive power is an alternative to the trade-off between data-independence and expressive power. Indeed, the following conjectured meta-theorem has been consistently confirmed (C is a time or space Turing complexity class at least linear) :

> If the class of deterministic queries in C can be expressed by a deterministic language in the presence of order, then the class N-C of deterministic and non-deterministic queries computable in C can be expressed by a non-deterministic language.

The idea behind the meta-theorem is that an arbitrary order which can be used to compute the queries in C can be generated by non-deterministic means in linear time/space.

4 The Impact of Order

In this section we discuss in detail the impact of order on the expressive power of query languages. As discussed above, we view the order assumption as essentially the same as suspending the data independence principle in a database. Since data independence is one of the main guiding principles in databases, it is important to understand its consequences on the expressiveness and complexity of query languages.

4.1 Expressiveness With Order

As exemplified by the *even* query, order can considerably affect the expressiveness of a language and the difficulty of computing some queries. Without the order assumption, no expressiveness results are known for the complexity classes of PTIME and below. With order, there are numerous such results. We mention here some of the most prominent ones.

A database is said to be *ordered* if one particular binary relation *succ* provides a successor relation on the constants occurring in the database.

We start with *fixpoint* and *while*.

Theorem 4.1 (i) *Fixpoint* expresses PTIME on ordered databases.
(ii) *While* expresses PSPACE on ordered databases.

Part (i) of the theorem appears in [I86, V82] and part (ii) is due to [V82].

Of course, complexity classes lower than PTIME are most useful in practice. There are numerous results on languages which express complexity classes below PTIME. To illustrate, we mention characterizations of LOGSPACE, non-deterministic logspace, denoted NLOGSPACE, and symmetric logspace, denoted SLOGSPACE. These results are due to Immerman [I87]. The classes LOGSPACE and NLOGSPACE are well known. We briefly review the meaning of SLOGSPACE. This class is based on *Symmetric Turing Machines* (STM), defined by Lewis and Papadimitriou [LP82]. An STM is a Turing Machine for which the "next move" relation is symmetric. Thus, the machine can always return to any previous configuration. The class SLOGSPACE consists of the queries for which the recognition problem is computable by an STM with logarithmic space in the size of the input.

The languages capturing these classes are extensions of FO with various kinds of transitive closure operators, which in turn are specialized forms of the fixpoint operator μ^{inf}. We begin with the transitive closure operator TC. The operator is applied to formulas $\phi(\vec{x},\vec{y})$, where $\vec{x}$ and $\vec{y}$ are both k-tuples of free variables in ϕ. Then $TC(\phi(\vec{x},\vec{y}))$ denotes the transitive closure of the binary relation of k-tuples $\langle\vec{x},\vec{y}\rangle$ defined by $\phi(\vec{x},\vec{y})$. We use two other variations of the TC operator. The first is called *deterministic transitive closure (DTC)*. Given a formula $\phi(\vec{x},\vec{y})$ as above,

$$DTC(\phi(\vec{x},\vec{y})) = TC[\phi(\vec{x},\vec{y}) \wedge \forall\vec{z}(\phi(\vec{x},\vec{z}) \rightarrow \vec{z} = \vec{y})].$$

Finally, the *symmetric transitive closure operator (STC)* applied to a formula $\phi(\vec{x},\vec{y})$ is defined by

$$STC(\phi(\vec{x},\vec{y})) = TC[\phi(\vec{x},\vec{y}) \vee \phi(\vec{y},\vec{x})].$$

The logics obtained by augmenting FO with each of the operators TC, DTC, STC, are denoted by $FO{+}TC$, $FO{+}DTC$, and $FO{+}STC$, respectively. We now have the following [I87].

Theorem 4.2 (i) $FO{+}TC$ expresses NLOGSPACE;
(ii) $FO{+}DTC$ expresses LOGSPACE; and,
(iii) $FO{+}posSTC$ expresses[4] SLOGSPACE.

4.2 Normal Form for While

We next discuss a normal form for the *while* queries, which provides a bridge between computation without order and computation with order. This helps understand the impact of order, and the cost of generic computation without order.

The normal form 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, one can obtain the result on the relationship of *fixpoint* and *while*, stated in Theorem 2.15. Thus, it is shown in [AV91a, AV91b] that *fixpoint* = *while* iff PTIME = PSPACE. The "only if" part follows from Theorem 4.1. The normal form is used for the "if" part as follows. Suppose PTIME = PSPACE. Let w be a *while* query. By the normal form, $w = fw'$, where f is a *fixpoint* query and w' is a *while* query whose computation is isomorphic to that of a *while* query on an ordered domain. Since w' is in PSPACE and PSPACE = PTIME, w' is in PTIME. By Theorem 4.1 (i), there exists a fixpoint query f' equivalent to w' on the ordered domain. Thus w is equivalent to ff' and is a *fixpoint* query.

[4] $FO{+}posSTC$ denotes the restriction of $FO{+}STC$ where STC does not appear within negations.

4.3 Generic Complexity

Generic computation without order raises specific complexity issues which cannot be captured by traditional mechanisms such as Turing Machines (TM). Indeed, there is now ample evidence of a fundamental mismatch between the Turing complexity of queries and their practical hardness.

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 any of the query languages discussed so far. Recall that the source of the difficulty is that all elements of the set must be treated uniformly throughout the computation. 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 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* on unordered sets, 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. Such a model, called *Generic Machine (GM)*, is defined in [AV91a]. The machine consists of a Turing Machine extended with a relational store. Designated relations contain the input at the beginning of the computation, and other relations hold the output at the end of the computation. 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, and for communication among them. The output is only obtained after all machines are merged into a single one, which ensures genericity of the global computation.

Based on GM, one 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, it was shown that the natural polynomial time and space restrictions of two complete languages, $while^{unsort}$ of [CH80] and $while^{invent}$ of [AV90] (described in Section 2.9), coincide respectively with GEN-PTIME and GEN-PSPACE. Similar results can be obtained for the object-oriented language IQL of [AK89]. 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).

It can be shown 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. Such results formalize the trade-off between complexity and computing with an abstract interface without order. In general, the generic complexity of a query is at least as high as its Turing complexity. We claim that the generic complexity is a more realistic complexity measure for languages using solely the abstract interface to the database. Indeed, the possibly lower Turing complexity is generally not realizable by uniform compilation of queries expressed in such a language.

Note that one can add as primitives to any "purely generic" language, any finite number of *oracle* queries with high generic complexity but low Turing complexity, such as *even*, or various forms of *counting* [I87b, I89]. One way to view the problem of capturing PTIME is as follows: is there a choice of such oracles which yields a language expressing exactly PTIME? We believe that the results on generic complexity provide useful tools for approaching this question.

5 Non-Determinism

As discussed in Section 3, a major difficulty in language design is to obtain deterministic languages which express low complexity classes of queries. For instance, it is conjectured that there is no deterministic language expressing exactly the queries computable in polynomial time, or, indeed, any class below polynomial time. This suggests that, for any tractable deterministic query language, there will be "easy" queries which the language will not express. The typical example of such query is the *even* query.

Recall that *even* becomes easy if order is available, or if the language provides a non-deterministic construct allowing to pick an arbitrary element from an unordered set. Indeed, it turns out that non-determinism offers one solution to the limitations in expressive power of deterministic languages, in the form of a trade-off: one gives up the guarantee of determinism in exchange for the ability to express "nice" classes of queries. In this section, we exhibit non-deterministic languages expressing exactly the (deterministic and non-deterministic) queries computable in polynomial time. Analogous results can be shown for lower complexity classes of queries.

There are many different ways to introduce non-determinism in a language. We will describe briefly two families of non-deterministic languages. One family consists of rule-based languages which are variations of Datalog$^{\neg}$ and Datalog$^{\neg\star}$ with fixpoint semantics. The non-determinism results from firing the rules one-by-one, based on a non-deterministic choice. A second family of languages consists of non-deterministic extensions of $FO{+}\mu$ and $FO{+}\mu^{inf}$. The non-determinism is provided by an operator called *witness*, based on the idea of choosing an arbitrary element from a set. It yields formulas with several different interpretations for each given structure. This provides a uniform way of obtaining non-deterministic counterparts for traditional deterministic logics. It was shown in [AV88, AV89] that the two families of languages are strongly related.

5.1 Non-deterministic Fixpoint Logics

The non-deterministic extensions of the fixpoint logics allow formulas that define *several* relations for each given structure. This is achieved by a non-deterministic operator on formulas, called the *witness* operator[5] (W). Intuitively, $W\vec{x}\phi(\vec{x},\,\vec{y})$ indicates that one "witness" $\vec{x}_y$ is chosen for each $\vec{y}$ satisfying $\exists\;\vec{x}\;\phi(\vec{x},\,\vec{y})$. For example, if $R=$ {[1,1],[2,1],[3,2]}, the formula $Wx(R(x,y))$ denotes the set of two relations {[1,1],[3,2]} and {[2,1],[3,2]}. More precisely, for each formula $\phi(\vec{x},\,\vec{y}))$ (where $\vec{x}$ and $\vec{y}$ are vectors of the variables which are free in ϕ), $W\vec{x}\phi(\vec{x},\,\vec{y}))$ is a formula (where the $\vec{x}$ remain free) defining the *set* of relations **I** such that for some **J** defined by ϕ: $\mathbf{I} \subset \mathbf{J}$; and for each $\vec{y}$ for which $[\vec{x},\,\vec{y}]$ is in **J** for some $\vec{x}$, there exists a *unique* $\vec{x}_y$ such that $[\vec{x}_y,\,\vec{y}]$ is in **I**.

It is also possible to describe the semantics of the W operator using functional dependencies: for each instance **J** defined by $\phi(\vec{x},\vec{y})$, $W\vec{x}(\phi(\vec{x},\vec{y}))$ defines all maximal sub-instances **I** of **J** such that the attributes corresponding to the variables in $\vec{y}$ form a key in **I**.

Note that, in general, $Wx(Wy\phi(x,y))$ is *not* equivalent[6] to $Wxy\phi(x,y)$; also, $Wx(Wy\phi(x,y))$ is not equivalent to $Wy(Wx\phi(x,y))$. To see the latter, let $\phi = R(x,y)$, where R is interpreted as {[0,1],[2,1],[2,3]}. Note first that {[0,1],[2,1]} and {[0,1],[2,3]} are the only possible interpretations of $WyR(x,y)$, and {[0,1]}, {[2,1]} and {[0,1],[2,3]} the only possible interpretations of

$$Wx(WyR(x,y)).$$

[5] The witness operator is related to Hilbert's ε-symbol [Le69]. Its semantics is quite different. In particular, the ε-symbol does not yield non-determinism.

[6] Two formulas are *equivalent* iff they define the same set of relations for each given structure.

It is easily seen that {[2,3]} belongs to the set of predicates defined by

$$Wy(WxR(x,y)),$$

so $Wx(WyR(x,y))$ and $Wy(WxR(x,y))$ are not equivalent.

The extension based on the witness operator is orthogonal to the fixpoint extensions of first-order logic corresponding to the deterministic languages. Thus, we will consider extensions of the $FO+\mu^{inf}$ and $FO+\mu$ fixpoint logics with the W operator, denoted $FO+\mu^{inf}+W$ and $FO+\mu+W$, respectively. We note that each "deterministic" logic has a natural W-extension. Thus, one can consider W-extensions of first-order logic, Horn clause logic (*Datalog*), etc. Following is a simple example in $FO+\mu+W$:

Example 5.1 Let G be a symmetric, binary relation. Consider the formula $\mu(\phi(G),G)(x,y)$ where

$$\phi(x,y) = [G(x,y) \wedge \neg Wxy(G(x,y) \wedge G(y,x))].$$

It defines the set of orientations G' of G, where one edge $[x,y]$ is retained for each $[x,y]$ and $[y,x]$ in G. One "redundant" edge is removed from G at each iteration. Note that an orientation G' of G cannot generally be defined by deterministic means, since a non-deterministic choice of the edges to be removed is generally required. □

5.2 Non-deterministic Rule-Based Languages

The non-deterministic rule-based languages we consider are extensions of Datalog. The Datalog extensions allow for the use of negation in bodies and heads of rules. Negations in heads of rules are interpreted as deletions. Rules are fired one instantiation at a time, until a fixpoint is reached. The non-deterministic choice of instantiation yields non-determinism. For instance, the program:

$$\neg G(x,y) \leftarrow G(x,y), G(y,x).$$

computes one of several possible orientations for the graph G. Each orientation of G is a possible outcome of the program. Note that one can obtain deterministic semantics to such programs by firing all applicable instantiations of rules simultaneously. With this semantics, the above program removes from G all cycles of length two.

We next define the syntax of this non-deterministic language, denoted $\mathcal{N}$-$Datalog^{\neg\star}$. Note that heads of rules may contain several literals, and equality can be used in bodies. (It can be shown that these features would be redundant with the deterministic semantics.) An instantiation of a rule can only be fired if its head is consistent, i.e. it contains no literal together with its negation.

Definition 5.2 A $\mathcal{N}$-$Datalog^{\neg\star}$ program is a finite set of rules of the form

$$A_1, ..., A_k \leftarrow B_1, ..., B_n$$

($k \geq 1$, $n \geq 0$), where each A_j is a literal of the form $(\neg)\ Q(x_1,...,x_m)$ ($m \geq 0$), and each B_i is a literal of the same form, or $(\neg)\ x_1 = x_2$ (the x_i's are variables or constants). It is required that each variable occurring in the head of a rule also occur positively bound in the body. □

If the literals in heads are all positive, the program is also a $\mathcal{N}$-$Datalog^{\neg}$ program.

5.3 Equivalence of Languages

In [AV88, AV89] the expressive power of the non-deterministic *Datalog*-like languages and fixpoint logics is characterized. In particular, it was shown that the *Datalog* languages are strongly related to their fixpoint counterparts. Indeed, the non-inflationary languages $FO+\mu+W$ and $\mathcal{N}$-$Datalog^{\neg\star}$, are equivalent. The symmetry between the *Datalog*-like languages and the fixpoint languages breaks down in the inflationary, non-deterministic case. Indeed, $\mathcal{N}$-$Datalog^{\neg}$ is strictly weaker than $FO+\mu^{inf}+W$; however, it can be augmented to compensate for the loss of expressive power in the following fashion. Universal quantification is allowed in bodies of rules and yields the language $\mathcal{N}$-$Datalog^{\neg}\forall$. For example, the query $P - \pi_A(\mathrm{Q})$ (which cannot be expressed in $\mathcal{N}$-$Datalog^{\neg}$) is computed by the $\mathcal{N}$-$Datalog^{\neg}\forall$ program:

$$T(x) \leftarrow \forall y[P(x), \neg Q(x,y)].$$

It was shown that $\mathcal{N}$-$Datalog^{\neg}\forall$ is equivalent to $FO+\mu^{inf}+W$.

5.4 Expressive Power

The non-deterministic languages considered here express both deterministic and non-deterministic queries. One can extend classical complexity classes C to corresponding classes $\mathcal{N}-C$ of deterministic *and* non-deterministic queries computable by (non-deterministic) Turing Machines with complexity C. The non-determinism in the Turing Machine is used here simply to produce the various possible outcomes of the query. Note that this is different from the traditional use of non-determinism in defining complexity classes of *deterministic* mappings. For instance, the non-deterministic analog of PTIME is denoted $\mathcal{N}$-PTIME. This is not to be confused with NP!

The power of the languages considered here w.r.t. their ability to express both deterministic and non-deterministic queries, is given by the following.

Theorem 5.3 (i) $FO+\mu+W$ and $\mathcal{N}$-$Datalog^{\neg\star}$ express precisely $\mathcal{N}$-PSPACE; and,
(ii) $\mathcal{N}$-$Datalog^{\neg}\forall$ and $FO+\mu^{inf}+W$ express precisely $\mathcal{N}$-PTIME.

As a consequence, the set of *deterministic* queries expressed by the languages $\mathcal{N}$-Datalog$^{\neg}\forall$ and $FO+\mu^{inf}+W$ is precisely PTIME. Similarly, the set of deterministic queries expressible by $\mathcal{N}$-Datalog$^{\neg\star}$ and $FO+\mu+W$ is precisely PSPACE.

The above does *not* provide *languages* that express precisely PTIME, PSPACE since non-deterministic queries can also be expressed and, as shown in [ASV90], it is undecidable if a program is deterministic. Instead, the result shows the power of non-deterministic constructs. Thus, augmenting a deterministic language L with the witness operator may allow expressing more *deterministic* queries than in L. Thus, $FO+\mu^{inf}+W$ can express all the PTIME queries, while $FO+\mu^{inf}$ alone cannot express the simple *even* query.

6 Expressiveness Above NP

Our discussion of expressiveness dealt so far with complexity classes of PTIME and below. We have seen that, for these classes, there are no known expressiveness results. That is, there are no known languages which capture them precisely. Using the *even* query, we argued that these limitations in the expressiveness of query languages are due to the difficulty of computing generically without order.

It turns out that the situation changes dramatically for complexity classes above NP. Indeed, expressiveness results abound in this range of complexity. This qualitative difference can again be explained by the impact of order. While for classes below PTIME the explicit presence of order results in increased expressive power, for classes above NP the presence of order is irrelevant. To understand this, consider again the query *even* on an unordered set. One can reduce the problem to that for an ordered set by constructing *all* orderings of elements in the set, then checking the parity by traversing the orderings. This is too expensive to be done within PTIME, but can be done within NP, by non-deterministically guessing an arbitrary ordering. Thus, if $\phi(succ)$ is a query making use of an ordering *succ*, one first guesses the relation *succ*, then one runs $\phi(succ)$ using *succ*.

SO and $\exists SO$. We mention two classical results characterizing NP [F74] and the polynomial hierarchy, PH [S77]. The class NP is characterized by *existential second-order logic*, $\exists SO$. These are formulas of the form $\exists R \phi(R)$, where R is a *relation*, and $\phi(R)$ a first-order formula using R. This matches the intuition that the inputs accepted by a non-deterministic Turing Machine can be described as follows: "*there exists* a sequence of non-deterministic choices in the computation, such that the input is accepted".

Extending the above result, the polynomial hierarchy is characterized by the full second order logic (SO), which allows arbitrary quantifications over relations.

Theorem 6.1 (i) $\exists SO$ expresses NP; (ii) SO expresses PH.

Note that an $\exists SO$ formula can be used to define orderings which may be needed in the computation. Thus, the presence of *succ* can be simulated as follows:

$$\exists\ succ\ ((\ succ \text{ is an ordering}) \wedge \phi(succ)).$$

The sentence "*succ* is an ordering" is clearly expressible in FO.

Complex objects. Recently, the use of *complex objects* has gained prominence, in the context of object-oriented databases. Complex objects are built recursively from atomic constants using a *set nesting* construct, and a *tuple* construct. Set nesting subsumes the power of the quantification on sets provided in SO. We briefly discuss languages with complex objects and some expressiveness results.

Complex objects are typed objects. We assume the existence of the following countably infinite and pairwise disjoint sets of atomic elements: relation names $\{R_1, R_2, ...\}$, attributes $\{A_1, A_2, ...\}$, constants $D = \{d_1, d_2, ...\}$. The abstract syntax τ and the interpretation $[\![\tau]\!]$ of *sorts* are given by:

1. $\tau\ =\ D\ \mid\ [B_1 : \tau, ..., B_k : \tau]\ \mid\ \{\tau\}$ ($k \geq 0$, $B_1, ..., B_k$ distinct attributes),
2. $[\![D]\!] = D$, $[\![\{\tau\}]\!] = \{\{v_1, ..., v_j\} \mid v_i \in [\![\tau]\!], i = 1, ..., j\}$,
3. $[\![[B_1 : \tau_1, ..., B_k : \tau_k]]\!] = \{[B_1 : v_1, ..., B_k : v_k] \mid v_j \in [\![\tau_j]\!], j = 1, ..., k\}$.

An element of a sort is called a *complex object*.

An example of a complex object database is given next.

<table>
<tr><td>

<table>
<tr><th>A</th><th>B</th></tr>
<tr><td>d_1</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_1</td><td>d_2</td></tr><tr><td>d_3</td><td>d_4</td></tr></table></td></tr>
<tr><td>d_1</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_3</td><td>d_4</td></tr><tr><td>d_5</td><td>d_6</td></tr></table></td></tr>
<tr><td>d_2</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_1</td><td>d_3</td></tr><tr><td>d_2</td><td>d_4</td></tr></table></td></tr>
</table>

$\mathbf{J}(R_1)$

</td><td>

<table>
<tr><th>A</th><th>A_1</th><th>A_2</th></tr>
<tr><td>d_1</td><td>d_1</td><td>d_2</td></tr>
<tr><td>d_1</td><td>d_3</td><td>d_4</td></tr>
<tr><td>d_1</td><td>d_5</td><td>d_6</td></tr>
<tr><td>d_2</td><td>d_1</td><td>d_3</td></tr>
<tr><td>d_2</td><td>d_2</td><td>d_4</td></tr>
</table>

$\mathbf{J}(R_2)$

</td><td>

<table>
<tr><th>A</th><th>B</th></tr>
<tr><td>d_1</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_1</td><td>d_2</td></tr><tr><td>d_3</td><td>d_4</td></tr><tr><td>d_5</td><td>d_6</td></tr></table></td></tr>
<tr><td>d_2</td><td><table><tr><th>A_1</th><th>A_2</th></tr><tr><td>d_1</td><td>d_3</td></tr><tr><td>d_2</td><td>d_4</td></tr></table></td></tr>
</table>

$\mathbf{J}(R_3)$

</td></tr>
</table>

Figure 1: A database instance

Example 6.2 Figure 1 shows an instance $\mathbf{J}$ of $(\{R_1, R_2, R_3\}, \mathbf{S})$ where

$$\begin{array}{r}\mathbf{S}(R_1) = \mathbf{S}(R_3) = [A : D, B : \{[A_1 : D, A_2 : D]\}] \quad \text{and} \\ \mathbf{S}(R_2) = [A : D, A_1 : D, A_2 : D].\Box\end{array}$$

Algebra, calculus and deductive languages for complex objects have been proposed. We describe informally the calculus, first defined in [KV84], based on an earlier proposal of [J82]. Other variations have been defined in [AB87, HS88, KRS85]. The calculus uses typed variables and typed relation symbols. Tuple $\langle\ \rangle$ and set $\{\ \}$ constructors are provided. Furthermore, besides equality, a new logical predicate is used, namely set inclusion $\subseteq$. If restricted to a single level of set nesting, the calculus yields exactly the power of SO [HS88].

An example of a query in the complex object calculus is given next.

Example 6.3 Consider the schema and the instance of Example 6.2. One can verify that $\mathbf{J}(R_2)$ is the answer to the following query applied to $\mathbf{J}(R_1)$:

$$\begin{aligned}\{x \mid \exists y, z, z', u, v, w(\quad & R_1(y) \wedge y = [A : z, B : z'] \\ & \wedge\ x = [A : z, A_1 : v, A_2 : w] \wedge [A_1 : v, A_2 : w] \in z')\}.\end{aligned}$$

$\Box$

Recall that the transitive closure of a binary relation cannot be computed in relational algebra/calculus. On the other hand, it can be computed in the complex object algebra/calculus. In the algebra, this is achieved using a *powerset* operation.

Example 6.4 Consider the database schema consisting of a single relation R_0 of sort $\tau = [A : D, B : D]$. A query expressing the transitive closure of R_0 can be expressed as follows:

1. $\{y \mid \forall x(closed(x) \wedge containsR_0(x) \Rightarrow y \in x)\}$; with
2. $closed(x) =$
   $\forall y, z, u, v, w(y \in x \wedge z \in x \wedge y = [A : u, B : v] \wedge z = [A : v, B : w] \Rightarrow [A : u, B : w] \in x)$;

3. $containsR_0(x) = \forall y(R_0(y) \Rightarrow y \in x)$

where $sort(x) = \{\tau\}$, $sort(y) = sort(z) = \tau$ and $sort(u) = sort(v) = sort(w) = D$. □

There has been much work on complex object languages (see the survey [AK89]). Most work on expressiveness of languages with complex objects has focused on their ability to express queries from flat databases to flat databases, using the complex objects only in the computation. It turns out that the calculus, algebra and the deductive languages have equivalent expressive power [AB87]. Furthermore, they all yield the class of "elementary queries" [HS88, KV88]. (A database query is *elementary* if it has elementary-recursive data complexity.) The expressive power is due essentially to the set nesting. Indeed, no additional power beyond FO is obtained without it [PG88].

Several investigations [HS88, KV88, GV91] deal with controlling the complexity of queries by restricting the set height of the types of variables used in the query. In particular, a hierarchy based on the set height of the variables used in queries was exhibited in [HS88], and an alternative hierarchy based on set height and quantification pattern is provided in [KV88]. Exact expressiveness results for similarly restricted languages are obtained in [GV91] using extensions of the fixpoint operators to the complex object calculus.

Acknowledgement: We are grateful to Seymour Ginsburg for his dedication and thoroughness while serving as the authors' PhD advisor, and for his willingness to pass on some of his professional insight. The example of scientific rigor and integrity that he provided continues to be an inspiration to us. Of course, Professor Ginsburg shares the responsibility for the mistakes in the two theses and in all subsequent papers by the authors.

References

[AB87] S. Abiteboul, C. Beeri. On the power of languages for the manipulation of complex objects, INRIA Research Report 846, (1988). Abstract in Proc. International Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt (1987).

[ABW86] Apt, K., H.Blair, A.Walker, Towards a theory of declarative knowledge, Proc. Workshop on Foundations of Deductive Databases and Logic Programming, Washington, D.C. (1986), pp. 546-629.

[ACY91] Afrati, F., S. Cosmadakis, M. Yannakakis, On Datalog vs. polynomial-time, Proc. 10th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1991), pp. 13-25.

[AK90] S. Abiteboul and P. Kanellakis, Query languages for complex object databases, Database Theory Column, SIGACT News, Vol. 21, No3 (1990)

[AK89] S. Abiteboul and P. Kanellakis, Object identity as a query language primitive, Proc. ACM SIGMOD Int'l. Conf. on Management of Data (1989), full version to appear in JACM.

[AG89] Ajtai, M., Y. Gurevich, Datalog vs. first-order logic, Proc. 30th IEEE Symp. on Foundations of Computer Science, (1989), pp. 142-147.

[AU79] Aho, A.V., J. Ullman, Universality of data retrieval languages, 6th ACM Symp. on Principles of Programming Languages (1979), pp. 110-117.

[AV88] Abiteboul, S., V. Vianu, Datalog extensions for database updates and queries, I.N.R.I.A. Technical Report No.715 (1988). To appear in JCSS.

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

[AV90] Abiteboul, S., V. Vianu, Procedural languages for database queries and updates, JCSS, 41,2 (1990) pp. 181-229.

[ASV90] Abiteboul, S., E. Simon, and V. Vianu, Nondeterministic languages to express deterministic transformations, Proc. 9th ACM Symp. on Principles of Database Systems (1990), pp. 218-229.

[AV91a] Abiteboul, S. V. Vianu, Generic computation and its complexity, Proc. *ACM Symposium on Theory of Computing*, 1991.

[AV91b] Abiteboul, S. V. Vianu, Computing with first-order logic, to appear in JCSS.

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

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

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

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

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

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

[Co88] Compton, K.J., 0-1 laws in logic and combinatorics, Proc. 1987 NATO Adv. Study Inst. on algorithms and order, Reidel (1988), pp. 353-383.

[D87] Dahlhaus, E., Skolem normal forms concerning the least fixpoint, in *Computation Theory and Logic*, E. Borger ed., Lecture Notes in Computer Science 270, Springer-Verlag (1987), pp. 101-106.

[DV91] Denninghof, K., V. Vianu, The power of methods with parallel semantics, Proc. Int'l. Conf. on Very Large Data Bases (1991), Barcelona, to appear.

[E61] Ehrenfeucht, A., An application of games to the completeness problem for formalized theories, Fund. Math, 49, 1961.

[F74] Fagin R., Generalized first-order spectra and polynomial-time recognizable sets, in *Complexity of Computation*, ed. R.Karp, SIAM-AMS Proc. 7 (1974), pp. 43-73.

[F76] Fagin R., Probabilities on finite models, J. of Symbolic Logic, 41(1):50-58 (March 1976).

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

[F54] Fraissé, R., Sur les classifications des systèmes de relations, Publ. Sci. Univ Alger, I:1, 1954.

[GV91] Grumbach, S., V. Vianu, Tractable query languages for complex object databases, Proc. 10th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1991), pp. 315-327.

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

[G84] Gurevich, Y., Toward logic taylored for computational complexity, Computation and Proof Theory, ed. M.M.Richter et al. Springer Verlag Lecture Notes in Math. 1104 (1984), 175-216.

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

[GS86] Gurevich, Y., and S.Shelah: Fixed-point extensions of first-order logic, Annals of Pure and Apllied Logic 32, North Holland (1986), 265-280. Also in 26th Symp. on Found. of Computer Science (1985), 346-353.

[H80] Harel, D., On folk theorems, CACM 23 (1980), 379-385.

[HS88] R. Hull, J. Su. On the expressive power of database queries with intermediate types". Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1988), pp. 39-51.

[I86] Immerman N., Relational queries computable in polynomial time, Information and Control 68 (1986), pp. 86-104.

[I87] Immerman N., Languages which capture complexity classes, SIAM J. Comp., 16,4 (1987) pp. 760-778.

[I87b] Immerman N., Expressibility as a complexity measure: results and directions, Yale Univ. Res. Rep. DCS-TR-538, (1987).

[I89] Cai J.-W., M. Fürer, N. Immerman, An optimal lower bound on the number of variables for graph identification, Proc. IEEE Symp. on Foundations of Computer Science (1989), pp. 612-617.

[J82] Jacobs, B., On database logic, J. of the ACM, 29,2, (1982), pp. 310-332.

[K91] Kanellakis, P.C., Elements of Relational Database Theory, Handbook of Theoretical Computer Science, North-Holland (1991).

[K87] Kolaitis, P.G., The expressive power of stratified logic programs, to appear in Information and Computation.

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

[KV89] Kolaitis, P., M. Vardi, The decision problem for the probabilities of higher-order properties, Proc. 19th ACM Symp. on Theory of Computing, (1987), pp. 425-435.

[KRS85] Korth, H.F., M.A.Roth, A. Silberschatz, Extended algebra and calculus for not 1NF relational databases, Technical Report, Dept. of Comp. Sci., Univ. of Texas at Austin (1985).

[KV84] G.M. Kuper, M.Y. Vardi. A new approach to database logic, Proc. 3rd ACM Symp. on Principles of Database Systems (1984), pp. 86-96.

[KV88] G.M. Kuper, M.Y. Vardi. On the complexity of queries in the logical data model". Proc. 2nd Int'l. Conf. on Database Theory, pp. 267-280 (1988).

[KV90a] Kolaitis, P., M. Vardi, 0-1 laws and decision problems for fragments of second-order logic, Information and Computation, 87:302-338 (1990).

[KV90b] Kolaitis, P., M. Vardi, 0-1 laws for infinitary logics, Proc. 5th IEEE Symp. on Logic in Computer Science, (1990), pp. 156-167.

[Le88] Leivant, D., Inductive definitions over finite structures, draft (1988).

[Le69] Leisenring, A.C., *Mathematical Logic and Hilbert's ε-symbol.* Gordon and Breach ed. (1969).

[LP82] Lewis, H., C.H.Papadimitriou, Symmetric space bounded computation, Theoretical Comp. Sci. 19 (1982), pp. 161-188.

[M74] Moschovakis, Y., Elementary induction on abstract structures, North Holland, 1974.

[MW88] Manchanda, S., D.S. Warren, A logic-based language for database updates, in *Foundations of Logic Programming and Deductive Databases*, ed. J.Minker, Morgan-Kaufman, Los Altos (1988).

[N86] Naqvi, S., A logic for negation in database systems, Proc. Workshop on Logic in Databases, Washington, D.C. (1986).

[NK88] Naqvi, S., R. Krishnamurthy, Database updates in logic programming, Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1988) pp. 251-262.

[PG88] Paredaens, J., D. Van Gucht, Possibilities and limitations of using flat operators in nested algebra expressions, Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (1988), pp. 29-38.

[S77] Stockmayer, L.J., The Polynomial Hierarchy, Theoretical Comp. Sci. 3 (1977) pp.1-22.

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

[VG86] Van Gelder, A., Negation as failure using tight derivations for general logic programs, 3rd IEEE Symp. on Logic Programming (1986), pp. 127-139.

[VRS88] Van Gelder, A., K.Ross, J.Schlipf, Unfounded sets and well-founded semantics for general logic programs, 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, (1988), pp. 221-230.

[V82] Vardi, M.Y., The complexity of relational query languages, Proc. 14th ACM Symp. on Theory of Computing (1982), pp. 137-146.

[Vi89] Vianu, V., Expressive power of query languages, Tutorial at the 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, (1989).