RULE-BASED LANGUAGES

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

August 11, 1995

Abstract

The paper presents a survey of the main formal rule-based languages and semantics. Both procedural (fixpoint) and declarative (model-theoretic) semantics are defined and discussed, including inflationary and noninflationary fixpoint semantics, and the semi-positive, stratified and well-founded semantics. The relative expressive power and complexity of the various languages are provided. Nondeterministic rule-based languages are also discussed, and it is shown how nondeterminism can circumvent some difficulties concerning the expressive power of the deterministic languages. Finally, languages with value invention (in the spirit of object-creation in oodbs) are presented and issues of expressive power specific to such languages are discussed.

1 Introduction

Rule-based languages lie at the core of several areas of central importance to databases and artificial intelligence, such as deductive databases, active databases, and production systems. This paper presents a survey of the main abstract rule-based languages. The emphasis is on the various semantics, the expressive power, and the complexity of the languages.

In terms of semantics, there are two main competing approaches to rule-based languages. The first, which we shall call the *logic programming* approach, attempts to provide declarative, model-theoretic semantics to programs. This paradigm is dominant in deductive databases. The second approach, which we call the *production systems* approach, provides procedural semantics based on forward chaining of rules. This approach is dominant in active databases and production systems. An underlying theme of this survey is a comparison between these competing approaches in the context of various languages.

All rule-based languages considered in this survey are variations of Datalog. We begin by introducing this language, which provides a nice but very simplified abstraction of recursion. In terms of the competing approaches alluded to earlier, Datalog is the "garden of Eden" of rule-based languages: there is a perfect marriage between the logic programming and

*Work supported in part by the National Science Foundation under grant IRI-9221268.

the production systems approach to semantics. The difficulties begin with the introduction of negation. We present first the procedural, production systems semantics to Datalog$^\neg$ and to a further extension denoted Datalog$^{\neg\neg}$ that allows for explicit retraction of facts. We next describe the main declarative semantics which have been proposed for Datalog$^\neg$ and its various restrictions: the semi-positive, stratified and well-founded semantics, and compare the expressive power of the languages with the various procedural and declarative semantics. One of the nicest results in regard to expressive power is the convergence of the procedural and declarative semantics to the well-known *fixpoint* queries.

Deterministic languages, including rule-based languages, have well-known limitations of expressive power. For example, there is no known languages that expresses precisely the PTIME queries. We consider two ways to circumvent these limitations. The first trades off the data independence principle for expressiveness, and is formalized by results on expressiveness in the presence of *order*. For example, Datalog$^\neg$ with inflationary or well-founded semantics is shown to express on ordered databases exactly the queries computable in polynomial time. The second, intimately related to the first, trades off determinism for expressiveness. Indeed, we exhibit nondeterministic rule-based languages that can express all (deterministic and nondeterministic) queries computable in polynomial time. We also argue that nondeterminism can be a very useful feature, independently of issues of expressiveness. Several practical production systems are in fact nondeterministic. In terms of semantics, we present nondeterministic languages where the nondeterminism arises from the firing of rule instantiations in arbitrary order. We also describe another approach based on a *choice* operator that yields outputs related to the "stable models" of the program.

Finally, we look at rule languages that allow for the invention of new values. Such rules arise in the object-oriented context, where object creation is a very useful and common operation. We present results on the impact of this feature on the expressiveness of rule-based languages. In particular, we exhibit a language that expresses all traditional queries, and is thus complete. We also point out limitations in the ability of rule-based languages to express nontraditional transformations that contain newly created objects in their results.

The paper is organized as follows. Some background is provided in Section 2. The production systems approach is surveyed next. This introduces Datalog, inflationary Datalog$^\neg$, and noninflationary Datalog$^{\neg\neg}$. The logic programming approach is developed in Section 4, including the semi-positive, stratified, and well-founded semantics for Datalog$^\neg$. The relative expressive power of the languages is provided in Section 5, including connections with complexity classes of queries. Nondeterministic languages are discussed in Section 6, and a value-inventing language is presented in Section 7. Lastly, some conclusions are presented, including a discussion of procedural vs. declarative semantics for rule-based languages.

This survey is inspired by the presentations in [Bid91a, AV91c, AHV95]. Many proofs can be found in [AHV95], and, of course, in the original papers, to which pointers are provided.

2 Background

In this section we review some terminology relating to relational databases. In particular, we recall some of the traditional query languages, including iterative extensions of first-order logic and relational algebra (the *fixpoint* [Mos74, CH82] and *while* [Cha81] queries).

We assume the reader is familiar with the basic concepts and terminology of relational database theory (see [Ull88, Ull89b, AHV95]). We also refer to [Kan91] for a survey of the field. We review briefly some of the basic terminology and notation.

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 *free tuple* over a relational schema R is a mapping from R into $\mathbf{dom} \cup \mathbf{var}$. 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 all instances over a schema $\mathbf{R}$ is denoted by $inst(\mathbf{R})$.

Note that, in logic terms, a database schema supplies a finite set of predicates, and a database instance provides an interpretation of the predicates into finite structures. Only finite structures are considered in this paper.

We are interested primarily in database queries and updates, which involve transformations of database instances into other database instances. We distinguish here between *deterministic* and *nondeterministic database transformations.* A *nondeterministic database transformation* is a subset of *inst(*$\mathbf{R}$*)* $\times$ *inst(*$\mathbf{S}$*)* for some $\mathbf{R}$, $\mathbf{S}$, and a *deterministic* database transformation is a mapping from *inst(*$\mathbf{R}$*)* to *inst(*$\mathbf{S}$*)*.

Database transformations are usually required to obey three conditions: *well-typedness, effective computability* and *genericity* [AU79, CH80, HY84]. 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 or update can only use information provided at the conceptual level of the database. 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/update. Formally, genericity requires that the graph of a database transformation be closed under isomorphisms of the domain.

We will refer to complexity classes of database transformations. 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. For each Turing machine complexity class C, there is a corresponding complexity class of (nondeterministic) transformations denoted (N)DB-C. In particular, the class of nondeterministic database transformations which can be computed by a (nondeterministic) Turing machine in polynomial time is denoted (N)DB-PTIME. It is important to distinguish between classes NDB-C of nondeterministic queries and classes of deterministic queries defined using nondeterministic devices. For example, by Savitch's theorem, PSPACE = NPSPACE, so DB-PSPACE = DB-NPSPACE. Both are classes of deterministic queries. However, NDB-PSPACE contains nondeterministic transformations, so DB-PSPACE $\neq$ NDB-PSPACE (and NDB-PSPACE $\neq$ DB-NPSPACE). Similarly, DB-NP is not to be confused with NDB-PTIME !

Given a program P (in a transformation language L), the mapping (or relation) between database instances that the program describes is called the *effect* of the program, and constitutes the semantics of the program.

Some query languages

Most practical query languages in relational databases are based on FO, first-order logic on relations, sometimes called *relational calculus*. FO has an algebraization called *relational algebra* [Cod70]. Relational algebra provides the following operations on relations: π_X (projection on attributes X), σ_C (selection of tuples satisfying condition C consisting of (in)equalities among attributes and/or constants), $\delta_{A \to B}$ (rename attribute A to B), $\bowtie$ (join of two relations), $-$ (difference), and $\cup$ (union).

There are many useful queries that FO cannot express, such as the transitive closure of a graph. Numerous extensions of FO with recursion have been proposed. Most of them converge towards two central classes of queries: *fixpoint* [Mos74, CH82] and *while* [Cha81]. These can be defined in various ways: by adding fixpoint operators to FO [Mos74, AV91a], looping constructs to relational algebra [Cha81, CH82], or by extensions of Datalog [AV91a]. We briefly review here the definition of *fixpoint* and *while* using looping constructs.

While and Fixpoint. *While* extends FO with recursion. It provides relation variables, statements of the form $R := \varphi$ where φ is an FO query, and a looping construct *while* φ *do* where φ is an FO condition. An equivalent variation uses loops of the form *while change do* which iterate the body as long as some change is made to some relation. *Fixpoint* is the same as *while* except the semantics of assignment is cumulative (i.e., an assignment denoted $R{+}{=}\ \varphi$ *adds* φ to the current content of R). This guarantees termination of *fixpoint* programs in polynomial time, whereas *while* programs require polynomial space.

3 The Production Systems approach

We describe here the languages Datalog, Datalog$^\neg$, and Datalog$^{\neg\neg}$, and their procedural semantics. The procedural semantics is intuitively very simple: the rules of the program are fired in parallel until a fixpoint is reached. We present this straightforward semantics for the three Datalog languages.

3.1 Datalog

Much of the activity in deductive databases has focused on a toy language called *Datalog*. Some of the early history of Datalog is discussed in [MW88]. Although limited, Datalog highlights some aspects of recursion present in many practical languages. Most of the optimization techniques in deductive databases have been developed around Datalog.

Before formally presenting Datalog, we present informally its syntax and procedural semantics. The fixpoint semantics that we consider here is due to [CH85]. However, it has been considered much earlier in the context of logic programming [vEK76, AvE82]. Following is a Datalog program P_{TC} that computes the transitive closure of a graph. The graph is represented in relation G and its transitive closure in relation T:

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

A Datalog program "defines" the relations occurring in heads of rules, from the other relations. The definition is recursive, so defined relations can also occur in bodies of rules.

Thus, a Datalog program is interpreted as a mapping from instances over the relations occurring in the bodies only, to instances over the relations occurring in the heads. For example, the program above maps a relation over G (a graph) to a relation over T (its transitive closure).

We now formally define the syntax of Datalog.

Definition 3.1 A *(Datalog) rule* is an expression of the form:

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

where $n \geq 1$, $R_1, \ldots, R_n$ are relation names, and $u_1, \ldots, u_n$ are free tuples (tuples of variables and constants). Each variable occurring in u_1 must occur in at least one of $u_2, \ldots, u_n$. A *Datalog program* is a finite set of Datalog rules.

The *head* of the rule is the expression $R_1(u_1)$; and $R_2(u_2), \ldots, R_n(u_n)$ forms the *body*.

The set of constants occurring in a Datalog program P is denoted $adom(P)$; and for an instance $\mathbf{I}$, we use $adom(P, \mathbf{I})$ as an abbreviation for $adom(P) \cup adom(\mathbf{I})$.

Let P be a Datalog program. An *extensional* relation is a relation occurring only in the body of the rules. An *intensional* relation is a relation occurring in the head of some rule of P. The *extensional (database) schema*, denoted $edb(P)$, consists of the set of all extensional relation names; whereas the *intensional schema* $idb(P)$ consists of all the intensional ones. The *schema* of P, denoted $sch(P)$ is the union of $edb(P)$ and $idb(P)$. The semantics of a Datalog program is a mapping from database instances over $edb(P)$ to database instances over $idb(P)$. In some contexts, we call the input data the "extensional database", and call the program the "intensional database". Note also that in the context of logic-based languages, the term *predicate* is often used in place of the term "relation name".

The procedural semantics of Datalog is defined next. We use an operator called the *immediate consequence* operator. The operator produces new facts starting from known facts, using the rules. An (active domain) *instantiation* of a rule $R_1(u_1) \leftarrow R_2(u_2), \ldots, R_n(u_n)$ is a rule $R_1(\nu(u_1)) \leftarrow R_2(\nu(u_2)), \ldots, R_n(\nu(u_n))$ where ν is a valuation which maps each variable into $adom(P, \mathbf{I})$.

Let P be a Datalog program and $\mathbf{K}$ an instance over $sch(P)$. A fact A is an *immediate consequence* for $\mathbf{K}$ and P if either $A \in \mathbf{K}(R)$ for some *edb* relation R, or $A \leftarrow A_1, \ldots, A_n$ is an instantiation of a rule in P and each A_i is in $\mathbf{K}$. The *immediate consequence operator* of P, denoted T_P, is the mapping from $inst(sch(P))$ to $inst(sch(P))$ defined as follows. For each $\mathbf{K}$, $T_P(\mathbf{K})$ consists of all facts A that are immediate consequences for $\mathbf{K}$ and P.

We next note some simple mathematical properties of the operator T_P over sets of instances. We first define two useful properties. For an operator T:

- T is *monotone* if for each $\mathbf{I}$, $\mathbf{J}$, $\mathbf{I} \subseteq \mathbf{J}$ implies $T(\mathbf{I}) \subseteq T(\mathbf{J})$.
- $\mathbf{K}$ is a *fixpoint* of T if $T(\mathbf{K}) = \mathbf{K}$.

The following can be easily shown:

Theorem 3.2 For each P and $\mathbf{I}$, T_P has a minimum fixpoint extending $\mathbf{I}$, denoted $P(\mathbf{I})$.

The minimum fixpoint $P(\mathbf{I})$ can be computed as follows. Given an instance $\mathbf{I}$ over $edb(P)$, one can compute $T_P(\mathbf{I}), T_P^2(\mathbf{I}), T_P^3(\mathbf{I})$, etc. Clearly,

$$\mathbf{I} \subseteq T_P(\mathbf{I}) \subseteq T_P^2(\mathbf{I}) \subseteq T_P^3(\mathbf{I}) \subseteq \ldots$$

Indeed, this follows immediately from the fact that $\mathbf{I} \subseteq T_P(\mathbf{I})$ and the monotonicity of T_P. Let N be the number of facts over predicates in $sch(P)$ and using elements in $\mathbf{I}$. The sequence $\{T_P^i(\mathbf{I})\}_i$ reaches a fixpoint after at most N steps, i.e. for each $i \geq N$, $T_P^i(\mathbf{I}) = T_P^N(\mathbf{I})$. In particular, $T_P(T_P^N(\mathbf{I})) = T_P^N(\mathbf{I})$, so $T_P^N(\mathbf{I})$ is a fixpoint of T_P. We denote this fixpoint by $T_P^\omega(\mathbf{I})$.

The fixpoint approach suggests a straightforward algorithm for the evaluation of Datalog. We explain the algorithm on an example. We extend relational algebra with a *while* operator that allows to iterate an algebraic expression while some condition holds.

Consider again the transitive closure query. We wish to compute the transitive closure of relation G in relation T. Suppose both relations are over AB. This computation is performed by the following program:

$$\begin{array}{l} T \ := \ G; \\ \text{while } q(T) \neq T \text{ do } T := q(T); \end{array}$$

where

$$q(T) = G \cup \pi_{AB}(\delta_{B \to C}(G) \bowtie \delta_{A \to C}(T))$$

(recall that δ is the attribute renaming operation in relational algebra, see Section 2).

A lot of redundant computation is performed when running the *while* program above. An array of optimization techniques for Datalog evaluation has been developed (see [BR88, Bid91b, CGT90, Ull89a, AHV95]. However, this is beyond the scope of this paper.

3.2 Datalog$^\neg$

Datalog$^\neg$ allows negations in bodies of rules. 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.

Example 3.3 We present a Datalog$^\neg$ program with input a graph in binary relation G. The program computes the relation $closer(x, y, x', y')$ defined as follows:

$$closer(x, y, x', y') = \{\langle x, y, x', y' \rangle \mid d_G(x, y) \leq d_G(x', y')\},$$

where $d_G(a, b)$ denotes the distance between nodes a and b in G. ($d(a, b)$ is infinite if there is no path from x to y.) The program is:

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

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 can be inferred. 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, the transitive closure of G is computed in T. Consider the consecutive stages in the evaluation of the program. Note that, if the fact $T(x, y)$ is inferred at stage n, then $d(x, y) = n$. So, if $T(x', y')$ has not been inferred yet, this means that the distance between x and y is less than that between x' and y'. Thus, if $T(x, y)$ and $\neg T(x', y')$ hold at some stage n, then $d(x, y) \leq n$ and $d(x', y') > n$ and $closer(x, y, x', y')$ is then inferred. $\square$

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

$$A \leftarrow L_1, ..., L_n$$

where: A is an atom and each L_i is either an atom B_i (in which case it is called *positive*) or a negated atom $\neg B_i$ (in which case it is called *negative*). (We use an active domain semantics for evaluating Datalog$^\neg$, and so do not require that the rules be range-restricted.)

A Datalog$^\neg$ *program* is a non-empty finite set of Datalog$^\neg$ rules. As for Datalog programs, $sch(P)$ denotes the database schema consisting of all relations involved in the program P; the relations occurring in heads of rules are the *idb* relations of P, and the others are the *edb* relations of P.

The procedural semantics of Datalog$^\neg$ is an extension of the fixpoint semantics of Datalog. Let $\mathbf{K}$ be an instance over $sch(P)$. Recall that an (active domain) *instantiation* of a rule $A \leftarrow L_1, \ldots, L_n$ is a rule $\nu(A) \leftarrow \nu(L_1), \ldots, \nu(L_n)$ where ν is a valuation which maps each variable into $adom(P, \mathbf{K})$. A fact A' is an *immediate consequence* for $\mathbf{K}$ and P if $A' \in \mathbf{K}(R)$ for some *edb* relation R, or $A' \leftarrow L'_1, ..., L'_n$ is an instantiation of a rule in P and: each positive L'_i is a fact in $\mathbf{K}$, and for each negative $L'_i = \neg A'_i$, $A'_i \notin \mathbf{K}$. The *immediate consequence operator* of P, denoted Γ_P, is now defined as follows. For each $\mathbf{K}$ over $sch(P)$,

$$\Gamma_P(\mathbf{K}) = \mathbf{K} \cup \{A \mid A \text{ is an immediate consequence for } \mathbf{K} \text{ and } P\}.$$

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 suggested in Example 3.3, each application of Γ_P is called a *stage* in the evaluation. From the definition of Γ_P, it folows that

$$\Gamma_P(\mathbf{I}) \subseteq \Gamma_P^2(\mathbf{I}) \subseteq \Gamma_P^3(\mathbf{I}) \subseteq \ldots$$

As for Datalog, the sequence reaches a fixpoint, denoted $\Gamma_P^\omega(\mathbf{I})$, after a finite number of steps. The restriction of this to the idb relations (or some subset thereof) is called the *image* (or *answer*) of P on $\mathbf{I}$.

In the procedural semantics described above, increasing sets of facts are inferred by firings of the rules. For that reason, this semantics is also referred to as the *inflationary* semantics for Datalog$^\neg$ (there is an "inflation" of tuples!). The language Datalog$^\neg$ with inflationary semantics is also referred to as *inflationary Datalog*$^\neg$. This semantics was first proposed in [AV88, KP88].

3.3 Datalog$^{\neg\neg}$

Recall that in Datalog$^\neg$ with inflationary semantics, a fact that has been inferred can never be retracted. Datalog$^{\neg\neg}$ allows explicit retraction of a previously inferred fact (thus, the semantics of Datalog$^{\neg\neg}$ is "noninflationary") [AV91a]. Syntactically, this is done using negations in heads of rules, interpreted as deletions of facts. This is close to some practical production systems languages. The resulting language is denoted by Datalog$^{\neg\neg}$, to indicate that negations are allowed in both heads and bodies of rules.

The immediate consequence operator Γ_P and semantics of a Datalog$^{\neg\neg}$ program are analogous to those for Datalog$^{\neg}$ with the following important proviso. If a negative literal $\neg A$ is inferred, the fact A is removed, unless A 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) give priority to negative facts, (ii) interpret the simultaneous inference of A and $\neg A$ as a "no-op", i.e., including A in the new instance only if it is there in the old one; and (iii) interpret the simultaneous inference of A and $\neg A$ as a contradiction which makes the result undefined. The chosen semantics has the advantage over (iii) that the result is always defined. In any case, the choice of semantics is not crucial: it results in equivalent languages.

With the semantics chosen above, 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.

3.4 The Rule Algebra

The examples of Datalog$^{\neg}$ programs shown earlier make it clear that the semantics of such programs is not always easy to understand. There is a simple mechanism which facilitates the specification by the user of various "customized" semantics. This is done by means of the *Rule Algebra*, which allows the specification of an order of firing of the rules, as well as firing up to a fixpoint in an inflationary or noninflationary manner. The Rule Algebra for logic programs was introduced in [IN88].

We present an inflationary and a noninflationary version of the rule algebra, although many variations are possible. For the inflationary version RA^+, the base expressions are individual Datalog$^{\neg}$ rules; the semantics associated to a rule is to apply its immediate consequence operator once in a cumulative fashion. Union ($\cup$) can be used to specify simultaneous application of a pair of rules or more complex programs. The expression $P;Q$ specifies the composition of P and Q; its semantics is to execute P once and then Q once. Inflationary iteration of program P is called for by $(P)^+$. The noninflationary version of the Rule Algebra, denoted RA, starts with Datalog$^{\neg}$ rules, but now with a noninflationary, "destructive" semantics. Union and composition are generalized in the natural fashion, and the noninflationary iterator, denoted '*', is used.

Example 3.4 Let P be the set of rules

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

and Q consist of the rule

$$CT(x,y) \leftarrow \neg T(x,y).$$

The RA^+ program $(P)^+;Q$ computes in CT the complement of the transitive closure of G. $\square$

It can be shown that RA^+ is equivalent to Datalog$^\neg$, and RA is equivalent to Datalog$^{\neg\neg}$ [AV91a]. Thus, an RA^+ program can be compiled into a (possibly much more complicated) Datalog$^\neg$ program. For instance, the RA^+ program in Example 3.4 is equivalent to the Datalog$^\neg$ program in Example 5.5. The advantage of the Rule Algebra lies in the ease of expressing various semantics using simple building blocks. In particular, RA^+ can easily be used to specify the stratified and well-founded semantics for Datalog$^\neg$ introduced in the next section.

4 The Logic Programming approach

We present here the declarative semantics of Datalog and Datalog$^\neg$. To our knowledge, no such semantics has been proposed for Datalog$^{\neg\neg}$.

4.1 Minimum model semantics for Datalog

The key idea of the model-theoretic approach is to view the program as a set of first-order sentences that describes the desired answer. For instance, the rules of P_{CT} yield the logical formulas:

$$\begin{array}{lll} (1) & \forall x,y & (T(x,y) \leftarrow G(x,y)) \\ (2) & \forall x,y,z & (T(x,y) \leftarrow (G(x,z) \wedge T(z,y))). \end{array}$$

The result T must satisfy the above sentences. However, this is not sufficient to uniquely determine the result, since it is easy to see that there are many Ts that satisfy the sentences. However, it turns out that the result becomes unique if one adds the following natural minimality requirement: T consists of the smallest set of facts that makes the sentences true. As it turns out, for each Datalog program and input, there is a unique minimal instance satisfying the sentences corresponding to the program and extending the input. This defines the semantics of a Datalog program. For example suppose that the instance contains:

$$G(a,b), G(b,c), G(c,d).$$

It turns out that $T(a,d)$ holds in each instance obeying (1,2) and where these three facts hold. In particular, it belongs to the minimum instance satisfying (1,2) that includes the input G.

Thus, the database instance constituting the result satisfies the sentences. Such an instance is also called a *model* of the sentences. However, the problem arises that there can be many (indeed, infinitely many) instances satisfying the sentences of a program. Thus, the sentences themselves do not uniquely identify the answer; it remains necessary to specify which of the models is the *intended* answer. This is usually done based on assumptions that are external to the sentences themselves. In this section we formalize: (i) the relationship between rules and logical sentences, (ii) the notion of model, and (iii) the concept of intended model.

We begin by associating logical sentences with rules, much like in the introductory discussion. To a Datalog rule

$$\rho : R_1(u_1) \leftarrow R_2(u_2), \ldots, R_n(u_n)$$

we associate the logical sentence:

$$\forall x_1, \ldots, x_m (R_1(u_1) \leftarrow R_2(u_2) \wedge \ldots \wedge R_n(u_n))$$

where $x_1, \ldots, x_m$ are the variables occurring in the rule; and "$\leftarrow$" is the standard logical *implication.* Observe that an instance $\mathbf{I}$ satisfies ρ, denoted $\mathbf{I} \models \rho$, if for each instantiation

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

such that $R_2(\nu(u_2)), \ldots, R_n(\nu(u_n))$ belong to $\mathbf{I}$, so does $R_1(\nu(u_1))$. In the following, we do not distinguish between a rule ρ and the associated sentence. For a program P, the conjunction of the sentences associated with the rules of P is denoted by Σ_P. It turns out that for each Datalog program P, and input $\mathbf{I}$, there is a minimum model of Σ_P extending $\mathbf{I}$. This model is the semantics of P on input $\mathbf{I}$ and is denoted by $P(\mathbf{I})$.

A surprising and elegant property of Datalog is that the declarative and procedural semantics of Datalog programs coincide. Thus, there is no real competition between the logic programming and production systems approaches in the case of Datalog; instead, they reinforce each other as sides of the same coin. Indeed, $P(\mathbf{I})$ (as defined by the model-theoretic semantics) is a fixpoint of T_P. In particular, it is the minimum fixpoint extending $\mathbf{I}$. As we shall see, this harmonious coexistence of declarative and procedural semantics ceases as soon as negation is introduced.

4.2 Model-theoretic semantics for Datalog$^\neg$

We review here the main declarative semantics for Datalog$^\neg$ (or fragments thereof): semi-positive Datalog$^\neg$, stratified Datalog$^\neg$, and the well-founded semantics for Datalog$^\neg$.

The basic problem

One might hope to extend the model-theoretic semantics of Datalog to Datalog$^\neg$ just as smoothly as the syntax. Unfortunately, things are less straightforward when negation is present. We illustrate informally the problems that arise.

As with Datalog, we can associate to a Datalog$^\neg$ program P the set Σ_P of FO sentences corresponding to the rules of P. Note first that, as with Datalog, Σ_P always has at least one model extending any given input $\mathbf{I}$. Indeed, let $B(P, \mathbf{I})$ be the instance where the *idb* relations contain all tuples with values in $\mathbf{I}$ or P. Clearly, $B(P, \mathbf{I})$ is a model of Σ_P.

For Datalog, the model-theoretic semantics of a program P is given by the unique minimal model of Σ_P extending the input. Unfortunately, this simple solution no longer works for Datalog$^\neg$, since uniqueness of a minimal model extending the input is not guaranteed.

Example 4.1 Let P_{pq} be the program $\{p \leftarrow \neg q, q \leftarrow \neg p\}$. Program P_{pq} has two distinct minimal models: $\{p\}$ and $\{q\}$. As another example, suppose that we want to compute the pairs of disconnected nodes in a graph G, i.e., we are interested in the *complement* of the transitive closure of a graph whose edges are given by a binary relation G. We might naturally be tempted to write the program P_{TCcomp}:

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

Let I be an input for predicate G, and let $\mathbf{J}$ over $sch(P_{TCcomp})$ be such that $\mathbf{J}(G) = I$, $\mathbf{J}(T) \supseteq I$, $\mathbf{J}(T)$ is transitively closed, and $\mathbf{J}(CT) = \{\langle x,y\rangle \mid x, y \text{ occur in } \mathbf{I}, \langle x,y\rangle \notin \mathbf{J}(T)\}$. Clearly, there may be more than one such $\mathbf{J}$, but one can verify that each one is a minimal model of $\Sigma_{P_{TCcomp}}$ satisfying $\mathbf{J}(G) = I$. $\Box$

When for a program P, Σ_P has several minimal models, one must specify which among them is the model *intended* to be the solution. To this end, various criteria of "niceness" of models have been proposed, that can hopefully distinguish the intended model from other candidates. We shall discuss several such criteria as we go along. Unfortunately, none of these criteria suffices to do the job. Moreover, upon reflection it is clear that no criteria can exist that would always permit identification of a unique intended model among several minimal models. This is because, as in the case of program P_{pq} of Example 4.1 above, the minimal models can be completely symmetric; in such cases there is no property that would separate one from the others, using just the information in the input or the program.

In summary, the approach we used for Datalog, based on minimum model semantics, breaks down when negation is present. We shall describe several solutions to the problem of giving semantics to Datalog$^\neg$ programs. We begin with the simplest case and build up from there.

Semi-positive Datalog$^\neg$

We consider now the *semi-positive* Datalog$^\neg$ programs, that only apply negation to *edb* relations. For example, the difference of R and R' can be defined by the one-rule program

$$\mathit{Diff}(x) \leftarrow R(x), \neg R'(x).$$

To give semantics to $\neg R'(x)$ we simply use the closed-world assumption: $\neg R'(x)$ holds iff x is in the active domain and $x \notin R'$. Since R' is an *edb* relation, its content is given by the database and the semantics of the program is clear. We elaborate on this next.

Definition 4.2 A Datalog$^\neg$ program P is *semi-positive* if, whenever a negative literal $\neg R'(x)$ occurs in the body of a rule in P, $R' \in edb(P)$.

As their name suggests, semi-positive programs are "almost positive". Indeed, one could eliminate negation from semi-positive programs by adding, for each *edb* relation R', a new *edb* relation $\overline{R'}$ holding the complement of R' (w.r.t. the active domain), and replacing $\neg R'(x)$ by $\overline{R'}(x)$. Thus, it is not surprising that semi-positive programs behave much like Datalog programs. The next result is easily shown.

Theorem 4.3 Let P be a semi-positive Datalog$^\neg$ program. For every instance $\mathbf{I}$ over $edb(P)$,

(i) Σ_P has a unique minimal model $\mathbf{J}$ satisfying $\mathbf{J}|edb(P) = \mathbf{I}$.

(ii) Γ_P has a unique minimal fixpoint $\mathbf{J}$ satisfying $\mathbf{J}|edb(P) = \mathbf{I}$.

(iii) The minimum model in (i) and the least fixpoint in (ii) are identical, and equal to the limit of the sequence $\{\Gamma_P^i(\mathbf{I})\}_{i>0}$.

Given a semi-positive Datalog$^\neg$ program P and an input $\mathbf{I}$, we denote by $P^{semi-pos}(\mathbf{I})$ the minimum model of Σ_P (or equivalently, the least fixpoint of Γ_P) whose restriction to $edb(P)$ equals $\mathbf{I}$.

An example of semi-positive program that is neither in Datalog nor in FO is given by:

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

This program computes the transitive closure of the complement of G. On the other hand, the program for the complement of transitive closure above is not a semi-positive program. However, it can naturally be viewed as the composition of two semi-positive programs: the program computing the transitive closure, followed by the program computing its complement. Stratification, studied next, may be viewed as the closure of semi-positive programs under composition. It will allow us to specify, for instance, the composition just described, computing the complement of transitive closure.

Stratified semantics for Datalog$^\neg$

We now consider a natural extension of semi-positive programs. In semi-positive programs, the use of negation is restricted to *edb* relations. Now suppose that we use some *defined* relations, much like views. Once a relation has been defined by some program, other programs can subsequently treat it as an *edb* relation and apply negation to it. This simple, natural idea underlies an important extension to semi-positive programs, called "stratified programs". Not surprisingly, this appealing semantics was independently proposed by quite a few investigators [CH85, ABW88, Lif88, Gel86].

Suppose we have a Datalog$^\neg$ program P. Each *idb* relation is defined by one or more rules of P. If we are able to "read" the program so that, for each *idb* relation R', the portion of P defining R' comes before the negation of R' is used, then we can simply compute R' before its negation is used, and we are done. For example, consider program P_{TCcomp} of Example 4.1. Clearly, we intended for T to be defined by the first two rules, before its negation is used in the rule defining CT. Thus, the first two rules are applied before the third. Such a way of "reading" P is called a "stratification" of P, defined next.

Definition 4.4 A *stratification* of a Datalog$^\neg$ program P is a sequence of Datalog$^\neg$ programs $P^1, \ldots, P^n$ such that for some mapping σ from $idb(P)$ to $[1..n]$:

(i) $\{P^1, \ldots, P^n\}$ is a partition of P;

(ii) for each predicate R, all the rules in P defining R are in $P^{\sigma(R)}$ (i.e., in the same program of the partition).

(iii) If $R(u) \leftarrow \ldots R'(v) \ldots$ is a rule in P, and R' is an *idb* relation, then $\sigma(R') \leq \sigma(R)$.

(iv) If $R(u) \leftarrow \ldots \neg R'(v) \ldots$ is a rule in P, and R' is an *idb* relation, then $\sigma(R') < \sigma(R)$.

Given a stratification $P^1, \ldots, P^n$ of P, each P^i is called a *stratum* of the stratification, and σ the *stratification mapping.*

Intuitively, a stratification of a program P provides a way of parsing P as a sequence of subprograms $P^1, \ldots, P^n$ each defining one or several *idb* relations. By (iii), if a relation R' is used positively in the definition of R then R' must be defined earlier or simultaneously with R (this allows recursion!). If the negation of R' is used in the definition of R, then by (iv) the definition of R' must come strictly before that of R.

Unfortunately, not every Datalog$^\neg$ program has a stratification. For example, there is no way to "read" program P_{pq} of Example 4.1 above, so that p is defined before q and q before p. Programs that have a stratification are called *stratifiable.* Thus, P_{pq} is not stratifiable. On the other hand, P_{TCcomp} is clearly stratifiable: the first stratum consists of the first two rules (defining T), and the second stratum consists of the third rule, defining CT using T.

There is a simple test for checking if a program is stratifiable. Not surprisingly, it involves testing for an acyclicity condition in definitions of relations using negation. Let P be a Datalog$^\neg$ program. The *precedence graph* G_P of P is the labeled graph whose nodes are the *idb* relations of P. Its edges are the following:

- if $R(u) \leftarrow \ldots R'(v) \ldots$ is a rule in P then $\langle R', R \rangle$ is an edge in G_P with label $+$ (called a *positive edge*).
- if $R(u) \leftarrow \ldots \neg R'(v) \ldots$ is a rule in P then $\langle R', R \rangle$ is an edge in G_P with label $-$ (called a *negative edge*).

Stratifiability of a program can be tested using its precedence graph as follows.

Proposition 4.5 A Datalog$^\neg$ program P is stratifiable iff its precedence graph G_P has no cycle containing a negative edge.

Clearly, the stratifiability test provided by Proposition 4.5 takes time polynomial in the size of the program P.

Consider a stratifiable program P with a stratification $\sigma = P^1, \ldots, P^n$. Using the stratification σ, we can now easily give a semantics to P using the well-understood semi-positive programs. Indeed, notice that for each program P^i in the stratification, if P^i uses the negation of R' then $R' \in edb(P^i)$ (note that $edb(P^i)$ generally contains some of the *idb* relations of P). Furthermore, R' is either in $edb(P)$ or it is defined by some P^j preceding P^i, i.e. $R' \in \cup_{j<i} idb(P^j)$. Thus, each program P^i is semi-positive relative to previously defined relations. Then the semantics of P is obtained by applying, in order, the programs P^i. More precisely, let $\mathbf{I}$ be an instance over $edb(P)$. Define the sequence of instances

$$\begin{array}{lcl} \mathbf{I}_0 & = & \mathbf{I} \\ \mathbf{I}_i & = & \mathbf{I}_{i-1} \cup P^i(\mathbf{I}_{i-1}|edb(P^i)), 0 < i \leq n. \end{array}$$

Note that $\mathbf{I}_i$ extends $\mathbf{I}_{i-1}$ by providing values to the relations defined by P^i; and that $P^i(\mathbf{I}_{i-1}|edb(P^i))$, or equivalently, $P^i(\mathbf{I}_{i-1})$, is the semantics of the semi-positive program P^i applied to the values of its *edb* relations provided by $\mathbf{I}_{i-1}$. Let us denote the final instance $\mathbf{I}_n$ thus obtained by $\sigma(\mathbf{I})$. This provides the *semantics of* a Datalog$^\neg$ program *under a stratification* σ.

It is easily seen that a Datalog$^\neg$ program can have more than one stratification. Will the different stratifications yield the same semantics? Fortunately, the answer is yes. Two stratifications of a Datalog$^\neg$ program are *equivalent* if they yield the same semantics on all inputs.

Theorem 4.6 [ABW88] Let P be a stratifiable Datalog$^\neg$ program. Then all stratifications of P are equivalent.

We have seen that stratification provides an elegant and simple approach to defining semantics of Datalog$^\neg$ programs. Nonetheless, it has two major limitations. First, it does not provide semantics to *all* Datalog$^\neg$ programs. Second, stratified Datalog$^\neg$ programs are not quite satisfactory with regard to expressive power. From a computational point of view, they provide recursion and negation and are inflationary. Therefore, one might expect that they express the *fixpoint* queries. Unfortunately, stratified Datalog$^\neg$ programs fall short of expressing all such queries. This will be shown in Section 5. Intuitively, this is due to the fact that the stratification condition prohibits recursive application of negation, whereas in other languages expressing *fixpoint* this computational restriction does not exist.

For these reasons, we finally consider another semantics for Datalog$^\neg$ programs, called *well-founded*. As we shall see, this provides semantics to all Datalog$^\neg$ programs and allows expressing all *fixpoint* queries. Furthermore, the well-founded and stratified semantics agree on stratified Datalog$^\neg$ programs.

Well-founded semantics for Datalog$^\neg$

The well-founded semantics relies on a fundamental revision of our expectations on the answer to a Datalog$^\neg$ program. So far, we required that the answer must provide information on the truth or falsehood of every fact. The well-founded semantics is based on the idea that a given program may not necessarily provide such information on all facts. Instead, some facts may simply be indifferent to it, and the answer should be allowed to say that the truth value of those facts is *unknown*. As it will turn out, relaxing expectations about the answer in this fashion allows us to provide a very natural semantics to *all* Datalog$^\neg$ programs. The price to pay is that the answer is no longer guaranteed to provide total information.

Another aspect of this approach is that it puts negative and positive facts on a more equal footing. One can no longer assume that $\neg R(u)$ is true simply because $R(u)$ is not in the answer. Instead, both negative and positive facts must be inferred. To formalize this, we shall introduce 3-valued instances, where the truth value of facts can be **true**, **false**, or **unknown**.

The section begins by introducing a largely declarative semantics for Datalog$^\neg$ programs. Next, an equivalent fixpoint semantics is developed. Finally, it is shown that the stratified and well-founded semantics agree on the family of stratified Datalog$^\neg$ programs.

The aim of giving semantics to a Datalog$^\neg$ program P will be to find an "appropriate" 3-valued model $\mathbf{I}$ of Σ_P. In considering what "appropriate" might mean, it is useful to recall the basic motivation underlying the logic programming approach to negation, as opposed to the purely computational approach. An important goal here is to model some form of natural reasoning process. In particular, consistency in the reasoning process is required. Specifically, one cannot use a fact and later infer its negation. This should be captured in the notion of "appropriateness" of a 3-valued model $\mathbf{I}$, and has two intuitive aspects:

- the positive facts of $\mathbf{I}$ must be inferred from P assuming the negative facts in $\mathbf{I}$; and,
- all negative facts that can be inferred from $\mathbf{I}$ must already be in $\mathbf{I}$.

A 3-valued model satisfying the above notion of "appropriateness" will be called a "3-stable model" of P. It turns out that generally, programs have several 3-stable models. Then it is natural to take as answer the certain (positive and negative) facts which belong to all such models, which turns out to yield, in some sense, the smallest 3-stable model. This is indeed how the well-founded semantics of P will be defined.

Example 4.7 The example concerns a game with states, $a, b, \ldots$. The game is between two players. The possible moves of the games are held in a binary relation *moves*. A tuple $\langle a, b\rangle$ in *moves* indicates that when in state a, one can choose to move to state b. A player loses if he/she is in a state from which there are no moves. The goal is to compute the set of winning states, i.e., the set of states such that there exists a winning strategy for a player in this state. These are obtained in a unary predicate *win*.

Consider the input $\mathbf{K}$ with the following value for *moves*:

$$\mathbf{K}(moves) = \{\langle b,c\rangle, \langle c,a\rangle, \langle a,b\rangle, \langle a,d\rangle, \langle d,e\rangle, \langle d,f\rangle, \langle f,g\rangle\}$$

It is easily seen that there are indeed winning strategies from states d (move to e) and f (move to g). Slightly more subtle is the fact that there is no winning strategy from any of states a, b, or c. Indeed, a given player can prevent the other from winning, essentially by forcing a non-terminating sequence of moves.

Now consider the following nonstratifiable program P_{win}:

$$win(x) \quad \leftarrow \quad moves(x,y), \neg win(y)$$

Intuitively, P_{win} states that a state x is in *win* if there is at least one state y that one can move to from x, for which the opposing player looses. We now exhibit a 3-valued model $\mathbf{J}$ of P_{win}, that agrees with $\mathbf{K}$ on *moves*. As will be seen, this will in fact be the well-founded semantics of P_{win} on input $\mathbf{K}$. Instance $\mathbf{J}$ is such that $\mathbf{J}(moves) = \mathbf{K}(moves)$ and the values of *win*-atoms are given as follows:

true	$win(d), win(f)$
false	$win(e), win(g)$
unknown	$win(a), win(b), win(c)$.

□

We now embark upon defining formally the well-founded semantics. We do this in three steps. First, we define the notion of 3-valued instance and extend the notion of truth value and satisfaction. Then we consider Datalog and show the existence of a minimum 3-valued model for each Datalog program. Finally, we consider Datalog$^\neg$, and the notion of 3-stable model, which is the basis of the well-founded semantics.

3-valued Instances

Dealing with three truth values instead of the usual two requires extending some of the basic notions like instance and model. As we shall see, this is quite straightforward. We will denote **true** by 1, **false** by 0, and **unknown** by 1/2.

Consider a Datalog$^\neg$ program P and a classical two valued instance $\mathbf{I}$. We denote by $P_{\mathbf{I}}$ the program obtained from P by adding to P unit clauses, i.e. rules $R(t) \leftarrow$ for each fact $R(t)$ in $\mathbf{I}$. Then $P(\mathbf{I}) = P_{\mathbf{I}}(\emptyset)$. For the moment, we shall deal with Datalog$^\neg$ programs such as these, whose input is included in the program. Let $\mathbf{B}(P)$ denote all facts of the form $R(a_1, \ldots, a_k)$ where R is a relation and $a_1, \ldots, a_k$ constants occurring in P. In particular, $\mathbf{B}(P_{\mathbf{I}}) = \mathbf{B}(P, \mathbf{I})$.

Let P be a Datalog$^\neg$ program. A *3-valued instance* $\mathbf{I}$ over $sch(P)$ is a total mapping from $\mathbf{B}(P)$ to $\{0, 1/2, 1\}$. We denote by $\mathbf{I}^1$, $\mathbf{I}^{1/2}$, and $\mathbf{I}^0$ the set of atoms in $\mathbf{B}(P)$ whose truth value is $1, 1/2$ and 0, respectively. A 3-valued instance $\mathbf{I}$ is *total* , or *2-valued*, if $\mathbf{I}^{1/2} = \emptyset$. There is a natural ordering $\prec$ among 3-valued instances over $sch(P)$, defined by:

$$\mathbf{I} \prec \mathbf{J} \text{ iff for each } A \in \mathbf{B}(P),\ \mathbf{I}(A) \leq \mathbf{J}(A).$$

Note that this is equivalent to $\mathbf{I}^1 \subseteq \mathbf{J}^1$ and $\mathbf{I}^0 \supseteq \mathbf{J}^0$, and that it generalizes containment for 2-valued instances.

Occasionally, we will represent a 3-valued instance by listing the positive and negative facts and omitting the undefined ones. For example, the 3-valued instance $\mathbf{I}$ where $\mathbf{I}(p) = 1, \mathbf{I}(q) = 1, \mathbf{I}(r) = 1/2, \mathbf{I}(s) = 0$ will also be written as $\mathbf{I} = \{p, q, \neg s\}$.

Given a 3-valued instance $\mathbf{I}$, we next define the truth value of boolean combinations of facts using the connectives: $\vee, \wedge, \neg, \leftarrow$. The truth value of a boolean combination α of facts is denoted by $\hat{\mathbf{I}}(\alpha)$, defined by:

$$\begin{array}{lcl}
\hat{\mathbf{I}}(\beta \wedge \gamma) & = & \min\{\hat{\mathbf{I}}(\beta), \hat{\mathbf{I}}(\gamma)\} \\
\hat{\mathbf{I}}(\beta \vee \gamma) & = & \max\{\hat{\mathbf{I}}(\beta), \hat{\mathbf{I}}(\gamma)\} \\
\hat{\mathbf{I}}(\neg \beta) & = & 1 - \hat{\mathbf{I}}(\beta) \\
\hat{\mathbf{I}}(\beta \leftarrow \gamma) & = & 1 \text{ if } \hat{\mathbf{I}}(\gamma) \leq \hat{\mathbf{I}}(\beta), \text{ and 0 otherwise.}
\end{array}$$

The reader should be careful: known facts about boolean operators in the two-valued context may not hold in this more complex one. For instance, note that the truth-value of $p \leftarrow q$ may be different from that of $p \vee \neg q$. To see that the definition above matches the intuition, one might want to verify that with the specific semantics of $\leftarrow$ used here, the instance $\mathbf{J}$ of Example 4.7 does satisfy (the ground instantiation of) $P_{win,\mathbf{K}}$. That would not be the case if we define the semantics of $\leftarrow$ in a more standard way using $p \leftarrow q \equiv p \vee \neg q$.

A 3-valued instance $\mathbf{I}$ over $sch(P)$ *satisfies* a boolean combination α of atoms in $\mathbf{B}(P)$ iff $\hat{\mathbf{I}}(\alpha) = 1$. Given a Datalog$^{(\neg)}$ program P, a *3-valued model* of Σ_P is a 3-valued instance over $sch(P)$ satisfying the set of implications corresponding to the rules in $ground(P)$.

Example 4.8 Recall the program P_{win} of Example 4.7 and the input instance $\mathbf{K}$ and output instance $\mathbf{J}$ presented there. Consider the ground sentences:

$$\begin{array}{lcl} win(a) & \leftarrow & moves(a,d), \neg win(d) \\ win(a) & \leftarrow & moves(a,b), \neg win(b) \end{array}$$

The first is true for $\mathbf{J}$, because $\hat{\mathbf{J}}(\neg win(d)) = 0$, $\hat{\mathbf{J}}(moves(a,d)) = 1$, $\hat{\mathbf{J}}(win(a)) = 1/2$, and $1/2 \geq 0$. The second is true because $\hat{\mathbf{J}}(\neg win(b)) = 1/2$, $\hat{\mathbf{J}}(moves(a,b)) = 1$, $\hat{\mathbf{J}}(win(a)) = 1/2$, and $1/2 \geq 1/2$.

Observe that on the other hand,

$$\hat{\mathbf{J}}(win(a) \vee \neg(moves(a,b) \wedge \neg win(b))) = 1/2.$$

□

3-valued minimal model for Datalog

We next extend the definition and semantics of Datalog programs to the context of 3-valued instances. Although Datalog programs do not contain negation, they will now be allowed to infer positive, unknown and false facts. The syntax of a *3-extended Datalog program* is the same as for Datalog, except that the truth values $0, 1/2, 1$ can occur as literals in bodies of rules. Given a 3-extended Datalog program P, the *3-valued immediate consequence operator 3-*T_P of P is a mapping on 3-valued instances over $sch(P)$ defined as follows. Given a 3-valued instance $\mathbf{I}$ and $A \in \mathbf{B}(P)$, *3-*$T_P(\mathbf{I})(A)$ is:

1 if there is a rule $A \leftarrow$ *body* in $ground(P)$ such that $\hat{\mathbf{I}}$(*body*)$= 1$,

0 if for each rule $A \leftarrow$ *body* in $ground(P)$, $\hat{\mathbf{I}}$(*body*)$= 0$ (and in particular, if there is no rule with A in head),

1/2 otherwise.

Example 4.9 Consider the 3-extended Datalog program $P = \{p \leftarrow 1/2;\ p \leftarrow q, 1/2;\ q \leftarrow p, r;\ q \leftarrow p, s;\ s \leftarrow q;\ r \leftarrow 1\}$. Then

$$\begin{array}{lcl} \textit{3-}T_P(\{\neg p, \neg q, \neg r, \neg s\}) & = & \{\neg q, r, \neg s\} \\ \textit{3-}T_P(\{\neg q, r, \neg s\}) & = & \{r, \neg s\} \\ \textit{3-}T_P(\{r, \neg s\}) & = & \{r\} \\ \textit{3-}T_P(\{r\}) & = & \{r\}. \end{array}$$

□

In the following, 3-valued instances are compared with respect to $\prec$. Thus, "least, "minimal", and "monotonic" are with respect to $\prec$ rather than the set inclusion used for classical 2-valued instances. In particular, note that the minimum 3-valued instance w.r.t. $\prec$ is that where all atoms are false. Let $\perp$ denote this particular instance.

With the above definitions, extended Datalog programs on 3-valued instances behave similarly to classical programs. Indeed, the next lemma can be easily verified.

Lemma 4.10 Let P be a 3-extended Datalog program. Then

1. $3\text{-}T_P$ is monotonic and the sequence $\{3\text{-}T_P^i(\bot)\}_{i>0}$ is increasing and converges to the least fixpoint of $3\text{-}T_P$;

2. P has a unique minimal 3-valued model that equals the least fixpoint of $3\text{-}T_P$.

The semantics of an extended Datalog program is the minimum 3-valued model of P. Analogous to conventional Datalog, we denote this by $P(\bot)$.

We are now ready to look at Datalog$^\neg$ programs and wish to formally define 3-stable models of a Datalog$^\neg$ program P. We "bootstrap" to the semantics of programs with negation, using the semantics for 3-extended Datalog programs described above. Let $\mathbf{I}$ be a 3-valued instance over $sch(P)$. We reduce the problem to that of applying a positive Datalog program, as follows. The *positivized ground version* of P given $\mathbf{I}$, denoted $pg(P,\mathbf{I})$, is the 3-extended Datalog program obtained from $ground(P)$ by replacing each negative premise $\neg A$ by $\hat{\mathbf{I}}(\neg A)$ (i.e., 0, 1 or 1/2). Since all negative literals in $ground(P)$ have been replaced by their truth value in $\mathbf{I}$, $pg(P,\mathbf{I})$ is now a 3-extended Datalog program (i.e, a program without negation). Its least fixpoint $pg(P,\mathbf{I})(\bot)$ contains all the facts that are consequences of P by assuming the values for the negative premises as given by $\mathbf{I}$. We denote $pg(P,\mathbf{I})(\bot)$ by $conseq_P(\mathbf{I})$. Thus, the intuitive conditions required of 3-stable models now amount to: $conseq_P(\mathbf{I}) = \mathbf{I}$.

Definition 4.11 Let P be a Datalog$^\neg$ program. A 3-valued instance $\mathbf{I}$ over $sch(P)$ is a *3-stable model* of P iff $conseq_P(\mathbf{I}) = \mathbf{I}$.

Observe an important distinction between $conseq_P$ and the immediate consequence operator used for inflationary Datalog$^\neg$. For inflationary Datalog$^\neg$, we assumed that $\neg A$ was true as long as A was not inferred. Here, we just assume in such case that A is unknown, and try to prove new facts. Of course, doing so requires the 3-valued approach.

Example 4.12 Consider the following Datalog$^\neg$ program P:

$$\begin{array}{lcl} p & \leftarrow & \neg r \\ q & \leftarrow & \neg r, p \\ s & \leftarrow & \neg t \\ t & \leftarrow & q, \neg s \\ u & \leftarrow & \neg t, p, s. \end{array}$$

The program has three 3-stable models (represented by listing the positive and negative facts and leaving out the unknown facts):

$$\begin{array}{lcl} \mathbf{I}_1 & = & \{p, q, t, \neg r, \neg s, \neg u\}, \\ \mathbf{I}_2 & = & \{p, q, s, \neg r, \neg t, \neg u\}, \\ \mathbf{I}_3 & = & \{p, q, \neg r\}. \end{array}$$

Let us check that $\mathbf{I}_3$ is a 3-stable model of P. The program $P' = pg(P, \mathbf{I}_3)$ is:

$$\begin{array}{lcl} p & \leftarrow & 1 \\ q & \leftarrow & 1, p \\ s & \leftarrow & 1/2 \\ t & \leftarrow & q, 1/2 \\ u & \leftarrow & 1/2, p, s. \end{array}$$

The minimum 3-valued model of $pg(P, \mathbf{I}_3)$ is obtained by iterating $\mathit{3}\text{-}T_{P'}(\bot)$ up to a fixpoint. Thus, we start with $\bot = \{\neg p, \neg q, \neg r, \neg s, \neg t, \neg u\}$. The first application of $\mathit{3}\text{-}T_{P'}$ yields $\mathit{3}\text{-}T_{P'}(\bot) = \{p, \neg q, \neg r, \neg t, \neg u\}$. Next, $(\mathit{3}\text{-}T_{P'})^2(\bot) = \{p, q, \neg r, \neg t\}$. Finally, $(\mathit{3}\text{-}T_{P'})^3(\bot) = (\mathit{3}\text{-}T_{P'})^4(\bot) = \{p, q, \neg r\}$. Thus,

$$\mathit{conseq}_P(\mathbf{I}_3) = pg(P, \mathbf{I}_3)(\bot) = (\mathit{3}\text{-}T_{P'})^3(\bot) = \mathbf{I}_3,$$

and $\mathbf{I}_3$ is a 3-stable model of P.

The reader is invited to verify that in Example 4.7, the instance $\mathbf{J}$ is a 3-stable model of the program $P_{win,\mathbf{K}}$, for the input instance $\mathbf{K}$ presented there. □

As seen from the example, Datalog$^\neg$ programs generally have several 3-stable models. We will show later that each Datalog$^\neg$ program has at least one 3-stable model. Therefore, it makes sense to let the final answer consist of the positive and negative facts belonging to all 3-stable models of the program. As we shall see, the 3-valued instance so obtained is itself a 3-stable model of the program.

Definition 4.13 Let P be a Datalog$^\neg$ program. The *well-founded semantics* of P is the 3-valued instance consisting of all positive and negative facts belonging to all 3-stable models of P. This is denoted by $P^{wf}(\emptyset)$. Given Datalog$^\neg$ program P and input instance $\mathbf{I}$, $P^{wf}_{\mathbf{I}}(\emptyset)$ is denoted $P^{wf}(\mathbf{I})$.

Thus the well-founded semantics of the program P in Example 4.12 is $P^{wf}(\emptyset) = \{p, q, \neg r\}$. We shall see below that in Example 4.7, $P^{wf}_{win}(\mathbf{K}) = \mathbf{J}$.

Note that the above description of the well-founded semantics, although effective, is quite inefficient. Indeed, the straightforward algorithm yielded by this description involves checking all possible 3-valued instances of a program, determining which are 3-stable models, and then taking their intersection. We next provide a simpler, efficient way of computing the well-founded semantics. It is based on an "alternating fixpoint" computation that converges to the well-founded semantics. As a side-effect, the proof of this fact shows that each Datalog$^\neg$ program has at least one 3-stable model, (and therefore the well-founded semantics is always defined). It also shows that the well-founded model is itself a 3-stable model, in some sense the smallest.

The idea of the computation is the following. We define an alternating sequence $\{\mathbf{I}_i\}_{i \geq 0}$ of 3-valued instances that are underestimates and overestimates of the facts known in every 3-stable model of P. The sequence is the following:

$$\begin{array}{lcl} \mathbf{I}_0 & = & \bot \\ \mathbf{I}_{i+1} & = & \mathit{conseq}_P(\mathbf{I}_i). \end{array}$$

Recall that $\bot$ is the least 3-valued instance and that all facts have value 0 in $\bot$. Also note that each of the $\mathbf{I}_i$ defined above is a total instance. This follows easily from the following facts:

- if $\mathbf{I}$ is total then $\mathit{conseq}_P(\mathbf{I})$ is total; and,
- the $\mathbf{I}_i$ are constructed starting from the total instance $\bot$ by repeated applications of conseq_P.

The intuition behind the construction of the sequence $\{\mathbf{I}_i\}_{i\geq 0}$ is the following. The sequence starts with $\perp$, which is an overestimate of the negative facts in the answer (it contains all negative facts). From this overestimate we compute $\mathbf{I}_1 = \mathit{conseq}_P(\perp)$, which includes all positive facts that can be inferred from $\perp$. This is clearly an overestimate of the positive facts in the answer, so the set of negative facts in $\mathbf{I}_1$ is an underestimate of the negative facts in the answer. Using this underestimate of the negative facts we compute $\mathbf{I}_2 = \mathit{conseq}_P(\mathbf{I}_1)$, whose positive facts will now be an underestimate of the positive facts in the answer. By continuing the process we see that the even-indexed instances provide underestimates of the positive facts in the answer and the odd-indexed ones provide underestimates of the negative facts in the answer. Then the limit of the even-indexed instances provides the positive facts in the answer and the limit of the odd-indexed instances provides the negative facts in the answer. This intuition will be made formal later on.

It is easy to see that $\mathit{conseq}_P(\mathbf{I})$ is antimonotonic, i.e. if $\mathbf{I} \prec \mathbf{J}$ then $\mathit{conseq}_P(\mathbf{J}) \prec \mathit{conseq}_P(\mathbf{I})$. From this and the facts that $\perp \prec \mathbf{I}_1$ and $\perp \prec \mathbf{I}_2$, it immediately follows that, for all $i > 0$:

$$\mathbf{I}_0 \prec \mathbf{I}_2 \ldots \prec \mathbf{I}_{2i} \prec \mathbf{I}_{2i+2} \prec \ldots \prec \mathbf{I}_{2i+1} \prec \mathbf{I}_{2i-1} \prec \ldots \prec \mathbf{I}_1.$$

Thus, the even subsequence is increasing and the odd one is decreasing. Since there are finitely many 3-valued instances relative to a given program P, each of these sequences becomes constant at some point. Let $\mathbf{I}_*$ denote the limit of the increasing sequence $\{\mathbf{I}_{2i}\}_{i\geq 0}$, and $\mathbf{I}^*$ the limit of the decreasing sequence $\{\mathbf{I}_{2i+1}\}_{i\geq 0}$. From the above inequalities it follows that $\mathbf{I}_* \prec \mathbf{I}^*$. Also, note that $\mathit{conseq}_P(\mathbf{I}_*) = \mathbf{I}^*$ and $\mathit{conseq}_P(\mathbf{I}^*) = \mathbf{I}_*$. Finally, let $\mathbf{I}_*^*$ denote the 3-valued instance consisting of the facts known in both $\mathbf{I}_*$ and $\mathbf{I}^*$, that is:

$$\mathbf{I}_*^*(A) = \begin{cases} 1 & \text{if } \mathbf{I}_*(A) = \mathbf{I}^*(A) = 1 \\ 0 & \text{if } \mathbf{I}_*(A) = \mathbf{I}^*(A) = 0 \text{ and} \\ 1/2 & \text{otherwise.} \end{cases}$$

Equivalently, $\mathbf{I}_*^* = (\mathbf{I}_*)^1 \cup (\mathbf{I}^*)^0$. It can be shown that:

Theorem 4.14 For each Datalog$^\neg$ program P,

1. $\mathbf{I}_*^*$ is a 3-stable model of P.

2. $P^{wf}(\emptyset) = \mathbf{I}_*^*$.

We illustrate the alternating fixpoint computation with several examples.

Example 4.15 (a) Consider again the program in Example 4.12. Let us perform the alternating fixpoint computation described above. We start with $\mathbf{I}_0 = \perp = \{\neg p, \neg q, \neg r, \neg s, \neg t, \neg u\}$. By applying conseq_P we obtain the following sequence of instances:

$$\begin{aligned} \mathbf{I}_1 &= \{p, q, \neg r, s, t, u\}, \\ \mathbf{I}_2 &= \{p, q, \neg r, \neg s, \neg t, \neg u\}, \\ \mathbf{I}_3 &= \{p, q, \neg r, s, t, u\}, \\ \mathbf{I}_4 &= \{p, q, \neg r, \neg s, \neg t, \neg u\}. \end{aligned}$$

Thus, $\mathbf{I}_* = \mathbf{I}_4 = \{p, q, \neg r, \neg s, \neg t, \neg u\}$ and $\mathbf{I}^* = \mathbf{I}_3 = \{p, q, \neg r, s, t, u\}$. Finally, $\mathbf{I}_*^* = \{p, q, \neg r\}$, which coincides with the well-founded semantics of P computed in Example

4.12.

(b) Recall now P_{win} and input $\mathbf{K}$ of Example 4.7. We compute $\mathbf{I}_*^*$ for the program $P_{win,\mathbf{I}}$. Note that for $\mathbf{I}_0$ the value of all *move* atoms is **false**, and for each $j \geq 1$, $\mathbf{I}_j$ agrees with the input $\mathbf{K}$ on the predicate *moves*; thus we do not show the *move* atoms here. For the *win* predicate, then, we have

$$\begin{aligned}
\mathbf{I}_1 &= \{win(a), win(b), win(c), win(d), \neg win(e), win(f), \neg win(g)\} \\
\mathbf{I}_2 &= \{\neg win(a), \neg win(b), \neg win(c), win(d), \neg win(e), win(f), \neg win(g)\} \\
\mathbf{I}_3 &= \mathbf{I}_1 \\
\mathbf{I}_4 &= \mathbf{I}_2
\end{aligned}$$

Thus

$$\begin{aligned}
\mathbf{I}_* &= \mathbf{I}_2 = \{\neg win(a), \neg win(b), \neg win(c), win(d), \neg win(e), win(f), \neg win(g)\} \\
\mathbf{I}^* &= \mathbf{I}_1 = \{win(a), win(b), win(c), win(d), \neg win(e), win(f), \neg win(g)\} \\
\mathbf{I}_*^* &= \{win(d), \neg win(e), win(f), \neg win(g)\}
\end{aligned}$$

which is the instance $\mathbf{J}$ of Example 4.7.

(c) Consider the database schema consisting of a binary relation G and a unary relation *good*, and the following program defining *bad* and *answer*:

$$\begin{aligned}
bad(x) &\leftarrow G(y, x), \neg good(y) \\
answer(x) &\leftarrow \neg bad(x).
\end{aligned}$$

Consider the instance $\mathbf{K}$ over G and *good* where

$$\begin{aligned}
\mathbf{K}(G) &= \{\langle b, c\rangle, \langle c, b\rangle, \langle c, d\rangle, \langle a, d\rangle, \langle a, e\rangle\}, \text{ and} \\
\mathbf{K}(good) &= \{\langle a\rangle\}.
\end{aligned}$$

We assume the facts of the database are added as unit clauses to P, yielding $P_{\mathbf{K}}$. Again, we perform the alternating fixpoint computation for $P_{\mathbf{K}}$. We start with $\mathbf{I}_0 = \bot$ (containing all negated atoms). Applying $conseq_{P_{\mathbf{K}}}$ yields the sequence $\{\mathbf{I}_i\}_{i>0}$ below:

	bad	*answer*
$\mathbf{I}_0$	$\emptyset$	$\emptyset$
$\mathbf{I}_1$	$\{\neg a, b, c, d, e\}$	$\{a, b, c, d, e\}$
$\mathbf{I}_2$	$\{\neg a, b, c, d, \neg e\}$	$\{a, \neg b, \neg c, \neg d, \neg e\}$
$\mathbf{I}_3$	$\{\neg a, b, c, d, \neg e\}$	$\{a, \neg b, \neg c, \neg d, e\}$
$\mathbf{I}_4$	$\{\neg a, b, c, d, \neg e\}$	$\{a, \neg b, \neg c, \neg d, e\}$

omitting (as in (b)) the facts relating to the *edb* predicates G and *good*, which do not change after step 1.

Thus, $\mathbf{I}_*^* = \mathbf{I}_* = \mathbf{I}^* = \mathbf{I}_3 = \mathbf{I}_4$. Note that P is stratified and its well-founded semantics coincides with its stratified semantics. As we shall see, this is not accidental. □

Since the well-founded semantics provides semantics to all Datalog$^\neg$ programs, it does so in particular for stratified programs. We have seen in Example 4.15 (c) one stratified program for which the stratified and well-founded semantics coincide. Fortunately, as shown next, the stratified and well-founded semantics are always compatible. Thus, if a program is stratified then the stratified and well-founded semantics agree.

A Datalog$^\neg$ program P is said to be *total* if $P^{wf}(\mathbf{I})$ is total for each input $\mathbf{I}$ over $edb(P)$.

Theorem 4.16 [GRS88] If P is a stratified datalog$^\neg$ program, then P is total under the well-founded semantics, and for each 2-valued instance $\mathbf{I}$ over $edb(P)$, $P^{wf}(\mathbf{I}) = P^{strat}(\mathbf{I})$.

As just seen, each stratifiable program is total under the well-founded semantics. However, as indicated by Example 4.15(b), a Datalog$^\neg$ program P may yield a 3-valued model $P^{wf}(\mathbf{I})$ on some inputs. Furthermore, there exist programs that are not stratified but whose well-founded models are nonetheless total. Unfortunately, there can be no effective characterization of those Datalog$^\neg$ programs whose well-founded semantics is total for all input databases. One can find sufficient syntactic conditions that guarantee the totality of the well-founded semantics, but this quickly becomes a tedious endeavor. The following interesting question remains unresolved:

Open problem: Is it the case that for each Datalog$^\neg$ program P, one can find another program whose well-founded semantics is total on all inputs and that produces the same positive facts as P?

Research on well-founded semantics, and the related notion of 3-stable model, has its roots in investigations of stable and default model semantics. Although formulated somewhat differently, the notion of stable/default model is equivalent to that of total 3-stable model [Prz90]. Stable model semantics was introduced in [GL88] and default model semantics in [BF87, BF88]. Stable semantics is based on Moore's autoepistemic logic [Moo85], and default semantics is based on Reiter's default logic [Rei80]. The equivalence between autoepistemic and default logic in the general case has been shown in [Kon88]. The equivalence between stable model semantics and default model semantics was shown in [BF88].

Several equivalent definitions of the well-founded semantics have been proposed. The definition used here comes from [Prz90]. The alternating fixpoint computation we described is essentially the same as in [Gel89]. Alternative procedures for computing the well-founded semantics are exhibited in [BF88, Prz89]. Historically, the first definition of the well-founded semantics was proposed in [GRS88, GRS91].

5 Expressive power and complexity

We have described a variety of rule-based languages: Datalog, Datalog$^\neg$ with procedural semantics (inflationary Datalog$^\neg$) and declarative semantics (semi-positive, stratified, and well-founded), and Datalog$^{\neg\neg}$ (with procedural semantics only). In this section we look at the relative expressive power and the complexity of these languages.

5.1 Relative expressiveness

Since FO cannot express transitive closure and since Datalog is monotonic, these two languages are incomparable. The exact power of Datalog has not been precisely characterized. It has been investigated in [AC89, ACY91, CH85, Shm87, LM89, KV90]. Clearly, Datalog expresses only monotonic queries, commutes with homomorphisms of the database (if there are no constants in the program), and can be evaluated in polynomial time. It is natural to wonder if Datalog expresses *precisely* those queries. The answer is negative. Indeed, [ACY91] show that the existence of a path whose length is a perfect square between two nodes is not expressible in Datalog$^{\neq}$ (Datalog augmented with inequalities $x \neq y$), so not in Datalog. This is a monotonic, polynomial-time query, commuting with homomorphisms.

Datalog is strictly included in semi-positive Datalog$^{\neg}$, which can express nonmonotonic queries using negation on edb predicates. Both semi-positive Datalog$^{\neg}$ and FO are included in stratified Datalog$^{\neg}$.

Stratified vs. well-founded semantics

The connection between stratified and well-founded semantics is not obvious and has in fact remained an open problem for some time. It is now known that stratified semantics is strictly weaker than well-founded semantics. The intuitive reason is that in stratified Datalog$^{\neg}$ there is no recursion through negation, so the number of applications of negation is bounded. In contrast, Datalog$^{\neg}$ with well-founded semantics allows recursion through negation, so there is no bound on the number of applications of negation. This distinction turns out to be crucial. The following result follows from two facts: (i) Datalog$^{\neg}$ with well-founded semantics expresses the *fixpoint* queries (Theorems 5.3, 5.2, and 5.4), and (ii) stratified Datalog$^{\neg}$ is strictly weaker than *fixpoint*. The latter result is due to [Kol91], making use of earlier results from [Dal87] and [CH82]. By putting (i) and (ii) together we obtain:

Theorem 5.1 There exists a query expressible by a total Datalog$^{\neg}$ program with well-founded semantics but not expressible by stratified Datalog$^{\neg}$.

Coming up with a query that separates stratified from well-founded semantics (and *fixpoint*) is nontrivial. We describe such a query next. The query concerns so called "game trees". This involves a game played on a given tree. The nodes of the tree are the possible positions in the game and the edges the possible moves from one position to another. Additionally, some leaves of the tree are labeled black. The game is between two players. A round of the game starting at node x begins with Player I making a move from x to one of its children y. Player II then makes a move from y, etc. The game ends when a leaf is reached. Player I wins if Player II picks a black leaf. For a given tree (with labels), Player I has a winning strategy for the game starting at node x if he can win starting at x no matter how Player II plays. We are interested in determining whether there exists such a winning strategy.

The game tree is represented as follows. The set of possible moves is given by a binary relation *move* and the set of black nodes by a unary relation *black*. Consider the query *winning* (not to be confused with the predicate *win* of Example 4.7) that asks if Player I has a winning strategy starting at the root of the tree. This query is not expressible in stratified Datalog$^{\neg}$. However, it is expressible by a Datalog$^{\neg}$ program with well-founded semantics (and in *fixpoint*).

It would be of interest to have simpler and more natural examples of queries expressible by well-founded Datalog$^{\neg}$ but not by stratified Datalog$^{\neg}$. Unfortunately (or fortunately?), there is a shortage of such examples, although several relatively simple queries are conjectured to have this property. For example, consider again the query on a graph:

$$closer(x, y, x', y') = \{\langle x, y, x', y'\rangle \mid d_G(x, y) \leq d_G(x', y')\},$$

where $d_G(a,b)$ denotes the distance between nodes a and b in G. We know from Example 3.3 that *closer* is expressible in inflationary Datalog$^\neg$, and therefore (by Theorem 5.3 to follow) also in Datalog$^\neg$ with well-founded semantics.

Open problem: Is *closer* expressible in stratified Datalog$^\neg$?

It is conjectured that the answer is negative.

Recently, a rather natural query on an object database was shown not to be expressible by stratified Datalog$^\neg$ [AVdB95]: it asks whether two given objects are deeply equal (we do not define here deep equality; intuitively, two objects are deeply equal if they cannot be distinguished by following pointers starting from the objects).

Well-founded vs. inflationary semantics

One of the nicest expressiveness results involving rule-based languages states that the well-founded and inflationary semantics for Datalog$^\neg$ have the same expressive power [Gel89]. This surprising and elegant result shows a convergence of procedural and declarative semantics for Datalog$^\neg$. Furthermore, both semantics express the well-known *fixpoint* queries. This bears witness, once again, to the naturaleness of this class of queries.

The following theorems make precise the conection between inflationary and well-founded semantics. The comparison has to be done with care, since in some sense we are comparing apples with oranges: inflationary semantics lives in the classical 2-valued world whereas well-founded semantics uses 3-valued logic. The first theorem shows in what sense well-founded semantics can be simulated by inflationary semantics. In the following, P^{inf} denotes the query defined by a Datalog$^\neg$ program P under inflationary semantics.

Theorem 5.2 Let P be a Datalog$^\neg$ program. There exists a Datalog$^\neg$ program Q such that $edb(P) = edb(Q)$ and:

1. Q contains, for each relation R in $sch(P)$, three relations R^ϵ_{answer}, where $\epsilon \in \{0, 1/2, 1\}$;
2. for each instance $\mathbf{I}$ over $edb(P)$, $u \in Q^{inf}(\mathbf{I})(R^\epsilon_{answer})$ iff $P^{wf}(\mathbf{I})(R(u)) = \epsilon$, for $\epsilon \in \{0, 1/2, 1\}$.

The next result concerns the simulation of inflationary by well-founded semantics. More precisely, the positive portion of the well-founded semantics yields the same facts as the inflationary semantics.

Theorem 5.3 Let P be a Datalog$^\neg$ program. There exists a Datalog$^\neg$ program Q such that $edb(Q) = edb(P)$, Q has an *idb* relation *answer*, and for each instance $\mathbf{I}$ over $\mathbf{R}$, the positive portion of *answer* in $P^{wf}(\mathbf{I})$ coincides with $Q(\mathbf{I})$.

In the above theorem, the *positive* portion of *answer* for $P^{wf}(\mathbf{I})$ coincides with $Q(\mathbf{I})$. However, $P^{wf}(\mathbf{I})$ is not guaranteed to be total, i.e., it may contain unknown facts. It remains open whether a program can be constructed that always provides a total answer, and whose positive portion coincides with the answer of Q (this is essentially the same as the open problem stated at the end of Section 4).

The connection with the *fixpoint* queries is provided by the following [AV88, AV91a]:

Theorem 5.4 Inflationary Datalog$^\neg$ expresses precisely the *fixpoint* queries.

The proof that inflationary Datalog$^\neg$ can be simulated in *fixpoint* is straightforward. The converse is however nontrivial. The simulation of *fixpoint* by inflationary Datalog$^\neg$ presents two main difficulties, related to the simulation of the control capabilities available in *fixpoint*. The first involves delaying the firing of a rule until after the completion of a fixpoint by another set of rules. Intuitively, this is hard because checking that the fixpoint has been reached involves checking the *non-existence* rather than the existence of some valuation, and Datalog$^\neg$ is more naturally geared towards checking the *existence* of valuations. The solution to this difficulty is illustrated in the following example.

Example 5.5 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, i.e., until the application of the transitivity rule yields no new tuples. To monitor this, the relations $old\text{-}T$, $old\text{-}T\text{-}except\text{-}final$ are used. $old\text{-}T$ follows the computation of T, but is one step behind it. The relation $old\text{-}T\text{-}except\text{-}final$ is identical to $old\text{-}T$, but includes a clause which prevents it from firing when T has reached its last iteration. Thus, $old\text{-}T$ and $old\text{-}T\text{-}except\text{-}final$ differ only in the iteration after the transitive closure T reaches its final value. In the subsequent iteration, the program recognizes that the fixpoint has been reached, and fires the rule computing the complement in relation CT. The program is:

$$
\begin{array}{lcl}
T(x,y) & \leftarrow & G(x,y) \\
T(x,y) & \leftarrow & G(x,z), T(z,y) \\
old\text{-}T(x,y) & \leftarrow & T(x,y) \\
old\text{-}T\text{-}except\text{-}final(x,y) & \leftarrow & T(x,y), T(x',z'), T(z',y'), \neg T(x',y') \\
CT(x,y) & \leftarrow & \neg T(x,y), old\text{-}T(x',y'), \\
 & & \quad \neg old\text{-}T\text{-}except\text{-}final(x',y').
\end{array}
$$

(It is assumed that G is not empty.) $\square$

The second difficulty concerns keeping track of iterations of the body in the computation of a loop. Given a loop

$$\textit{while change do body}$$

the simulation of *body* itself may involve numerous relations, whose behavior may be "sabotaged" by an overly zealous application of iteration. To overcome this we separate the "internal" computation of the body from the "external" iteration, as illustrated in the following example.

Example 5.6 Let G be a binary relation schema. Consider the *fixpoint* program

$$
\begin{array}{l}
good \mathrel{+}= \emptyset; \\
\textit{while change do} \\
good \mathrel{+}= \varphi
\end{array}
$$

where

$$\varphi = \forall y\ (G(y,x) \rightarrow good(y)).$$

Note that the query computes the set of nodes in G that are not reachable from a cycle. (In other words, the nodes such that the lengths of paths leading to them are bounded.) One iteration is achieved by the Datalog$^{\neg}$ program P:

$$\begin{array}{lcl} bad(x) & \leftarrow & G(y,x), \neg good(y) \\ delay & \leftarrow & \\ good(x) & \leftarrow & delay, \neg bad(x) \end{array}$$

Simply iterating P does not yield the desired result. Intuitively, the relations *delay* and *bad*, which are used as "scratch paper" in the computation of a single iteration of the loop, cannot be re-initialized, and so cannot be effectively re-used to perform the computation of subsequent iterations.

To surmount this problem, we essentially create a version of P for each iteration The versions are distinguished by using "timestamps". The nodes themselves serve as timestamps. The timestamps marking iteration i are the values newly introduced in relation *good* at iteration $i-1$. Relations *delay* and *delay-stamped* are used to delay the derivation of new tuples in *good* until *bad* and *bad-stamped* (respectively) have been computed in the current iteration. The process continues until no new values are introduced in an iteration. The full program is the union of the three rules given above, which perform the first iteration, and the following rules, which perform the iteration with timestamp t:

$$\begin{array}{lcl} bad\text{-}stamped(x,t) & \leftarrow & G(y,x), \neg good(y), good(t) \\ delay\text{-}stamped(t) & \leftarrow & good(t) \\ good(x) & \leftarrow & delay\text{-}stamped(t), \neg bad\text{-}stamped(x,t). \end{array}$$

□

5.2 Datalog$^{\neg\neg}$

By a method similar to the above, it can be shown that Datalog$^{\neg\neg}$ expresses precisely the *while* queries [AV91a].

How about the relationship between Datalog$^{\neg}$ and Datalog$^{\neg\neg}$? Clearly, Datalog$^{\neg}$ is subsumed by Datalog$^{\neg\neg}$. Intuitively, it is tempting to believe that Datalog$^{\neg\neg}$ is more powerful than Datalog$^{\neg}$ (considering only terminating queries). However, this is far from obvious. Indeed, the following is a consequence of results in [AV91b] on the relationship between *fixpoint* and *while*:

Theorem 5.7 Inflationary (or well-founded) Datalog$^{\neg}$ is strictly less expressive than Datalog$^{\neg\neg}$ iff PTIME $\neq$ PSPACE.

Since it is open whether PTIME $\neq$ PSPACE, the relationship between inflationary (or well-founded) Datalog$^{\neg}$ and Datalog$^{\neg\neg}$ remains open. However, the strict inclusion is conjectured to be true.

The relative expressive power of the various rule-based languages is summarized in Figure 1. The arrow $\Uparrow$ indicates strict inclusion and the arrow $\uparrow$ indicates strict inclusion iff PTIME $\neq$ PSPACE.

$$
\begin{array}{c}
\text{Datalog}^{\neg\neg} \equiv \textit{while} \\
\uparrow \\
\text{well-founded semantics Datalog}^{\neg} \equiv \ \text{inflationary Datalog}^{\neg} \equiv \textit{fixpoint} \\
\Uparrow \\
\text{stratified Datalog}^{\neg} \\
\Uparrow \\
\text{semi-positive Datalog}^{\neg} \\
\Uparrow \\
\text{Datalog}
\end{array}
$$

Figure 1: Relative expressive power of $\text{Datalog}^{(\neg)(\neg)}$ languages.

5.3 Limitations in expressive power

Although the languages discussed here are quite powerful, they have certain shortcomings with regard to expressive power. Indeed, there are very "simple" queries that none of the languages can express. The prototypical example is the evenness query on a unary relation:

$$even(R) = \begin{cases} true & \text{if } \mid R \mid \text{ is even} \\ false & \text{if } \mid R \mid \text{ is odd.} \end{cases}$$

This difficulty is not specific to rule-based languages. Indeed, it extends to most deterministic languages. To understand the difficulty involved, consider the natural way to compute the query: remove elements from R one at a time, and keep a binary counter. However, the elements of R are conceptually undistinguishable one from another, so no deterministic language that adheres to the data independence principle can perform the algorithm just described. There are two ways out: (i) sacrifice data independence, or (ii) sacrifice determinism by allowing a nondeterministic construct to pick an arbitrary element from a set. We will explore each of these trade-offs in turn. Suspending the data independence principle is modeled by access to an *order* among the elements in the database (which is a reasonable mathematical metaphor for access to the additional symmetry-breaking information provided by the internal storage).

Intuitively, there is a strong connection between the use of order (i.e., information on the internal storage) and nondeterminism. If a query is implemented using information on internal storage, then the answer may depend on such information and thus appear nondeterministic at the conceptual level.

5.4 The impact of order

The assumption that databases are ordered can have dramatic impact on the expressive power of languages. In ordered databases, the schema contains a special binary relation *succ* providing a successor relation among all constants in the database. With this assumption, it turns out that stratified Datalog$^{\neg}$, inflationary Datalog$^{\neg}$, and Datalog$^{\neg}$ with well-founded semantics are all equivalent and express precisely DB-PTIME. Furthermore, the apparently much weaker semi-positive Datalog$^{\neg}$ is (almost) as powerful as these languages. The "almost" is due to a technicality concerning the order: we also need to assume that the minimum and maximum constants are explicitly given. Surprisingly, these constants, that can be computed with a first order query if *succ* is given, cannot be computed with semi-positive programs.

Theorem 5.8 Stratified Datalog$^\neg$, Datalog$^\neg$ with well-founded semantics, and inflationary Datalog$^\neg$ are equivalent on ordered databases and express exactly the DB-PTIME queries. They are also equivalent to semi-positive Datalog$^\neg$ on ordered databases with *min* and *max* and express exactly the DB-PTIME queries.

The result that semi-positive Datalog$^\neg$ expresses DB-PTIME on ordered databases is due to [Pap85]. The result that inflationary Datalog$^\neg$ expresses DB-PTIME follows from results in [AV91a] (equivalence of inflationary Datalog$^\neg$ and *fixpoint*) and [Imm86, Var82] (who showed that *fixpoint* expresses DB-PTIME on ordered databases).

Note that without the order assumption, there is no known deterministic language that expresses precisely the DB-PTIME queries. Indeed, the existence of such a language remains one of the main open problems in the theory of query languages. It is conjectured that no language expressing DB-PTIME exists.

The following characterizes the power of Datalog$^{\neg\neg}$ on ordered databases. It follows from a result of [AV91a] showing the equivalence of Datalog$^{\neg\neg}$ and *while*, and from a result of [Var82] showing that *while* expresses DB-PSPACE on ordered databases.

Theorem 5.9 Datalog$^{\neg\neg}$ expresses exactly the DB-PSPACE queries on ordered databases.

6 Nondeterministic rule-based languages

The arguments in favor of nondeterministic languages are both practical and theoretical. The first is that nondeterminism is already present in various forms. There are natural nondeterministic queries and updates, whose implementation using deterministic languages is contrived and inefficient. There are well-known applications in Artificial Intelligence which naturally lead to nondeterminism, and expert systems shells (such as KEE or OPS5 [For81]) whose rule-based components work nondeterministically. The theoretical arguments for nondeterminism involve primarily the expressive power of nondeterministic languages. Indeed, the use of nondeterminism circumvents some of the problems discussed above, associated with deterministic languages. As discussed, it is conjectured that there is no deterministic language expressing exactly the queries computable in polynomial time. On the other hand, there are nondeterministic languages expressing exactly the (deterministic and nondeterministic) queries computable in polynomial time. See [AV91c] for a survey on nondeterminism in logic-based languages, which inspired this presentation.

6.1 Nondeterministic Datalog$^{\neg(\neg)}$

We next consider nondeterministic versions of the Datalog$^{\neg(\neg)}$ languages. Recall that the procedural, deterministic semantics for these languages was the result of evaluating programs by repeatedly firing all rules *in parallel*, up to a fixpoint. The nondeterministic semantics is obtained by firing one instantiation of a rule at a time, based on a nondeterministic choice. For instance consider the program:

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

With deterministic semantics, the program removes all cycles of length two. With the nondeterministic semantics, the program computes one of several possible "orientations" for the graph G (i.e., for every pair of edges (x,y) and (y,x) in G, one of the edges is removed).

We first define the syntax of the nondeterministic version of Datalog$^{\neg\neg}$, denoted N-Datalog$^{\neg\neg}$ [AV91a]. The difference with the deterministic version is that heads of rules may contain several literals, and equality can be used in bodies. It can be seen that these features would be redundant with the deterministic semantics.

Definition 6.1 A N-Datalog$^{\neg\neg}$ 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. □

To formally define the nondeterministic semantics, we introduce the notion of (nondeterministic) immediate successor of a set of facts using a rule. Let r be an N-Datalog$^{\neg\neg}$ rule. Let I be a set of facts and r' be a ground instance of r such that (i) each literal of the body of r' is true in I, (ii) the head of r' is consistent and (iii) each variable is valuated to some constant occurring in I. Then the instance J obtained from I by deleting the facts A such that $\neg A$ is in the head of r', and inserting the facts A in the head of r', is called an *immediate successor* of I using r.

By condition (ii) above, a ground instance of a rule is not considered if its head contains a ground literal and its negation.

Definition 6.2 Let P be an N-Datalog$^{\neg\neg}$ program. The *effect* of P is a relation over sets of facts defined as follows: for each I, (I,J) is in *eff(P)* iff there exists a sequence $I_0 = I, ..., I_n = J$ such that (i) for each i, I_{i+1} is an immediate successor of I_i using some r in P, and (ii) there is no immediate successor $J' \neq J$ of J using some rule in P. □

The language N-Datalog$^{\neg}$ is a specialization of N-Datalog$^{\neg\neg}$ obtained by disallowing negative literals in heads of rules (thus, negation can only occur in bodies of rules).

6.2 Expressive power

We present several results on the expressive power of N-Datalog$^{\neg}$ and N-Datalog$^{\neg\neg}$. As discussed earlier, the most significant result involves expressibility of NDB-PTIME.

We consider first the expressive power of N-Datalog$^{\neg\neg}$.

Theorem 6.3 [AV91a] N-Datalog$^{\neg\neg}$ expresses exactly NDB-PSPACE.

We consider next the expressive power of N-Datalog$^\neg$. It is easy to see that each N-Datalog$^\neg$ transformation is in NDB-PTIME. It turns out that there are simple NDB-PTIME queries that cannot be expressed in N-Datalog$^\neg$. We show this next, and then show how N-Datalog$^\neg$ can be augmented to increase the expressive power to NDB-PTIME.

The strict inclusion in NDB-PTIME is shown using the following example [AV91a].

Example 6.4 Let $\mathbf{R} = \{P(A), Q(AB)\}$. It can be shown that there is no N-Datalog$^\neg$ program which computes $P - \pi_A(Q)$. □

The precise characterization of the power of N-Datalog$^\neg$ is open. We note, however, that N-Datalog$^\neg$ expresses exactly NDB-PTIME in the presence of order.

As seen in the example above, there are very simple transformations that N-Datalog$^\neg$ cannot compute. We now look at the origin of this weakness and show how it can be corrected. Note that N-Datalog$^\neg$ does not provide sufficient control capability to simulate the composition of two programs. Indeed, $P - \pi_A(Q)$ can be obtained as the composition of the mappings defined by the following two rules:

$$\begin{array}{lll} T(x) & \leftarrow & Q(x,y), \quad \text{and} \\ answer(x) & \leftarrow & P(x), \neg T(x). \end{array}$$

The weak control capability of N-Datalog$^\neg$ makes it impossible for programs in this language to simulate the explicit control necessary to compute NDB-PTIME transformations. Note that, in the case of N-Datalog$^{\neg\neg}$, the control needed is provided by deletions. For example, the query in Example 6.4 is computed by the following N-Datalog$^{\neg\neg}$ program:

$$\begin{array}{lll} answer(x) & \leftarrow & P(x) \\ \neg answer(x), \neg P(x) & \leftarrow & Q(x,y). \end{array}$$

The constructs we add to N-Datalog$^\neg$ essentially provide sufficient control to simulate composition (in an inflationary manner). We consider two alternative constructs. The first construct allows for an "inconsistency" symbol $\perp$ to appear in heads of rules. The resulting language is denoted N-Datalog$^\neg\perp$. The idea is that if such a symbol is derived in a computation, that particular computation is abandoned. The second construct is universal quantification in bodies of rules and yields the language N-Datalog$^\neg\forall$. We first present N-Datalog$^\neg\perp$, then N-Datalog$^\neg\forall$. These languages are from [AV91a].

N-Datalog$^\neg\perp$: The language N-Datalog$^\neg$ is extended with the symbol $\perp$ that can occur only as a literal in the head of rules. A pair (I,J) is in the effect of a N-Datalog$^\neg\perp$ program iff J is obtained by a computation where $\perp$ is not derived.

N-Datalog$^\neg\forall$: The language N-Datalog$^\neg$ is extended to allow rules of the form:

$$A_1, ..., A_q \leftarrow \forall \vec{x} B_1, ..., B_n,$$

where $\vec{x}$ is a sequence of variables occurring *only* in the body of the rule. Let $\vec{y}$ be the vector of the variables occurring in $B_1, ..., B_n$ and not in $\vec{x}$, and v be a valuation of $\vec{y}$. The rule is fired with valuation v if for each extension $\overline{v}$ of v to the variables in $\vec{x}$ (which valuates variables in $\vec{x}$ in the active domain), $\overline{v}B_1 \wedge ... \wedge \overline{v}B_n$ holds.

To illustrate these two languages, we show how to compute the query of Example 6.4 with N-Datalog$^\neg\forall$ or N-Datalog$^\neg\perp$ programs.

Example 6.5 The mapping $P - \pi_A(\mathrm{Q})$ is computed by the following N-Datalog$^{\neg\forall}$ program:

$$answer(x) \leftarrow \forall y P(x), \neg Q(x,y).$$

A N-Datalog$^{\neg\perp}$ program computing the same query is:

$$\begin{array}{lcl} PROJ(x) & \leftarrow & \neg \textit{done-with-proj}, Q(x,y) \\ \textit{done-with-proj} & \leftarrow & \\ \perp & \leftarrow & \textit{done-with-proj}, Q(x,y), \neg PROJ(x) \\ answer(x) & \leftarrow & \textit{done-with-proj}, P(x), \neg PROJ(x). \end{array}$$

□

Intuitively, in N-Datalog$^{\neg\forall}$, one can check that a stage is completed (using $\forall$) before proceeding to the next one; this allows simulating composition. In N-Datalog$^{\neg\perp}$, a detected error leads to the derivation of $\perp$. The following shows that in fact these constructs provide sufficient power to bridge the gap between N-Datalog$^{\neg}$ and NDB-PTIME.

Theorem 6.6 [AV91a] For each query τ the following are equivalent:

- τ is in NDB-PTIME,
- τ is defined by a N-Datalog$^{\neg\perp}$ program, and
- τ is defined by a N-Datalog$^{\neg\forall}$ program.

6.3 The choice operator

Another way to provide nondeterminism in rule-based languages is provided by the *choice* operator first presented in [KN88]. This construct has been included in the language LDL, an implementation of Datalog$^{\neg}$ [NT89]. Variations of the choice operator, and its connection with stable models of Datalog$^{\neg}$ programs, are further studied in [SZ90, GPSZ91]. The expressive power of the choice operator in the context of Datalog is investigated in [CGP93]

The following extension of Datalog$^{\neq}$ with a variation of the *choice* operator is introduced in [CGP93]. This particular variation is called *dynamic choice*. Datalog$^{\neq}$ programs are extended by allowing atoms of the form *choice(X, Y)* in rules of bodies, where X and Y are disjoint sets of variables occurring in regular atoms of the rule. Several *choice* atoms can appear in one rule. The language obtained is called Datalog$^{\neq}$+*choice*. The semantics is the following. The *choice* atoms render the immediate consequence operator of a Datalog$^{\neq}$+*choice* program P nondeterministic. In each application of T_P, a subset of the applicable valuations is chosen so that for each rule containing an occurrence *choice(X, Y)*, the functional dependency $X \rightarrow Y$ holds. That is, one instantiation for the Y variables is chosen for each instantiation of the X variables. Moreover, the nondeterministic choices operated at each application of T_P for a given occurrence of a *choose* atom *extend* the choices made in previous applications of T_P for that atom. (Thus, *choose* has a global nature.) Although negation is not used in Datalog$^{\neq}$+*choice*, it can be simulated, as seen in the following example.

Example 6.7 The following Datalog$^{\neq}$+*choice* program computes in $\bar{P}$ the complement of a nonempty relation P with respect to a universal relation T of the same arity [CGP93]:

$$
\begin{array}{lcl}
TAG(X,0) & \leftarrow & P(X) \\
TAG(X,1) & \leftarrow & T(X), COMP(Y,0) \\
COMP(X,I) & \leftarrow & TAG(X,I),\ choose(X,I) \\
\bar{P}(X) & \leftarrow & COMP(X,1)
\end{array}
$$

The role of *choose* in the above program is very simple. When first applied, it associates to each X in P the tag $I = 0$. At the second application, it chooses a tag of 0 or 1 for all tuples in T. However, tuples in P have already been tagged by 0 in the previous application of *choose*, so the tuples tagged by 1 are precisely those in the complement. □

The main result on the expressive power of dynamic choice is the following:

Theorem 6.8 [CGP93] Datalog$^{\neq}$+*choice* expresses exactly NDB-PTIME.

6.4 Connections with Determinism

Recall that one of the motivations for considering nondeterministic languages is their ability to express more *deterministic* transformations. In this section we consider the ability of various nondeterministic languages to express deterministic transformations. We also consider the deterministic transformations definable by considering the "possible" and "certain" answers of a nondeterministic transformation, in the spirit of queries on databases with incomplete information. The results are from [ASV90].

The notion of *functional fragment* expressed by a language is defined next.

Definition 6.9 The *functional fragment* of a (nondeterministic) language is the set of deterministic transformations which are effects of programs in the language. The functional fragment of a language L is denoted *funct*(L). □

In the previous section we characterized the nondeterministic transformations expressible in the various languages. These results can be used to characterize the functional fragments expressible in these languages. Thus, we have:

Theorem 6.10

(1) funct(N-Datalog$^{\neg\forall}$) = funct(N-Datalog$^{\neg\bot}$) = DB-PTIME.

(2) funct(N-Datalog$^{\neg\neg}$) = DB-PSPACE.

An alternative way of obtaining deterministic transformations using nondeterministic programs is suggested by the work of [IL84] on incomplete information. Indeed, there is a natural connection between incomplete information and nondeterminism. As noted in [AG85], incomplete information can be seen as resulting from incompletely specified (therefore nondeterministic) updates. The notions of *possible* and *certain* answers in [IL84] suggest the following definition:

Definition 6.11 Given a nondeterministic program P, the image of an input $\mathbf{I}$ under P with the *possibility* semantics (denoted $poss(\mathbf{I}, P)$) and the *certainty* semantics (denoted $cert(\mathbf{I}, P)$) are defined by:

$$poss(\mathbf{I}, P) = \bigcup\{\mathbf{J} \mid (\mathbf{I}, \mathbf{J}) \in \mathit{eff(P)}\} \quad \text{and} \quad cert(\mathbf{I}, P) = \bigcap\{\mathbf{J} \mid (\mathbf{I}, \mathbf{J}) \in \mathit{eff(P)}\}.$$

The deterministic transformation expressed by a program P under possibility semantics is denoted *poss(P)*, and under certainty semantics *cert(P)*. For a language L,

$$poss(L) = \{poss(P) \mid P \in L\} \text{ and } cert(L) = \{cert(P) \mid P \in L\}.$$

The *poss* or *cert* semantics yield significant power:

Theorem 6.12

(1) $poss(\text{N-Datalog}^{\neg\forall}) = poss(\text{N-Datalog}^{\neg\bot}) =$ DB-NP.
(2) $cert(\text{N-Datalog}^{\neg\forall}) = cert(\text{N-Datalog}^{\neg\bot}) =$ DB-CO-NP.
(3) $cert(\text{N-Datalog}^{\neg\neg}) = poss(\text{N-Datalog}^{\neg\neg}) =$ DB-PSPACE.

Note that, for N-Datalog$^{\neg\neg}$, the *poss* and *cert* semantics do not yield additional power. In particular, these semantics can be simulated within the functional fragment of N-Datalog$^{\neg\neg}$.

7 Complete rule-based languages

All the languages considered so far express queries within (N)DB-PSPACE. Intuitively, this is so because each program uses a fixed number of relations of fixed arity, which are filled in the course of the computation with tuples over the elements of the input. Thus, such programs can build an amount of space which is no more than polynomial in the number of elements of the input.

Suppose that we wish to have a *complete* language, i.e. a language expressing all queries. One way to break the polynomial "space barrier" is to allow programs to invent new values in the course of the computation. Besides the computational justification, this is useful in object-oriented databases, where the creation of new object identifiers is a useful and very common feature.

We first exhibit an extension of Datalog$^{\neg}$ that is complete in the traditional sense: it expresses all deterministic queries whose outputs are relations using only elements in the input. This extension uses new values in the course of the computation, but not in the answer. We informally describe this language, denoted by Datalog$^{\neg}_{new}$. The syntax is the same as that of Datalog$^{\neg}$, except that variables that do not appear in body of a rule may appear in its head. The inflationary semantics of this language is similar to that of Datalog$^{\neg}$. The only difference consists in the use of the variables that occur only in heads of rules: these are valuated *outside* the current active domain, thus resulting in the "invention" of new values. The choice of the particular new value is nondeterministic. However this is the *only* source of nondeterminism. If the final result contains only values from the input, then the query defined by such a program is deterministic. Furthermore, Datalog$^{\neg}_{new}$ is complete:

Theorem 7.1 [AV91a] For each query there is a Datalog$^{\neg}_{new}$ program that expresses it.

Note that there are Datalog$^{\neg}_{new}$ programs that do not express queries in the traditional sense, since their results may contain newly invented values. However, such transformations are very useful in some contexts, such as object-oriented databases. The notion of completeness can be naturally extended to queries that contain new values in their result [AK89]. One might expect that Datalog$^{\neg}_{new}$ is complete with respect to this extended notion of completeness. However, this turns out not to be the case. Indeed, a query not expressible by Datalog$^{\neg}_{new}$ is exhibited in the following example.

Example 7.2 [AK89] Let q be the query with input schema $\mathbf{R} = \{S\}$, where S is unary, and output G where G is binary. Let q be defined as follows: for each input I over S, if $I = \{a, b\}$ then $q(I) = \{\langle\psi_0, \psi_1\rangle, \langle\psi_1, \psi_2\rangle, \langle\psi_2, \psi_3\rangle, \langle\psi_3, \psi_0\rangle, \langle\psi_0, b\rangle, \langle\psi_1, a\rangle, \langle\psi_2, b\rangle, \langle\psi_3, a\rangle\}$ for some new elements $\psi_0, \psi_1, \psi_2, \psi_3$, and $q(I) = \emptyset$ otherwise (Figure 2). This query is not expressible by Datalog$^{\neg}_{new}$. $\square$

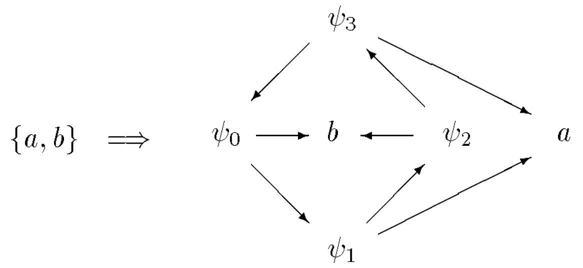

Figure 2: A query not expressible in $while_{new}$.

There is an elegant necessary condition for a query to be computable by Datalog$^{\neg}_{new}$ [AP92, dBGAG92]. We present this condition, then show how it can be used to prove that the query in Example 7.2 is not expressible by Datalog$^{\neg}_{new}$. The condition involves input-output pairs of Datalog$^{\neg}_{new}$ programs. For each input-output pair $\langle I, J\rangle$, the condition requires a simple connection between the automorphism group of I and that of J. For an instance K, let $Aut(K)$ denote the set of automorphisms of K. For a pair of instances K, K', $Aut(\langle K, K'\rangle)$ denotes the bijections on $\mathbf{adom}(K \cup K')$ that are automorphisms of both K and K'.

Theorem 7.3 [dBGAG92] A query q is expressible in Datalog$^{\neg}_{new}$ only if for each input-output pair $\langle I, J\rangle$ in q there exists a mapping h from $Aut(I)$ to $Aut(\langle I, J\rangle)$ such that for each $\tau, \mu \in Aut(I)$,

(i) τ and $h(\tau)$ coincide on I;

(ii) $h(\tau \circ \mu) = h(\tau) \circ h(\mu)$; and[1],

(iii) $h(\mathbf{id}_I) = \mathbf{id}_{\langle I, J\rangle}$.

[1] $\mathbf{id}$ denotes the identity mapping

We note that this becomes a necessary and sufficient condition for an extension of Datalog$^{\neg}_{new}$ called IQL, which provides complex values, including sets [dBGAG92]. IQL is more powerful than Datalog$^{\neg}_{new}$.

A mapping h as above is called an *extension homomorphism* from $Aut(I)$ to $Aut(\langle I, J\rangle)$. To see an example of the usefulness of the above condition, consider the query q in Example 7.2. One can show that q is not expressible in Datalog$^{\neg}_{new}$ as follows. Let $\langle I, J\rangle$ be the input-output pair of Figure 2. The proof is by contradiction. Suppose there is a Datalog$^{\neg}_{new}$ query which produces J on input I. By Theorem 7.3, there is an extension homomorphism h from $Aut(I)$ to $Aut(\langle I, J\rangle)$. Let μ be the automorphism of I exchanging a and b. Note that $\mu^{-1} = \mu$, so $\mu \circ \mu = \mathbf{id}_I$. Consider $h(\mu)(\psi_0)$. Clearly, $h(\mu)(\psi_0) \in \{\psi_1, \psi_3\}$. Suppose $h(\mu)(\psi_0) = \psi_1$ (the other case is similar). Then clearly, $h(\mu)(\psi_1) = \psi_2$. Consider now $h(\mu \circ \mu)(\psi_0)$. We have, on one hand,

$$\begin{aligned} h(\mu \circ \mu)(\psi_0) &= (h(\mu) \circ h(\mu))(\psi_0) \\ &= h(\mu)(\psi_1) \\ &= \psi_2 \end{aligned}$$

and on the other hand

$$\begin{aligned} h(\mu \circ \mu)(\psi_0) &= h(\mathbf{id}_I)(\psi_0) \\ &= \mathbf{id}_{\langle I,J\rangle}(\psi_0) \\ &= \psi_0, \end{aligned}$$

which is a contradiction, since $\psi_0 \neq \psi_2$. So, q is not expressible in Datalog$^{\neg}_{new}$.

It is possible to obtain a language expressing all queries with new values in results by adding to Datalog$^{\neg}_{new}$ a *choose* operator that allows the selection (nondeterministically but in a determinate manner) of one object out of a set of objects that are "isomorphic" [AK89] . However, this is a highly complex construct, since it requires the ability to check for isomorphism of graphs. A tractable construct that yields completeness is exhibited in [DV93]. However, the construct shown there is global in nature (it applies to the entire database). The search for simpler, local constructs that yield a language complete with value invention is an active area of research.

8 Conclusions

We have described a variety of rule-based languages, with semantics following two main paradigms: the procedural, "production systems" approach, and the declarative, "logic programming" approach. The two paradigms are philosophically very different. The procedural approach provides semantics that are extremely simple to describe. However, the semantics are purely computational. In contrast, the declarative semantics attempt to model a natural reasoning process. In particular, consistency in the reasoning process is required: one cannot use a fact and later infer its negation, as can happen in the procedural semantics. The declarative approach attempts to describe the solution as a model of the program satisfying certain desirable properties, such as minimality.

What conclusion can we draw from the results presented here? As we have seen, the declarative and procedural semantics for Datalog$^{\neg}$ converge, in terms of expressive power, to the *fixpoint* queries. Therefore, any comparison between the procedural and declarative paradigms is necessarily based on subjective preference of one style over another. We attempt such a (risky!) comparison in the next paragraphs.

Perhaps the first question to ask about "declarative" semantics is to what extent the various semantics proposed are truly declarative. The minimum model semantics for Datalog is as purely declararative as one can get. How about languages beyond Datalog? Once negation is present, the lack of a unique minimal model leads to the need to specify an "intended" model among several possible candidates. The stratified semantics is very natural, but is it declaratice, or procedural? We would venture that stratified semantics is in fact best understood in procedural terms (compute relation R, then use its negation, etc), rather than in model theoretic terms (using preferred models, not discussed here). Beyond stratified semantics, the criteria for selecting the intended model become increasingly contrived. Furthermore, the common justification that the chosen criterion captures a fictitious programmer's natural reasoning process rests on shallow ground.

Procedural semantics is simple to understand, but, as already mentioned, does not yield a solution easily justifiable in model theoretic terms, beyond the fact that the solution is one model of the program. Also, despite the simplicity of the computational semantics, larger programs can become hard to understand, as their effect is very sensitive to timing (e.g., see Example 3.3). One way around this that seems quite appealing is provided by the rule algebras RA (and RA$^+$). In the algebras, complicated programs can be build up using simple programs as building blocks. This breaks down the complexity of the program and allows for a very flexible specification mechanism. Furthermore, all declarative semantics can be easily specified using the rule algebras (e.g. the stratified semantics is expressed as the composition of the strata in some stratification, and the well-founded semantics is expressed by a program mimicking the alternating fixpoint computation of the semantics).

The final test for the various semantics has to rest with programmers. In that respect, the procedural approach seems to be leading for the time being: there are production systems in use (and expert systems whose shells have a procedural rule-based component), while deductive database systems are limited to prototypes (see [RU95] for a survey of deductive database implementations). Active databases, still at an incipient stage, have adopted the procedural approach to rule semantics.

In terms of expressiveness, we pointed out some limitations in the expressive power of the deterministic languages considered. The nondeterministic languages circumvent some of these limitations. In particular, we exhibited several languages that express NDB-PTIME. Thus, the nondeterminism provided by such languages not only allows expressing nondeterministic queries which may be useful on their own, but additionally allows expressing more *deterministic* queries. This point out to an interesting trade-off between expressiveness and determinism, which is intimately related to the trade-off between expressiveness and the data independence principle.

Lastly, languages expressing all classical queries were exhibited. The expressive power is due to their ability to invent new values in the course of the computation. Apart from computational reasons, this feature is very useful in object-oriented languages with object creation. We discussed an extended notion of completeness that takes into account queries with new values in their answer, and pointed out some difficulties in obtaining languages that are complete with respect to this extended class of queries.

We limited ourselves here to languages that operate primarily within the relational framework. Many recent extensions have been left out of this survey. We mention three particularly active areas:

- *Deductive object-oriented languages.* In such languages, object-oriented features are added to rule-based languages. A typical example is the language IQL [AK89]. A different approach (based on rules and Datalog with negation) has been used in [ALUW93] to provide semantics to a number of variations of schemas with methods. The connection between methods and rule-based languages is also considered in [DV91]. The *DOOD* (Deductive Object-Oriented Databases) conference provides a forum for research in this area.

- *Active databases:* these are most often centered around rule-based languages. "Active" databases generally support the automatic triggering of updates as a response to user-requested or system-generated updates. Most active database systems, e.g., [CCCR+90, Coh86, MD89, Han89, SKdM92, SJGP90, WF90] use a paradigm of *rules* to specify the actions to be taken, in a manner reminiscent of expert systems. Active databases and techniques have been shown useful for constraint maintenance [Mor83, CW90, CTF88], incremental update of materialized views [CW91], database security [SJGP90]; and hold the promise of providing new solutions to the view and derived data update problem [CHM94] and issues in database interoperability [CW93, Cha94, Wie92]. See [WC95] for a survey of the main active database systems.

- *Constraint logic programming:* Programming with constraints has been studied for some time in topic areas ranging from linear programming to AI to logic programming. Although the declarative spirit of both constraint programming and database query languages leads to a natural marriage, it is only recently that the combination of the two paradigms has been seriously studied [KKR90]. This was probably a consequence of the success of constraints in the field of logic programming (see e.g., [JL87] and [Lel87, Coh90] for surveys). Recent work on constraint database languages includes [Kup93, GS94].

Overall, the rule-based paradigm has demonstrated its flexibility and vitality. It will most likely continue to provide a useful tool to the computer scientist for the foreseeable future.

References

[ABW88] K.R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, CA, 1988.

[AC89] F. Afrati and S.S. Cosmadakis. Expressiveness of restricted recursive queries. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 113–126, 1989.

[ACY91] F. Afrati, S. Cosmadakis, and M. Yannakakis. On datalog vs. polynomial time. In *Proc. ACM Symp. on Principles of Database Systems*, pages 13–25, 1991.

[AG85] S. Abiteboul and G. Grahne. Update semantics for incomplete databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 1–12, 1985.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, Mass., 1995.

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

[ALUW93] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 32–41, 1993.

[AP92] M. Andries and J. Paredaens. A language for generic graph-transformations. In *Proc. Int'l. Workshop WG 91*, pages 63–74. Springer-Verlag, 1992.

[ASV90] S. Abiteboul, E. Simon, and V. Vianu. Non-deterministic languages to express deterministic transformations. In *Proc. ACM Symp. on Principles of Database Systems*, pages 218–229, 1990.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 110–117, 1979.

[AV88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proc. ACM Symp. on Principles of Database Systems*, pages 240–250, 1988.

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

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

[AV91c] S. Abiteboul and V. Vianu. Non-determinism in logic-based languages. *Annals of Math. and Artif. Int.*, 3:151–186, 1991.

[AVdB95] S. Abiteboul and J. Van den Bussche. Deep equality revisited. In *Proceedings 4th International Conference on Deductive and Object-Oriented Databases*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[AvE82] K. Apt and M. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, 1982.

[BF87] N. Bidoit and C. Froidevaux. Minimalism subsumes default logic and circumscription. In *Proc. IEEE Conf. on Logic in Computer Science*, pages 89–97, 1987.

[BF88] N. Biboit and C. Froidevaux. General logic databases and programs: Default logic semantics and stratification. Technical report, LRI, Université de Paris-Sud, Orsay, 1988. To appear in *J. Information and Computation*.

[Bid91a] N. Bidoit. *Bases de Données Déductives (Présentation de Datalog)*. Armand Colin, Paris, 1991.

[Bid91b] N. Bidoit. Negation in rule-based database languages: A survey. *Theoretical Computer Science*, 78:3–83, 1991.

[BR88] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In M. Stonebraker, editor, *Readings in Database Systems*, pages 507–555. Morgan-Kaufmann, 1988. An earlier version of this work appears in *Proc. ACM SIGMOD Conf. on Management of Data*, pp. 16–52, 1986.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 225–236, 1990.

[CGP93] L. Corciulo, F. Giannotti, and D. Pedreschi. Datalog with non-deterministic choice computes NDB-PTIME. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 93.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer-Verlag, Berlin Heidelberg, 1990.

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

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

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

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

[Cha94] T.-P. Chang. *On Incremental Update Propagation Between Object-based Databases*. PhD thesis, University of Southern Calfornia, Los Angeles, California, 1994.

[CHM94] I-M. A. Chen, R. Hull, and D. McLeod. Local ambiguity and derived data update. In *Fourth Intl. Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 77–86, 1994.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Comm. of the ACM*, 13(6):377–387, 1970.

[Coh86] D. Cohen. Programming by specification and annotation. In *Proc. of AAAI*, 1986.

[Coh90] J. Cohen. Constraint logic programming languages. In *Comm. of the ACM*, volume 33(7), pages 69–90, 1990.

[CTF88] M. A. Casanova, L. Tucherman, and A. L. Furtado. Enforcing inclusion dependencies and referential integrity. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 38–49, 1988.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, 1990.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[CW93] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proc. of Intl. Conf. on Very Large Data Bases*, 1993.

[Dal87] E. Dalhaus. Skolem normal forms concerning the least fixpoint. In E. Börger, editor, *Computation Theory and Logic*, volume 270, pages 101–106. Springer Verlag, Lecture Notes in Computer Science, Berlin/New York, 1987.

[dBGAG92] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating query languages. In *IEEE Conf. on Foundations of Computer Science*, pages 372–379, 1992.

[DV91] K. Denninghoff and V. Vianu. The power of methods with parallel semantics. In *vldb*, pages 221–232, 1991.

[DV93] K. Denninghoff and V. Vianu. Database method schemas and object creation. In *Proc. ACM Symp. on Principles of Database Systems*, pages 265–275, 1993.

[For81] C. L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, 1981.

[Gel86] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In *IEEE Symp. on Logic Programming*, pages 127–139, 1986.

[Gel89] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. ACM Symp. on Principles of Database Systems*, pages 1–11, 1989.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programs. In *Intl. Conf. on Logic Programming*, pages 1070–1080, 1988.

[GPSZ91] F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Nondeterminism in deductive databases. In *Proc. of Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 129–146, Los Altos, CA, 1991. Springer Verlag, Lecture Notes in Computer Science 566.

[GRS88] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. In *Proc. ACM Symp. on Principles of Database Systems*, pages 221–230, 1988.

[GRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:620–650, 1991.

[GS94] S. Grumbach and J. Su. Finitely representable databases. In *Proc. ACM Symp. on Principles of Database Systems*, 1994.

[Han89] E.H. Hanson. An initial report on the design of ariel: a dbms with an integrated production rule system. In *Sigmod Record, 18(3)*, pages 12–19, 1989.

[HY84] R. Hull and C. K. Yap. The Format model: A theory of database organization. *Journal of the ACM*, 31(3):518–537, 1984.

[IL84] T. Imielinski and W. Lipski. The relational model of data and cylindric algebras. *Journal of Computer and System Sciences*, 28(1):80–102, 1984.

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

[IN88] T. Imielinski and S. Naqvi. Explicit control of logic programs through rule algebra. In *Proc. ACM Symp. on Principles of Database Systems*, pages 103–116, 1988.

[JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

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

[KKR90] P. Kanellakis, G Kuper, and P. Revesz. Constraint query languages. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pages 299–313, Nashville, 1990.

[KN88] R. Krishnamurthy and S.A. Naqvi. Nondeterministic choice in datalog. In *5th Int'l. Conf. on Data and Knowledge Bases*, pages 416–424, Los Altos, CA, 1988. Morgan Kaufmann.

[Kol91] P. G. Kolaitis. The expressive power of stratified logic programs. *Information and Computation*, 90(1):50–66, 1991.

[Kon88] S. Konolige. On the relation between default and autoepistemic logic. *Artificial Intelligence*, 35(3):343–382, 1988.

[KP88] P. G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint? In *Proc. ACM Symp. on Principles of Database Systems*, pages 231–239, 1988.

[Kup93] G.M. Kuper. Aggregation in constraint databases. In *Proc. First Workshop on Principles and Practice of Constraint Programming*, 1993.

[KV90] P. G. Kolaitis and M. Y. Vardi. On the expressive power of datalog: tools and a case study. In *Proc. ACM Symp. on Principles of Database Systems*, pages 61–71, 1990.

[Lel87] W. Leler. *Constraint Programming Languages.* Addison Wesley, 1987.

[Lif88] V. Lifschitz. On the declarative semantics of logic programs with negation. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 177–192. Morgan Kaufmann, Los Altos, CA, 1988.

[LM89] V.S. Lakshmanan and A.O. Mendelzon. Inductive pebble games and the inductive power of datalog. In *Proc. ACM Symp. on Principles of Database Systems*, pages 301–311, 1989.

[MD89] D. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 215–224, 1989.

[Moo85] R.C. Moore. Semantics considerations on non-monotonic logic. *Artificial Intelligence*, 25:75–94, 1985.

[Mor83] M. Morgenstern. Active databaswes as a paradigm for enhanced computing environments. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 34–42, 1983.

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

[MW88] D. Maier and D. S. Warren. *Computing with Logic: Logic Programming with Prolog*. Benjamin Cummings, Menlo Park, CA, 1988.

[NT89] S. Naqvi and S. Tsur. *A language for data and knowledge bases*. Computer Science Press, Rockville, Maryland, 1989.

[Pap85] C.P. Papadimitriou. A note on the expressive power of prolog. *Bulletin of the EATCS*, 26:21–23, 1985.

[Prz89] T. Przymusinski. Every logic program has a natural stratification and an iterated least fixpoint model. In *Proc. ACM Symp. on Principles of Database Systems*, pages 11–21, 1989.

[Prz90] T. Przymusinski. Well-founded semantics coincides with three-valued stable semantics. *Fundamenta Informaticae*, XIII:445–463, 1990.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1):80–132, 1980.

[RU95] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *J. Logic Programming*, pages 125–149, 1995.

[Shm87] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proc. ACM Symp. on Principles of Database Systems*, pages 237–249, 1987.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 281–290, 1990.

[SKdM92] E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on top of a relational dbms. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 315–326, 1992.

[SZ90] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. ACM Symp. on Principles of Database Systems*, pages 205–217, 1990.

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

[Ull89a] J. D. Ullman. Bottom-up beats top-down for datalog. In *Proc. ACM Symp. on Principles of Database Systems*, pages 140–149, 1989.

[Ull89b] J.D. Ullman. *Principles of Database and Knowledge Base Systems, Volume II: The New Technologies.* Computer Science Press, 1989.

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

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[WC95] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan-Kaufmann, Inc, San Francisco, California, To appear, 1995.

[WF90] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 259–264, 1990.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.