Complexity and Expressive Power of Logic Programming

Evgeny Dantsin (University of Manchester, United Kingdom)
and
Thomas Eiter (Technische Universität Wien, Austria)
and
Georg Gottlob (Technische Universität Wien, Austria)
and
Andrei Voronkov (University of Manchester, United Kingdom)

Association for Computing Machinery, Inc., 1515 Broadway, New York, NY 10036, USA
Tel: (212) 555-1212; Fax: (212) 555-2000

This paper surveys various complexity and expressiveness results on different forms of logic programming. The main focus is on decidable forms of logic programming, in particular, propositional logic programming and datalog, but we also mention general logic programming with function symbols. Next to classical results on plain logic programming (pure Horn clause programs), more recent results on various important extensions of logic programming are surveyed. These include logic programming with different forms of negation, disjunctive logic programming, logic programming with equality, and constraint logic programming.

Categories and Subject Descriptors: F.1.3 [**Theory of Computation**]: Complexity Measures and Classes—*Machine-Independent Complexity*; F.4.1 [**Theory of Computation**]: Mathematical Logic—*Computational Logic*; H.2.3 [**Information Management**]: Languages—*Query Languages*

General Terms: logic programming, expressive power, complexity

Additional Key Words and Phrases: query languages, nonmonotonic logic, datalog

Contents

This article is an extended version of [Dantsin et al. 1997].

1. INTRODUCTION

Logic programming is a well-known declarative method of knowledge representation and programming based on the idea that the language of first-order logic is well-suited for both representing data and describing desired outputs [Kowalski 1974]. Logic programming was developed in the early 1970's based on work in automated theorem proving [Green 1969; Kowalski and Kuehner 1971], in particular, on Robinson's *resolution principle* [Robinson 1965].

A pure logic program consists of a set of *rules*, also called definite Horn clauses. Each such rule has the form *head* $\leftarrow$ *body*, where *head* is a logical atom and *body* is a conjunction of logical atoms. The logical semantics of such a rule is given by the implication *body* $\Rightarrow$ *head* (for a more precise account, see Section 2). Note that the semantics of a pure logic program is completely independent of the order in which its clauses are given, and of the order of the single atoms in each rule body.

With the advent of the programming language Prolog [Colmerauer et al. 1973], the paradigm of logic programming became soon ready for practical use. Many applications in different areas were and are successfully implemented in Prolog. Note that Prolog is — in a sense — only an approximation to fully declarative logic programming. In fact, the clause matching and backtracking algorithms at the core of Prolog are sensitive to the ordering of the clauses in a program and of the atoms in a rule body.

While Prolog has become a popular programming language taught in many computer science curricula, research focuses more on pure logic programming and on extensions thereof. Even in some application areas such as *knowledge representation* (a subfield of artificial intelligence) and *databases* there is a predominant need for full declarativeness, and hence for pure logic programming. In knowledge representation, declarative extensions of pure logic programming, such as negation in rule bodies and disjunction in rule heads, are used to formalize common sense reasoning. In the database context, the query language *datalog* was designed and intensively studied (see [Ullman 1988; Ullman 1989; Ceri et al. 1990]).

There are many interesting complexity results on logic programming. These results are not limited to "classical" complexity theory but also comprise expressiveness results in the sense of *descriptive complexity theory*. For example, it was shown that (a slight extension of) datalog cannot just express *some*, but actually *all* polynomially computable queries on ordered databases and only those. Thus datalog precisely *expresses* or *captures* the complexity class P on ordered databases. Similar results were obtained for many variants and extensions of datalog. It turned out that all major variants of datalog can be characterized by suitable complexity classes. As a consequence, complexity theory has become a very important tool for comparing logic programming formalisms.

This paper surveys various complexity and expressiveness results on different forms of (purely declarative) logic programming. The aim of the paper is twofold. First, a broad survey and many pointers to the literature are given. Second, in order to give a flavor of complexity issues in logic programming, a few fundamental topics are explained in greater detail, in particular, the basic results on plain logic programming (Section 4) and some fundamental issues related to descriptive complexity (Section 7). These two sections are written in a more tutorial style and contain several proofs, while the other sections are written in a rather succinct survey style.

Note that the present paper does not consist of an encyclopedic listing of all published complexity results on logic programming, but rather of a more or less subjective choice of results. Many interesting results are not mentioned for space reasons, e.g., results on abductive logic programming [Eiter et al. 1997a; Inoue and Sakama 1993; Sakama and Inoue 1994b; Marek et al. 1996], on intuitionistic logic programming [Bonner 1990; Bonner 1997], and on Prolog [Dikovsky 1993]; see also other surveys, e.g., [Cadoli and Schaerf 1993; Schlipf 1995a].

The paper is organized as follows. Section 2 defines syntax and semantics of logic programs, describe datalog and introduce complexity measures. Computational models and complexity notation are discussed in Section 3. Section 4 presents the main complexity results on plain logic programming and datalog. Section 5 discusses various semantics for logic programming with negation and respective complexity results. Section 6 deals with disjunctive logic programming. Section 7 studies the expressive power of datalog and logic programming with complex values. Section 8 characterizes the complexity of unification. Section 9 deals with logic programming extended by equality. Finally, Section 10 describes complexity results on constraint logic programming.

This article is an extended version of [Dantsin et al. 1997].

2. PRELIMINARIES

In this section, we introduce some basic concepts of logic programming. Due to space reasons, the presentation is necessarily succinct; for a more detailed treatment, see [Lloyd 1987; Apt 1990; Apt and Bol 1994; Baral and Gelfond 1994].

We use letters $p, q, \ldots$ for predicate symbols, $X, Y, Z, \ldots$ for variables, $f, g, h, \ldots$ for function symbols, and $a, b, c, \ldots$ for constants; a bold face version of a letter denotes a list of symbols of the respective type. In logic programs, we sometimes denote predicate and function symbols by arbitrary strings.

2.1 Syntax of logic programs

Logic programs are formulated in a language $\mathcal{L}$ of *predicates* and *functions* of non-negative arity; 0-ary functions are *constants*. A language $\mathcal{L}$ is *function-free* if it contains no functions of arity greater than 0.

A *term* is inductively defined as follows: each variable X and each constant c is a term, and if f is an n-ary function symbol and $t_1, \ldots\ t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. A term is *ground* if no variable occurs in it. The *Herbrand universe* of $\mathcal{L}$, denoted $U_{\mathcal{L}}$, is the set of all ground terms which can be formed with the functions and constants in $\mathcal{L}$.

An *atom* is a formula $p(t_1, \ldots, t_n)$, where p is a *predicate* symbol of arity n and each t_i is a term. An atom is *ground* if all t_i are ground. The *Herbrand base* of a language $\mathcal{L}$, denoted $B_{\mathcal{L}}$, is the set of all ground atoms that can be formed with predicates from $\mathcal{L}$ and terms from $U_{\mathcal{L}}$.

A *Horn clause* is a rule of the form

$$A_0 \leftarrow A_1, \ldots, A_m \qquad (m \geq 0),$$

where each A_i is an atom. The parts on the left and on the right of "$\leftarrow$" are called the *head* and the *body* of the rule, respectively. A rule r of the form $A_0 \leftarrow$, i.e., whose body is empty, is called a *fact*, and if A_0 is a ground atom, then r is called a *ground fact*.

A *logic program* is a finite set of Horn clauses. A clause or logic program is *ground* if it contains no variables.

With each logic program P, we associate the language $\mathcal{L}(P)$ that consists of the predicates, functions and constants occurring in P. If no constant occurs in P, we add some constant to $\mathcal{L}(P)$ to have a non-empty domain. Unless stated otherwise,

$\mathcal{L}(P)$ is the underlying language, and we use simplified notation U_P and B_P for $U_{\mathcal{L}(P)}$ and $B_{\mathcal{L}(P)}$, respectively.

A *Herbrand interpretation* of a logic program P is any subset $I \subseteq B_P$ of its Herbrand base. Intuitively, the atoms in I are true, while all others are false. A *Herbrand model* of P is a Herbrand interpretation of P such that for each rule $A_0 \leftarrow A_1, \ldots, A_m$ in P, this interpretation satisfies the logical formula $\forall \mathbf{X}((A_1 \wedge \cdots \wedge A_m) \Rightarrow A_0)$, where $\mathbf{X}$ is a list of the variables in the rule.

Propositional logic programs are logic programs in which all predicates have arity 0, i.e., all atoms are propositional ones.

Example 1. Here is an example of a propositional logic program, which captures knowledge (in a simplified form) about a steam engine equipped with three signal gauges.

$$
\begin{aligned}
\mathit{shut\_down} &\leftarrow \mathit{overheat}\\
\mathit{shut\_down} &\leftarrow \mathit{leak}\\
\mathit{leak} &\leftarrow \mathit{valve\_closed}, \mathit{pressure\_loss}\\
\mathit{valve\_closed} &\leftarrow \mathit{signal\_1}\\
\mathit{pressure\_loss} &\leftarrow \mathit{signal\_2}\\
\mathit{overheat} &\leftarrow \mathit{signal\_3}\\
\mathit{signal\_1} &\leftarrow\\
\mathit{signal\_2} &\leftarrow
\end{aligned}
$$

Informally, the rules of the program tell that the system has to be shut down if it is in a dangerous state. Such states are connected to causes and signals by respective rules. The facts *signal_1* and *signal_2* state that signals #1 and #2, respectively, are being observed.

Note that if P is a propositional logic program then B_P is a set of propositional atoms. Any interpretation of P is a subset of B_P.

2.2 Semantics of logic programs

The notions of a Herbrand interpretation and model can be generalized for infinite sets of clauses in a natural way. Let P be a set (finite or infinite) of ground clauses. Such a set P defines an operator $T_P : 2^{B_P} \to 2^{B_P}$, where 2^{B_P} denotes the set of all Herbrand interpretations of P, by

$$
\begin{aligned}
T_P(I) = \{A_0 \in B_P \mid\ & P \text{ contains a rule } A_0 \leftarrow A_1, \ldots, A_m\\
& \text{such that } \{A_1, \ldots, A_m\} \subseteq I \textit{ holds } \}.
\end{aligned}
$$

This operator is called the *immediate consequence operator*; intuitively, it yields all atoms that can be derived by a single application of some rule in P given the atoms in I.

Since T_P is monotone, by the Knaster-Tarski Theorem it has a least fixpoint, denoted by T_P^∞; since, moreover, T_P is also continuous, by Kleene's Theorem T_P^∞ is the limit of the sequence $T_P^0 = \emptyset$, $T_P^{i+1} = T_P(T_P^i)$, $i \geq 0$.

A ground atom A is called a *consequence* of a set P of clauses if $A \in T_P^\infty$ (we write $P \models A$). Also, we say that a negated ground atom $\neg A$ is a *consequence* of P and write $P \models \neg A$ if $A \notin T_P^\infty$. Note that $\models$ differs from the standard logical consequence relation. The *semantics* of a set P of ground clauses, denoted $\mathcal{M}(P)$, is defined as the following set consisting of atoms and negated atoms:

$$\begin{aligned}\mathcal{M}(P) &= \{A \mid P \models A\} \cup \{\neg A \mid P \models \neg A\} \\ &= T_P^\infty \cup \{\neg A \mid A \in B_P \setminus T_P^\infty\}.\end{aligned}$$

Example 2. (See Example 1.) For the program P above, we have

$$\begin{aligned}&T_P^0 = \emptyset, \\ &T_P^1 = \{\mathit{signal\_1}, \mathit{signal\_2}\}, \\ &T_P^2 = T_P^1 \cup \{\mathit{valve\_closed}, \mathit{pressure\_loss}\}, \\ &T_P^3 = T_P^2 \cup \{\mathit{leak}\}, \\ &T_P^4 = T_P^\infty = T_P^3 \cup \{\mathit{shutdown}\}.\end{aligned}$$

Thus, the least fixpoint is reached in four steps; e.g., $P \models \mathit{shutdown}$ and $P \models \neg \mathit{overheat}$.

For each set P of clauses, T_P^∞ coincides with the unique *least Herbrand model of* P, where a model M is smaller than a model N, if M is a proper subset of N [van Emden and Kowalski 1976].

The semantics of nonpropositional logic programs is now defined as follows. Let the *grounding* of a clause r in a language $\mathcal{L}$, denoted $\mathit{ground}(r, \mathcal{L})$, be the set of all clauses obtained from r by all possible substitutions of elements of $U_\mathcal{L}$ for the variables in r. For any logic program P, we define

$$\mathit{ground}(P, \mathcal{L}) = \bigcup_{r \in P} \mathit{ground}(r, \mathcal{L}),$$

and we write $\mathit{ground}(P)$ for $\mathit{ground}(P, \mathcal{L}(P))$. The operator $T_P : 2^{B_P} \to 2^{B_P}$ associated with P is defined by $T_P = T_{\mathit{ground}(P)}$. Accordingly, $\mathcal{M}(P) = \mathcal{M}(\mathit{ground}(P))$.

Example 3. Let P be the program

$$\begin{aligned}p(a) &\leftarrow \\ p(f(x)) &\leftarrow p(x)\end{aligned}$$

Then, $U_P = \{a, f(a), f(f(a)), \ldots\}$ and $\mathit{ground}(P)$ contains the clauses $p(a) \leftarrow$, $p(f(a)) \leftarrow p(a)$, $p(f(f(a))) \leftarrow p(f(a))$, ... The least fixpoint of T_P is

$$T_P^\infty = T_{\mathit{ground}(P)}^\infty = \{p(f^n(a)) \mid n \geq 0\}.$$

Hence, e.g., $P \models p(f(f(a)))$.

In practice, generating $\mathit{ground}(P)$ is often cumbersome, since, even in case of function-free languages, it is in general exponential in the size of P. Moreover, it is not always necessary to compute $\mathcal{M}(P)$ in order to determine whether $P \models A$ for some particular atom A. For these reasons, completely different strategies of deriving atoms from a logic program have been developed. These strategies are based on variants of the famous *Resolution Principle* of [Robinson 1965]. The major variant is SLD-resolution [Kowalski and Kuehner 1971; Apt and van Emden 1982].

Roughly, SLD-resolution can be described as follows. A *goal* is a conjunction of atoms, and a *substitution* is a function ϑ that maps variables $v_1, \ldots, v_n$ to terms

$t_1, \ldots, t_n$. The result of simultaneous replacement of variables v_i by terms t_i in an expression E is denoted by $E\vartheta$. For a given goal G and a program P, SLD-resolution tries to find a substitution ϑ such that $G\vartheta$ logically follows from P. The initial goal is repeatedly transformed until the empty goal is obtained. Each transformation step is based on the application of the resolution rule to a *selected atom* B_i from the goal $B_1, \ldots, B_m$ and a clause $A_0 \leftarrow A_1, \ldots, A_n$ from P. SLD-resolution tries to *unify* B_i with the head A_0, i.e., to find a substitution ϑ such that $A_0\vartheta = B_i\vartheta$. Such a substitution ϑ is called a *unifier* of A_0 and B_i. If a unifier ϑ exists, a most general such ϑ (which is essentially unique) is chosen and the goal is transformed into

$$(B_1, \ldots, B_{i-1}, A_1, \ldots, A_n, B_{i+1}, \ldots, B_m)\vartheta.$$

For a more precise account see [Apt 1990; Lloyd 1987]; for resolution on general clauses, see e.g., [Leitsch 1997]. The complexity of unification will be dealt with in Section 8.

2.3 Datalog

The interest in using logic in databases gave rise to the field of deductive databases; see [Minker 1996] for a comprehensive overview of the development of this area. It appeared that logic programming is a suitable formalism for querying relational databases. In this context, the logic programming based query language datalog and various extensions thereof have been defined.

In the context of logic programming, relational databases are identified with sets of ground facts $p(c_1, \ldots, c_n)$. Intuitively, all ground facts with the same predicate symbol p represent a data relation. The set of all predicate symbols occurring in the database together with a possibly infinite *domain* for the argument constants is called the *schema* of the database. With each database D, we associate a finite universe U_D of constants which encompasses at least all constants appearing in D, but possibly more. In the classical database context, U_D is often identified with the set of all constants appearing in D. But in the datalog context, a larger universe U_D may be suitable in case one wants to derive assertions about items that do not explicitly occur in the database.

To understand how datalog works, let us consider a clarifying example.

Example 4. Consider a database D containing the ground facts

$$\begin{aligned} \mathit{father}(\mathit{john}, \mathit{mary}) &\leftarrow \\ \mathit{father}(\mathit{joe}, \mathit{kurt}) &\leftarrow \\ \mathit{mother}(\mathit{mary}, \mathit{joe}) &\leftarrow \\ \mathit{mother}(\mathit{tina}, \mathit{kurt}) &\leftarrow \end{aligned}$$

The schema of this database is the set of relation symbols $\{\mathit{father}, \mathit{mother}\}$ together with the domain *STRING* of all alphanumeric strings. With this database, we associate the finite universe $U_D = \{\ \mathit{john}, \mathit{mary}, \mathit{joe}, \mathit{tina}, \mathit{kurt}, \mathit{susan}\}$. Note that *susan* does not appear in the database but is included in the universe U_D.

The following datalog program (or query) P computes all ancestor relationships relative to this database:

$$
\begin{aligned}
parent(X,Y) &\leftarrow father(X,Y)\\
parent(X,Y) &\leftarrow mother(X,Y)\\
ancestor(X,Y) &\leftarrow parent(X,Y)\\
ancestor(X,Y) &\leftarrow parent(X,Z), ancestor(Z,Y)\\
person(X) &\leftarrow
\end{aligned}
$$

In the program P, *father* and *mother* are the *input predicates*, also called *database predicates*. Their interpretation is fixed by the given input database D. The predicates *ancestor* and *person* are *output predicates*, and the predicate *parent* is an *auxiliary predicate*. Intuitively, the output predicates are those which are computed as the visible result of the query, while the auxiliary predicates are introduced for representing some intermediate results, which are not to be considered part of the final result.

The datalog program P on input database D computes a result database R with the schema $\{ancestor, person\}$ containing among others the following ground facts:

$$
\begin{aligned}
&ancestor(mary, joe),\\
&ancestor(john, joe),\\
&person(john),\\
&person(susan).
\end{aligned}
$$

The last fact is in R because *susan* is included as a constant in U_D. However, the fact *person*(*harry*) is not in R, because *harry* is not a constant in the finite universe U_D of the database D.

Formally, a *database schema* $\mathcal{D}$ consists of a finite set $Rels(\mathcal{D})$ of relation names with associated arities and a (possibly countable infinite) domain $Dom(\mathcal{D})$. For each database schema $\mathcal{D}$, we denote by $HB(\mathcal{D})$ the Herbrand base corresponding to the function-free language whose predicate symbols are $Rels(\mathcal{D})$ and whose constant symbols are $Dom(\mathcal{D})$.

A *database* (also, *database instance*) D over a schema $\mathcal{D}$ is given by a finite subset of the Herbrand base $D \subseteq HB(\mathcal{D})$ together with an associated finite universe $U_D \subseteq Dom(\mathcal{D})$, containing all constants actually appearing in D. By abuse of notation, we also write D instead of $\langle D, U_D\rangle$. We denote by $D|p$ the extension of the relation $p \in Rels(\mathcal{D})$ in D. Moreover, $INST(\mathcal{D})$ denotes the set of all databases over $\mathcal{D}$.

A *datalog query* or a *datalog program* is a function-free logic program P with three associated database schemas: the input schema $\mathcal{D}_{in}$, the output schema $\mathcal{D}_{out}$ and the complete schema $\mathcal{D}$, such that the following is satisfied:

$$
\begin{aligned}
&Dom(\mathcal{D}_{in}) = Dom(\mathcal{D}_{out}) = Dom(\mathcal{D}),\\
&Rels(\mathcal{D}_{in}) \subseteq Rels(\mathcal{D}),\\
&Rels(\mathcal{D}_{out}) \subseteq Rels(\mathcal{D}), \text{ and}\\
&Rels(\mathcal{D}_{in}) \cap Rels(\mathcal{D}_{out}) = \emptyset.
\end{aligned}
$$

Moreover, each predicate symbol appearing in P is contained in $Rels(\mathcal{D})$ and no predicate symbol from $\mathcal{D}_{in}$ appears in a rule head of P (the latter means that the predicates of the input database are never redefined by a datalog program).

The formal semantics of a datalog program P over the input schema $\mathcal{D}_{in}$, output schema $\mathcal{D}_{out}$, and complete schema $\mathcal{D}$ is given by a partial mapping from instances of $\mathcal{D}_{in}$ to instances of $\mathcal{D}_{out}$ over the same universe. A result instance of $\mathcal{D}_{out}$ is regarded as the result of the query. More formally, $\mathcal{M}_P : \mathit{INST}(\mathcal{D}_{in}) \to \mathit{INST}(\mathcal{D}_{out})$ is defined for all instances $D_{in} \in \mathit{INST}(\mathcal{D}_{in})$ such that all constants occurring in P appear in $U_{\mathcal{D}_{in}}$, and maps every such D_{in} to the database $D_{out} = \mathcal{M}_P(D_{in})$ such that $U_{D_{out}} = U_{D_{in}}$ and, for every relation $p \in \mathit{Rels}(\mathcal{D}_{out})$,

$$D_{out}|p = \{\mathbf{a} \mid p(\mathbf{a}) \in \mathcal{M}(\mathit{ground}(P \cup D_{in}, \mathcal{L}(P, D_{in})))\},$$

where $\mathcal{M}$ and *ground* are defined as in Section 2.2, and $\mathcal{L}(P, D_{in})$ is the language of P extended by all constants in the universe $U_{D_{in}}$. For all ground atoms $A \in \mathit{HB}(D_{out})$, we write $P \cup D_{in} \models A$ if $A \in \mathcal{M}_P(D_{in})$ and write $P \cup D_{in} \models \neg A$ if $A \notin \mathcal{M}_P(D_{in})$.

The semantics of datalog is thus *inherited* from the semantics of logic programming. In a similar way, the semantics of various extensions of datalog is inherited from the corresponding extensions of logic programming.

There are three main kinds of complexity connected to plain datalog and its various extensions [Vardi 1982]:

- The ***data complexity*** is the complexity of checking whether $D_{in} \cup P \models A$ when datalog programs P are *fixed*, while input databases D_{in} and ground atoms A are an *input*.
- The ***program complexity*** (also called *expression complexity*) is the complexity of checking whether $D_{in} \cup P \models A$ when input databases D_{in} are *fixed*, while datalog programs P and ground atoms A are an *input*.
- The ***combined complexity*** is the complexity of checking whether $D_{in} \cup P \models A$ when input databases D_{in}, datalog programs P and ground atoms A are an *input*.

Note that for plain datalog, as well as for all other versions of datalog considered in this paper, the combined complexity is equivalent to the program complexity with respect to polynomial-time reductions. This is due to the fact that with respect to the derivation of ground atoms, each pair $\langle D_{in}, P\rangle$ can be easily reduced to the pair $\langle D_\emptyset, P^*\rangle$, where $D_\emptyset$ is the empty database instance associated with a universe of two constants c_1 and c_2, and P^* is obtained from $P \cup D_{in}$ by straightforward encoding of the universe $U_{D_{in}}$ using n-tuples over $\{c_1, c_2\}$, where $n = \lceil |U_{D_{in}}| \rceil$. For this reason, we mostly disregard the combined complexity in the material concerning datalog. We remark, however, that due to a fixed universe, program complexity may allow for slightly sharper upper bounds than the combined complexity (e.g., ETIME vs EXPTIME).

Another approach to measuring complexity of query languages is the *parametric complexity* [Papadimitriou and Yannakakis 1997]. In this approach, the complexity is expressed as a function of some "reasonable" *parameters*. An example of such a parameter is the number of variables appearing in the query (interest in this parameter is motivated by [Vardi 1995], where it is shown that data and program complexity become close when the number of query variables is bounded).

As for logic programming in general, a generalization of the combined complexity may be regarded as the main complexity measure. Below, when we speak about the complexity of a fragment of logic programming, we mean the following kind of complexity:

- The ***complexity*** of (some fragment of) logic programming is the complexity of checking whether $P \models A$ for *variable* both programs P and ground atoms A.

3. COMPLEXITY CLASSES

This section contains definitions of the standard complexity classes encountered in this survey and provides other related definitions (we follow the notation of [Johnson 1990]). A detailed exposition of most complexity notions can be found e.g. in [Papadimitriou 1994].

3.1 Turing machines

Deterministic Turing machines.. Informally, we think of a Turing machine as a device able to read from and write on a semi-infinite *tape*, whose contents may be locally accessed and changed in a computation. Formally, a *deterministic Turing machine (DTM)* is defined as a quadruple (S, Σ, δ, s_0), where S is a finite set of *states*, Σ is a finite alphabet of *symbols*, δ is a *transition function*, and $s_0 \in S$ is the *initial state*. The alphabet Σ contains a special symbol ␣ called the *blank*. The transition function δ is a map

$$\delta : \quad S \times \Sigma \quad \rightarrow \quad (S \cup \{\mathtt{halt}, \mathtt{yes}, \mathtt{no}\}) \times \Sigma \times \{\mathtt{-1},\ \mathtt{0},\ \mathtt{+1}\},$$

where `halt`, `yes`, and `no` denote three additional states not occurring in S, and `-1, 0, +1` denote *motion directions*. It is assumed here, without loss of generality, that the machine is well-behaved and never moves off the tape, i.e., $d \neq \mathtt{-1}$ whenever the cursor is on the leftmost cell; this can be ensured by proper design of δ.[1]

Let T be a DTM (Σ, S, δ, s_0). The tape of T is divided into *cells* containing symbols of Σ. There is a *cursor* that may move along the tape. At the start, T is in the initial state s_0, and the cursor points to the leftmost cell of the tape. An *input string* I is written on the tape as follows: the first $|I|$ cells $c_0, \ldots, c_{|I|-1}$ of the tape, where $|I|$ denotes the length of I, contains the symbols of I, and all other cells contain ␣.

The machine takes successive *steps* of computation according to δ. Namely, assume that T is in a state $s \in S$ and the cursor points to the symbol $\sigma \in \Sigma$ on the tape. Let

$$\delta(s, \sigma) = (s', \sigma', d).$$

Then T changes its current state to s', overwrites σ' on σ, and moves the cursor according to d. Namely, if $d = \mathtt{-1}$ or $d = \mathtt{+1}$, then the cursor moves to the previous cell or the next one, respectively; if $d = \mathtt{0}$, then the cursor remains in the same position.

When any of the states `halt`, `yes` or `no` is reached, T halts. We say that T *accepts* the input I if T halts in `yes`. Similarly, we say that T *rejects* the input in the case of halting in `no`. If `halt` is reached, we say that the *output* of T on I is

[1] Some texts assume that Σ has a special symbol which marks the left end of the tape. This symbol can be eliminated by a proper redesign of the machine. For the purpose of this paper, the simpler model without a left end marker is convenient.

computed. This output, denoted by $T(I)$, is defined as the string contained in the initial segment of the tape which ends before the first blank.

Nondeterministic Turing machines.. Like a DTM, a *nondeterministic Turing machine, or NDTM,* is defined as a quadruple (S, Σ, Δ, s_0), where S, Σ, s_0 are the same as before. Possible operations of the machine are described by Δ, which is no longer a function. Instead, Δ is a relation:

$$\Delta \subseteq (S \times \Sigma) \times (S \cup \{\texttt{halt}, \texttt{yes}, \texttt{no}\}) \times \Sigma \times \{\texttt{-1, 0, +1}\}.$$

A tuple whose first two members are s and σ respectively, specifies the action of the NDTM when its current state is s and the symbol pointed at by its cursor is σ. If the number of such tuples is greater than one, the NDTM nondeterministically chooses any of them and operates accordingly.

Unlike the case of a DTM, the definition of acceptance and rejection by a NDTM is asymmetric. We say that a NDTM *accepts* an input if there is at least one sequence of choices leading to the state `yes`. A NDTM *rejects* an input if no sequence of choices can lead to `yes`.

Time and space bounds.. The *time* expended by a DTM T on an input I is defined as the number of steps taken by T on I from the start to halting. If T does not halt on I, the time is considered to be infinite. For a NDTM T, we define the *time* expended by T on I as 1, if T does not accept I (respectively, computes no output for I), and otherwise as the minimum over the number of steps in any accepting (respectively, output producing) computation of T.

The *space* required by a DTM T on I is the number of cells visited by the cursor during the computation on I. In the case of a NDTM, the *space* is defined as 1, if T does not accept I (respectively, computes no output for I), and otherwise as the minimum number of cells visited on the tape over all accepting (respectively, output producing) computations.

Let T be a DTM or a NDTM. Let f be a function from the positive integers to themselves. We say that T *halts in time* $O(f(n))$ if there exist positive integers c and n_0 such that the time expended by T on any input of length n is not greater than $cf(n)$ for all $n \geq n_0$. Likewise, we say that T halts *within space* $O(f(n))$ if the space required by T on any input of length n is not greater than $cf(n)$ for all $n \geq n_0$, where c and n_0 are positive integers.

Assume that a DTM (NDTM) T halts in time $O(n^d)$, where d is a positive integer. Then we call T a *polynomial-time DTM (NDTM)* and we say that T halts in *polynomial time*. Similarly, if T halts within space $O(n^d)$, we call T a *polynomial-space DTM (NDTM)*.

3.2 Notation for complexity classes

As above, let Σ be a finite alphabet containing $\sqcup$. Let $\Sigma' = \Sigma \setminus \{\sqcup\}$, and let $L \subseteq \Sigma'^*$ be a *language* in Σ', i.e. a set of finite strings over Σ'. Let T be a DTM or a NDTM such that (i) if $x \in L$ then T accepts x, and (ii) if $x \notin L$ then T rejects x. Then we say that T *decides* L. In addition, if T halts in time $O(f(n))$, we say that T decides L *in time* $O(f(n))$. Similarly, if T halts within space $O(f(n))$, we say that T decides L *within space* $O(f(n))$.

Observe that if $f(n)$ is a sublinear function, then a Turing machine which halts within space $f(n)$ can not read the whole input string, nor produce a large output. To remedy this problem, a Turing machine T is equipped with a read-only input-tape and a write-only output tape besides the work tape, which contain the input string and the output computed by T, respectively. The time and space requirement of T is defined as above, where only the space used on the work tape counts. In case T halts within sublinear time $f(n)$, random access to the input symbols on the input-tape is provided using a further tape which serves as an index register. In the following, we assume that multi-tape machines as described may be used for deciding languages within sublinear bounds.

Let f be a function on positive integers. We define the following sets of languages:

$$\begin{aligned}
\mathsf{TIME}(f(n)) &= \{L \mid L \text{ is decided by some DTM in time } O(f(n))\},\\
\mathsf{NTIME}(f(n)) &= \{L \mid L \text{ is decided by some NDTM in time } O(f(n))\},\\
\mathsf{SPACE}(f(n)) &= \{L \mid L \text{ is decided by some DTM within space } O(f(n))\},\\
\mathsf{NSPACE}(f(n)) &= \{L \mid L \text{ is decided by some NDTM within space } O(f(n))\}.
\end{aligned}$$

All these sets are examples of *complexity classes*, other examples will be given below. Note that some functions f can lead to complexity classes with unnatural properties (see [Papadimitriou 1994] for details). However, for "normal" functions such as polynomials, exponents or logarithms, the corresponding complexity classes are "normal" too.

Complexity classes of most interest are not classes corresponding to particular functions but their unions such as, for example, the union $\bigcup_{d>0}\mathsf{TIME}(n^d)$ taken over all polynomials of the form n^d. The following abbreviations are used to denote main complexity classes of such a kind:

$$\begin{aligned}
\mathsf{P} &= \textstyle\bigcup_{d>0}\mathsf{TIME}(n^d),\\
\mathsf{NP} &= \textstyle\bigcup_{d>0}\mathsf{NTIME}(n^d),\\
\mathsf{EXPTIME} &= \textstyle\bigcup_{d>0}\mathsf{TIME}(2^{n^d}),\\
\mathsf{NEXPTIME} &= \textstyle\bigcup_{d>0}\mathsf{NTIME}(2^{n^d}),\\
\mathsf{PSPACE} &= \textstyle\bigcup_{d>0}\mathsf{SPACE}(n^d),\\
\mathsf{EXPSPACE} &= \textstyle\bigcup_{d>0}\mathsf{SPACE}(2^{n^d}),\\
\mathsf{L} &= \mathsf{SPACE}(\log n),\\
\mathsf{NL} &= \mathsf{NSPACE}(\log n).
\end{aligned}$$

The list contains no abbreviations for the nondeterministic counterparts of PSPACE and $\mathsf{EXPSPACE}$ because $\bigcup_{d>0}\mathsf{NSPACE}(n^d)$ coincides with the class PSPACE and $\bigcup_{d>0}\mathsf{NSPACE}(2^{n^d})$ coincides with the class $\mathsf{EXPSPACE}$ [Savitch 1970].

Complementary classes.. Any complexity class $\mathcal{C}$ has its *complementary class* denoted by co-$\mathcal{C}$ and defined as follows. For every language L in Σ', let $\overline{L}$ denote its *complement*, i.e. the set $\Sigma'^* \setminus L$. Then co-$\mathcal{C}$ is $\{\overline{L} \mid L \in \mathcal{C}\}$.

The polynomial hierarchy.. To define the polynomial hierarchy classes, we need to introduce oracle Turing machines. Let A be a language. An *oracle DTM* T^A, also called a *DTM with oracle* A, can be thought of as an ordinary DTM augmented by an additional write-only *query tape* and additional three states `query`, $\in$ and $\notin$. When T^A is not in the state `query`, the computation proceeds as usual (in addition, T^A can write on the query tape). When T^A is in `query`, T^A changes this state to $\in$ or $\notin$ depending on whether the string written on the query tape belongs to A or not; furthermore, the query tape is instantaneously erased. Like the case of an ordinary DTM, the expended time is the number of steps and the required space is the number of cells used on the tape and the query tape. An *oracle NDTM* is defined as the same augmentation of a NDTM.

Let $\mathcal{C}$ be a set of languages. We define complexity classes $\mathsf{P}^{\mathcal{C}}$ and $\mathsf{NP}^{\mathcal{C}}$ as follows. For a language L, we have $L \in \mathsf{P}^{\mathcal{C}}$ (or $L \in \mathsf{NP}^{\mathcal{C}}$) if and only if there is some language $A \in \mathcal{C}$ and some polynomial-time oracle DTM (or NDTM) T^A such that T^A decides L.

The *polynomial hierarchy* consists of classes Δ_i^p, Σ_i^p, and Π_i^p defined by the following equalities:

$$\begin{aligned}
&\Delta_0^p = \Sigma_0^p = \Pi_0^p = \mathsf{P},\\
&\Delta_{i+1}^p = \mathsf{P}^{\Sigma_i^p},\\
&\Sigma_{i+1}^p = \mathsf{NP}^{\Sigma_i^p},\\
&\Pi_{i+1}^p = \text{co-}\Sigma_{i+1}^p,
\end{aligned}$$

for all $i \geq 0$. The class PH is defined as $\bigcup_{i\geq 0} \Sigma_i^p$.

Exponential time.. Besides $\mathsf{EXPTIME}$ and $\mathsf{NEXPTIME}$, we mention in this paper some other classes that characterize computation in exponential time. The classes ETIME and NETIME are defined as

$$\bigcup_{d>0} \mathsf{TIME}(2^{dn}) \text{ and } \bigcup_{d>0} \mathsf{NTIME}(2^{dn})$$

respectively; they capture linear exponents instead of polynomial exponents. The class $\mathsf{EXPTIME}$ can be viewed as 1-$\mathsf{EXPTIME}$ where 1 means the first level of exponentiation. Double exponents, triple exponents, etc. are captured by the classes 2-$\mathsf{EXPTIME}$, 3-$\mathsf{EXPTIME}$ etc. defined as

$$\bigcup_{d>0} \mathsf{TIME}(2^{2^{n^d}}),\ \bigcup_{d>0} \mathsf{TIME}(2^{2^{2^{n^d}}}),\ \ldots.$$

Their nondeterministic counterparts are defined in the same way but with the replacement of $\mathsf{TIME}(f(n))$ by $\mathsf{NTIME}(f(n))$. The class $\mathsf{ELEMENTARY}$ is defined to be the union of classes k-$\mathsf{EXPTIME}$ over all $k > 0$.

3.3 Reductions

Let L_1 and L_2 be languages. Assume that there is a DTM R such that

(1) For all input strings x, we have $x \in L_1$ if and only if $R(x) \in L_2$, where $R(x)$ denotes the output of R on input x.

(2) R halts within space $O(\log n)$.

Then R is called a *logarithmic-space reduction* from L_1 to L_2 and we say that L_1 is *reducible* to L_2.

Let $\mathcal{C}$ be a set of languages. A language L is called *$\mathcal{C}$-hard* if any language L' in $\mathcal{C}$ is reducible to L. If L is $\mathcal{C}$-hard and $L \in \mathcal{C}$ then L is called *$\mathcal{C}$-complete* or *complete for $\mathcal{C}$*.

Besides the above notion of a reduction, complexity theory also considers many other kinds of reductions, for example, polynomial-time many-one reductions or polynomial-time Turing reductions (which are both weaker, i.e., more liberal kinds of reductions). *In this paper, unless otherwise stated, a reduction means a logarithmic-space reduction.* We note that in several cases, results that we shall review have been stated for polynomial-time many-one reductions, but the proofs establish that they hold under logarithmic-space reduction.

Sometimes reductions are considered that are *tighter* than logarithmic-space reductions. Since such reductions are only of minor importance to this paper, they will be shortly described in appropriate places below. Note, however, that in case of such tight reductions, as well as in case of computation with sublinear resource constraints, the particular representation of the problem input as a string I may be a matter of concern. However, for most of the problems that we describe, and in particular those having complexity at least P, this is not an issue; any "reasonable" representation is appropriate, see e.g. [Johnson 1990].

4. COMPLEXITY OF PLAIN LOGIC PROGRAMMING

In this section, we survey some basic results on the complexity of plain logic programming. This section is written in a slightly more tutorial style than the following sections in order to help both readers not familiar with logic programming and readers not too familiar with complexity theory to grasp some key issues relating complexity theory and logic programming.

4.1 Simulation of deterministic Turing machines by logic programs

Let T be a DTM. Consider the computation of T on an input string I. The purpose of this section is to describe a logic program L and a goal G such that $L \models G$ if and only if T accepts I in at most N steps.

The transition function δ of a DTM with a single tape can be represented by a table whose rows are tuples $t = \langle s, \sigma, s', \sigma', d\rangle$. Such a tuple t expresses the following if-then-rule:

> **if** at some time instant τ the DTM is in state s, the cursor points to cell number π, and this cell contains symbol σ
> **then** at instant $\tau + 1$ the DTM is in state s', cell number π contains symbol σ', and the cursor points to cell number $\pi + d$.

It is possible to describe the complete evolution of a DTM T on input string I from its initial configuration at time instant 0 to the configuration at instant N by a propositional logic program $L(T, I, N)$. To achieve this, we define the following classes of propositional atoms:

$\boldsymbol{symbol}_\alpha[\tau, \pi]$ for $0 \leq \tau \leq N$, $0 \leq \pi \leq N$ and $\alpha \in \Sigma$. Intuitive meaning: at instant τ of the computation, cell number π contains symbol α.

cursor$[\tau, \pi]$ for $0 \leq \tau \leq N$ and $0 \leq \pi \leq N$. Intuitive meaning: at instant τ the cursor points to cell number π.

state$_s[\tau]$ for $0 \leq \tau \leq N$ and $s \in S$. Intuitive meaning: at instant τ the DTM T is in state s.

accept Intuitive meaning: T has reached state `yes`.

Let us denote by I_k the k-th symbol of the string $I = I_0 \cdots I_{|I|-1}$. The initial configuration of T on input I is reflected by the following *initialization facts* in $L(T, I, N)$:

$$\begin{array}{rll} symbol_\sigma[0, \pi] & \leftarrow & \text{for } 0 \leq \pi < |I|, \text{ where } I_\pi = \sigma \\ symbol_\sqcup[0, \pi] & \leftarrow & \text{for } |I| \leq \pi \leq N \\ cursor[0, 0] & \leftarrow & \\ state_{s_0}[0] & \leftarrow & \end{array}$$

Each entry $\langle s, \sigma, s', \sigma', d\rangle$ of the transition table δ is translated into the following propositional Horn clauses, which we call the *transition rules*. The clauses are asserted for each value of τ and π such that $0 \leq \tau < N$, $0 \leq \pi < N$, and $0 \leq \pi + d$.

$$\begin{array}{rcl} symbol_{\sigma'}[\tau+1, \pi] & \leftarrow & state_s[\tau], symbol_\sigma[\tau, \pi], cursor[\tau, \pi] \\ cursor[\tau+1, \pi+d] & \leftarrow & state_s[\tau], symbol_\sigma[\tau, \pi], cursor[\tau, \pi] \\ state_{s'}[\tau+1] & \leftarrow & state_s[\tau], symbol_\sigma[\tau, \pi], cursor[\tau, \pi] \end{array}$$

These clauses almost perfectly describe what is happening during a state transition from an instant τ to an instant $\tau + 1$. However, it should not be forgotten that those tape cells which are not changed during the transition keep their old values at instant $\tau + 1$. This must be reflected by what we term *inertia rules*. These rules are asserted for each time instant τ and tape cells numbers π, π', where $0 \leq \tau < N$, $0 \leq \pi < \pi' \leq N$, and have the following form:

$$\begin{array}{rcl} symbol_\sigma[\tau+1, \pi] & \leftarrow & symbol_\sigma[\tau, \pi], cursor[\tau, \pi'] \\ symbol_\sigma[\tau+1, \pi'] & \leftarrow & symbol_\sigma[\tau, \pi'], cursor[\tau, \pi] \end{array}$$

Finally, a group of clauses termed *accept rules* derives the propositional atom *accept*, whenever an accepting configuration is reached.

$$accept \leftarrow state_{\texttt{yes}}[\tau] \qquad \text{for } 0 \leq \tau \leq N.$$

Denote by L the logic program $L(T, I, N)$. Note that $T_L^0 = \emptyset$ and that T_L^1 contains the initial configuration of T at time instant 0. By construction, the least fixpoint T_L^∞ of L is reached at T_L^{N+2}, and the ground atoms added to T_L^τ, $2 \leq \tau \leq N+1$, i.e., those in $T_L^\tau \setminus T_L^{\tau-1}$, describe the configuration of T on the input I at the time instant $\tau - 1$. The fixpoint T_L^∞ contains *accept* if and only if an accepting configuration has been reached by T in at most N computation steps. We thus have:

Lemma 4.1 $L(T, I, N) \models accept$ *if and only if the DTM T accepts the input string I within N steps.*

A somewhat different simulation of deterministic multi-tape Turing machines by logic programs was given by [Itai and Makowsky 1987]. These authors also note that simulating Turing machines by Horn clause theories, and, more generally, by logical deduction has a long history:

"*The idea of simulating Turing machines by logical deduction goes back to Turing's original paper [Turing 1937]. Turing introduced his abstract machine concept at a time when computations were considered to be something mechanical, and felt it was necessary to show that logical deduction can be reduced to such a mechanistic model of computation. However, this reduction uses full first-order logic. A reduction using only universal Horn formulas (with function symbols) appears buried in the exposition of [Scholz and Hasenjaeger 1961]. It also forms the basis of the theory of formal systems, as presented by Smullyan in his thesis [Smullyan 1961]. The idea of coding Turing machines by logic Horn formulas appears explicitly in [Büchi 1962] and has been used since 1971 in a series of papers by Aandera, Börger, and Lewis [Aanderaa and Börger 1979; Börger 1971; Börger 1974; Börger 1984; Lewis 1979] to obtain undecidability and complexity results. Since then, various authors have rediscovered that such a reduction is possible and have used this observation to show that logic programming is computationally complete. The earliest reference we have found that states this result explicitly is [Andréka and Németi 1978]; a slightly weaker result appears in [Tärnlund 1977].*"

Yet another translation and further references can be found in the recent book [Börger et al. 1997].

4.2 Propositional logic programming

The simulation of a DTM by a propositional logic program, as described in Section 4.1 is almost all we need in order to determine the complexity of propositional logic programming, i.e., the complexity of deciding whether $P \models A$ holds for a given logic program P and ground atom A.

Theorem 4.2 *(implicit in [Jones and Laaser 1976; Vardi 1982; Immerman 1986]) Propositional logic programming is* P*-complete.*

Proof. (1) *Membership.* It is obvious that the least fixpoint T_P^∞ of the operator T_P, given program P, can be computed in polynomial time: the number of iterations (i.e. applications of T_P) is bounded by the number of rules plus one. Each iteration step is clearly feasible in polynomial time.

(2) *Hardness.* Let A be a language in P. Thus A is decidable in $q(n)$ steps by a DTM T for some polynomial q. Transform each instance I of A to the corresponding logic program $L(T, I, q(|I|))$ as described in Section 4.1. By Lemma 4.1, $L(T, I, q(|I|)) \models$ *accept* if and only if T has reached an accepting state within $q(n)$ steps. The translation from I to $L(T, I, q(|I|))$ is very simple and is clearly feasible in logarithmic space, since all rules of $L(T, I, q(|I|))$ can be generated independently of each other and each has size logarithmic in $|I|$; note that the numbers τ and π have $O(\log |I|)$ bits, while all other syntactic constituents of a rule have constant size. We have thus shown that every language A in P is logspace reducible to propositional logic programming. Hence, propositional logic programming is P-hard.

□

Obviously, this theorem can be proved by simpler reductions from other P-complete problems, for example from the monotone circuit value problem (see [Papadimitriou 1994]). However, our proof from first principles provides a basic framework from which further results will be derived by slight adaptations in the sequel.

Notice that in a standard programming environment, propositional logic programming is feasible in linear time by using appropriate data structures, as follows from results about deciding Horn satisfiability [Dowling and Gallier 1984; Itai and Makowsky 1987]. This does not mean that all problems in P are solvable in linear time; first, the model of computation used in [Dowling and Gallier 1984] is the RAM machine, and second logarithmic-space reductions may in general polynomially increase the input.

Theorem 4.2 holds under stronger reductions. In fact, it holds under the requirement that the logspace reduction is also a *polylogtime reduction (PLT)*. Briefly, a map $f : \Pi \rightarrow \Pi'$ from a problem Π to a problem Π' is a PLT-reduction, if there are polylogtime deterministic Turing machines N and M such that for all w, $N(w) = |f(w)|$ and for all w and n, $M(w,n) = Bit(n, f(w))$, i.e., the n-th bit of $f(w)$ (see e.g. [Veith 1998] for details). (Recall that N and M have separate input tapes whose cells can be accessed by use of an index register tape.) Since the above encoding of a DTM into logic programming is highly regular, it is easily seen that it is a PLT reduction.

Syntactical restrictions on programs lead to completeness for classes inside P. Let LP(k) denote logic programming where each clause has at most k atoms in the body. Then, by results in [Vardi 1982; Immerman 1987], one easily obtains:

Theorem 4.3 *LP*(1) *is* NL*-complete.*

Proof. (Sketch)

(1) *Membership* The membership part can be established by reducing this problem to graph reachability, i.e., given a directed graph $G = (V, E)$ and vertices $s, t \in V$, decide whether t is reachable from s. Since graph reachability is in NL and NL is closed under logarithmic-space reductions (i.e., reducibility of a problem A to a problem B in NL implies that A is in NL), it follows that LP(1) is in NL.
For a program P from LP(1), the question whether $P \models A$ is equivalent to the node *true* (representing truth) is reachable from the node A in the directed graph $G = (V, E)$ as follows. The vertex set V is the set of atoms in P plus *true*; the edge set E contains an edge (A, B) directed from A to B for every rule $A \leftarrow B$ in P, and an edge (A, true) for every fact $A \leftarrow$ in P. Clearly, the graph G is constructible from P in logarithmic space. Thus, the problem is in NL.

(2) *Hardness* Conversely, graph reachability is easily transformed into $P \models A$ for a program in LP(1). Since graph reachability is NL-complete (thus NL-hard), the result is established.

□

Observe that the above DTM encoding can be easily modified to programs in LP(2). Hence, LP(2) is P-complete.

Further syntactical restrictions on LP(1) yield problems complete for L (of course, under reductions stronger than logspace reductions), which we omit here.

4.3 Complexity of datalog

Let us now turn to datalog, and let us first consider the data complexity. Grounding P on an input database D yields polynomially many clauses in the size of D; hence, the complexity of propositional logic programming is an upper bound for the data complexity. The same holds for the variants of datalog we shall consider in the sequel. The complexity of propositional logic programming is also a lower bound. Thus,

Theorem 4.4 *(implicit in [Vardi 1982; Immerman 1986]) Datalog is data complete for* P.

In fact, this result follows from the proof of Theorem 7.2 below. An alternative proof of P-hardness can be given by writing a simple datalog *meta-interpreter* for propositional LP(k), where k is a constant.

Represent rules $A_0 \leftarrow A_1, \ldots, A_i$, where $0 \leq i \leq k$, by tuples $\langle A_0, \ldots, A_i \rangle$ in an $(i+1)$-ary relation R_i on the propositional atoms. Then, a program P in LP(k) which is stored this way in a database $D(P)$ can be evaluated by a fixed datalog program $P_{MI}(k)$ which contains for each relation R_i, $0 \leq i \leq k$, a rule

$$T(X_0) \leftarrow T(X_1), \ldots, T(X_i), R_i(X_0, \ldots, X_i).$$

Here $T(x)$ intuitively means that atom x is true. Then, $P \models A$ just if $P_{MI} \cup P(D) \models T(A)$. P-hardness of the data complexity of datalog is then immediate from Theorem 4.2.

The program complexity is exponentially higher.

Theorem 4.5 *(implicit in [Vardi 1982; Immerman 1986]) Datalog is program complete for* EXPTIME.

PROOF. (Sketch)

(1) *Membership.* Grounding P on D leads to a propositional program P' whose size is exponential in the size of the fixed input database D. Hence, by Theorem 4.2, the program complexity is in EXPTIME.

(2) *Hardness.* In order to prove EXPTIME-hardness, we show that if a DTM T halts in less than $N = 2^{n^k}$ steps on a given input I where $|I| = n$, then T can be simulated by a datalog program over a fixed input database D. In fact, we use $D_\emptyset$, i.e., the empty database with the universe $U = \{0, 1\}$.

We employ the scheme of the DTM encoding into logic programming from above, but use the predicates $symbol_\sigma(X, Y)$, $cursor(X, Y)$ and $state_s(X)$ instead of the propositional letters $symbol_\sigma[X, Y]$, $cursor[X, Y]$ and $state_s[X]$ respectively. The time points τ and tape positions π from 0 to $2^m - 1$, $m = n^k$, are represented by m-ary tuples over U, on which the functions $\tau + 1$ and $\pi + d$ are realized by means of the successor $Succ^m$ from a linear order $\leq^m$ on U^m.

For an inductive definition, suppose $Succ^i(\mathbf{X},\mathbf{Y})$, $First^i(\mathbf{X})$, and $Last^i(\mathbf{X})$ tell the successor, the first, and the last element from a linear order $\leq^i$ on U^i, where $\mathbf{X}$ and $\mathbf{Y}$ have arity i. Then, use rules

$$\begin{aligned}
Succ^{i+1}(Z,\mathbf{X},Z,\mathbf{Y}) &\leftarrow Succ^i(\mathbf{X},\mathbf{Y})\\
Succ^{i+1}(Z,\mathbf{X},Z',\mathbf{Y}) &\leftarrow Succ^1(Z,Z'), Last^i(\mathbf{X}), First^i(\mathbf{Y})\\
First^{i+1}(Z,\mathbf{X}) &\leftarrow First^1(Z), First^i(\mathbf{X})\\
Last^{i+1}(Z,\mathbf{X}) &\leftarrow Last^1(z), Last^i(\mathbf{X})
\end{aligned}$$

Here $Succ^1(X,Y)$, $First^1(X)$, and $Last^1(X)$ on $U^1 = U$ must be provided. For our reduction, we use the usual ordering $0 \leq^1 1$ and provide those relations by the ground facts $Succ^1(0,1)$, $First^1(0)$, and $Last^1(1)$.

The initialization facts $symbol_\sigma[0,\pi]$ are readily translated into the datalog rules

$$symbol_\sigma(\mathbf{X},\mathbf{t}) \leftarrow First^m(\mathbf{X}),$$

where $\mathbf{t}$ represents the position π, and similarly the facts $cursor[0,0]$ and $state_{s_0}[0]$. The remaining initialization facts $symbol_\sqcup[0,\pi]$, where $|I| \leq \pi \leq N$, are translated to the rule

$$symbol_\sqcup(\mathbf{X},\mathbf{Y}) \leftarrow First^m(\mathbf{X}),\ \leq^m(\mathbf{t},\mathbf{Y})$$

where $\mathbf{t}$ represents the number $|I|$; the order $\leq^m$ is easily defined from $Succ^m$ by two clauses

$$\begin{aligned}
\leq^m(\mathbf{X},\mathbf{X}) &\leftarrow \mathbf{X}\\
\leq^m(\mathbf{X},\mathbf{Y}) &\leftarrow Succ^m(\mathbf{X},\mathbf{Z}),\ \leq^m(\mathbf{Z},\mathbf{Y})
\end{aligned}$$

The transition and inertia rules are easily translated into datalog rules. For realizing $\tau+1$ and $\pi+d$, use in the body atoms $Succ^m(\mathbf{X},\mathbf{X}')$. For example, the clause

$$symbol_{\sigma'}[\tau+1,\pi] \leftarrow state_s[\tau], symbol_\sigma[\tau,\pi], cursor[\tau,\pi]$$

is translated into

$$symbol_{\sigma'}(\mathbf{X}',\mathbf{Y}) \leftarrow state_s(\mathbf{X}), symbol_\sigma(\mathbf{X},\mathbf{Y}), cursor(\mathbf{X},\mathbf{Y}), Succ^m(\mathbf{X},\mathbf{X}').$$

The translation of the accept rules is straightforward.

For the resulting datalog program P', it holds that $P' \cup D_\emptyset \models accept$ if and only if T accepts input I in at most N steps. It is easy to see that P' can be constructed from T and I in logarithmic space. Hence, datalog has EXPTIME-hard program complexity.

Note that straightforward simplifications in the construction are possible, which we omit here, as part of it will be reused below. □

Instead of using a generic reduction, the hardness part of this theorem can also be obtained by applying complexity upgrading techniques [Papadimitriou and Yannakakis 1986; Balcázar et al. 1992]. We briefly outline this in the rest of this section.

This technique utilizes a conversion lemma [Balcázar et al. 1992] of the form "If Π X-reduces to Π', then $s(\Pi)$ Y-reduces to $s(\Pi')$"; here $s(\Pi)$ is the succinct variant of Π, where the instances I of Π are given by a Boolean circuit C_I which computes the bits of I (see [Balcázar et al. 1992] for details). The strongest form of the conversion lemma appears in [Veith 1998], where X is PLT and Y is monotone projection reducibility [Immerman 1987]. Informally, monotone projection reductions are reductions that transform a relational data structure A into a relational data structure B such that each tuple in B is the projection of a single tuple in A. This tuple is determined by a quantifier-free formula using just equality and successor. Note that this reduction is uniform in the sense that the formula is the same for all tuples of A. Monotone projection reductions are computable in logarithmic time, which means that the size of B and the value of each bit position in data structure B can determined in time logarithmic in the size of A on a RAM. They are tighter than both PLT reductions and *first-order reductions* where arbitrary first-order formulae (and not just projections) can be used in the transformations. For details, see [Immerman 1987].

The conversion lemma gives rise to an upgrading theorem, which has been subsequently sharpened [Balcázar et al. 1992; Eiter et al. 1994; Gottlob et al. 1995; Veith 1998] and is stated below in the strongest form of [Veith 1998]. For a complexity class $\mathcal{C}$, denote $long(\mathcal{C}) = \{long(L) \mid L \in \mathcal{C}\}$, where $long(L) = \bigcup_{bin(n) \in 1L} \{0,1\}^n$, i.e., contains all strings of length n such that n, in binary and with the leading 1 omitted, belongs to L.

Theorem 4.6 *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be complexity classes such that $long(\mathcal{C}_1) \subseteq \mathcal{C}_2$. If Π is hard for $\mathcal{C}_2$ under PLT-reduction, then $s(\Pi)$ is hard for $\mathcal{C}_1$ under monotone projection reduction.*

We remark that since monotone projection reduction is very weak, a special encoding of succinct problems is necessary. From the observations in Section 4.2, we then obtain that $s(\mathrm{LP}(2))$ is EXPTIME-hard under monotone projection reductions, where each program P is stored in the database $D(P)$, which is represented by a binary string in the standard way.

$s(\mathrm{LP}(2))$ can be reduced to evaluating a datalog program P^* over a fixed database as follows. From a succinct instance of LP(2), i.e., a Boolean circuit C_I for $I = D(P)$, Boolean circuits C_i for computing R_i, $0 \leq i \leq 2$ can be constructed which use negation merely on input gates.

Each such circuit $C_i(\mathbf{X})$ can be simulated by straightforward datalog rules. For example, an $\wedge$-gate g_i with input from gates g_j and g_k is described by a rule $g_i(\mathbf{X}) \leftarrow g_j(\mathbf{X}), g_k(\mathbf{X})$, and an $\vee$-gate g_i is described by the rules $g_i(\mathbf{X}) \leftarrow g_j(\mathbf{X})$ and $g_i(\mathbf{X}) \leftarrow g_k(\mathbf{X})$. Observe that Boolean circuits with arbitrary use of negation can be easily simulated in stratified datalog [Kolaitis and Papadimitriou 1991] or disjunctive datalog [Eiter et al. 1997].

The desired program P^* comprises the rules for the Boolean circuits C_i and the rules of the meta-interpreter $P_{MI}(k)$, which are adapted for a binary encoding of the domain $U_{D(P)}$ of the database $D(P)$ by using binary tuples of arity $\lceil \log |U_{D(P)}| \rceil$. This construction is feasible in logarithmic space, from which EXPTIME-hard program complexity of datalog follows. We refer the reader to [Eiter et al. 1994; Eiter et al. 1997; Gottlob et al. 1995] for the technical details.

4.4 Logic programming with functions

Let us see what happens if we allow function symbols in logic programs. In this case, entailment of an atom is no longer decidable. To prove it, we can, for example, reduce Hilbert's Tenth Problem to query answering in full logic programming. Natural numbers can be represented using the constant 0 and the successor function s. Addition and multiplication are expressed by the following simple logic program:

$$\begin{aligned} X+0=X &\leftarrow \\ X+s(Y)=s(Z) &\leftarrow X+Y=Z \\ X\times 0=0 &\leftarrow \\ X\times s(Y)=Z &\leftarrow X\times Y=U,\ U+X=Z \end{aligned}$$

Now, undecidability of full logic programming follows from the undecidability of diophantine equations [Matiyasevič 1970]. More precisely, it shows that full logic programming can express r.e.-complete languages. On the other hand, the least fixpoint T_P^∞ of any logic program P is clearly a r.e. set. This shows r.e.-completeness of logic programming.

Theorem 4.7 *([Andréka and Németi 1978; Tärnlund 1977]) Logic programming is r.e.-complete.*[2]

Of course, this theorem may as well be proved by a simple encoding of Turing machines similar to the encoding in the proof of Theorem 4.5 (use terms $f^n(c)$, $n \geq 0$, for representing cell positions and time instants). It is interesting to note that [Smullyan 1956] asserted –quite some time before the first proposals to logic programming – a closely related result which essentially says that, in our terms, the minimal model semantics of logic programming over arithmetic yields the r.e. sets.

Theorem 4.7 was generalized in [Voronkov 1995] for more expressive S-semantics and C-semantics [Falaschi et al. 1989]. On the other hand, it was sharpened to syntactical classes of logic programs. E.g., [Tärnlund 1977] used binary Horn clause programs to simulate a universal Turing machine. By a transformation from binary Horn clause programs, [Sebelík and Štěpánek 1982] showed that a class of logic programs called stratifiable (in a sense different from the one in Section 5.1) is r.e.-complete. Furthermore, [Štěpánek and Štěpánková 1986] proved that (an inessential variant of) PRIMLOG (see [Markusz and Kaposi 1982]) is r.e.-complete, which restricts considerably the size of AND- and OR-branching and allows to use recursion explicitly in only a single clause of particular type. The proof shows that all μ-recursive functions can be expressed within this fragment.

A natural decidable fragment of logic programming with functions are *nonrecursive programs*, in which intuitively no predicate depends syntactically on itself (see Section 5.1 for a definition). Their complexity is characterized by the following theorem.

[2] In the context of recursion theory, reducibility of a language (or problem) L_1 to L_2 is understood in terms of a Turing reduction, i.e., L_1 can be decided by a DTM with oracle L_2, rather than logarithmic-space reduction.

Theorem 4.8 *([Dantsin and Voronkov 1997]) Nonrecursive logic programming is* NEXPTIME*-complete.*

The membership is established by applying SLD-resolution with constraints. The size of the derivation turns out to be exponential. NEXPTIME-hardness is proved by reduction from the tiling problem for the square $2^n \times 2^n$.

Some other fragments of logic programming with function symbols are known to be decidable too. For example, the following result was established in [Shapiro 1984], by using a simulation of alternating Turing machines by logic programs and vice versa.

Theorem 4.9 *([Shapiro 1984]) Logic programming with function symbols is* PSPACE*-complete, if each rule is restricted as follows: the body contains only one atom, the size of the head is greater than or equal to that of the body, and the number of occurrences of any variable in the body is less than or equal to the number of its occurrences in the head.*

The simulation assumed that the input to an alternating Turing machine is written on the work-tape. Extending the simulation by a distinguished input-tape, [Štěpánek and Štěpánková 1986] showed that the class of logic programs having logarithmic (respectively, polynomial) goal-size complexity is P-complete (respectively, EXPTIME-complete). Here, the goal-size complexity is the maximal size of any subgoal (in terms of symbols) occurring in the proof tree of a goal. Related notions of complexity and normal forms of programs, defined in terms of computation trees [Štěpánková and Štěpánek 1984], are studied in [Ochozka et al. 1988].

We refer to [Blair 1982; Fitting 1987a; Fitting 1987b] for further material on recursion-theoretic issues related to logic programming.

4.5 Further issues

Besides data and combined complexity, many other complexity aspects of logic program have been investigated, in particular in the context of datalog. We discuss here some of issues that have received broad attention.

Sirups.. A strongly restricted class of logic programs often considered in the literature is the class of *single rule programs* (sirups) or programs consisting of *one* recursive rule and some nonrecursive (initialization) rules or atoms.

For a long time, the decidability of the following problem was open: Given an LP P (with function symbols) that consists of a unique recursive rule and a set of ground atoms, and given a ground goal G, does it hold that $P \models G$? This problem is equivalent to the *Horn clause implication problem*, i.e., checking whether the universal closure of a Horn clause C_1 logically implies the universal closure of a Horn clause C_2. The problem was shown to be undecidable in [Marcinkowski and Pacholski 1992]. Some decidable special cases of this problem were studied in [Gottlob 1987; Leitsch and Gottlob 1990; Leitsch 1990].

Several undecidability results of inference and satisfiability problems for various restricted forms of sirups with non-ground atoms or with nonrecursive rules can be found in [Devienne 1990; Devienne et al. 1993; Hanschke and Würtz 1993; Devienne et al. 1996].

Datalog sirups are EXPTIME complete with respect to *program* and *combined complexity*; this remains true even for datalog sirups consisting of a unique rule and no facts [Gottlob and Papadimitriou 1999]. It follows that deciding whether (the universal closure of) a datalog clause logically implies (the universal closure of) another datalog clause is EXPTIME complete, too. The problem of evaluating a *nonrecursive* Horn clause (with or without function symbols) over a set of ground facts is NP-complete [Chandra and Merlin 1977] (even for a fixed set of ground facts). (Here by "evaluation", we mean determining whether a rule fires.) This problem is computationally equivalent to the problem of evaluating a Boolean *conjunctive query* over a database, i.e., a datalog clause whose body contains only input predicates, and also to the well known NP-complete clause subsumption problem [Garey and Johnson 1979] (see below). The parametric complexity of conjunctive queries is studied on [Papadimitriou and Yannakakis 1997].

With respect to *data complexity*, datalog sirups are complete for P, and thus in general inherently sequential, cf. [Kanellakis 1988]. There are, however, many interesting special cases in which sirup queries can be evaluated in parallel.

Inside P *and parallelization issues..* In [Ullman and van Gelder 1988] the *polynomial fringe property* is studied. Roughly, a datalog program P has the polynomial fringe property if it is guaranteed that for each database D and goal G such that $P \cup D \models G$, there is a derivation tree whose *fringe* (i.e., set of leaves) is of polynomial size. The data complexity of datalog programs with the polynomial fringe property is in LOGCFL, which is the class of all languages (that is, problems) that are reducible in logarithmic space to a context-free language. LOGCFL is a subclass of NC^2, and thus contains highly parallelizable problems [Johnson 1990]; furthermore, programs whose fringe is superpolynomial (i.e., $O(2^{\log^k n})$) are in NC [Ullman and van Gelder 1988; Kanellakis 1988]. Here NC^2 is the second level of the NC-hierarchy of complexity classes NC^i. These classes are defined by families of uniform Boolean circuits of depth $O(\log^i n)$ [Johnson 1990]. An example of programs with the polynomial fringe property are linearly recursive sirups; however, there also exist nonlinear sirups that are not equivalent to any linear sirup and are still in NC [Afrati and Cosmadakis 1989].

In [Kanellakis 1988], the *polynomial (superpolynomial) tree-size property for width* k is considered. Roughly, a datalog program has this property if every derivable atom can be obtained by a width-k derivation tree of polynomial (superpolynomial) size. A width-k derivation tree is a generalized derivation tree, where each node may represent up to k ground atoms. For width $k = 1$, the polynomial (resp., superpolynomial) tree-size property coincides with the polynomial (resp., superpolynomial) fringe property; however, for higher widths, the former properly generalizes the latter. [Kanellakis 1988] shows that the data complexity of datalog programs having the polynomial (resp., superpolynomial) tree-size property for any fixed constant width is in LOGCFL (resp., in NC).

The *hypergraph* (V, E) associated with a Horn clause or conjunctive query has as set V of vertices the set of variables occurring in the rule; its set E of hyperedges contains for each atom A in the rule body a hyperedge consisting of the variables occurring in A. If the hypergraph associated with a nonrecursive rule is *acyclic*, the evaluation problem is feasible in polynomial time [Yannakakis 1981] and is actually complete for LOGCFL and thus highly parallelizable [Gottlob et al. 1998]. For generalizations of this result to various types of nearly acyclic hypergraphs, see [Gottlob et al. 1999a].

While determining whether a datalog program is parallelizable, i.e., has data complexity in NC, is in general undecidable [Ullman and van Gelder 1988; Gaifman et al. 1987], the problem has been completely resolved by [Afrati and Papadimitriou 1993] for an interesting and relevant class of sirups called *simple chain queries.* These are logic programs with a single recursive rule whose right hand side consists of binary relations forming a chain. An example of such a rule, involving a database predicate a, is

$$s(X, Y) \leftarrow a(X, Z_1), s(Z_1, Z_2), s(Z_2, Z_3), a(Z_3, Y).$$

[Afrati and Papadimitriou 1993] show that (unless $\mathsf{P} = \mathsf{NC}$) simple chain queries are either complete for P or in NC. They give a precise characterization of the P-complete and NC-computable simple chain queries.

Boundedness.. Many papers have been devoted to the decidability of the *boundedness problem* for datalog programs. A datalog program P is *bounded*, if there exists a constant k such that for all databases D, the number of iteration steps needed in order to compute the least fixed point $\mathcal{M}(ground(P \cup D, \mathcal{L}(P, D)))$ is bounded by k and is thus independent of D (it depends on P only). Boundedness is an interesting property, because as shown in [Ajtai and Gurevich 1994], a datalog program is bounded if and only if it is equivalent to a first-order query. For important related results on the equivalence of recursive and nonrecursive datalog queries, see [Chaudhuri and Vardi 1997]. The undecidability of the boundedness for general datalog programs was shown in [Gaifman et al. 1987], for linear recursive queries in [Vardi 1988], and for sirups in [Abiteboul 1989]. There is a very large number of papers discussing the decidability of boundedness issues, both for syntactic restrictions of datalog programs or sirups and for variants of boundedness such as *uniform boundedness.* Good surveys of early work are given in [Kanellakis 1988] and in [Kanellakis 1990]. The following is an incomplete list of papers where important results and further relevant references on decidability issues of boundedness or uniform boundedness can be found: [Hillebrand et al. 1995; Marcinkowski 1996b; Marcinkowski 1996a; Marcinkowski 1999]. Sufficient conditions for boundedness were given in [Minker and Nicolas 1983; Sagiv 1985; Ioannidis 1986; Vardi 1988; Naughton 1989; Cosmadakis 1989; Naughton and Sagiv 1987; Naughton and Sagiv 1991].

Containment, equivalence, and subsumption.. Issues that have been studied repeatedly in the context of query optimization are query equivalence and containment. *Query containment* is the problem, given two datalog programs P_1 and P_2 having the same input schema $\mathcal{D}_{in}$ and output schema $\mathcal{D}_{out}$, whether for every input database D_{in}, the output of P_1 is contained in the output of P_2, i.e, $\mathcal{M}_{P_1}(D_{in})|p \subseteq \mathcal{M}_{P_2}(D_{in})|p$ holds, for every relation $p \in \mathcal{D}_{out}$. As shown by [Shmueli 1987], containment and equivalence are undecidable for datalog programs; however, a stronger form of uniform containment is decidable [Sagiv 1988].

In the case where P_1 and P_2 contain only conjunctive queries, containment and equivalence are NP-complete [Sagiv and Yannakakis 1980], and remain NP-complete even if P_1 and P_2 consist of single conjunctive queries [Chandra and Merlin 1977]. If the domain has a linear order $\leq$ and comparison literals $t_1 \leq t_2$, $t_1 < t_2$, and $t_1 \neq t_2$ may be used in rule bodies, then the containment problem for single conjunctive queries is Π_2^p-complete [van der Meyden 1997]; this result generalizes to sets of conjunctive queries. As shown in [van der Meyden 1997], conjunctive query containment is still co-NP-complete if the database relations are monadic, but polynomial if an additional sequentiality restrictions is imposed on order literals.

Containment of a nonrecursive datalog program P_1 in a recursive datalog program P_2 is decidable, since P_1 can be rewritten to a set of conjunctive queries, and deciding whether a conjunctive query is contained in an arbitrary (recursive) datalog program is EXPTIME-complete [Cosmadakis and Kanellakis 1986; Chandra et al. 1981]. [Chaudhuri and Vardi 1994] have investigated the converse problem, i.e., containment of a recursive datalog program P_1 in a nonrecursive datalog program P_2. They showed that the problem is 3-EXPTIME-complete in general and 2-EXPTIME-complete if P_2 is a set of conjunctive queries. Furthermore, they showed that deciding equivalence of a recursive and a nonrecursive datalog program is 3-EXPTIME-complete.

We observe that the containment problem for conjunctive queries is equivalent to the clause subsumption problem. A clause C *subsumes* a clause D, if there exists a substitution θ such that $C\theta \subseteq D$; subsumption algorithms are discussed in [Gottlob and Leitsch 1985b; Gottlob and Leitsch 1985a; Bachmair et al. 1996]. This equivalence extends to sets of conjunctive queries, i.e., in essence to nonrecursive datalog programs [Sagiv and Yannakakis 1980]. For a discussion of subsumption-based and other notions of equivalence for logic programs, see [Maher 1988].

The clause subsumption problem plays a very important role in the field of *inductive logic programming (ILP)* [Muggleton 1992]. For complexity results on ILP consult [Kietz and Dzeroski 1994; Gottlob et al. 1997]. A problem related to clause subsumption is *clause condensation*: given a clause C, find a smallest subset of C which subsumes C. Complexity results and algorithms for clause condensation can be found in [Gottlob and Fermüller 1993]. The complexity of the clause evaluation problem and of other related problems on *generalized Herbrand interpretations*, which may contain nonground atoms, is studied in [Gottlob and Pichler 1999].

5. COMPLEXITY OF LOGIC PROGRAMMING WITH NEGATION

5.1 Stratified negation

A *literal* L is either an atom A (called a *positive literal*) or a negated atom $\neg A$ (called a *negative literal*). Literals A and $\neg A$ are *complementary*; for any literal L, we denote by $\neg.L$ its complementary literal, and for any set Lit of literals, $\neg.Lit = \{\neg.L \mid L \in Lit\}$.

A *normal clause* is a rule of the form

$$A \leftarrow L_1, \ldots, L_m \qquad (m \geq 0) \tag{1}$$

where A is an atom and each L_i is a literal. A *normal logic program* is a finite set of normal clauses.

The semantics of normal logic programs is not straightforward, and numerous proposals exist, cf. [Bidoit 1991; Apt and Bol 1994]. However, there is general consensus for stratified normal logic programs.

A normal logic program P is *stratified*, see [Apt et al. 1988], if there is an assignment $str(\cdot)$ of integers 0,1,... to the predicates p in P, such that for each clause r in P the following holds: If p is the predicate in the head of r and q the predicate in an L_i from the body, then $str(p) \geq str(q)$ if L_i is positive, and $str(p) > str(q)$ if L_i is negative.

Example 5. Reconsider the steam turbine scenario in Example 1, and let us add the following rules to the program there:

$$\begin{aligned} \mathit{check\_sensors} &\leftarrow \mathit{signal\_error} \\ \mathit{signal\_error} &\leftarrow \mathit{valve\_closed}, \neg \mathit{signal\_1} \\ \mathit{signal\_error} &\leftarrow \mathit{pressure\_loss}, \neg \mathit{signal\_2} \\ \mathit{signal\_error} &\leftarrow \mathit{overheat}, \neg \mathit{signal\_3} \end{aligned}$$

These rules express knowledge about potential signal errors, which must handled by checking the sensors. The augmented program P is stratified: E.g. for the assignment $str(A) = 1$ for $A \in \{\mathit{check\_sensors},\ \mathit{signal\_error}\}$ and $str(B) = 0$ for any other atom B occurring in P, the condition of stratification is satisfied.

The *reduct* of a normal logic program P by a Herbrand interpretation I [Gelfond and Lifschitz 1988], denoted P^I, is the set of ground clauses obtained from $ground(P)$ as follows: first remove every clause r with a negative literal L in the body such that $\neg.L \in I$, and then remove all negative literals from the remaining rules. Notice that P^I is a set of ground *Horn* clauses.

The semantics of a stratified normal program P is then defined as follows. Take an arbitrary stratification str. Denote by $P_{=k}$ the set of rules r such that $str(p) = k$, where p is the head predicate of r. Define a sequence of Herbrand interpretations: $M_0 = \emptyset$, and M_{k+1} is the least Herbrand model of $P_{=k}^{M_k} \cup M_k$ for $k \geq 0$. Finally, let

$$\mathcal{M}_{str}(P) = \bigcup_i M_i \cup \{\neg A \mid A \notin \bigcup_i M_i\}.$$

The semantics $\mathcal{M}_{str}$ does not depend on the stratification str [Apt et al. 1988]. Note that in the propositional case $\mathcal{M}_{str}(P)$ is polynomially computable.

Example 6. We consider the program P in Example 5. For the stratification $str(\cdot)$ of P given there, $P_{=0}$ contains the clauses listed in Example 1, and $P_{=1}$ the clauses introduced in Example 5. Then,

$$\begin{aligned} &M_0 = \emptyset \quad P_{=0}^{M_0} = P_0, \\ &M_1 = T_{P_0}^{\infty} \ P_{=1}^{M_1} = \{\mathit{check\_sensors} \leftarrow \mathit{signal\_error},\ \mathit{signal\_error} \leftarrow \mathit{overheat}\} \\ &M_2 = T_{P_0}^{\infty} \end{aligned}$$

where $T_{P_0}^{\infty} = \{\mathit{signal\_1},\ \mathit{signal\_2},\ \mathit{valve\_closed},\ \mathit{pressure\_loss},\ \mathit{leak},\ \mathit{shutdown}\}$. Thus, $\mathcal{M}_{str}(P) = T_{P_0}^{\infty} \cup \{\neg \mathit{signal\_3},\ \neg \mathit{overheat},\ \neg \mathit{signal\_error},\ \neg \mathit{check\_sensors}\}$.

Theorem 5.1 *(implicit in [Apt et al. 1988]) Stratified propositional logic programming with negation is* P*-complete. Stratified datalog with negation is data complete for* P *and program complete for* EXPTIME*.*

For full logic programming, stratified negation yields the arithmetical hierarchy.

Theorem 5.2 *([Apt and Blair 1988]) Logic programming with n levels of stratified negation is Σ^0_{n+1}-complete.*

Recall here that Σ^0_{n+1} denotes the relations over the natural numbers that are definable in arithmetic by means of a first-order formula

$$\phi(\mathbf{Y}) = \exists \mathbf{X}_0 \forall \mathbf{X}_1 \cdots Q_k \mathbf{X}_n \psi(\mathbf{X}_0, \ldots, \mathbf{X}_n, \mathbf{Y})$$

with free variables $\mathbf{Y}$, where the quantifiers alternate and ψ is quantifier-free; in particular, Σ^0_1 contains the r.e. sets. Further complexity results on stratification can be found in [Cholak and Blair 1994; Palopoli 1992].

A particular case of stratified negation are nonrecursive logic programs. A program is *nonrecursive* if and only if it has a stratification such that each predicate p occurs in its defining stratum $P_{=str(p)}$ only in the heads of rules.

Theorem 5.3 *(implicit in [Immerman 1987; Vardi 1982]) Nonrecursive propositional logic programming with negation is* P*-complete. Nonrecursive datalog with negation is program complete for* PSPACE*, and its data complexity is in the class* AC^0*, which contains the languages recognized by unbounded fan-in circuits of polynomial size and constant depth [Johnson 1990].*

[Vorobyov and Voronkov 1998] classified the complexity of nonrecursive logic programming depending on the signature, presence of negation and range-restriction. A clause P is called *range-restricted* if every variable occurring in this clause also occurs in a positive literal in the body. A program P is range-restricted if so is every clause in P. Range-restricted clauses have a number of good properties, for example *domain-independence*. Before presenting the results of [Vorobyov and Voronkov 1998], we explain the notation for signatures used in their paper. The tuple (k, l, m) denotes the signature with k constants, l unary function symbols and m function symbols of arity ≥ 2. The complexity of nonrecursive logic programming is summarized in Table 1.

In this table $\mathsf{TA}(f(n), g(n))$ means the class of functions computable on alternating Turing machines [Chandra et al. 1981] using $g(O(n))$ alternations with time $f(O(n))$ on every branch. Such classes are closed under *polylin* (and *loglin*) reductions, i.e., those running in polynomial time (respectively, logarithmic space), with output linearly bounded by the input. Such complexity classes arise in connection with the complexity characterization of logical theories [Berman 1977; Berman 1980].

To define the classes $\mathsf{NONELEMENTARY}(n)$, we define functions $e_n(m)$ by recursion: $e_0(m) = m$ and $e_{n+1}(m) = 2^{e_n(m)}$. Note that ELEMENTARY is the class of languages decided within time $e_k(0)$ for some fixed k. Then $\mathsf{NONELEMENTARY}(n)$ is the class of languages decided with lower and upper time bounds $e_{cn}(0)$ and $e_{dn}(0)$ respectively for some $c, d > 0$. In all cases in the table we have completeness in the corresponding complexity class, except for $\mathsf{NONELEMENTARY}(n)$ (in this case both lower and upper bounds are linearly growing towers of 2's).

Thus, there is a huge difference between nonrecursive datalog with negation and nonrecursive logic programming with negation in their program complexity, namely PSPACE vs. $\mathsf{NONELEMENTARY}(n)$. At the same time, as [Vardi 1982] and the following result show, both the languages have polynomial data complexity.

signature	$(\geq 2, 0, 0)$	$(\_, 1, 0)$	$(\_, \geq 2, 0)$	$(\_, \_, \geq 1)$
	not range-restricted			
no negation	PSPACE	PSPACE	NEXPTIME	NEXPTIME
with negation	PSPACE	PSPACE	$\mathsf{TA}(2^{O(n/\log n)}, O(n/\log n))$	$\mathsf{NONELEMENTARY}(n)$
	range-restricted			
no negation	PSPACE	PSPACE	PSPACE	NEXPTIME
with negation	PSPACE	PSPACE	PSPACE	$\mathsf{TA}(2^{n/\log n}, n/\log n)$

Table 1. Summary of results.

Theorem 5.4 *([Dantsin and Voronkov 2000]) Nonrecursive logic programming with negation has polynomial data complexity.*

5.2 Well-founded negation

Roughly speaking, the *well-founded semantics (WFS)* [van Gelder et al. 1991] assigns value "unknown" to an atom A, if it is defined by unstratified negation. Briefly, WFS can be defined as follows [Baral and Subrahmanian 1993]. Let $F_P(I)$ be the operator $F_P(I) = T^{\infty}_{P^I}$. Since $F_P(I)$ is anti-monotone, $F^2_P(I)$ is monotone, and thus has a least and a greatest fixpoint, denoted by $F^2_P{\uparrow}^{\infty}$ and $F^2_P{\downarrow}^{\infty}$, respectively. Then, the meaning of a program P under WFS, $\mathcal{M}_{wfs}(P)$, is

$$\mathcal{M}_{wfs}(P) = F^2_P{\uparrow}^{\infty} \cup \{\neg A \mid A \notin F^2_P{\downarrow}^{\infty}\}.$$

Note that on stratified programs, WFS and stratified semantics coincide.

Theorem 5.5 *(implicit in [van Gelder 1989; van Gelder et al. 1991]) Propositional logic programming with negation under WFS is* P*-complete. Datalog with negation under WFS is data complete for* P *and program complete for* EXPTIME*.*

The question whether $P \models_{wfs} A$ can be decided in linear time is open [Berman et al. 1995]. A fragment of datalog with well-founded negation that has linear data complexity and, under certain restrictions, also linear combined complexity, was recently identified and studied in [Gottlob et al. 2000b; Gottlob et al. 2000a]. This fragment, called *datalog LITE*, is well-suited for expressing temporal properties of a finite state system represented as a Kripke structure. It is more expressive than CTL and some other well-known temporal logics used in automatic verification.

For full logic programming, the following is known.

Theorem 5.6 *([Schlipf 1995b]) Logic programming with negation under WFS is* Π^1_1*-complete.*

The class Π^1_1 belongs to the *analytical hierarchy* (in a relational form) and contains those relations which are definable by a second-order formula $\Phi(\mathbf{X}) = \forall \mathbf{P} \phi(\mathbf{P}; \mathbf{X})$, where $\mathbf{P}$ is a tuple of predicate variables and ϕ is a first-order formula with free variables $\mathbf{X}$. For more details about this class in the context of logic programming, see e.g. [Schlipf 1995b; Eiter and Gottlob 1997].

5.3 Stable model semantics

An interpretation I of a normal logic program P is a *stable model* of P [Gelfond and Lifschitz 1988] if $I = T^{\infty}_{P^I}$, i.e., I is the least Herbrand model of P^I.

In general, a normal logic program P may have zero, one, or multiple stable models.

Example 7. Let P be the following non-stratified program:

$$\begin{array}{l} sleep \leftarrow \neg work \\ work \leftarrow \neg sleep \end{array}$$

Then $M_1 = \{sleep\}$ and $M_2 = \{work\}$ are the stable models of P.

Denote by SM(P) the set of stable models of P. The meaning $\mathcal{M}_{st}$ of P under the *stable model semantics (SMS)* is

$$\mathcal{M}_{st}(P) = \bigcap_{M \in \mathrm{SM}(P)} (M \cup \neg.(B_P \setminus M)).$$

Note that every stratified P has a unique stable model, and its stratified and stable semantics coincide. Unstratified rules increase complexity.

Theorem 5.7 *([Marek and Truszczyński 1991], [Bidoit and Froidevaux 1991]) Given a propositional normal logic program P, deciding whether $SM(P) \neq \emptyset$ is* NP*-complete.*

Proof. (1) *Membership.* Clearly, P^I is polynomial time computable from P and I. Hence, a stable model M of P can be guessed and checked in polynomial time.

(2) *Hardness.* Modify the DTM encoding in Section 4 for a nondeterministic Turing machine T as follows. For each state s and symbol σ, introduce atoms $B_{s,\sigma,1}[\tau], \ldots, B_{s,\sigma,k}[\tau]$ for all $1 \leq \tau < N$ and transitions $\langle s, \sigma, s_i, \sigma'_i, d_i \rangle$, where $1 \leq i \leq k$. Add $B_{s,\sigma,i}[\tau]$ in the bodies of the transition rules for $\langle s, \sigma, s_i, \sigma'_i, d_i \rangle$ and the rule

$$\begin{aligned} B_{s,\sigma,i}[\tau] \ \leftarrow \ & \neg B_{s,\sigma,1}[\tau], \ldots, \neg B_{s,\sigma,i-1}[\tau], \\ & \neg B_{s,\sigma,i+1}[\tau], \ldots, \neg B_{s,\sigma,k}[\tau]. \end{aligned}$$

Intuitively, these rules nondeterministically select precisely one of the possible transitions for s and σ at time instant τ, whose transition rules are enabled via $B_{s,\sigma,i}[\tau]$. Finally, add a rule

$$accept \leftarrow \neg accept.$$

It ensures that *accept* is true in every stable model. The stable models M of the resulting program correspond to the accepting runs of T.

□

As an easy consequence, we obtain

Theorem 5.8 *([Marek and Truszczyński 1991]; [Schlipf 1995b] and [Kolaitis and Papadimitriou 1991]) Logic programming with negation under SMS is* co-NP*-complete. Datalog with negation under SMS is data complete for* co-NP *and program complete for* co-NEXPTIME.

The co-NEXPTIME result for program complexity, which is not stated in [Schlipf 1995b], follows from an analogous result for datalog under fixpoint models in [Kolaitis and Papadimitriou 1991] and a simple, elegant transformation of this semantics to SMS [Schlipf 1995b].

For full logic programming, SMS has the same complexity as WFS.

Theorem 5.9 *([Schlipf 1995b; Marek et al. 1994]) Logic programming with negation under SMS is Π^1_1-complete.*

Further results on stable models of recursive (rather than only finite) logic programs can be found in [Marek et al. 1992].

Beyond inference, further complexity aspects of stable models have been analyzed, including compact representations of stable models and the well-founded semantics of nonground logic programs [Gottlob et al. 1996; Eiter et al. 1998], and optimization issues such as determining symmetries across stable models [Eiter et al. 1997b].

5.4 Inflationary and noninflationary semantics

The *inflationary semantics (INFS)* [Abiteboul and Vianu 1991a; Abiteboul et al. 1995] is inspired by inflationary fixpoint logic [Gurevich and Shelah 1986]. In place of T_P^∞, it uses the limit $\widetilde{T}_P^\infty$ of the sequence

$$\begin{aligned} \widetilde{T}_P^0 &= \emptyset, \\ \widetilde{T}_P^{i+1} &= \widehat{T}_P(\widetilde{T}_P^i), \quad \text{if } i \geq 0, \end{aligned}$$

where $\widetilde{T}_P$ is the *inflationary operator* $\widetilde{T}(I) = I \cup T_{P^I}(I)$. Clearly, $\widetilde{T}_P^\infty$ is computable in polynomial time for a propositional program P. Moreover, $\widetilde{T}_P^\infty$ coincides with T_P^∞ for Horn clause programs P. Therefore, by the above results,

Theorem 5.10 *([Abiteboul and Vianu 1991a]; implicit in [Gurevich and Shelah 1986]) Logic programming with negation under INFS is* P*-complete. Datalog with negation under INFS is data complete for* P *and program complete for* EXPTIME.

The *noninflationary semantics (NINFS)* [Abiteboul and Vianu 1991a], in the version of [Abiteboul and Vianu 1995, page 373], uses in place of T_P^∞ the limit $\widehat{T}_P^\infty$ of the sequence

$$\begin{aligned} \widehat{T}_P^0 &= \emptyset, \\ \widehat{T}_P^{i+1} &= \widehat{T}_P(\widehat{T}_P^i), \ \text{if } i \geq 0, \end{aligned}$$

where $\widehat{T}_P(I) = T_{P^I}(I)$, if it exists; otherwise, $\widehat{T}_P^\infty$ is undefined. Similar equivalent algebraic query languages have been earlier described in [Chandra and Harel 1982; Vardi 1982]. In particular, datalog under NINFS is equivalent to partial fixpoint logic [Abiteboul and Vianu 1991a; Abiteboul et al. 1995].

As easily seen, T_P^∞ is for a propositional program P computable in polynomial space; this bound is tight.

Theorem 5.11 *([Abiteboul and Vianu 1991a; Abiteboul et al. 1995]) Logic programming with negation under NINFS is* PSPACE*-complete. Datalog with negation under NINFS is data complete for* PSPACE *and program complete for* EXPSPACE.

5.5 Further semantics of negation

A number of interesting further semantics for logic programming with negation have been defined, among them partial stable models, maximal partial stable models, regular models, perfect models, 2- and 3-valued completion semantics, and fixpoint models; see e.g. [Schlipf 1995b; You and Yuan 1995; Przymusinski 1988a; Kolaitis and Papadimitriou 1991]. There is no space to discuss these semantics here; see e.g. [Schlipf 1995b; Saccà 1995; Dudakov 1999; Kolaitis and Papadimitriou 1991] for more details and complexity results. However, we remark that when a logic program has a perfect model, then this model is unique [Przymusinski 1988a; Przymusinski 1988b]. As recently shown in [Dudakov 1999], propositional logic programming under perfect model semantics is in Δ_2^p, and its precise complexity can be characterized through an interesting variant of the DTM with an oracle for the classical propositional satisfiability problem (SAT): if the SAT-instance in the query has more than one satisfying assignment, then the machine immediately rejects the input (i.e., changes its state to no rather than to $\in$). Deciding whether a given propositional logic program P has a perfect model (resp., $P \models A$ under perfect models), is complete for the class of languages accepted by such machines in polynomial time (resp., for the complementary class).

Extensions of logic programming with negation have been proposed which handle different kinds of negation, namely strong and default negation, see e.g. [Gelfond and Lifschitz 1991; Pearce and Wagner 1991]. The semantics we have considered above use default negation as the single kind of negation. Different kinds of negation increase the suitability of logic programming as a knowledge representation formalism [Baral and Gelfond 1994].

In the approach of [Gelfond and Lifschitz 1991], strong negation is interpreted as classical negation. E.g., the rule

$$\mathit{flies}(X) \leftarrow \sim \neg \mathit{flies}(X), \mathit{bird}(X)$$

naturally expresses that a bird flies by default; here, "$\sim$" is default negation and "$\neg$" is classical negation. The language of *extended logic programs* treats literals with classical negation as atoms, on which default negation may be applied. The notion of *answer set* for such a program is defined by a natural generalization of the concept of stable model [Gelfond and Lifschitz 1991].

As for the complexity, there is no increase for extended logic programs over normal logic programs under SMS.

Theorem 5.12 *([Ben-Eliyahu and Dechter 1994]) Given a propositional extended logic program P, deciding whether P has an answer set is* NP*-complete, and extended logic programming is* co-NP*-complete.*

Complexity results on extended logic programs with rule priorities can be found in [Brewka and Eiter 1998], and for an extension of logic programming using hierarchical modules in [Buccafurri et al. 1998].

6. DISJUNCTIVE LOGIC PROGRAMMING

Informally, *disjunctive logic programming (DLP)* extends logic programming by adding disjunction in the rule heads, in order to allow more natural and flexible knowledge representation. For example,

$$male(X) \vee female(X) \leftarrow person(X)$$

naturally represents that any person is either male or female.

A *disjunctive logic program* is a set of clauses

$$A_1 \vee \cdots \vee A_k \leftarrow L_1, \ldots, L_m \qquad (k \geq 1, m \geq 0), \tag{2}$$

where each A_i is an atom and each L_j is a literal. For a background, see [Lobo et al. 1992] and the more recent [Minker 1994].

The semantics of negation-free disjunctive logic programs is based on *minimal* Herbrand models, as the least (unique minimal) model does not exist in general.

Example 8. Let P consist of the single clause $p \vee q \leftarrow$. Then, P has the two minimal models $M_1 = \{p\}$ and $M_2 = \{q\}$.

Denote by $\mathrm{MM}(P)$ the set of minimal Herbrand models of P. The *Generalized Closed World Assumption (GCWA)* [Minker 1982] for negation-free P amounts to the meaning $\mathcal{M}_{GCWA}(P) = \{L \mid \mathrm{MM}(P) \models L\}$.

Example 9. Consider the following propositional program P', describing the behavior of a reviewer while reviewing a paper:

$$\begin{aligned} good \vee bad &\leftarrow paper \\ happy &\leftarrow good \\ angry &\leftarrow bad \\ smoke &\leftarrow happy \\ smoke &\leftarrow angry \\ paper &\leftarrow \end{aligned}$$

The following models of P' are minimal:

$$\begin{aligned} M_1 &= \{paper, good, happy, smoke\} \text{ and} \\ M_2 &= \{paper, bad, angry, smoke\}. \end{aligned}$$

Under GCWA, we have $P \models_{GCWA} smoke$, while $P \not\models_{GCWA} good$ and $P \not\models_{GCWA} \neg good$.

Theorem 6.1 *([Eiter and Gottlob 1993; Eiter et al. 1994]) Let P be a propositional negation-free disjunctive logic program and A be a propositional atom. (i) Deciding whether $P \models_{GCWA} A$ is* co-NP*-complete. (ii) Deciding whether $P \models_{GCWA} \neg A$ is Π_2^p-complete.*

PROOF. It is not hard to argue that for an atom A, we have $P \models_{GCWA} A$ if and only if $P \models_{PC} A$, where $\models_{PC}$ is the classical logical consequence relation. In addition, it is not hard to argue that any set of clauses can be represented by a suitable disjunctive logic program. Hence, by the well-known NP-completeness of SAT, part (i) is obvious.

Let us thus consider part (ii).

(1) *Membership.* We have $P \not\models_{GCWA} \neg A$ if and only if there exists an $M \in \mathrm{MM}(P)$ such that $M \not\models \neg A$, i.e., $A \in M$. Clearly, a guess for M can be verified with an oracle for NP in polynomial time; from this, membership of the problem in Π_2^p follows.

(2) *Hardness.* We show Π_2^p-hardness by an encoding of alternating Turing machines (ATM) [Chandra et al. 1981]. Recall that an ATM T has its set of states

partitioned into existential ($\exists$) and universal ($\forall$) states. If the machine reaches an $\exists$-state (respectively, $\forall$-state) s in a run, then the input is accepted if the computation continued in some (respectively, all) of the possible successor configurations is accepting. As in our simulations above, we assume that T has a single tape.

The polynomial-time bounded ATMs which start in a $\forall$-state s_0 and have one alternation, i.e., precisely one transition from a $\forall$-state to an $\exists$-state in each run (and no reverse transition), solve precisely the problems in Π_2^p [Chandra et al. 1981].

By adapting the construction in the proof of Theorem 5.7, we show how any such machine T on input I can be simulated by a disjunctive logic program P under GCWA. Without loss of generality, we assume that each run of T is polynomial-time bounded [Balcázar et al. 1990].

We start from the clauses constructed for the NTM T on input I in the proof of Theorem 5.7, from which we drop the clause $accept \leftarrow \neg accept$ and replace the clauses

$$\begin{aligned} B_{s,\sigma,i}[\tau] \;\leftarrow\; & \neg B_{s,\sigma,1}[\tau], \ldots, \neg B_{s,\sigma,i-1}[\tau], \\ & \neg B_{s,\sigma,i+1}[\tau], \ldots, \neg B_{s,\sigma,k}[\tau]. \end{aligned}$$

for s and σ by the logically equivalent disjunctive clause

$$B_{s,\sigma,1}[\tau] \vee \cdots \vee B_{s,\sigma,k}[\tau] \leftarrow \quad .$$

Intuitively, in a minimal model precisely one of the atoms $B_{s,\sigma,i}[\tau]$ will be present, which means that one of the possible branchings is followed in a run. The current clauses constitute a propositional program which derives *accept* under GCWA if and only if T would accept I if all its states were universal. We need to respect the $\exists$-states, however. For each $\exists$-state s and time point $\tau > 0$, we set up the following clauses, where s' is any $\exists$-state, $\tau \leq \tau' \leq N$, $0 \leq \pi \leq N$, and $1 \leq i \leq k$:

$$\begin{aligned} state_{s'}[\tau'] \;&\leftarrow\; naccept, state_s[\tau] \\ symbol_\sigma[\tau', \pi] \;&\leftarrow\; naccept, state_s[\tau] \\ cursor[\tau', \pi] \;&\leftarrow\; naccept, state_s[\tau] \\ B_{s,\sigma,i}[\tau'] \;&\leftarrow\; naccept, state_s[\tau]. \end{aligned}$$

Intuitively, these rules state that if a nonaccepting run enters an $\exists$-state, i.e., *naccept* is true, then all relevant facts involving a time point $\tau' \geq \tau$ are true. This way, nonaccepting runs are corrupted. Finally, we set up for each nonaccepting terminal $\exists$-state s the clauses

$$naccept \;\leftarrow\; state_s[\tau], \quad 0 < \tau \leq N.$$

These clauses state that *naccept* is true if the run ends in a nonaccepting state. Let P^+ be the resulting program. The minimal models M of P^+ which *do not contain naccept* correspond to the accepting runs of T.

It can be seen that the minimal models of P^+ which contain *naccept* correspond to the partial runs of T from the initial state s_0 to an $\exists$-state s from which no completion of the run ending in an accepting state is possible. This implies that P^+ has some minimal model M containing *naccept* precisely if T, by definition, does not accept input I. Consequently, $P^+ \models_{GCWA} \neg \mathit{naccept}$, i.e., *naccept* is in no minimal model of P^+, if and only if T accepts input I. It is clear that the program P^+ can be constructed in logarithmic space. Consequently, deciding $P \models_{GCWA} \neg A$ is Π_2^p-hard.

□

Note that many problems in the field of nonmonotonic reasoning are Π_2^p-complete, e.g. [Gottlob 1992; Eiter and Gottlob 1992; Eiter and Gottlob 1995a].

Stable negation naturally extends to disjunctive logic programs, by adopting that I is a *(disjunctive) stable model* of a disjunctive logic program P if and only if $I \in \mathrm{MM}(P^I)$ [Przymusinski 1991; Gelfond and Lifschitz 1991]. The disjunctive stable model semantics subsumes the disjunctive stratified semantics [Przymusinski 1988a]. For well-founded semantics, no such natural extension is known; the semantics in [Brass and Dix 1995; Przymusinski 1995] are the most appealing attempts in this direction.

Clearly, P^I is easily computed, and $P^I = P$ if P is negation-free. Thus,

Theorem 6.2 *([Eiter and Gottlob 1995b; Eiter et al. 1994; Eiter et al. 1997]) Propositional DLP under SMS is* Π_2^p *complete. Disjunctive datalog under SMS is data complete for* Π_2^p *and program complete for* co-NEXPTIME$^{\mathsf{NP}}$.

The latter result was derived by utilizing complexity upgrading techniques as described above in Section 4.3. We remark that a sophisticated algorithm for computing stable models of propositional disjunctive logic programs, which mirrors the complexity of the problem in its structure, is described in [Leone et al. 1997].

For full DLP, we have:

Theorem 6.3 *([Chomicki and Subrahmanian 1990]) DLP under GCWA is* Π_2^0-*complete.*

Theorem 6.4 *([Eiter and Gottlob 1995b]) Full DLP under SMS is* Π_1^1-*complete.*

Thus, disjunction adds complexity under GCWA and under SMS in finite Herbrand universes (unless co-NP $= \Pi_2^p$), but not in infinite ones. This is intuitively explained by the fact that DLP under SMS corresponds to a weak fragment of Π_2^1 which can be recursively translated to Π_1^1.

Many other semantics for DLP have been analyzed. For some of them, the complexity is lower than for SMS, for example for the coinciding possible worlds and possible model semantics [Chan 1993; Sakama and Inoue 1994a], as well as for the causal model semantics [Dix et al. 1996], which are all co-NP-complete. Others have higher complexity, for example the regular model semantics and the maximal partial stable model semantics [Eiter et al. 1998]. However, typically they are Π_2^p-complete in the propositional case.

Extended disjunctive logic programs (EDLPs), which have default and classical negation, are defined analogously to the case of non-disjunctive logic programs [Gelfond and Lifschitz 1991]. The notion of answer set is generalized in the same way as stable model from a non-disjunctive program to a disjunctive one. There is no complexity increase over disjunctive stable models; in particular, extended disjunctive logic programming is Π_2^p-complete in the propositional case [Eiter and Gottlob 1995b].

Fragments of EDLPs that have lower complexity are known. The most important such fragment are *headcycle-free programs*. Informally, an EDLP P is headcycle-free, if there are no two distinct atoms A and B which mutually depend on each other through positive recursion (i.e., default negation is disregarded), such that A and B occur in the head of the same rule of P. As shown in [Ben-Eliyahu and Dechter 1994], extended disjunctive logic programming for headcycle-free programs is co-NP-complete, and thus polynomial-time transformable to (disjunction-free) normal logic programming under stable model semantics.

A generalization of EDLPs by allowing default negation in the head has been studied in [Inoue and Sakama 1998]. As the authors show, the complexity of both arbitrary and headcycle-free programs does not increase. Other extensions of disjunctive logic programming and their complexities are studied in e.g. [Marek et al. 1995; Minker and Ruiz 1994; Buccafurri et al. 1997; Buccafurri et al. 1998; Buccafurri et al. 2000; Rosati 1997; Rosati 1998]. In particular, [Buccafurri et al. 1997; Buccafurri et al. 2000] analyzes the effect of different kinds of constraints on stable models. Weak constraints may be violated at a penalty, leading to a cost-based notion of stable models whose complexity is characterized as an optimization problem. In [Buccafurri et al. 1998], disjunctive logic programs are extended by classical negation and modularization with inheritance; as shown, these features do not increase the complexity. The papers [Rosati 1997; Rosati 1998] address the complexity of using epistemic operators such as minimal knowledge and belief in disjunctive logic programs.

7. EXPRESSIVE POWER OF LOGIC PROGRAMMING

The expressive power of query languages such as datalog is a topic common to database theory [Abiteboul et al. 1995] and finite model theory [Ebbinghaus and Flum 1995] that has attracted much attention by both communities. By the expressive power of a (formal) *query language*, we understand the set of all queries expressible in that language. Note that we will not only mention query languages used in database systems, but also formalisms used in formal logic and finite model theory such as first and second-order logic over finite structures or fixpoint logic (for precise definitions consult [Ebbinghaus and Flum 1995]).

In general, a *query* q defines a mapping $\mathcal{M}_q$ that assigns to each suitable input database D_{in} (over a fixed input schema) a result database $D_{out} = \mathcal{M}_q(D_{in})$ (over a fixed output schema); more logically speaking, a query defines global relations [Gurevich 1988]. For reasons of representation independence, a query should, in addition, be *generic*, i.e., invariant under isomorphisms. This means that if τ is a permutation of the domain $Dom(\mathcal{D})$, then $\mathcal{M}(\tau(D_{in})) = \tau(D_{out})$. Thus, when we speak about queries, we always mean generic queries.

Formally, the *expressive power* of a query language Q is the set of mappings $\mathcal{M}_q$ for all queries q expressible in the language Q by some *query expression* (program) E; this syntactic expression is commonly identified with the semantic query it defines, and simply (in abuse of definition) called a query.

There are two important research tasks in this context. The first is comparing two query languages Q_1 and Q_2 in their expressive power. One may prove, for instance, that $Q_1 \subsetneqq Q_2$, which means that the set of all queries expressible in Q_1 is a proper subset of the queries expressible in Q_2, and hence, Q_2 is strictly more expressive than Q_1. Or one may show that two query languages Q_1 and Q_2 have the same expressive power, denoted by $Q_1 = Q_2$, and so on.

The second research task, more related to complexity theory, is determining the absolute expressive power of a query language. This is mostly achieved by proving that a given query language Q is able to express exactly all queries whose evaluation complexity is in a complexity class $\mathcal{C}$. In this case, we say that Q *captures* $\mathcal{C}$ and write simply $Q = \mathcal{C}$. The *evaluation complexity* of a query is the complexity of checking whether a given atom belongs to the query result, or, in the case of Boolean queries, whether the query evaluates to *true* [Vardi 1982; Gurevich 1988].

Note that there is a substantial difference between showing that the query evaluation problem for a certain query language Q is $\mathcal{C}$-complete and showing that Q captures $\mathcal{C}$. If the evaluation problem for Q is $\mathcal{C}$-complete, then *at least one* $\mathcal{C}$-hard query is expressible in Q. If Q captures $\mathcal{C}$, then Q expresses *all* queries evaluable in $\mathcal{C}$ (including, of course, all $\mathcal{C}$-hard queries). Thus, usually proving that Q captures $\mathcal{C}$ is much more involved than proving that evaluating Q-queries is $\mathcal{C}$-hard. Note also that it is possible that a query language Q captures a complexity class $\mathcal{C}$ for which no complete problems exist or for which no such problems are known. As an example, second-order logic over finite structures captures the polynomial hierarchy PH, for which no complete problem is known. However, the existence of a complete problem of PH would imply that it collapses at some finite level, which is widely believed to be false.

The subdiscipline of database theory and finite model theory dealing with the description of the expressive power of query languages and related logical formalisms via complexity classes is called *descriptive complexity theory* [Immerman 1987; Leivant 1989; Immerman 1999]. An early foundational result in this field was [Fagin 1974]'s theorem stating that existential second-order logic captures NP. In the eighties and nineties, descriptive complexity theory has become a flourishing discipline with many deep and useful results.

To prove that a query language Q captures a machine-based complexity class $\mathcal{C}$, one usually shows that each $\mathcal{C}$-machine with (encodings of) finite structures as inputs that computes a generic query can be represented by an expression in language Q. There is, however, a slight mismatch between ordinary machines and logical queries. A Turing machine works on a string encoding of the input database D. Such an encoding provides an implicit *linear order* on D, in particular, on all elements of the universe U_D. The Turing machine can take profit of this order and use this order in its computations (as long as genericity is obeyed). On the other hand, in logic or database theory, the universe U_D is a pure set and thus unordered. For "powerful" query languages of inherent nondeterministic nature at the level of NP this is not a problem, since an ordering on U_D can be nondeterministically guessed. However, for many query languages, in particular, for those corresponding to complexity classes below NP, generating a linear order is not feasible. Therefore, one often assumes that a linear ordering of the universe elements is predefined, i.e., given explicitly in the input database. More specifically, by *ordered databases* or *ordered finite structures*, we mean databases whose schemas contain special relation symbols *Succ*, *First*, and *Last*, that are always interpreted such that $Succ(x, y)$ is a successor relation of some linear order and $First(x)$ determines the first element and $Last(x)$ the last element in this order. The importance of predefined linear orderings becomes evident in the next two theorems.

Before coming to the theorems, we must highlight another small mismatch between the Turing machine and the datalog setting. A Turing machine can consider each input bit independently of its value. On the other hand, a plain datalog program is not able to detect that some atom is *not* a part of the input database. This is due to the representational peculiarity that only positive information is present in a database, and that the negative information is understood via the closed world assumption. To compensate this deficiency, we will slightly augment the syntax of datalog. *Throughout this section, we will assume that input predicates may appear negated in datalog rule bodies; the resulting language is* datalog$^+$. This extremely limited form of negation is much weaker than stratified negation, and could be easily circumvented by adopting a different representation for databases.

Theorem 7.1 *(a fortiori from [Chandra and Harel 1982]) Datalog*$^+ \subsetneq \mathsf{P}$.

PROOF. (Hint.) Show that there exists no datalog$^+$ program P that can tell whether the universe U of the input database has an even number of elements. □

Clearly, plain datalog (without negation of the input predicates) can only define *monotonic queries*, i.e., the output grows monotonically with the input, and thus datalog can not express all queries computable in polynomial time. The natural question is thus to ask whether datalog expresses all monotone queries computable in polynomial time. As shown in [Afrati et al. 1995], the answer is negative. In particular, datalog$^{\neq}$ (i.e., datalog augmented by inequality) can not express whether a given set of linear constraints of the form $x + y + z = 1$ or $x = 0$ is inconsistent, even on ordered databases [Afrati et al. 1995]. Furthermore, deciding whether a directed graph has path with length a *perfect square* is not expressible in datalog$^{+,\neq}$ (datalog$^+$ with inequality). The language datalog$^{\neq}$ was first studied by [Shmueli 1987], who showed that is more expressive than plain datalog. Properties and expressiveness aspects of this language have been further studied e.g. in [Gaifman et al. 1987; Lakshmanan and Mendelzon 1989; Ajtai and Gurevich 1994; Kolaitis and Vardi 1995; Afrati 1997].

The *perfect square* query is expressible in datalog$^{+,\neq}$ on ordered databases, however. This is a corollary to the next result.

Theorem 7.2 *([Papadimitriou 1985; Grädel 1992]; implicit in [Vardi 1982; Immerman 1986; Leivant 1989]) On ordered databases, datalog*$^+$ *captures* P.

PROOF. (Sketch) By Theorem 5.1, query answering for a fixed datalog$^+$ program is in P. It thus remains to show that each polynomial-time DTM T on finite input databases $D \in INST(\mathcal{D}_{in})$ can be simulated by a datalog$^+$ program. To show this, we first make some simplifying assumptions.

(1) The universe U_D is an initial segment $[0, n-1]$ of the integers, and *Succ*, *First*, and *Last* are from the natural linear ordering over this segment.

(2) The input database schema $\mathcal{D}_{in}$ consists of a single binary relation G, plus the predefined predicates *Succ*, *First*, *Last*. In other words, D is always (an ordered) graph $\langle U, G\rangle$.

(3) T operates in $< n^k$ steps, where $n = |U| > 1$.

(4) T computes a Boolean (0-ary) predicate.

The simulation is akin to the simulation used in the proofs of Theorems 4.2 and 4.5.

Recall the framework of Section 4.1. In the spirit of this framework, it suffices to encode n^k time-points τ and tape-cell numbers π within a fixed datalog program. This is achieved by considering k-tuples $\mathbf{X} = \langle X_1, \ldots, X_k\rangle$ of variables X_i ranging over U. Each such k-tuple encodes the integer $int(\mathbf{X}) = \sum_{i=1}^{k} X_i \cdot n^{k-i}$.

At time point 0 the tape of T contains an encoding of the input graph. Recall that in Section 4.1 this was reflected by the following initialization facts

$$symbol_\sigma[0, \pi] \leftarrow \qquad \text{for } 0 \leq \pi < |I|, \text{ where } I_\pi = \sigma.$$

Before translating these rules into appropriate datalog rules, we shall spend a word about how input graphs are usually represented by a binary strings. A graph $\langle U, G\rangle$ is encoded by binary string $enc(U, G)$ of length $|U|^2$: if $G(i, j)$ is true for $i, j \in U = [0, n-1]$ then the bit number $i * n + j$ of $enc(U, G)$ is 1, otherwise this bit is 0. The bit positions of $enc(U, G)$ are exactly the integers from 0 to $n^2 - 1$. These integers are represented by all k-tuples $\langle 0^{k-2}, a, b\rangle$ such that $a, b \in U$. Moreover, the bit-position $int(\langle 0^{k-2}, X, Y\rangle)$ of $enc(U, G)$ is 1 if and only if $G(X, Y)$ is true in the input database and 0 otherwise.

The above initialization rules can therefore be translated into the datalog rules

$$\begin{array}{rcl} symbol_1[0^k, 0^{k-2}, X, Y] & \leftarrow & G(X, Y) \\ symbol_0[0^k, 0^{k-2}, X, Y] & \leftarrow & \neg G(X, Y) \end{array}$$

Intuitively, the first rule says that at time point $0 = int(0^k)$, the bit number $int(\langle 0^{k-2}, X, Y\rangle)$ on the tape is 1 if $G(X, Y)$ is true. The second rule states that the same bit is false if $G(X, Y)$ is false. Note that the second rule applies negation to an input predicate. *Only this rule in the entire datalog$^+$ program uses negation.* Clearly, these two rules simulate that at time point 0, the cells $c_0, \ldots, c_{n^2-1}$ contain precisely the string $enc(U, G)$.

The other initialization rules described in Section 4.1 are also easily translated into appropriate datalog rules. Let us now see how the other rules are translated into datalog.

From the linear order given by $Succ(X, Y)$, $First(X)$, and $Last(X)$, it is easy to define by datalog clauses a linear order $\leq^k$ on k-tuples $Succ^k(\mathbf{X}, \mathbf{Y})$, $First^k(\mathbf{X})$, $Last^k(\mathbf{X})$ (see the proof of Theorem 4.5), by using $Succ^1 = Succ$, $First^1 = First$ and $Last^1 = Last$. By using $Succ^k$, transition rules, inertia rules and the accept rules are easily translated into datalog as in the proof of Theorem 4.5.

The output schema of the resulting datalog program P^+ is defined to be $\mathcal{D}_{out} = \{accept\}$. It is clear that this program evaluates to *true* on input $D = \langle U, G\rangle$, i.e., $P^+ \cup D \models accept$ if and only if T accepts $enc(U, G)$.

The generalization to a setting where the simplifying assumptions 1–3 are not made is rather straightforward and is omitted. Assumption 4 can also be easily lifted to the computation of output predicates. We consider here the case where the output scheme $\mathcal{D}_{out}$ contains a single binary relation R. Then, the output database D' computed by T, which is a graph $\langle U, R\rangle$, can be encoded similarly to the input database as a binary string $enc(U,R)$ of length $|U|^2$. We may suppose that when the machine enters the halt state, this string is contained in the first $|U|^2$ cells of the tape. To obtain the positive facts of the output relation R, we add the following rule:

$$R(X,Y) \leftarrow symbol_1[\mathbf{Y}, 0^{k-2}, X, Y]),\ state_{\mathtt{halt}}[\mathbf{Y}]$$

□

We remark that a result similar to Theorem 7.2 was independently obtained by [Livchak 1983].

Let us now state somewhat more succinctly further interesting results on datalog. A prominent query language is *fixpoint logic* (*FPL*), which is the extension of first-order logic by a least fixpoint operator $lfp(\mathbf{X}, \varphi, S)$, where S is a $|\mathbf{X}|$-ary predicate occurring positively in the formula $\varphi = \varphi(\mathbf{X}; S)$, and $\mathbf{X}$ is a tuple of free variables in φ; intuitively, it returns the least fixpoint of the operator Γ defined by $\Gamma(S) = \{\mathbf{a} \mid D \models \varphi(\mathbf{a}; S)\}$. We refer to [Chandra and Harel 1982; Abiteboul et al. 1995; Ebbinghaus and Flum 1995] for details. As shown in [Chandra and Harel 1982], *FPL* expresses a proper subset of the queries in P. Datalog$^+$ relates to *FPL* as follows.

Theorem 7.3 *([Chandra and Harel 1985]) Datalog$^+$ = FPL$^+(\exists)$, i.e., Datalog$^+$ coincides with the fragment of FPL having negation restricted to database relations and only existential quantifiers.*

As for expressibility in first-order logic, [Ajtai and Gurevich 1994] have shown that a datalog query is equivalent to a first-order formula if and only if it is bounded, and thus expressible in existential first-order logic.

Adding stratified negation does not preserve the equivalence of datalog and fixpoint logic in Theorem 7.3.

Theorem 7.4 *([Kolaitis 1991]; implicit in [Dahlhaus 1987]) Stratified datalog $\subsetneq$ FPL.*

This theorem is not obvious. In fact, for some time coincidence of the two languages was assumed, based on a respective statement in [Chandra and Harel 1985].

The nonrecursive fragment of datalog coincides with well-known database query languages.

Theorem 7.5 *(cf. [Abiteboul et al. 1995]) Nonrecursive range-restricted datalog with negation = relational algebra = relational calculus. Nonrecursive datalog with negation = first-order logic (without function symbols).*

The expressive power of relational algebra is equivalent to that of a fragment of the database query language SQL (essentially, SQL without grouping and aggregate functions). The expressive power of SQL is discussed in [Libkin and Wong 1994; Dong et al. 1997; Libkin 1997].

Unstratified negation yields higher expressive power.

Theorem 7.6 (i) *Datalog under WFS = FPL ([van Gelder 1989]).*
(ii) *Datalog under INFS = FPL ([Abiteboul and Vianu 1991a], using [Gurevich and Shelah 1986]).*

As recently shown, the first result holds also for total WFS (i.e., the well-founded model is always total) [Flum et al. 1997].

We remark that the variants of datalog mentioned above can only define queries which are expressible in infinitary logic with finitely many variables ($L^{\omega}_{\infty\omega}$) [Kolaitis and Vardi 1995]. It is known that $L^{\omega}_{\infty\omega}$ has a 0-1 law, i.e., every query definable in this language is either almost surely true or almost surely false, if the size of the universe grows to infinity [Kolaitis and Vardi 1992]. It is easy to see that the boolean *Even-query* q_E, which tells if the domain of a given input database D_{in} (over a fixed schema) contains an even number of elements, is not almost surely true or almost surely false. Thus, *a fortiori*, this query– which is computable in polynomial time– is not expressible in the above variants of datalog.

On ordered databases, Theorem 7.2 and the theorems in Section 5 imply

Theorem 7.7 *On ordered databases, the following query languages capture* P*: stratified datalog, datalog under INFS, and datalog under WFS.*

Syntactical restrictions allow us to capture classes within P. Let datalog$^+(1)$ be the fragment of datalog$^+$ where each rule has most one nondatabase predicate in the body, and let datalog$^+(1,d)$ be the fragment of datalog$^+(1)$ where each predicate occurs in at most one rule head.

Theorem 7.8 *([Grädel 1992; Veith 1994]) On ordered databases, datalog$^+(1)$ captures* NL *and datalog$^+(1,d)$ captures* L.

Due to inherent nondeterminism, stable semantics is much more expressive.

Theorem 7.9 *([Schlipf 1995b]) Datalog under SMS captures* co-NP.

Note that for this result an order on the input database is not needed. Informally, in each stable model such an ordering can be guessed and checked by the program. By [Fagin 1974]'s Theorem, this implies that datalog under SMS is equivalent to the existential fragment of second-order logic over finite structures.

Theorem 7.10 *([Abiteboul and Vianu 1991a]) On ordered databases, datalog under NINFS captures* PSPACE.

Here ordering is needed. An interesting result in this context, formulated in terms of datalog, is the following [Abiteboul and Vianu 1991a]: datalog under INFS = datalog under NINFS *on arbitrary finite databases* if and only if P = PSPACE. While the "only if" direction is obvious, the proof of the "if" direction is involved. It is one of the rare examples that translates open relationships between deterministic complexity classes into corresponding relationships between query languages.

We next briefly address the expressive power of disjunctive logic programs.

Theorem 7.11 *([Eiter et al. 1994; Eiter et al. 1997]) Disjunctive datalog under SMS captures Π_2^p.*

It appears that fragments of disjunctive datalog have interesting properties. While disjunctive datalog$^{+,\neq}$ expresses only a subset of the queries in co-NP (e.g., it can not express the Even-query), it expresses all of Σ_2^p under the credulous notion of consequence, i.e., $P \models_c A$ if A is true in some stable model. Furthermore, under credulous consequence every query in nondisjunctive datalog$^{+,\neq}$ is expressible in disjunctive datalog^{+}, even though the inequality predicate can not be recognized.

Finally, we consider full logic programs. In this case, the input databases are arbitrary (not necessarily recursive) relations on the genuine (infinite) Herbrand universe of the program.

Theorem 7.12 *[Schlipf 1995b; Eiter and Gottlob 1997] Each of logic programming under WFS, logic programming under SMS, and DLP under SMS captures Π_1^1.*

Thus, different from the function-free case, adding disjunction does not increase the expressive power of normal logic programs. The reason is that disjunctive logic programs can be expressed in a weak fragment of the class Π_2^1 of second-order logic, which in the case of an infinite Herbrand universe can be coded to the Π_1^1 fragment.

For further expressiveness results on logic programs see e.g. [Schlipf 1995b; Saccà 1995; Saccà 1997; Greco and Saccà 1997; Greco and Saccà 1996; Eiter et al. 1998; Cadoli and Palopoli 1998]. In particular, co-NP can be captured by a variant of circumscribed datalog [Cadoli and Palopoli 1998], and further classes of the polynomial hierarchy can be captured by variants of stable models [Saccà 1995; Saccà 1997; Eiter et al. 1998; Buccafurri et al. 1997] as well as through modular logic programming [Eiter et al. 1997; Eiter et al. 2000; Buccafurri et al. 1998]. Results on the expressiveness of the stable model semantics over disjunctive databases, which are given by sets of ground clauses rather than facts, can be found in [Bonatti and Eiter 1996].

We conclude this subsection with a brief look on expressiveness results for nondeterministic queries. A *nondeterministic query* maps an input database to one from a set of possible output databases; it can be viewed as a multi-valued function. For example, a query which returns as output a Hamiltonian cycle of given input graph is a nondeterministic query. The (deterministic) queries that we have considered above are a special case of nondeterministic queries.

It has been shown that the class NDB-P of nondeterministic queries which are computable in polynomial time can be captured by suitable nondeterministic variants of datalog, e.g., by a procedure-style variants [Abiteboul and Vianu 1991a], by datalog$^{\neq}$ (datalog with inequality) extended with a choice operator, or by datalog with stable models [Corciulo et al. 1993; Giannotti and Pedreschi 1998]. Also NDB-PSPACE, the class of nondeterministic queries computable in polynomial space, is captured by a nondeterministic variant of datalog [Abiteboul and Vianu 1991a]. For a tutorial survey of such and related deterministic languages, we recommend

[Vianu 1997]. For further issues on nondeterministic queries, we refer to [Giannotti et al. 1997; Grumbach and Lacroix 1997; Leone et al. 1999].

7.1 The order mismatch and relational machines

Many results on capturing the complexity classes by logical languages suffer from the *order mismatch*. For example, the results by Immerman and Vardi (Theorems 7.7 and 7.10) show that $\mathsf{P} = \mathsf{PSPACE}$ if and only if Datalog under INFS and Datalog under NINFS coincide on *ordered databases*. The order appears when we code the input for a standard computational device, like a Turing machine, while the semantics of Datalog and logic is defined directly in terms of logical structures, where no order on elements is given.

To overcome this mismatch, [Abiteboul and Vianu 1991b; Abiteboul and Vianu 1995] introduced *relational complexity theory*, where computations on unordered structures are modeled by *relational machines*. In [Abiteboul and Vianu 1991b; Abiteboul and Vianu 1995; Abiteboul et al. 1997] several relational complexity classes are introduced, such as P_r (*relational polynomial time*), NP_r (*relational nondeterministic polynomial time*), PSPACE_r (*relational polynomial space*) and $\mathsf{EXPTIME}_r$ (*relational exponential time*). It follows that all separation results among the standard complexity classes translate into separation results among relational complexity classes. For example, $\mathsf{P} = \mathsf{NP}$ if and only if $\mathsf{P}_r = \mathsf{NP}_r$.

It happens that Datalog under various semantics captures the relational complexity classes on unordered databases. For example (cf. Theorems 7.7 and 7.10), we have

Theorem 7.13 *Datalog under INFS captures* P_r. *Datalog under NINFS captures* PSPACE_r.

Note that together with the correspondence of the separation results between the standard complexity classes and the relational complexity classes, this theorem implies that Datalog under INFS coincides with Datalog under NINFS if and only if $\mathsf{P} = \mathsf{PSPACE}$. Therefore, the results of [Abiteboul and Vianu 1991b; Abiteboul and Vianu 1995; Abiteboul et al. 1997] provide an order-free correspondence between questions in computational and descriptive complexity.

7.2 Expressive power of logic programming with complex values

The expressive power of datalog queries is defined in terms of input and output databases, i.e., finite sets of tuples. In order to extend the notion of expressive power to logic programming with complex values, we need to define what we mean by an input. For example, in the case of plain logic programming, an input may be a finite set of ground terms, i.e. a finite set of trees. In the case of logic programming with sets, an input may be a set whose elements may be sets too and so on.

Various models and languages for dealing with complex values in databases have been proposed, e.g. [Abiteboul and Kanellakis 1989; Abiteboul and Grumbach 1988; Kifer and Wu 1993; Kifer et al. 1995; Abiteboul and Beeri 1995; Buneman et al. 1995; Suciu 1997; Greco et al. 1995; Libkin et al. 1996; Abiteboul et al. 1995]. The functional approach to such languages dominates the logic programming one. To extend variants of nested relational algebra as in [Buneman et al. 1995] to datalog, bounded fixpoint constructs have been proposed [Suciu 1997], as well as deflationary fixpoint constructs [Colby and Libkin 1997].

The comparative expressive power of languages for complex values is studied in e.g. [Abiteboul and Grumbach 1988; Vadaparty 1991; Suciu 1997; Abiteboul and Beeri 1995; Dantsin and Voronkov 2000]. For example, [Abiteboul and Beeri 1995] introduce a model for restricted combinations of tuples and sets and several corresponding query languages, including the algebraic and logic programming ones. It is proved that all these languages define the same class of queries. [Dantsin and Voronkov 2000] show that nonrecursive logic programming with negation has the same expressive power as nonrecursive datalog with negation (under a natural representation of inputs). Thus, the use of recursive data structures, namely trees, in nonrecursive datalog gives no gain in the expressiveness. It follows from this result that nonrecursive logic programming with negation is polynomial-time. [McAllester 1993; Givan and McAllester 2000] study logic programs without negation in which every term occurring in the head of a clause also occurs in its body. It is proved that this class captures P on ground terms (one can define a linear order on the set of ground terms using logic programs of this kind).

The absolute expressive power of languages for complex values is also studied in [Sazonov 1993; Suciu 1997; Sazonov and Lisitsa 1995; Grumbach and Vianu 1995; Gyssens et al. 1995; Lisitsa and Sazonov 1997]; further issues, such as expressibility of particular queries or faithful extension of datalog, are studied in [Libkin and Wong 1995; Wong 1996; Paredaens and van Gucht 1992].

Results on the expressive power of different forms of logic programming with constraints can be found e.g. in [Cosmadakis and Kuper 1994; Kanellakis et al. 1995; Benedikt et al. 1996; Vandeurzen et al. 1996].

Unlike research on the expressive power of datalog, there is no mainstream in research on the expressive power of logic programming with complex values. Extension of declarative query languages by complex values is more actively studied in database theory.

8. UNIFICATION AND ITS COMPLEXITY

What is the complexity of query answering for very simple logic programs consisting of one fact? This problem leads us to the problem of solving equations over terms, known as the *unification problem*. Unification lies in the very heart of implementations of logic programming and automated reasoning systems.

Atoms or terms s and t are called *unifiable* if there exists a substitution ϑ that makes them equal, i.e., the terms $s\vartheta$ and $t\vartheta$ coincide; such a substitution ϑ is called a *unifier* of s and t. The unification problem is the following decision problem: given terms s and t, are they unifiable?

[Robinson 1965] described an algorithm that solves this problem and, if the answer is positive, computes a most general unifier of given two terms. His algorithm had exponential time and space complexity mainly because of the representation of terms by strings of symbols. Using better representations (for example, by directed acyclic graphs), Robinson's algorithm was improved to linear time algorithms, e.g. [Martelli and Montanari 1976; Paterson and Wegman 1978].

Theorem 8.1 *([Dwork et al. 1984; Yasuura 1984; Dwork et al. 1988]) The uni-*

fication problem is P*-complete.*

P-hardness of the unification problem was proved by reductions from some versions of the circuit value problem in [Dwork et al. 1984; Yasuura 1984; Dwork et al. 1988]. (Note that [Lewis and Statman 1982] states that unifiability is complete for co-NL; however, [Dwork et al. 1984] gives a counterexample to the proof in [Lewis and Statman 1982].)

Also, many quadratic time and almost linear time unification algorithms have been proposed because these algorithms are often more suitable for applications and generalizations (see a survey of the main unification algorithms in [Baader and Siekmann 1994]). Here we mention only [Martelli and Montanari 1982]'s algorithm based on ideas going back to [Herbrand 1972]'s famous work. Modifications of this algorithm are widely used for unification in equational theories and rewriting systems. The time complexity of Martelli and Montanari's algorithm is $O(nA^{-1}(n))$ where A^{-1} is a function inverse to Ackermann's function and thus A^{-1} grows very slowly.

9. LOGIC PROGRAMMING WITH EQUALITY

The relational model of data deals with simple values, namely tuples consisting of atomic components. Various generalizations and formalisms have been proposed to handle more complex values like nested tuples, tuples of sets, etc; see Section 7.2 and [Abiteboul and Beeri 1995]. Most of these formalisms can be expressed in terms of logic programming with equality [Gallier and Raatz 1986; Gallier and Raatz 1989; Hölldobler 1989; Hanus 1994; Degtyarev and Voronkov 1996] and constraint logic programming considered in Section 10.

9.1 Equational theories

Let $\mathcal{L}$ be a language containing the equality predicate $=$. By an *equation* over $\mathcal{L}$ we mean an atom $s = t$ where s and t are terms in $\mathcal{L}$. An *equational theory* E over $\mathcal{L}$ is a set of equations closed under the logical consequence relation, i.e., a set satisfying the following conditions: (i) E contains the equation $x = x$; (ii) if E contains $s = t$ then E contains $t = s$; (iii) if E contains $r = s$ and $s = t$ then E contains $r = t$; (iv) if E contains $s_1 = t_1, \ldots, s_n = t_n$ then E contains $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ for each n-ary function symbol $f \in \mathcal{L}$; and (v) if E contains $s = t$ then E contains $s\vartheta = t\vartheta$ for all substitutions ϑ.

The syntax of *logic programs over an equational theory* E coincides with that of ordinary logic programs. Their semantics is defined as a generalization of the semantics of logic programming so that terms are identified if they are equal in E.

Example 10. We demonstrate logic programs with equality by a logic program processing finite sets. Finite sets are a typical example of complex values handled in databases. We represent finite sets by ground terms as follows: (i) the constant $\{\}$ denotes the empty set, (ii) if s represents a set and t is a ground term then $\{t \mid s\}$ represents the set $\{t\} \cup s$ (where $\{t\}$ and s are not necessarily disjoint). However the equality on sets is defined not as identity of terms but as equality in the equational theory in which terms are considered to be equal if and only if they represent equal sets (we omit the axiomatization of this theory).

Consider a very simple program that checks whether two given sets have a nonempty intersection. This program consists of one fact

$$non\_empty\_intersection(\{X \mid Y_1\}, \{X \mid Y_2\}) \leftarrow .$$

For example, to check that the sets $\{1, 3, 5\}$ and $\{4, 1, 7\}$ have a common member, we ask the query $non\_empty\_intersection(\{1, 3, 5\}, \{4, 1, 7\})$. The answer will be positive. Indeed, the following system of equations has solutions in the equational theory of sets:

$$\{X \mid Y_1\} = \{1, 3, 5\},\ \{X \mid Y_2\} = \{4, 1, 7\}.$$

For example, set $X = 1$, $Y_1 = \{3, 5\}$, $Y_2 = \{4, 7\}$.

Note that if we represent sets by lists in plain logic programming without equality, any encoding of *non_empty_intersection* will require recursion.

The complexity of logic programs over E depends on the complexity of solving systems of term equations in E. The problem of whether a system of term equations is solvable in an equational theory E is known as the problem of *simultaneous E-unification*.

A substitution ϑ is called an *E-unifier* of terms s and t if the equation $s\vartheta = t\vartheta$ is a logical consequence of the theory E. By the *E-unification problem* we mean the problem of whether there exists an E-unifier of two given terms. Ordinary unification can be viewed as E-unification where E contains only trivial equations $t = t$. It is natural to think of an E-unifier of s and t as a *solution* to the equation $s = t$ in the theory E.

9.2 Complexity of E-unification

Solving equations is a traditional subject of all mathematics. Since any result on solving equation systems can be viewed as a result on E-unification, it is thus practically impossible to overview all results on the complexity of E-unification. Therefore, we restrict this survey to only few cases closely connected with logic programming. The general theory of E-unification may be found e.g. in [Baader and Siekmann 1994].

Let E be an equational theory over $\mathcal{L}$ and $\cdot$ be a binary function symbol in $\mathcal{L}$ (written in the infix form). We call $\cdot$ an *associative* symbol if E contains the equation $x \cdot (y \cdot z) = (x \cdot y) \cdot z$, where x, y and z are variables. Similarly, $\cdot$ is called an *AC-symbol* (an abbreviation for an associative-commutative symbol) if $\cdot$ is associative and, in addition, E contains $x \cdot y = y \cdot x$. If $\cdot$ is an AC-symbol and E contains $x \cdot x = x$, we call $\cdot$ an *ACI-symbol* (I stands for idempotence). Also, $\cdot$ is called an *AC1-symbol* (or an *ACI1-symbol*) if $\cdot$ is an AC-symbol (an ACI-symbol respectively) and E contains the equation $x \cdot 1 = x$ where 1 is a constant belonging to $\mathcal{L}$.

Theorem 9.1 *([Makanin 1977; Baader and Schulz 1992; Benanav et al. 1987; Kościelski and Pacholski 1996]) Let E be an equational theory defining a function symbol $\cdot$ in $\mathcal{L}$ as an associative symbol (E contains all logical consequences of $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ and no other equations). The following upper and lower bounds on the complexity of the E-unification problem hold: (i) this problem is in 3-*NEXPTIME*, (ii) this problem is* NP*-hard.*

Basically, all algorithms for unification under associativity are based on [Makanin 1977]'s algorithm for word equations. The 3-NEXPTIME upper bound is obtained in [Kościelski and Pacholski 1996].

The following theorem characterizes other popular kinds of equational theories.

Theorem 9.2 *([Kapur and Narendran 1986; Kapur and Narendran 1992; Baader and Schulz 1996]) Let E be an equational theory defining some symbols as one of the following: AC-symbols, ACI-symbols, $AC1$-symbol, or $ACI1$-symbols (there can be one or more of these kinds of symbols). Suppose the theory E contains no other equations. Then the E-unification problem is* NP*-complete.*

9.3 Complexity of nonrecursive logic programming with equality

In the case of ordinary unification, there is a simple way to reduce solvability of finite systems of equations to solvability of single equations. However, these two kinds of solvability are not equivalent for some theories: there exists an equational theory E such that the solvability problem for one equation is decidable, while solvability for (finite) systems of equations is undecidable [Narendran and Otto 1990].

Simultaneous E-unification determines decidability of nonrecursive logic programming over E.

Theorem 9.3 *(implicit in [Dantsin and Voronkov 1997]) Let E be an equational theory. Nonrecursive logic programming over E is decidable if and only if the problem of simultaneous E-unification is decidable.*

An equational theory E is called NP-*solvable* if the problem of solvability of equation systems in E is in NP. For example, the equational theory of finite sets mentioned above, the equational theory of bags (i.e. finite multisets) and the equational theory of trees (containing only equations $t = t$) are NP-solvable [Dantsin and Voronkov 1999].

Theorem 9.4 *([Dantsin and Voronkov 1997; Dantsin and Voronkov 1999]) Nonrecursive logic programming over an* NP*-solvable equational theory E is in* NEXPTIME*. Moreover, if E is a theory of trees, or bags, or finite sets, or any combination of them, then nonrecursive logic programming over E is also* NEXPTIME*-complete.*

10. CONSTRAINT LOGIC PROGRAMMING

Informally, *constraint logic programming (CLP)* extends logic programming by involving additional conditions on terms. These conditions are expressed by *constraints*, i.e., equations, disequations, inequations etc. over terms. The semantics of such constraints is predefined and does not depend on logic programs.

Example 11. We illustrate CLP by the standard example. Suppose that we would like to solve the following puzzle:

$$\begin{array}{ccccc} + & S & E & N & D \\ & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

All these letters are variables ranging over decimal digits $0, 1, \ldots, 9$. As usual, different letters denote different digits and $S, M \neq 0$. This puzzle can be solved by a constraint logic program over the domain of integers $(Z, =, \neq, \leq, +, \times, 0, 1, \ldots)$. Informally, this program can be written as follows.

$$
\begin{array}{l}
\mathit{find}(S, E, N, D, \quad M, O, R, E, \quad M, O, N, E, Y) \leftarrow \\
\quad 1 \leq S \leq 9, \ \ldots, \ 0 \leq Y \leq 9, \\
\quad S \neq E, \ \ldots, \ R \neq Y, \\
\quad 1000 \cdot S + 100 \cdot E + 10 \cdot N + D + \\
\quad 1000 \cdot M + 100 \cdot O + 10 \cdot R + E = \\
\quad 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y
\end{array}
$$

The query $\mathit{find}(S, E, N, D, \quad M, O, R, E, \quad M, O, N, E, Y)$ will be answered by the only solution

$$
\begin{array}{cccccc}
+ & & 9 & 5 & 6 & 7 \\
 & & 1 & 0 & 8 & 5 \\
\hline
 & 1 & 0 & 6 & 5 & 2
\end{array}
$$

A ***structure*** is defined by an interpretation I of a language $\mathcal{L}$ in a nonempty set D. For example, we shall consider the structure defined by the standard interpretation of the language consisting of the constant 0, the successor function symbol s and the equality predicate $=$ on the set N of natural numbers. This structure is denoted by $(\mathsf{N}, =, s, 0)$. Other examples of structures are obtained by replacing N by the sets Z (the integers), Q (the rational numbers), R (the reals) or C (the complex numbers). Below we denote structures in a similar way, keeping in mind the standard interpretation of arithmetic function symbols in number sets. The symbols $\times$ and $/$ stand for multiplication and division respectively. We use $k \cdot x$ to denote unary functions of multiplication by particular numbers (of the corresponding domain); x^k is used similarly. All structures under consideration are assumed to contain the equality symbol.

Let S be a structure. An atom $c(t_1, \ldots, t_k)$ where $t_1, \ldots, t_k$ are terms in the language of S is called a ***constraint***. By a ***constraint logic program over S*** we mean a finite set of rules

$$p(\mathbf{X}) \leftarrow c_1, \ldots, c_m, q_1(\mathbf{X}_1), \ldots, q_n(\mathbf{X}_n)$$

where $c_1, \ldots, c_m$ are constraints, $p, q_1, \ldots, q_n$ are predicate symbols not occurring in the language of S, and $\mathbf{X}, \mathbf{X}_1, \ldots, \mathbf{X}_n$ are lists of variables. The semantics of CLP is defined as a natural generalization of semantics of logic programming, e.g. [Jaffar and Maher 1994]. If S contains function symbols interpreted as tree constructors (i.e. equality of corresponding terms is interpreted as ordinary unification) then CLP over S is an extension of logic programming. Otherwise, CLP over S can be regarded as an extension of Datalog by constraints.

10.1 Complexity of constraint logic programming

There are two sources of complexity in CLP: complexity of solving systems of constraints and complexity coming from the logic programming scheme. However, interaction of these two components can lead to complexity much higher than merely the sum of their complexities. For example, Datalog (which is EXPTIME-complete) with linear arithmetic constraints (whose satisfiability problem is in NP for integers and in P for rational numbers and reals) is undecidable.

Theorem 10.1 *([Cox et al. 1990]) CLP over* $(\mathsf{N}, =, s, 0)$ *is r.e.-complete. The same holds for each of* Z*,* Q*,* R*, and* C *instead of* N*.*

The proof uses the fact that CLP over $(\mathsf{N}, =, s, 0, 1)$ allows one to define addition and multiplication in terms of successor. Thus, diophantine equations can be expressed in this fragment of CLP.

On the other hand, simpler constraints, namely constraints over ordered infinite domains (of some particular kind), do not increase the complexity of Datalog.

Theorem 10.2 *([Cox and McAloon 1993]) CLP over* $(\mathsf{Z}, =, <, 0, \pm 1, \pm 2, \ldots)$ *is* EXPTIME*-complete. The same holds for* Q *or* R *instead of* Z*.*

Decidable fragments of CLP over more complex structures are obtained by restrictions imposed on constraint logic programs. For example, we consider a *conservative CLP* in which rules satisfy the restriction: all variables occurring in the body occur in the head.

Theorem 10.3 *([Cox et al. 1990]) Conservative CLP is* EXPTIME*-complete over each of the following structures:*

$(\mathsf{Q}, =, \leq, <, +, -, k \cdot x, 0, 1, \ldots)$*, i.e. linear inequations over the rational numbers;*
$(\mathsf{R}, =, \leq, <, +, -, k \cdot x, 0, 1, \ldots)$*, i.e. linear inequations over the reals;*
$(\mathsf{R}, =, \leq, <, +, -, \times, /, x^k, 0, 1, \ldots)$*, i.e. polynomial inequations over the reals;*
$(\mathsf{C}, =, +, -, \times, /, x^k, 0, 1, \ldots)$*, i.e. polynomial equations over the complex numbers.*

The proof is based on the known results on the complexity of algorithms for the corresponding algebraic structures [Canny 1988; Renegar 1988; Grigoryev and Vorobjov 1988; Ierardi 1989]. If we allow nonground queries, EXPTIME-completeness has to be replaced by NEXPTIME-completeness.

A very general formalism for logic programming with constraints is the *constraint database model* introduced by [Kanellakis et al. 1990]. They define a *constraint database* as a quantifier-free formula over a given mathematical structure (e.g. the field of the real numbers). In the simplest case, this could be a finite relational database, but in general, a constraint database finitely represents an infinite number of tuples. They investigate the data complexity of first-order logic (FO) and datalog over constraint databases and prove that for the case of the real field, FO queries over constraint databases are in the parallel complexity class NC, while datalog queries are in P. For finite databases, [Benedikt and Libkin 1996] improved the NC upper bound to the parallel class TC^0, which contains the languages recognized by constant depth threshold circuits [Johnson 1990].

10.2 Expressiveness of constraints

There are various different settings in which expressiveness issues of logic programming formalisms with constraints have been studied. Expressiveness of first-order logic and of datalog with constraints is currently an intensive research area of Database Theory. Many important papers on this subject can be found in the proceedings of recent PODS, ICDT or LICS conferences.[3] A detailed and uniform treatment is beyond the scope of this paper. In this section, we limit ourselves to a brief description of a number of relevant references, most closely related to the setting of [Kanellakis et al. 1990].

A main research issue was the question whether properties such as *parity* that cannot be expressed in FO or stratified datalog (without order) could be expressed in the respective formalisms extended by constraints. This question has two different interpretations, depending on how we interpret the variables in a query. The *active interpretation* restricts the domain of possible values for a variable to those values that effectively appear in the database (i.e., to the *active domain*). The *natural interpretation* does not make this restriction and allows a variable to be interpreted by any value of the underlying domain (e.g. the reals). Note that these two interpretations coincide for classical relational calculus [Hull and Su 1994; Benedikt and Libkin 1997].

For the active interpretation of first-order constraint queries, the above question was solved independently by [Benedikt et al. 1996] and by [Otto and van den Bussche 1996]. It was shown that the generic queries expressible by FO with constraints are contained in those expressible by FO plus linear order. In particular, it follows that parity is not expressible in the constraint setting. The expressiveness problem for datalog with constraints was resolved in [Benedikt and Libkin 1997] by using Ramsey Theory. In analogy to the results for first-order logic, it was shown that datalog with constraints is not more expressive than datalog plus linear order.

For the natural interpretation, it was shown in [Grumbach and Su 1995] that every recursive query is definable by FO with polynomial constraints over the *natural numbers*. As shown in [Kanellakis and Goldin 1994; Grumbach et al. 1995], and [Benedikt et al. 1996], similar results do not hold for the *reals*. In particular, in [Benedikt et al. 1996] it was shown that over the field of reals, every generic query of first-order logic with constraints can be rewritten as an equivalent query that uses only the natural order "$<$". From this result, together with results in [Paredaens et al. 1998], it follows that every generic query of first-order logic with constraints under the natural interpretation can be expressed as an equivalent query under the active interpretation. Therefore, the same expressivity bound as for the active interpretation holds (see the previous paragraph); in particular, parity cannot be expressed.

In [Benedikt and Libkin 1996] and [Benedikt and Libkin 1997] it was shown that for polynomial constraints over the reals, the active and the natural semantics actually coincide. This result can be generalized – with some care – to fixpoint logic and datalog [Benedikt and Libkin 1997]. If function symbols are allowed to occur in the bodies of datalog rules, then every recursive query is expressible. However, if a hybrid approach is taken, where the fixpoint computation is restricted to the active domain of a database, while quantification refers to the natural domain,

[3]PODS=ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems; ICDT = International Conference on Database Theory; LICS = IEEE Symposium on Logic in Computer Science.

then a similar collapse as for FO also happens for fixpoint logic and datalog. These results for the reals generalize to a large class of other structures with quantifier elimination.

Acknowledgments

We would like to thank Jan van den Bussche, Michael Gelfond, Leonid Libkin, Domenico Saccá, Moshe Vardi, Helmut Veith and our colleagues for comments on a previous version of this paper and suggestions for improvement. In particular, we appreciate the many and extremely valuable comments of Leonid Libkin. We are also grateful to Wolfgang Faber for pointing out corrections on the bibliography.

This work has been funded by the Austrian Science Fund under Project N Z29-INF and by an EPSRC grant.

REFERENCES

Aanderaa, S. and Börger, E. 1979. The Horn complexity of boolean functions and Cook's problem. In B. Mayoh and F. Jensen Eds., *Proceedings 5th Scandinavian Logic Symposium*, pp. 231–256. Aalborg, Denmark: Aalborg University Press.

Abiteboul, S. 1989. Boundedness is undecidable for datalog programs with a single recursive rule. *Information Processing Letters 32*, 6, 281–287.

Abiteboul, S. and Beeri, C. 1995. The power of languages for the manipulation of complex values. *VLDB Journal 4*, 4, 727–794.

Abiteboul, S. and Grumbach, S. 1988. Col: A logic-based language for complex objects. In J. Schmidt, S. Ceri, and M. Missikoff Eds., *Advances in Database Technology - EDBT'88, Proceedings of the International Conference on Extending Database Technology, Venice, Italy, March 14-18, 1988*, Volume 303 of *Lecture Notes in Computer Science* (March 1988), pp. 271–293. Springer Verlag.

Abiteboul, S., Hull, R., and Vianu, V. 1995. *Foundation of Databases.* Addison-Wesley Publishing Co.

Abiteboul, S. and Kanellakis, P. C. 1989. Object identity as a query language primitive. In J. Clifford, B. G. Lindsay, and D. Maier Eds., *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989* (1989), pp. 159–173. ACM Press. Extended version appeared as [Abiteboul and Kanellakis 1998].

Abiteboul, S. and Kanellakis, P. C. 1998. Object identity as a query language primitive. *Journal of the Association for Computing Machinery 45*, 5 (Sept.), 798–842.

Abiteboul, S., Vardi, M. Y., and Vianu, V. 1997. Fixpoint logics, relational machines, and computational complexity. *Journal of the Association for Computing Machinery 44*, 1 (Jan.), 30–56.

Abiteboul, S. and Vianu, V. 1991a. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences 43*, 1 (Aug.), 62–124.

Abiteboul, S. and Vianu, V. 1991b. Generic computation and its complexity. In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing (STOC'91)* (New Orleans, Louisiana, May 1991), pp. 209–219. Extended version appeared as [Abiteboul and Vianu 1995].

Abiteboul, S. and Vianu, V. 1995. Computing with first-order logic. *Journal of Computer and System Sciences 50*, 2 (April), 309–335.

Afrati, F. N. 1997. Bounded arity datalog($\neq$) queries on graphs. *Journal of Computer and System Sciences 55*, 2 (Oct.), 210–228.

Afrati, F. N. and Cosmadakis, S. S. 1989. Expressiveness of restricted recursive queries. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing (STOC'89)* (Seattle, Washington, May 1989), pp. 113–126.

Afrati, F. N., Cosmadakis, S. S., and Yannakakis, M. 1995. On Datalog vs polynomial time. *Journal of Computer and System Sciences 51*, 2 (Oct.), 177–196.

Afrati, F. N. and Papadimitriou, C. H. 1993. The parallel complexity of simple logic programs. *Journal of the Association for Computing Machinery 40*, 4 (Sept.), 891–916.

Ajtai, M. and Gurevich, Y. 1994. Datalog vs first-order logic. *Journal of Computer and System Sciences 49*, 3 (Dec.), 562–588.

Andréka, H. and Németi, I. 1978. The generalized completeness of Horn predicate logic as a programming language. *Acta Cybernetica 4*, 3–10. (This is the published version of a 1975 report entitled "General Completeness of PROLOG".).

Apt, K. R. 1990. Logic programming. In J. van Leeuwen Ed., *Handbook of Theoretical Computer Science*, Volume B: Formal Models and Sematics, Chapter 10, pp. 493–574. Elsevier Science and The MIT Press.

Apt, K. R. and Blair, H. A. 1988. Arithmetic classification of perfect models of stratified programs. In R. A. Kowalski and K. A. Bowen Eds., *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (ICLP/SLP 1988)* (Aug. 1988), pp. 765–779. The MIT Press.

Apt, K. R., Blair, H. A., and Walker, A. 1988. Towards a theory of declarative knowledge. In J. Minker Ed., *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Washington, DC, USA: Morgan Kaufmann.

Apt, K. R. and Bol, R. N. 1994. Logic programming and negation: A survey. *Journal of Logic Programming 19/20*, 9–71.

Apt, K. R. and van Emden, M. H. 1982. Contributions to the theory of logic programming. *Journal of the Association for Computing Machinery 29*, 3, 841–862.

Baader, F. and Schulz, K. U. 1992. Unification in the union of disjoint equational theories: Combining decision procedures. In D. Kapur Ed., *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, Volume 607 of *Lecture Notes in Artificial Intelligence* (June 1992), pp. 50–65.

Baader, F. and Schulz, K. U. 1996. Unification in the union of disjoint equational theories: Combining decision procedures. *Journal of Symbolic Computation 21*, 2 (Feb.), 211–243.

Baader, F. and Siekmann, J. H. 1994. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson Eds., *Handbook of Logic in Artificial Intelligence and Logic Programming*, pp. 41–125. Oxford University Press.

Bachmair, L., Chen, T., Ramakrishnan, C., and Ramakrishnan, I. 1996. Subsumption algorithms based on search trees. In H. Kirchner Ed., *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Linkping, Sweden, April, 22-24, 1996, Proceedings*, Volume 1059 of *Lecture Notes in Computer Science* (April 1996), pp. 135–148. Springer Verlag.

Balcázar, J. L., Díaz, J., and Gabarró, J. 1990. *Structural Complexity II*, Volume 22 of *EATCS: Monographs in Theoretical Computer Science*. Springer Verlag.

Balcázar, J. L., Lozano, A., and Torán, J. 1992. The complexity of algorithmic problems on succinct instances. In R. Baeta-Yates and U. Manber Eds., *Computer Science*, pp. 351–377. New York: Plenum Press.

Baral, C. and Gelfond, M. 1994. Logic programming and knowledge representation. *Journal of Logic Programming 19/20*, 73–148.

Baral, C. and Subrahmanian, V. 1993. Dualities between alternative semantics for logic programming and nonmonotonic reasoning. *Journal of Automated Reasoning 10*, 3 (June), 399–420.

Ben-Eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence 12*, I–II, 53–87.

Benanav, D., Kapur, D., and Narendran, P. 1987. Complexity of matching problems. *Journal of Symbolic Computation 3*, 1/2 (Feb./April), 203–216.

Benedikt, M., Dong, G., Libkin, L., and Wong, L. 1996. Relational expressive power of constraint query languages. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, 1996, Montreal, Canada (PODS'96)* (June 1996), pp. 5–16. ACM Press.

Benedikt, M. and Libkin, L. 1996. On the structure of queries in constraint query languages. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, 27-30 July 1996 (LICS'96)* (July 1996), pp. 25–34. IEEE Computer Society Press.

Benedikt, M. and Libkin, L. 1997. Languages for relational databases over interpreted structures. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona (PODS'97)* (May 1997), pp. 87–98. ACM Press.

Berman, K. A., Schlipf, J. S., and Franco, J. V. 1995. Computing well-founded semantics faster. In V. W. Marek, A. Nerode, and M. Truszczyński Eds., *Logic Programming and Nonmonotonic Reasoning, Third International Conference, LPNMR'95, Lexington, KY, USA, June 26-28, 1995, Proceedings*, Volume 982 of *Lecture Notes in Computer Science* (June 1995), pp. 113–126. Springer Verlag.

Berman, L. 1977. Precise bounds for Presburger arithmetic and the reals with addition: Preliminary report. In *18th Annual Symposium on Foundations of Computer Science (FOCS'77)* (Providence, Rhode Island, Oct./Nov. 1977), pp. 95–99. IEEE.

Berman, L. 1980. The complexity of logical theories. *Theoretical Computer Science 11*, 71–77.

Bidoit, N. 1991. Negation in rule-based database languages: a survey. *Theoretical Computer Science 78*, 1 (Jan.), 3–83.

Bidoit, N. and Froidevaux, C. 1991. Negation by default and unstratifiable logic programs. *Theoretical Computer Science 78*, 1 (Jan.), 85–112.

Blair, H. A. 1982. The recursion-theoretical complexity of the semantics of predicate logic as a programming language. *Information and Control 54*, 1/2 (Jan.), 25–47.

Bonatti, P. A. and Eiter, T. 1996. Querying disjunctive databases through nonmonotonic logics. *Theoretical Computer Science 160*, 1&2 (June), 321–363.

Bonner, A. J. 1990. Hypothetical datalog: Complexity and expressibility. *Theoretical Computer Science 76*, 1, 3–51.

Bonner, A. J. 1997. Intuitionistic deductive databases and the polynomial time hierarchy. *Journal of Logic Programming 33*, 1 (Oct.), 1–47.

Börger, E. 1971. *Reduktionstypen in Krom- und Hornformeln.* Ph. D. thesis, Fachbereich Mathematik der Universität Münster.

Börger, E. 1974. Beitrag zur Reduktion des Entscheidungsproblems auf Klassen von Hornformeln mit kurzen Alternationen. *Archiv für mathematische Logik und Grudlagenforschung 16*, 1–2, 67–84.

Börger, E. 1984. Decision problems in predicate logic. In G. Lolli, G. Longo, and A. Marcja Eds., *Logic Colloquium '82*, Volume 112 of *Studies in Logic and the Foundations of Mathematics* (1984), pp. 263–301. North Holland.

Börger, E., Grädel, E., and Gurevich, Y. 1997. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer Verlag, Berlin.

Brass, S. and Dix, J. 1995. Disjunctive semantics based upon partial and bottom-up evaluation. In L. Sterling Ed., *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, June 13-16, 1995, Tokyo, Japan (ICLP'95)* (June 1995), pp. 199–213. MIT Press.

Brewka, G. and Eiter, T. 1998. Preferred answer sets for extended logic programs. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro Eds., *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, June 2-5, 1998* (June 1998), pp. 86–97. Morgan Kaufmann. Full paper: [Brewka and Eiter 1999].

Brewka, G. and Eiter, T. 1999. Preferred answer sets for extended logic programs. *Artificial Intelligence 109*, 1–2 (April), 297–356.

Buccafurri, F., Greco, S., and Saccà, D. 1997. The expressive power of unique total stable model semantics. In P. Degano, R. Corrieri, and A. Marchetti-Spaccamela Eds., *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997*, Volume 1256 of *Lecture Notes in Computer Science* (July 1997), pp. 849–859. Springer Verlag.

Buccafurri, F., Leone, N., and Rullo, P. 1997. Strong and weak constraints in disjunctive datalog. In J. Dix, U. Furbach, and A. Nerode Eds., *Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning, LPNMR'97, Dagstuhl, Germany, July 28-31, 1997*, Number 1265 in Lecture Notes in Computer Science (July 1997), pp. 2–17. Springer Verlag. Full paper: [Buccafurri et al. 2000].

Buccafurri, F., Leone, N., and Rullo, P. 1998. Disjunctive ordered logic: Semantics and expressiveness. In A. G. Cohn, L. Schubert, and S. C. Shapiro Eds., *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, June 2-5, 1998* (June 1998), pp. 418–431. Morgan Kaufmann. Full paper: [Buccafurri et al. 1999].

Buccafurri, F., Leone, N., and Rullo, P. 1999. Semantics and expressiveness of disjunctive ordered logic. *Annals of Mathematics and Artificial Intelligence 25*, 3–4, 311–337.

Buccafurri, F., Leone, N., and Rullo, P. 2000. Enhancing disjunctive datalog by constraints. *IEEE Transactions on Knowledge and Data Engineering 12*, 5, 845–860.

Büchi, J. 1962. Turing machines and the Entscheidungsproblem. *Mathematische Annalen 148*, 201–213.

Buneman, P., Naqvi, S. A., Tannen, V., and Wong, L. 1995. Principles of programming with complex objects and collection types. *Theoretical Computer Science 149*, 1 (Sept.), 3–48.

Cadoli, M. and Palopoli, L. 1998. Circumscribing DATALOG: Expressive power and complexity. *Theoretical Computer Science 193*, 1–2 (Feb.), 215–244.

Cadoli, M. and Schaerf, M. 1993. A survey of complexity results for nonmonotonic logics. *Journal of Logic Programming 17*, 2/3&4 (Nov.), 127–160.

Canny, J. 1988. Some algebraic and geometric computations in PSPACE. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC'88)* (Chicago, Illinois, 2–4 May 1988), pp. 460–467.

Ceri, S., Gottlob, G., and Tanca, L. 1990. *Logic Programming and Databases*. Surveys in Computer Science. Springer Verlag.

Chan, E. P. F. 1993. A possible worlds semantics for disjunctive databases. *IEEE Transactions on Knowledge and Data Engineering 5*, 2 (April), 282–292.

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

Chandra, A. K. and Harel, D. 1985. Horn clause queries and generalizations. *Journal of Logic Programming 2*, 1 (April), 1–15.

Chandra, A. K., Kozen, D., and Stockmeyer, L. J. 1981. Alternation. *Journal of the Association for Computing Machinery 28*, 1 (Jan.), 114–133.

Chandra, A. K., Lewis, H. R., and Makowsky, J. A. 1981. Embedded implicational dependencies and their inference problem. In *Conference Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computation (STOC'81)* (Milwaukee, Wisconsin, 11–13 May 1981), pp. 342–354.

Chandra, A. K. and Merlin, P. M. 1977. Optimal implementation of conjunctive queries in relational data bases. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing (STOC'77)* (Boulder, Colorado, 2–4 May 1977), pp. 77–90.

Chaudhuri, S. and Vardi, M. Y. 1994. On the complexity of equivalence between recursive and nonrecursive datalog programs. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota (PODS'94)* (May 1994), pp. 107–116. ACM Press.

Chaudhuri, S. and Vardi, M. Y. 1997. On the equivalence of recursive and nonrecursive datalog programs. *Journal of Computer and System Sciences 54*, 1 (Feb.), 61–78.

Cholak, P. and Blair, H. A. 1994. The complexity of local stratification. *Fundamenta Informaticae 21*, 4 (Oct.), 333–344.

Chomicki, J. and Subrahmanian, V. 1990. Generalized closed world assumption is Π_2^0-complete. *Information Processing Letters 34*, 6 (May), 289–291.

Colby, L. S. and Libkin, L. 1997. Tractable iteration mechanisms for bag languages. In F. N. Afrati and P. Kolaitis Eds., *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, Volume 1186 of *Lecture Notes in Computer Science* (Jan. 1997), pp. 461–475. Springer Verlag.

Colmerauer, A., Kanoui, H., Roussel, P., and Passero, R. 1973. Un système de communication homme-machine en Francais. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille II.

Corciulo, L., Giannotti, F., and Pedreschi, D. 1993. Datalog with non-deterministic choice computes NDB-PTIME. In S. Ceri, K. Tanaka, and S. Tsur Eds., *Deductive and Object-Oriented Databases, Third International Conference, DOOD'93, Phoenix, Arizona, USA, December 6-8, 1993, Proceedings*, Volume 760 of *Lecture Notes in Computer Science* (Dec. 1993), pp. 49–66. Springer Verlag.

Cosmadakis, S. S. 1989. On the first-order expressibility of recursive queries. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania (PODS'89)* (March 1989), pp. 311–323. ACM Press.

Cosmadakis, S. S. and Kanellakis, P. C. 1986. Parallel evaluation of recursive rule queries. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts (PODS'86)* (March 1986), pp. 280–293. ACM Press.

Cosmadakis, S. S. and Kuper, G. M. 1994. Expressiveness of first-order constraint languages. Technical Report ECRC-94-13, European Computer Industry Research Center.

Cox, J. and McAloon, K. 1993. Decision procedures for constraint-based extensions of datalog. In F. Benhamou and A. Colmerauer Eds., *Constraint Logic Programming, Selected Research*, pp. 17–32. The MIT Press.

Cox, J., McAloon, K., and Tretkoff, C. 1990. Computational complexity and constraint logic programming languages. In S. K. Debray and M. V. Hermenegildo Eds., *Logic Programming, Proceedings of the 1990 North American Conference, Austin,Texas, October 29 - November 1, 1990 (SLP/NACLP'90)* (Oct./Dec. 1990), pp. 401–415. The MIT Press.

Dahlhaus, E. 1987. Skolem normal forms concerning the least fixpoint. In E. Börger Ed., *Computation Theory and Logic, In Memory of Dieter Rödding*, Volume 270 of *Lecture Notes in Computer Science* (1987), pp. 101–106. Springer Verlag.

Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. 1997. Complexity and expressive power of logic programming. In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity, June 24-27, 1997, Ulm, Germany, (CCC'97)* (June 1997), pp. 82–101. IEEE Computer Society Press.

Dantsin, E. and Voronkov, A. 1997. Complexity of query answering in logic databases with complex values. In S. I. Adian and A. Nerode Eds., *Logical Foundations of Computer Science, 4th International Symposium, LFCS'97 Yaroslavl, Russia, July 6-12, 1997, Proceedings*, Volume 1234 of *Lecture Notes in Computer Science* (July 1997), pp. 56–66. Springer Verlag.

Dantsin, E. and Voronkov, A. 1999. A nondeterministic polynomial-time unification algorithm for bags, sets and trees. In W. Thomas Ed., *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, Volume 1578 of *Lecture Notes in Computer Science* (March 1999), pp. 180–196. Springer Verlag.

DANTSIN, E. AND VORONKOV, A. 2000. Expressive power and data complexity of query languages for trees and lists. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA (PODS 2000)* (May 2000), pp. 157–165. ACM Press.

DAVIS, M. Ed. 1965. *The Undecidable.* Raven Press, New York.

DEGTYAREV, A. AND VORONKOV, A. 1996. A note on semantics of logics programs with equality based on complete sets of E-unifiers. *Journal of Logic Programming 28*, 3 (Sept.), 207–216.

DEVIENNE, P. 1990. Weighted graphs: A tool for studying the halting problem and time complexity in term rewrite systems and logic programming. *Theoretical Computer Science 75*, 1&2, 157–215.

DEVIENNE, P., LEBÈGUE, P., PARRAIN, A., ROUTIER, J.-C., AND WÜRTZ, J. 1996. Smallest Horn clause programs. *Journal of Logic Programming 27*, 3 (June), 227–267.

DEVIENNE, P., LEBÈGUE, P., AND ROUTIER, J.-C. 1993. Halting problem of one binary Horn clause is undecidable. In P. ENJALBERT, A. FINKEL, AND K. W. WAGNER Eds., *STACS 93, 10th Annual Symposium on Theoretical Aspects of Computer Science, Wrzburg, Germany, February 25-27, 1993, Proceedings*, Number 665 in Lecture Notes in Computer Science (Feb. 1993), pp. 48–57. Springer Verlag.

DIKOVSKY, A. J. 1993. On the computational complexity of prolog programs. *Theoretical Computer Science 119*, 1, 63–102.

DIX, J., GOTTLOB, G., AND MAREK, V. W. 1996. Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae 28*, 1–2 (Nov.), 87–100.

DONG, G., LIBKIN, L., AND WONG, L. 1997. Local properties of query languages. In F. N. AFRATI AND P. KOLAITIS Eds., *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, Volume 1186 of *Lecture Notes in Computer Science* (1997), pp. 140–154. Springer Verlag.

DOWLING, W. F. AND GALLIER, J. H. 1984. Linear-time algorithms for testing the satisfiability of propositional Horn theories. *Journal of Logic Programming 1*, 3 (Oct.), 267–284.

DUDAKOV, S. 1999. On the complexity of perfect models of logic programs. *Fundamenta Informaticae 39*, 3 (July–Aug.), 249–258.

DWORK, C., KANELLAKIS, P. C., AND MITCHELL, J. C. 1984. On the sequential nature of unification. *Journal of Logic Programming 1*, 1 (June), 35–50.

DWORK, C., KANELLAKIS, P. C., AND STOCKMEYER, L. J. 1988. Parallel algorithms for term matching. *SIAM Journal on Computing 17*, 4 (Aug.), 711–731.

EBBINGHAUS, H.-D. AND FLUM, J. 1995. *Finite Model Theory.* Perspectives in Mathematical Logic. Springer Verlag.

EITER, T. AND GOTTLOB, G. 1992. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artificial Intelligence 57*, 2–3, 227–270.

EITER, T. AND GOTTLOB, G. 1993. Propositional circumscription and extended closed world reasoning are $\mathbf{\Pi}_2^p$-complete. *Theoretical Computer Science 114*, 2, 231–245. Addendum 118:315.

EITER, T. AND GOTTLOB, G. 1995a. The complexity of logic-based abduction. *Journal of the Association for Computing Machinery 42*, 1 (Jan.), 3–42.

EITER, T. AND GOTTLOB, G. 1995b. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence 15*, 3–4, 289–323.

EITER, T. AND GOTTLOB, G. 1997. Expressiveness of stable model semantics for disjunctive logic programs with functions. *Journal of Logic Programming 33*, 2 (Nov.), 167–178.

EITER, T., GOTTLOB, G., AND LEONE, N. 1997a. Abduction from logic programs: Semantics and complexity. *Theoretical Computer Science 189*, 1–2 (Dec.), 129–177.

EITER, T., GOTTLOB, G., AND LEONE, N. 1997b. On the indiscernibility of individuals in logic programming. *Journal of Logic and Computation 7*, 6 (Dec.), 805–824.

EITER, T., GOTTLOB, G., AND MANNILA, H. 1994. Adding disjunction to Datalog. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of*

Database Systems, May 24-26, 1994, Minneapolis, Minnesota (PODS'94) (May 1994), pp. 267–278. ACM Press.

EITER, T., GOTTLOB, G., AND MANNILA, H. 1997. Disjunctive Datalog. *ACM Transactions on Database Systems 22*, 3 (Sept.), 364–418.

EITER, T., GOTTLOB, G., AND VEITH, H. 1997. Modular logic programming and generalized quantifiers. In J. DIX, U. FURBACH, AND A. NERODE Eds., *Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning, LPNMR'97, Dagstuhl, Germany, July 28-31, 1997*, Volume 1265 of *Lecture Notes in Computer Science* (July 1997), pp. 290–309. Springer Verlag.

EITER, T., GOTTLOB, G., AND VEITH, H. 2000. Generalized quantifiers in logic programs. In J. VÄÄNÄNEN Ed., *Generalized Quantifiers and Computation, 9th European Summer School in Logic, Language, and Information, ESSLLI'97 Workshop, Aix-en-Provence, France, August 11-22, 1997, Revised Lectures*, Number 1754 in Lecture Notes in Computer Science (2000), pp. 72–98. Springer.

EITER, T., LEONE, N., AND SACCÀ, D. 1998. Expressive power and complexity of partial models for disjunctive deductive databases. *Theoretical Computer Science 206*, 1–2 (Oct.), 181–218.

EITER, T., LU, J. J., AND SUBRAHMANIAN, V. 1998. A first-order representation of stable models. *The European Journal on Artificial Intelligence (AI Communications) 11*, 1, 53–73.

FAGIN, R. 1974. Generalized first-order spectra and polynomial-time recognizable sets. In R. M. KARP Ed., *Complexity of Computation* (1974), pp. 43–74. American Mathematical Society.

FALASCHI, M., LEVI, G., PALAMIDESSI, C., AND MARTELLI, M. 1989. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science 69*, 3, 289–318.

FITTING, M. 1987a. *Computability Theory, Semantics, and Logic Programming.* Oxford University Press.

FITTING, M. 1987b. Enumeration operators and modular logic programming. *Journal of Logic Programming 4*, 1 (March), 11–21.

FLUM, J., KUBIERSCHKY, M., AND LUDÄSCHER, B. 1997. Total and partial well-founded datalog coincide. In F. N. AFRATI AND P. KOLAITIS Eds., *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, Volume 1186 of *Lecture Notes in Computer Science* (Jan. 1997), pp. 113–124. Springer Verlag.

GAIFMAN, H., MAIRSON, H. G., SAGIV, Y., AND VARDI, M. Y. 1987. Undecidable optimization problems for database logic programs. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987* (June 1987), pp. 106–115. IEEE Computer Society Press.

GALLIER, J. H. AND RAATZ, S. 1986. SLD-resolution methods for Horn clauses with equality based on E-unification. In *Proceedings of the 1986 Symposium on Logic Programming, Salt Lake City, Utah, September 22-25, 1986 (SLP'86)* (Sept. 1986), pp. 168–179.

GALLIER, J. H. AND RAATZ, S. 1989. Extending SLD-resolution to equational Horn clauses using E-unification. *Journal of Logic Programming 6*, 1&2 (Jan./March), 3–43.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability.* Freeman, San Francisco.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In R. A. KOWALSKI AND K. A. BOWEN Eds., *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (ICLP/SLP 1988)* (Aug. 1988), pp. 1070–1080. The MIT Press.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 3–4, 365–386.

GIANNOTTI, F., GRECO, S., SACCÀ, D., AND ZANIOLO, C. 1997. Programming with non-determinism in deductive databases. *Annals of Mathematics and Artificial Intelligence 19*, 1–2, 97–125.

GIANNOTTI, F. AND PEDRESCHI, D. 1998. Datalog with non-deterministic choice computes NDB-PTIME. *Journal of Logic Programming 35*, 1 (April), 79–110.

GIVAN, R. AND MCALLESTER, D. 2000. Polynomial-time computation via local inference relations. submitted.

GOTTLOB, G. 1987. Subsumption and implication. *Information Processing Letters 24*, 2 (Jan.), 109–111.

GOTTLOB, G. 1992. Complexity results for nonmonotonic logics. *Journal of Logic and Computation 2*, 3 (June), 397–425.

GOTTLOB, G. AND FERMÜLLER, C. G. 1993. Removing redundancy from a clause. *Artificial Intelligence 61*, 2, 263–289.

GOTTLOB, G., GRÄDEL, E., AND VEITH, H. 2000a. Datalog LITE: A deductive query language with linear time model checking. Manuscript, submitted for publication.

GOTTLOB, G., GRÄDEL, E., AND VEITH, H. 2000b. Linear time Datalog and branching time logic. In J. MINKER Ed., *Logic-Based Artificial Intelligence*. Kluwer.

GOTTLOB, G. AND LEITSCH, A. 1985a. Fast subsumption algorithms. In B. F. CAVINESS Ed., *Proceedings of the European Conference on Computer Algebra (EUROCAL '85): volume 2: research contributions*, Volume 204 of *Lecture Notes in Computer Science* (Linz, Austria, 1985), pp. 64–77. Springer Verlag.

GOTTLOB, G. AND LEITSCH, A. 1985b. On the efficiency of subsumption algorithms. *Journal of the Association for Computing Machinery 32*, 2 (April), 280–295.

GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 1997. On the complexity of some inductive logic programming problems. In N. LAVRAČ AND S. DŽEROSKI Eds., *Inductive Logic Programming, 7th International Workshop, ILP-97, Prague, Czech Republic, September 17-20, 1997, Proceedings*, Volume 1297 of *LNAI* (Berlin, Sept. 1997), pp. 17–32. Springer. Full version: [Gottlob et al. 1999b].

GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 1998. The complexity of acyclic conjunctive queries. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA* (Nov. 1998), pp. 706–715. IEEE Computer Society Press.

GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 1999a. Hypertree decompositions and tractable queries. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania* (May/June 1999), pp. 21–32. ACM Press.

GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 1999b. On the complexity of some inductive logic programming problems. *New Generation Computing 17*, 1, 53–75.

GOTTLOB, G., LEONE, N., AND VEITH, H. 1995. Second order logic and the weak exponential hierarchies. In J. WIEDERMANN AND P. HÁJEK Eds., *Mathematical Foundations of Computer Science 1995, 20th International Symposium, MFCS'95, Prague, Czech Republic, August 28 - September 1, 1995, Proceedings*, Volume 969 of *Lecture Notes in Computer Science* (Aug./Sept. 1995), pp. 66–81. Full paper: [Gottlob et al. 1999].

GOTTLOB, G., LEONE, N., AND VEITH, H. 1999. Succinctness as a source of complexity in logical formalisms. *Annals of Pure and Applied Logic 97*, 1–3 (March), 231–260.

GOTTLOB, G., MARCUS, S., NERODE, A., SALZER, G., AND SUBRAHMANIAN, V. 1996. A non-ground realization of the stable and well-founded semantics. *Theoretical Computer Science 166*, 1&2 (Oct.), 221–262.

GOTTLOB, G. AND PAPADIMITRIOU, C. H. 1999. On the complexity of single-rule datalog queries. In H. GANZINGER, D. A. MCALLESTER, AND A. VORONKOV Eds., *Logic Programming and Automated Reasoning, 6th International Conference, LPAR'99, Tbilisi, Georgia, September 6-10, 1999, Proceedings*, Volume 1705 of *Lecture Notes in Computer Science* (1999), pp. 201–222. Springer Verlag. Full paper submitted for publication.

GOTTLOB, G. AND PICHLER, R. 1999. Working with arms: Complexity results on atomic representations of herbrand models. In *14th Annual IEEE Symposium on Logic in Computer Science, 2-5 July, 1999, Trento, Italy (LICS'99)* (July 1999), pp. 306–315. IEEE Computer Society Press. Full paper in *Information and Computation*, to appear.

GRÄDEL, E. 1992. Capturing complexity classes with fragments of second order logic. *Theoretical Computer Science 101*, 1, 35–57.

GRECO, S., PALOPOLI, L., AND SPADAFORA, E. 1995. Extending datalog with arrays. *Data and Knowledge Engineering 17*, 1 (Oct.), 31–57.

GRECO, S. AND SACCÀ, D. 1996. The expressive power of "possible-is-certain" semantics (extended abstract). In J. JAFFAR AND R. H. C. YAP Eds., *Concurrency and Parallelism, Programming, Networking, and Security: Second Asian Computing Science Conference, ASIAN '96, Singapore, December 2-5, 1996, Proceedings*, Volume 1179 of *Lecture Notes in Computer Science* (1996), pp. 33–42. Springer Verlag.

GRECO, S. AND SACCÀ, D. 1997. Deterministic semantics for datalog¬: Complexity and expressive power. In F. BRY, R. RAMAKRISHNAN, AND K. RAMAMOHANAROA Eds., *Deductive and Object-Oriented Databases, 5th International Conference, DOOD'97, Montreux, Switzerland, December 8-12, 1997, Proceedings*, Volume 1341 of *Lecture Notes in Computer Science* (1997), pp. 337–350. Springer Verlag.

GREEN, C. 1969. *The Application of Theorem Proving to Question-Answering Systems.* Ph. D. thesis, Computer Science Department, Stanford University.

GRIGORYEV, D. AND VOROBJOV, N. 1988. Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation 5*, 1/2 (Feb./April), 37–64.

GRUMBACH, S. AND LACROIX, Z. 1997. On non-determinism in machines and languages. *Annals of Mathematics and Artificial Intelligence 19*, 1–2, 169–213.

GRUMBACH, S. AND SU, J. 1995. First-order definability over constraint databases. In U. MONTANARI AND F. ROSSI Eds., *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings*, Volume 976 of *Lecture Notes in Computer Science*, pp. 121–136. Springer Verlag.

GRUMBACH, S., SU, J., AND TOLLU, C. 1995. Linear constraint query languages: Expressive power and complexity. In D. LEIVANT Ed., *Logic and Computational Complexity. Selected Papers. Logic and Computational Complexity, International Workshop LCC '94, Indianapolis, Indiana, USA, 13-16 October 1994*, Volume 960 of *Lecture Notes in Computer Science*, pp. 426–446. Springer Verlag.

GRUMBACH, S. AND VIANU, V. 1995. Tractable query languages for complex object databases. *Journal of Computer and System Sciences 51*, 2 (Oct.), 149–167.

GUREVICH, Y. 1988. Logic and the challenge of computer science. In E. BÖRGER Ed., *Current Trends in Theoretical Computer Science*, Chapter 1, pp. 1–57. Computer Science Press.

GUREVICH, Y. AND SHELAH, S. 1986. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic 32*, 265–280.

GYSSENS, M., VAN GUCHT, D., AND SUCIU, D. 1995. The restricted and bounded fixpoint closures of the nested relational algebra are equivalent. In P. ATZENI AND V. TANNEN Eds., *Database Programming Languages (DBPL-5), Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995*, Electronic Workshops in Computing (1995). Springer Verlag.

HANSCHKE, P. AND WÜRTZ, J. 1993. Satisfiability of the smallest binary program. *Information Processing Letters 45*, 5 (April), 237–241.

HANUS, M. 1994. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming 19/20*, 583–628.

HERBRAND, J. 1972. *Logical Writings.* Harvard University Press.

HILLEBRAND, G. G., KANELLAKIS, P. C., MAIRSON, H. G., AND VARDI, M. Y. 1995. Undecidable boundedness problems for datalog programs. *Journal of Logic Programming 25*, 2 (Nov.), 163–190.

HÖLLDOBLER, S. 1989. *Foundations of Equational Logic Programming*, Volume 353 of *Lecture Notes in Artificial Intelligence.* Springer Verlag.

HULL, R. AND SU, J. 1994. Domain independence and the relational calculus. *Acta Informatica 31*, 6, 513–524.

IERARDI, D. 1989. Quantifier elimination in the theory of an algebraically-closed field. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing (STOC'89)* (Seattle, Washington, May 1989), pp. 138–147.

IMMERMAN, N. 1986. Relational queries computable in polynomial time. *Information and Control 68*, 1–3 (Jan./Feb./March), 86–104.

IMMERMAN, N. 1987. Languages that capture complexity classes. *SIAM Journal on Computing 16*, 4 (Aug.), 760–778.

IMMERMAN, N. 1999. *Descriptive Complexity*. Graduate Texts in Computer Science. Springer Verlag, New York.

INOUE, K. AND SAKAMA, C. 1993. Transforming abductive logic programs to disjunctive programs. In D. S. WARREN Ed., *Logic Programming, Proceedings of the Tenth International Conference on Logic Programming, June 21-25, 1993, Budapest, Hungary (ICLP'93)* (June 1993), pp. 335–353. The MIT Press.

INOUE, K. AND SAKAMA, C. 1998. Negation as failure in the head. *Journal of Logic Programming 35*, 1 (April), 39–78.

IOANNIDIS, Y. E. 1986. A time bound on the materialization of some recursively defined views. *Algorithmica 1*, 3, 361–385.

ITAI, A. AND MAKOWSKY, J. A. 1987. Unification as a complexity measure for logic programming. *Journal of Logic Programming 4*, 2 (June), 105–117.

JAFFAR, J. AND MAHER, M. J. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

JOHNSON, D. S. 1990. A catalog of complexity classes. In J. VAN LEEUWEN Ed., *Handbook of Theoretical Computer Science*, Volume A: Algorithms and Complexity, Chapter 2, pp. 67–161. Elsevier Science and The MIT Press.

JONES, N. D. AND LAASER, W. T. 1976. Complete problems for deterministic polynomial time. *Theoretical Computer Science 3*, 1, 105–117.

KANELLAKIS, P. C. 1988. Logic programming and parallel complexity. In J. MINKER Ed., *Foundations of Deductive Databases and Logic Programming*, pp. 547–586. Washington, DC, USA: Morgan Kaufmann.

KANELLAKIS, P. C. 1990. Elements of relational database theory. In J. VAN LEEUWEN Ed., *Handbook of Theoretical Computer Science*, Volume B: Formal Models and Sematics, Chapter 17, pp. 1073–1156. Elsevier Science and The MIT Press.

KANELLAKIS, P. C. AND GOLDIN, D. Q. 1994. Constraint programming and database query languages. In M. HAGIYA AND J. C. MITCHELL Eds., *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, Volume 789 of *Lecture Notes in Computer Science* (April 1994), pp. 96–120. Springer Verlag.

KANELLAKIS, P. C., KUPER, G. M., AND REVESZ, P. Z. 1990. Constraint query languages. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee (PODS'90)* (April 1990), pp. 299–313. ACM Press.

KANELLAKIS, P. C., KUPER, G. M., AND REVESZ, P. Z. 1995. Constraint query languages. *Journal of Computer and System Sciences 51*, 1 (Aug.), 26–52.

KAPUR, D. AND NARENDRAN, P. 1986. NP-completeness of the set unification and matching problems. In J. H. SIEKMANN Ed., *8th International Conference on Automated Deduction, Oxford, England, July 27 - August 1, 1986, Proceedings (CADE-8)*, Volume 230 of *Lecture Notes in Computer Science* (1986), pp. 489–495. Springer Verlag.

KAPUR, D. AND NARENDRAN, P. 1992. Complexity of unification problems with associative-commutative operators. *Journal of Automated Reasoning 9*, 2 (Oct.), 261–288.

KIETZ, J.-U. AND DZEROSKI, S. 1994. Inductive logic programming and learnability. *SIGART Bulletin 5*, 1 (Jan.), 22–32.

KIFER, M., LAUSEN, G., AND WU, J. 1995. Logical foundations of object-oriented and frame-based languages. *Journal of the Association for Computing Machinery 42*, 4 (July), 741–843.

KIFER, M. AND WU, J. 1993. A logic for programming with complex objects. *Journal of Computer and System Sciences 47*, 1 (Aug.), 77–120.

KOLAITIS, P. G. 1991. The expressive power of stratified logic programs. *Information and Computation 90*, 1 (Jan.), 50–66.

KOLAITIS, P. G. AND PAPADIMITRIOU, C. H. 1991. Why not negation by fixpoint? *Journal of Computer and System Sciences 43*, 1 (Aug.), 125–144.

KOLAITIS, P. G. AND VARDI, M. Y. 1992. Fixpoint logic vs. infinitary logic in finite-model theory. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science (LICS'92)* (Santa Cruz, California, 22–25 June 1992), pp. 46–57. IEEE Computer Society Press.

KOLAITIS, P. G. AND VARDI, M. Y. 1995. On the expressive power of datalog: Tools and a case study. *Journal of Computer and System Sciences 51*, 1 (Aug.), 110–134.

KOŚCIELSKI, A. AND PACHOLSKI, L. 1996. Complexity of Makanin's algorithm. *Journal of the Association for Computing Machinery 43*, 4 (July), 670–684.

KOWALSKI, R. A. 1974. Predicate logic as programming language. In J. L. ROSENFELD Ed., *Information Processing 74, Proceedings of IFIP Congress 74, Stockholm, Sweden, August 5-10, 1974 (IFIP'74)* (Amsterdam, Aug. 1974), pp. 569–574. North Holland.

KOWALSKI, R. A. AND KUEHNER, D. 1971. Linear resolution with selection function. *Artificial Intelligence 2*, 3/4, 227–260.

LAKSHMANAN, V. AND MENDELZON, A. O. 1989. Inductive pebble games and the expressive power of datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania (PODS'89)* (March 1989), pp. 301–310. ACM Press.

LEITSCH, A. 1990. Deciding Horn classes by hyperresolution. In E. BÖRGER, H. KLEINE BÜNING, AND M. M. RICHTER Eds., *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings*, Volume 440 of *Lecture Notes in Computer Science* (1990), pp. 225–241. Springer Verlag.

LEITSCH, A. 1997. *The Resolution Calculus*. Springer Verlag.

LEITSCH, A. AND GOTTLOB, G. 1990. Deciding Horn clause implication problems by ordered semantic resolution. In F. GARDIN AND G. MAURI Eds., *Computational Intelligence II, Proceedings of the International Symposium, Milan, Italy, 25-27 Sept. 1989* (Amsterdam, The Netherlands, 1990), pp. 19–26. North Holland.

LEIVANT, D. 1989. Descriptive characterizations of computational complexity. *Journal of Computer and System Sciences 39*, 1 (Aug.), 51–83.

LEONE, N., PALOPOLI, L., AND SACCÀ, D. 1999. On the complexity of search queries. In T. POLLE, T. RIPKE, AND K.-D. SCHEWE Eds., *Fundamentals of Information Systems, Papers from the Seventh Workshop on Foundations of Models and Languages for Data and Objects, Ostfriesland, Germany, October 5-9, 1998 (FMLDO'98)* (1999), pp. 113–127. Kluwer.

LEONE, N., RULLO, P., AND SCARCELLO, F. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation 135*, 2 (June), 69–112.

LEWIS, H. R. 1979. *Unsolvable Classes of Quantificational Formulas*. Addison-Wesley, Reading, Mass.

LEWIS, H. R. AND STATMAN, R. 1982. Unifiability is complete for co-NLogSpace. *Information Processing Letters 15*, 5 (Dec.), 220–222.

LIBKIN, L. 1997. On the forms of locality over finite models. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997 (LICS'97)* (June/July 1997), pp. 204–215. IEEE Computer Society Press.

LIBKIN, L., MACHLIN, R., AND WONG, L. 1996. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In H. V. JAGADISH AND I. S. MUMICK Eds., *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996* (June 1996), pp. 228–239. ACM Press.

LIBKIN, L. AND WONG, L. 1994. New techniques for studying set languages, bag languages and aggregate functions. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota (PODS'94)* (May 1994), pp. 155–166. ACM Press.

LIBKIN, L. AND WONG, L. 1995. On representation and querying incomplete information in databases with bags. *Information Processing Letters 56*, 4 (Nov.), 209–214.

LISITSA, A. AND SAZONOV, V. Y. 1997. Delta-languages for sets and LOGSPACE computable graph transformers. *Theoretical Computer Science 175*, 1 (April), 183–222.

LIVCHAK, A. 1983. The relational model for process control. *Automatic Documentation and Mathematical Linguistics 4*, 27–29. In Russian.

LLOYD, J. W. 1987. *Foundations of Logic Programming (2nd edition)*. Springer Verlag.

LOBO, J., MINKER, J., AND RAJASEKAR, A. 1992. *Foundations of Disjunctive Logic Programming*. Logic Programming Series. MIT Press.

MAHER, M. J. 1988. Equivalences of logic programs. In J. MINKER Ed., *Foundations of Deductive Databases and Logic Programming*, pp. 627–658. Washington, DC, USA: Morgan Kaufmann.

MAKANIN, G. 1977. The problem of solvability of equations in free semigroups. *Matematiceskij Sbornik 103*, 2, 147–236. In Russian. English Translation in: *Math. USSR Sbornik 32*, 2, 129–198, 1977.

MARCINKOWSKI, J. 1996a. The 3 frenchmen method proves undecidability of the uniform boundedness for single recursive rule ternary DATALOG programs. In C. PUECH AND R. REISCHUK Eds., *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, February 22-24, 1996, Proceedings*, Volume 1046 of *Lecture Notes in Computer Science* (Feb. 1996), pp. 427–438. Springer Verlag.

MARCINKOWSKI, J. 1996b. DATALOG SIRUPs uniform boundedness is undecidable. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, 27-30 July 1996 (LICS'96)* (July 1996), pp. 13–24. IEEE Computer Society Press.

MARCINKOWSKI, J. 1999. Achilles, turtle, and undecidable boundedness problems for small DATALOG programs. *SIAM Journal on Computing 29*, 1, 231–257.

MARCINKOWSKI, J. AND PACHOLSKI, L. 1992. Undecidability of the Horn-clause implication problem. In *33rd Annual Symposium on Foundations of Computer Science (FOCS'92)* (Pittsburgh, Pennsylvania, 24–27 Oct. 1992), pp. 354–362. IEEE Computer Society Press.

MAREK, V. W., NERODE, A., AND REMMEL, J. B. 1992. How complicated is the set of stable models of a recursive logic program? *Annals of Pure and Applied Logic 56*, 1–3 (April), 119–135.

MAREK, V. W., NERODE, A., AND REMMEL, J. B. 1994. The stable models of a predicate logic program. *Journal of Logic Programming 21*, 3 (Nov.), 129–153.

MAREK, V. W., NERODE, A., AND REMMEL, J. B. 1996. On the complexity of abduction. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, 27-30 July 1996 (LICS'96)* (July 1996), pp. 513–522. IEEE Computer Society Press.

MAREK, V. W., RAJASEKAR, A., AND TRUSZCZYŃSKI, M. 1995. Complexity of computing with extended propositional logic programs. *Annals of Mathematics and Artificial Intelligence 15*, 3–4, 357–378.

MAREK, V. W. AND TRUSZCZYŃSKI, M. 1991. Autoepistemic logic. *Journal of the Association for Computing Machinery 38*, 3 (July), 588–619.

MARKUSZ, Z. AND KAPOSI, A. A. 1982. A design methodology in prolog programming. In M. VAN CANEGHEM Ed., *Proceedings of the First International Logic Programming Conference, September, 14-17th, 1982. Faculté des Science de Luminy, ADDP-GIA, Marseille, France (ICLP'82)* (Sept. 1982), pp. 139–145.

MARTELLI, A. AND MONTANARI, U. 1976. Unification in linear time and space: a structured presentation. Technical Report B76-16 (Nota Interna), IEI, Consiglio Nazionale delle Ricerche, Pisa, Italy.

MARTELLI, A. AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems 4*, 2 (April), 258–282.

MATIYASEVIČ, Y. V. 1970. Enumerable sets are Dophantine. *Doklady Akademii Nauk SSSR 191*, 2, 279–282. In Russian. English Translation in: *Soviet Mathematical Doklady 11*, 354–357, 1970.

MCALLESTER, D. 1993. Automatic recognition of tractability in inference relations. *Journal of the Association for Computing Machinery 40*, 2, 284–303.

MINKER, J. 1982. On indefinite data bases and the closed world assumption. In D. W. LOVELAND Ed., *6th Conference on Automated Deduction, New York, USA, June 7-9, 1982, Proceedings*, Number 138 in Lecture Notes in Computer Science (June 1982), pp. 292–308. Springer Verlag.

MINKER, J. Ed. 1988. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, Washington, DC, USA.

MINKER, J. 1994. Overview of disjunctive logic programming. *Annals of Mathematics and Artificial Intelligence 12*, 1,2 (Dec.), 1–24.

MINKER, J. 1996. Logic and databases: A 20 year retrospective. In *Logic in Databases, International Workshop LID'96, San Miniato, Italy, July 1-2, 1996, Proceedings*, Volume 1154 of *Lecture Notes in Computer Science* (1996), pp. 3–57. Springer.

MINKER, J. AND NICOLAS, J.-M. 1983. On recursive axioms in deductive databases. *Information Systems 8*, 1, 1–13.

MINKER, J. AND RUIZ, C. 1994. Semantics for disjunctive logic programming with explicit and default negation. *Fundamenta Informaticae 20*, 1/2/3 (March/April/May), 145–192.

MUGGLETON, S. 1992. Inductive logic programming. In S. MUGGLETON Ed., *Inductive Logic Programming*, pp. 3–28. Academic Press.

NARENDRAN, P. AND OTTO, F. 1990. Some results on equational unification. In M. E. STICKEL Ed., *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, Volume 449 of *Lecture Notes in Computer Science* (July 1990), pp. 276–291. Springer Verlag.

NAUGHTON, J. F. 1989. Data independent recursion in deductive databases. *Journal of Computer and System Sciences 38*, 2 (April), 259–289.

NAUGHTON, J. F. AND SAGIV, Y. 1987. A decidable class of bounded recursions. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California (PODS'87)* (March 1987), pp. 227–236. ACM Press.

NAUGHTON, J. F. AND SAGIV, Y. 1991. A simple characterization of uniform boundedness for a class of recursions. *Journal of Logic Programming 10*, 1/2/3&4 (Jan.), 233–253.

OCHOZKA, V., ŠTĚPÁNKOVÁ, O., AND ŠTĚPÁNEK, P. 1988. Normal forms and the complexity of computations of logic programs. In E. BÖRGER, H. KLEINE BÜNING, AND M. M. RICHTER Eds., *CSL '88, 2nd Workshop on Computer Science Logic, Duisburg, Germany, October 3-7, 1988, Proceedings*, Volume 385 of *Lecture Notes in Computer Science*, pp. 357–371. Springer Verlag.

OTTO, M. AND VAN DEN BUSSCHE, J. 1996. First-order queries on databases embedded in an infinite structure. *Information Processing Letters 60*, 1 (Oct.), 37–41.

PALOPOLI, L. 1992. Testing logic programs for local stratification. *Theoretical Computer Science 103*, 2, 205–234.

PAPADIMITRIOU, C. H. AND YANNAKAKIS, M. 1997. On the complexity of database queries. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona (PODS'97)* (May 1997), pp. 12–19. ACM Press.

PAPADIMITRIOU, C. H. 1985. A note on the expressive power of prolog. *Bulletin of the EATCS 26*, 21–23.

PAPADIMITRIOU, C. H. 1994. *Computational Complexity.* Addison-Wesley.

PAPADIMITRIOU, C. H. AND YANNAKAKIS, M. 1986. A note on succinct representations of graphs. *Information and Control 71*, 3 (Dec.), 181–185.

Paredaens, J., van den Bussche, J., and van Gucht, D. 1998. First-order queries on finite structures over the reals. *SIAM Journal on Computing 27*, 6 (Dec.), 1747–1763.

Paredaens, J. and van Gucht, D. 1992. Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems 17*, 1 (March), 65–93.

Paterson, M. S. and Wegman, M. N. 1978. Linear unification. *Journal of Computer and System Sciences 16*, 2 (April), 158–167.

Pearce, D. and Wagner, G. 1991. Logic programming with strong negation. In P. Schroeder-Heister Ed., *Extensions of Logic Programming, International Workshop, Tübingen, FRG, December 8-10, 1989, Proceedings (ELP'89)*, Number 475 in LNCS (1991), pp. 311–326. Springer.

Przymusinski, T. C. 1988a. On the declarative semantics of deductive databases and logic programs. In J. Minker Ed., *Foundations of Deductive Databases and Logic Programming*, pp. 193–216. Washington, DC, USA: Morgan Kaufmann.

Przymusinski, T. C. 1988b. Perfect model semantics. In R. A. Kowalski and K. A. Bowen Eds., *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (ICLP/SLP'88)* (Aug. 1988), pp. 1081–1096. MIT Press.

Przymusinski, T. C. 1991. Stable semantics for disjunctive programs. *New Generation Computing 9*, 3/4, 401–424.

Przymusinski, T. C. 1995. Static semantics for normal and disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence 14*, 2–4 (Sept.), 323–357.

Renegar, J. 1988. A faster PSPACE algorithm for deciding the existential theory of the reals. In *29th Annual Symposium on Foundations of Computer Science* (White Plains, New York, 24–26 Oct. 1988), pp. 291–295. IEEE.

Robinson, J. 1965. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery 12*, 1 (Jan.), 23–41.

Rosati, R. 1997. Reasoning with minimal belief and negation as failure: Algorithms and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island* (July 1997), pp. 430–435. AAAI Press/MIT Press.

Rosati, R. 1998. Expressiveness vs. complexity in nonmonotonic knowledge bases: Propositional case. In H. Prade Ed., *13th European Conference on Artificial Intelligence, Brighton, UK, August 23-28 1998, Proceedings (ECAI'98)* (Chichester, Aug. 1998), pp. 47–48.

Saccà, D. 1995. Deterministic and nondeterministic stable model semantics for unbound DATALOG queries. In G. Gottlob and M. Y. Vardi Eds., *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, Volume 893 of *Lecture Notes in Computer Science* (Jan. 1995), pp. 353–367. Springer Verlag.

Saccà, D. 1997. The expressive powers of stable models for bound and unbound DATALOG queries. *Journal of Computer and System Sciences 54*, 3 (June), 441–464.

Sagiv, Y. 1985. On computing restricted projections of representative instances. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 25-27, 1985, Portland, Oregon (PODS'85)* (March 1985), pp. 171–180. ACM Press.

Sagiv, Y. 1988. Optimizing datalog programs. In J. Minker Ed., *Foundations of Deductive Databases and Logic Programming*, pp. 659–698. Washington, DC, USA: Morgan Kaufmann.

Sagiv, Y. and Yannakakis, M. 1980. Equivalence among relational expressions with the union and difference operators. *Journal of the Association for Computing Machinery 27*, 4 (Oct.), 633–655.

Sakama, C. and Inoue, K. 1994a. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning 13*, 1, 145–172.

SAKAMA, C. AND INOUE, K. 1994b. On the equivalence between disjunctive and abductive logic programs. In P. VAN HENTENRYCK Ed., *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, June 13-18, 1994, Santa Marherita Ligure, Italy (ICLP'94)* (June 1994), pp. 489–503. The MIT Press.

SAVITCH, W. J. 1970. Relationship between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences 4*, 2 (April), 177–192.

SAZONOV, V. Y. 1993. Hereditarily-finite sets, data bases and polynomial-time computability. *Theoretical Computer Science 119*, 1, 187–214.

SAZONOV, V. Y. AND LISITSA, A. 1995. Delta-languages for sets and sub-PTIME graph transformers. In G. GOTTLOB AND M. Y. VARDI Eds., *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, Volume 893 of *Lecture Notes in Computer Science* (1995), pp. 125–138. Springer Verlag.

SCHLIPF, J. S. 1990. The expressive powers of the logic programming semantics. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee (PODS'90)* (April 1990), pp. 196–204. ACM Press.

SCHLIPF, J. S. 1995a. Complexity and undecidability results for logic programming. *Annals of Mathematics and Artificial Intelligence 15*, 3–4, 257–288.

SCHLIPF, J. S. 1995b. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences 51*, 1, 64–86. Abstract: [Schlipf 1990].

SCHOLZ, H. AND HASENJAEGER, G. 1961. *Grundzüge der mathematischen Logik.* Springer, Berlin.

SEBELÍK, J. AND ŠTĚPÁNEK, P. 1982. Horn clause programs for recursive functions. In K. L.CLARK AND S.-Å. TÄRNLUND Eds., *Logic Programming*, pp. 325–340. Academic Press.

SHAPIRO, E. Y. 1984. Alternation and the computational complexity of logic programs. *Journal of Logic Programming 1*, 1 (June), 19–33.

SHMUELI, O. 1987. Decidability and expressiveness aspects of logic queries. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California (PODS'87)* (March 1987), pp. 237–249. ACM Press.

SMULLYAN, R. 1961. *Theory of Formal Systems.* Princeton University Press, Princeton, New Jersey. Annals of Mathematical Studies Vol 47.

SMULLYAN, R. M. 1956. On definability by recursion (Abstract 782t). *Bulletin AMS 62*, 601.

ŠTĚPÁNEK, P. AND ŠTĚPÁNKOVÁ, O. 1986. Logic programs and alternation. In E. Y. SHAPIRO Ed., *Third International Conference on Logic Programming, Imperial College of Science and Technology, London, United Kingdom, July 14-18, 1986, Proceedings (ICLP'86)*, Number 225 in Lecture Notes in Computer Science, pp. 99–106. Springer Verlag.

ŠTĚPÁNKOVÁ, O. AND ŠTĚPÁNEK, P. 1984. Transformations of logic programs. *Journal of Logic Programming 1*, 4 (Dec.), 305–318.

SUCIU, D. 1997. Fixpoints for complex objects. *Theoretical Computer Science 176*, 1–2 (April), 283–328.

TÄRNLUND, S.-Å. 1977. Horn clause computability. *BIT 17*, 2, 215–226.

TURING, A. M. 1936-1937. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society Series 2*, 42, 230–265. Corrections ibid. 544–546. Reprinted in [Davis 1965].

ULLMAN, J. D. 1988. *Principles of Database and Knowledge-Base Systems*, Volume I. Computer Science Press, New York, N.Y.

ULLMAN, J. D. 1989. *Principles of Database and Knowledge-Base Systems*, Volume II. Computer Science Press, New York, N.Y.

ULLMAN, J. D. AND VAN GELDER, A. 1988. Parallel complexity of logical query programs. *Algorithmica 3*, 5–42.

VADAPARTY, K. V. 1991. On the power of rule-based languages with sets. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database*

Systems, May 29-31, 1991, Denver, Colorado (PODS'91) (May 1991), pp. 26–36. ACM Press.

van der Meyden, R. 1992. The complexity of querying indefinite data about linearly ordered domains. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California (PODS'92)* (June 1992), pp. 331–345. ACM Press.

van der Meyden, R. 1997. The complexity of querying indefinite data about linearly ordered domains. *Journal of Computer and System Sciences 54*, 1 (Feb.), 113–135. Abstract: [van der Meyden 1992].

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery 23*, 4 (Oct.), 733–742.

van Gelder, A. 1989. The alternating fixpoint of logic programs with negation. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania (PODS'89)* (March 1989), pp. 1–10. ACM Press.

van Gelder, A., Ross, K. A., and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery 38*, 3 (July), 620–650.

van Leeuwen, J. Ed. 1990. *Handbook of Theoretical Computer Science*, Volume B: Formal Models and Sematics. Elsevier Science and The MIT Press.

Vandeurzen, L., Gyssens, M., and van Gucht, D. 1996. On query languages for linear queries definable with polynomial constraints. In *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, Number 1118 in Lecture Notes in Computer Science (Aug. 1996), pp. 468–481.

Vardi, M. Y. 1982. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC'82)* (San Francisco, California, 5–7 May 1982), pp. 137–146.

Vardi, M. Y. 1988. Decidability and undecidability results for boundedness of linear recursive queries. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 21-23, 1988, Austin, Texas (PODS'88)* (March 1988), pp. 341–351. ACM Press.

Vardi, M. Y. 1995. On the complexity of bounded-variable queries. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California (PODS'95)* (May 1995), pp. 266–276. ACM Press.

Veith, H. 1994. Logical reducibilities in finite model theory. Master's thesis, Information Systems Department, TU Vienna, Austria.

Veith, H. 1998. Succinct representation, leaf languages, and projection reductions. *Information and Computation 142*, 2 (May), 207–236.

Vianu, V. 1997. Rule-based languages. *Annals of Mathematics and Artificial Intelligence 19*, 1–2, 215–259.

Vorobyov, S. G. and Voronkov, A. 1998. Complexity of nonrecursive logic programs with complex values. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington (PODS'98)* (June 1998), pp. 244–253. ACM Press.

Voronkov, A. 1995. On computability by logic programs. *Annals of Mathematics and Artificial Intelligence 15*, 3–4, 437–456.

Wong, L. 1996. Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences 52*, 3 (June), 495–505.

Yannakakis, M. 1981. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings (VLDB'81)* (Sept. 1981), pp. 82–94. IEEE Computer Society Press.

YASUURA, H. 1984. On parallel computational complexity of unification. In *Fifth Generation Computer Systems 1984, Proceedings of the International Conference on Fifth Generation Computer Systems 1984, Tokyo, Japan, November 6-9, 1984 (FGCS'84)* (1984), pp. 235–243. Institute for New Generation Computer Technology (ICOT): OHMSHA and North-Holland.

YOU, J.-H. AND YUAN, L.-Y. 1995. On the equivalence of semantics for normal logic programming. *Journal of Logic Programming 22*, 3 (March), 211–222.