Complexity and Expressive Power of Logic Programming

Evgeny Dantsin[1] Thomas Eiter[2] Georg Gottlob[3] Andrei Voronkov[4]

Abstract

This paper surveys various complexity 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. The complexity of the unification problem is also addressed.

1. Introduction

Logic programming (LP) 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 [87]. LP was developed in the early 1970's based on work in automated theorem proving [68, 88], in particular, on Robinson's *resolution principle* [113]. 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.

[1]Steklov Institute of Mathematics at St. Petersburg, Fontanka 27, St. Petersburg 191011, Russia. Currently at Uppsala University. Email dantsin@pdmi.ras.ru. Supported by grants from INTAS, RFBR and the Swedish Institute.

[2]AG Informatik, University of Gießen, Arndtstraße 2, D-35392 Gießen, Germany. Email eiter@informatik.uni-giessen.de.

[3]Information Systems Department, TU Vienna, Paniglgasse 16, A-1040 Vienna. Email gottlob@dbai.tuwien.ac.at.

[4]Computing Science Department, Uppsala University, P.O. Box 311, S-751 05, Uppsala, Sweden. Email voronkov@csd.uu.se. Supported by a TFR grant.

With the advent of the programming language Prolog [32], 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 LP. 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 LP 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 LP. 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 [26, 122]). This query language — based on function-free pure LP — allows a user to formulate recursive queries that cannot be expressed with standard query languages such as SQL-2.

There are many interesting complexity results on LP. 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 $\mathbf{P}$ 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) LP. The aim of the paper is twofold. First, a broad survey and many pointers to the literature are given. Second, a few fundamental topics are explained in greater detail, in particular, the basic results on plain LP (Section 3) and some fundamental issues related to descriptive complexity (Section 6). 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. There are many interesting results which we cannot mention for space reasons; such results may be found in other surveys, such as, e.g., [24, 118]. For example, results on abductive logic programming [52, 53], on intuitionistic logic programming [22], and on Prolog [41].

The paper is organized as follows. In Section 2 a short introduction to LP is given. We introduce datalog and distinguish between the notions of *data complexity*, *program complexity*, and *combined complexity* of classes of datalog programs. Section 3 presents the main complexity results on plain LP and datalog. Section 4 discusses the complexity of LP with negated atoms in rule bodies. Section 5 deals with disjunctive logic programming. Section 6 discusses the expressive power of datalog and of various datalog extensions. Section 7 reports on the complexity of the unification problem. Section 8 deals with LP extended by equality. Finally, Section 9 deals with the complexity of *constraint logic programming* and with the expressive power of logic programming with complex values.

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 [94, 6, 9, 15].

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.

2.1. Syntax of logic programs

Logic programs are formulated in a language $\mathcal{L}$ of predicates and functions of nonnegative arity; 0-ary functions are constants. A language $\mathcal{L}$ is *function-free* if it contains no function symbols 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 symbols 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 by 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 $\mathcal{L}$ is the set of all ground atoms that can be formed by using predicates from $\mathcal{L}$ and terms from $U_{\mathcal{L}}$.

A *Horn clause* is a rule of the form

$$A_0 \leftarrow A_1, \ldots, A_m \qquad (m \geq 0) \tag{1}$$

where each A_i is an atom. The parts on the left and on the right of "$\leftarrow$" are 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 all terms in it are ground.

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 $\mathcal{L}(P)$ for technical reasons. 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:

$$\begin{aligned}
\mathit{shut_down} &\leftarrow \mathit{overheat} \\
\mathit{shut_down} &\leftarrow \mathit{leak} \\
\mathit{leak} &\leftarrow \mathit{valve_closed}, \mathit{pressure_loss} \\
\mathit{valve_closed} &\leftarrow \mathit{signal_1} \\
\mathit{pressure_loss} &\leftarrow \mathit{signal_2} \\
\mathit{overheat} &\leftarrow \mathit{signal_3} \\
\mathit{signal_1} &\leftarrow \\
\mathit{signal_2} &\leftarrow
\end{aligned}$$

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 the propositional atoms.

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 $T_P : 2^{B_P} \mapsto 2^{B_P}$, where 2^{B_P} denotes the set of all Herbrand interpretations of P, by

$$T_P(I) = \{A_0 \in B_P \mid P \text{ contains a rule } A_0 \leftarrow A_1, \ldots, A_m \text{ such that } \{A_1, \ldots, A_m\} \subseteq I\}$$

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^∞, which 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 a *consequence* of a set P of clauses if $A \in T_P^\infty$ (we write $P \models A$). Also, by definition, a negated ground atom $\neg A$ is a *consequence* of P, denoted $P \models \neg A$, if $A \notin T_P^\infty$. The *semantics* of a set P of ground clauses is defined as the following set $\mathcal{M}(P)$ 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 1 (ctd) For program P above, we have

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

Thus, the least fixpoint is reached in four steps; e.g., $P \models shut\_down$ and $P \models \neg overheat$.

It appears that 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 [123].

The semantics of arbitrary logic programs is now defined as follows. Let the *grounding* of a clause r in a language $\mathcal{L}$, denoted $ground(r, \mathcal{L})$, 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

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

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

Example 2 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 $ground(P)$ 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

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

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

In practice, generating $ground(P)$ 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 Robinson's famous *Resolution Principle* [113]. The major variant is SLD-resolution [88, 10].

Roughly, SLD-resolution can be described as follows. A *goal* is a conjunction of atoms. 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 such a unifier ϑ is found, 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 [6, 94]; for resolution on general clauses, see e.g. [89]. The complexity of unification will be dealt with in Section 7.

2.3. Datalog

Logic programming is a suitable formalism for querying relational databases. In this context, the LP-based query language *datalog* and various extensions thereof have been defined. Over traditional query languages such as relational algebra or SQL-2, datalog has the advantage of being able to express *recursive queries*.

In the context of LP, 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 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 state a clarifying example.

Example 3 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:

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

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 $\{\mathit{ancestor}, \mathit{person}\}$ containing among others the following ground facts: $\mathit{ancestor}(\mathit{mary}, \mathit{joe})$, $\mathit{ancestor}(\mathit{john}, \mathit{joe})$, $\mathit{person}(\mathit{john})$, $\mathit{person}(\mathit{susan})$. The last fact is in R because *susan* is included as a constant in U_D. However, the fact $\mathit{person}(\mathit{harry})$ 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 $\mathit{Rels}(\mathcal{D})$ of relation names with associated arities and a (possibly infinite) domain $\mathit{Dom}(\mathcal{D})$. For each database schema $\mathcal{D}$, we denote by $\mathit{HB}(\mathcal{D})$ the Herbrand base corresponding to the function-free language whose predicate symbols are $\mathit{Rels}(\mathcal{D})$ and whose constant symbols are $\mathit{Dom}(\mathcal{D})$.

A *database* (also, *database instance*) D over a schema $\mathcal{D}$ is given by a finite subset of the Herbrand base $D \subseteq \mathit{HB}(\mathcal{D})$ together with an associated finite universe U_D such that $C \subseteq U_D \subseteq \mathit{Dom}(\mathcal{D})$, where C denotes the set of 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 $p \in \mathit{Rels}(\mathcal{D})$ in D. Moreover, $\mathit{INST}(\mathcal{D})$ 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: $\mathit{Dom}(\mathcal{D}_{in}) = \mathit{Dom}(\mathcal{D}_{out}) = \mathit{Dom}(\mathcal{D})$ and $\mathit{Rels}(\mathcal{D}_{in}) \subseteq \mathit{Rels}(\mathcal{D})$ and $\mathit{Rels}(\mathcal{D}_{out}) \subseteq \mathit{Rels}(\mathcal{D})$ and $\mathit{Rels}(\mathcal{D}_{in}) \cap \mathit{Rels}(\mathcal{D}_{out}) = \emptyset$. Moreover, each predicate symbol appearing in P is contained in $\mathit{Rels}(\mathcal{D})$ and no predicate symbol from $\mathcal{D}_{in}$ appears in a rule head of P (the latter means that the input database is never modified 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, $\mathcal{M}_P : \mathit{INST}(\mathcal{D}_{in}) \mapsto \mathit{INST}(\mathcal{D}_{out})$ 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 $p \in \mathit{Rels}(\mathcal{D}_{out})$,

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

where $\mathcal{M}$ and *ground* are defined as in Section 2.2 and $\mathcal{L}(P, D_{in})$ is the language of $P \cup D_{in}$ 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 LP. In a similar way, the semantics of various extensions of datalog is inherited from the corresponding extensions of logic programming.

There are three interesting complexity issues connected to plain datalog and its various extensions.

- The **data complexity** is the complexity of checking whether $D_{in} \cup P \models A$ for a *fixed* datalog program P and *variable* input databases D_{in} and ground atoms A.
- The **program complexity** is the complexity of checking whether $D_{in} \cup P \models A$ for *variable* datalog programs P and ground atoms A over a *fixed* input database D_{in}. We recall that if D_{in} is fixed, then the set of constants that may appear in P and A is fixed too.

- The **combined complexity** is the complexity of checking whether $D_{in} \cup P \models A$ for *variable* datalog programs P, ground atoms A, and input database D_{in}.

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 w.r.t. polynomial-time reductions. This is due to the fact that w.r.t. 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., **DETIME** vs **DEXPTIME**).

As for LP 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 LP, we mean the following kind of complexity:

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

3. Complexity of plain logic programming

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

3.1. Simulation of Deterministic Turing machines by logic programs

Formally, a *deterministic Turing machine (DTM)* is a quintuple $T = \langle \Sigma, S, \delta, s_0, S^+\rangle$, where Σ is a finite alphabet of tape symbols, containing also the special blank symbol $\#$, S is a finite set of states, $\delta : (S \times \Sigma) \longrightarrow \Sigma \times \{-1, 0, 1\} \times S$ is the transition function, $s_0 \in S$ is the initial state, and $S^+ \subseteq S$ is the set of accepting states; without loss of generality we assume that every accepting state is a terminal state, i.e., whenever T enters an accepting state, it remains in this state and stops running.

A DTM has a semi-infinite worktape whose cells $c_0, c_1, c_2 \ldots$ are on input I initialized as follows. Cells $c_0, \ldots, c_{|I|-1}$ contain the symbols of string I, where $|I|$ is the length of I, and all other cells contain $\#$.

The transition function δ is represented by a table whose rows are quintuples $\langle s, \alpha, \alpha', d, s'\rangle$, whose meaning is stated as follows as an if-then-rule:

> **if** at some instant τ of time T is in state s, the workhead is positioned at cell c_π, and cell c_π holds symbol α
> **then** at instant $\tau + 1$, T is in state s', the cell c_π holds symbol α', and the workhead is positioned at $c_\pi + d$.

Here, it is assumed without loss of generality that $d \neq -1$ whenever $\pi = 0$, i.e., the workhead never moves left of c_0.

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 $LP(T, I, N)$. For achieving this, we define various classes of propositional atoms:

$CC_\alpha[\tau, \pi]$ for $0 \leq \tau \leq N$, $0 \leq \pi \leq N$, and $\alpha \in \Sigma$. Intuitive meaning: At instant τ of the computation, cell π contains symbol α.

$WP[\tau, \pi]$ for $0 \leq \tau \leq N$, and $0 \leq \pi \leq N$. Intuitive meaning: At instant τ the workhead is positioned at cell number π.

$ST_s[\tau]$ for $0 \leq \tau \leq N$, and $s \in S$. Intuitive meaning: At instant τ the machine is in state s.

$ACCEPT$**:** the machine has reached an accepting state.

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

$$\begin{array}{rl}
CC_\alpha[0, \pi] \leftarrow & \text{for } 0 \leq \pi < |I|, \text{ where } I_\pi = \alpha \\
CC_\#[0, \pi] \leftarrow & \text{for } |I| \leq \pi \leq N \\
WP[0, 0] \leftarrow & \\
ST_{s_0}[0] \leftarrow &
\end{array}$$

Each entry $\langle s, \alpha, \alpha', d, s'\rangle$ 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 $0 \leq \pi + d \leq N$.

$$\begin{array}{rl}
CC_{\alpha'}[\tau + 1, \pi] & \leftarrow ST_s[\tau], CC_\alpha[\tau, \pi], WP[\tau, \pi] \\
WP[\tau + 1, \pi + d] & \leftarrow ST_s[\tau], CC_\alpha[\tau, \pi], WP[\tau, \pi] \\
ST_{s'}[\tau + 1] & \leftarrow ST_s[\tau], CC_\alpha[\tau, \pi], WP[\tau, \pi]
\end{array}$$

These clauses almost perfectly describe what is happening during a state transition from instant τ to instant $\tau + 1$. However, it should not be forgotten that those tape cells which are not changed during the transition keep there 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 c_π, c'_π, where $0 \leq \tau < N$, $0 \leq \pi < \pi' \leq N$, and have the following form:

$$CC_\alpha[\tau+1,\pi] \leftarrow CC_\alpha[\tau,\pi], WP[\tau,\pi']$$
$$CC_\alpha[\tau+1,\pi'] \leftarrow CC_\alpha[\tau,\pi'], WP[\tau,\pi]$$

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

$$ACCEPT \leftarrow ST_s[\tau] \qquad \text{for } 0 \leq \tau \leq N,\, s \in S^+$$

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

Lemma 3.1 *$LP(T,I,N) \models ACCEPT$ if and only if machine T accepts the input string I within N steps.*

3.2. Complexity of propositional LP

The simulation of a DTM by a propositional logic program, as described in Section 3.1 is almost all we need in order to determine the complexity of propositional LP, i.e., the complexity of deciding whether $P \models A$ holds for a given logic program P and ground atom A.

Theorem 3.2 (implicit in [80, 127, 76]) *Propositional LP is $\mathbf{P}$-complete under logspace reductions.*

Proof. a) *Membership*. It 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.

b) *Hardness*. Let A be a language in $\mathbf{P}$. 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 $LP(T,I,q(|I|))$ as described in Section 3.1. By Lemma 3.1, $LP(T,I,q(|I|)) \models ACCEPT$ if and only if T has reached an accepting state within $q(n)$ steps. The translation from I to $LP(T,I,q(|I|))$ is very simple and is clearly feasible in logarithmic space, since all rules of $LP(T,I,q(|I|))$ can be generated independently of each other and each has size logarithmic in the input; 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 $\mathbf{P}$ is logspace reducible to propositional LP. Hence, logic programming is $\mathbf{P}$-hard under logspace reductions. ■

Obviously, this theorem can be proved by simpler reductions from other $\mathbf{P}$-complete problems, e.g. from the monotone circuit value problem; however, our proof from first principles unveils the computational nature of LP and provides a basic framework form which further results will be derived by slight adaptations in the sequel.

Notice that in a standard programming environment, propositional LP is feasible in linear time by using appropriate data structures, as follows from results about deciding Horn satisfiability [43]. This does not mean that all problems in $\mathbf{P}$ are solvable in linear time; first, the model of computation used in [43] is the RAM machine, and second polynomial-time reductions may in general polynomially increase the input.

Theorem 3.2 holds under stronger reductions. In fact, it holds under the requirement that the logspace reduction is also a polylogtime reduction (PLT). Briefly, a map $f: \Pi \mapsto \Pi'$ from problem Π to problem Π' is a PLT-reduction, if there are polylogtime deterministic direct access Turing machines (DDATMs) N, 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. [129] for details). (Recall that a DDATM has a separate input tape whose cells can be indirectly accessed by use of an index register tape.) Since the above encoding of a DTM into LP is highly regular, it is easily seen that it is a PLT reduction.

Syntactical restrictions on programs lead to completeness for classes inside $\mathbf{P}$. Let $LP(k)$ denote logic restricted to programs where each clause has at most k atoms in the body. Then, by results in [127, 77], one easily obtains

Theorem 3.3 *$LP(1)$ is $\mathbf{NL}$-complete under logspace reductions.*

Notice that the above DTM encoding can be easily modified to programs in $LP(2)$. Hence, $LP(2)$ is $\mathbf{P}$-complete.

Further restrictions yield problems complete for $\mathbf{L}$ (of course, under reductions stronger than logspace reductions), which we omit here.

3.3. Complexity of datalog

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

Theorem 3.4 (implicit in [127, 76]) *Datalog is data complete in $\mathbf{P}$.*

In fact, this result follows from the proof of Theorem 6.2. An alternative proof of $\mathbf{P}$-hardness can be given by writing a simple datalog *meta-interpreter* for propositional $LP(k)$, where k is a constant.

Represent rules $A_0 \leftarrow A_1, \ldots, A_i$, $0 \leq i \leq k$, 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 $LP(k)$ 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, $P \models A$ precisely if $P_{MI} \cup P(D) \models T(A)$. $\mathbf{P}$-hardness of the data complexity of datalog is immediate from Theorem 3.2.

The program complexity is exponentially higher.

Theorem 3.5 (implicit in [127, 76]) *Datalog is program complete in* **DEXPTIME**.

Proof. (Sketch) a) *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 3.2, the program complexity is in **DEXPTIME**.

b) *Hardness*. In order to prove **DEXPTIME**-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 LP from above, but use the predicates $CC_\alpha(x, y)$, $WP(x, y)$, $ST_s(x)$ instead of the propositional letters $CC_\alpha[\tau, \pi]$, $WP[\tau, \pi]$, $ST_s[\tau]$, 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 $CC_\alpha[0, \pi]$ are readily translated into the datalog rules $CC_\alpha(\mathbf{x}, \mathbf{t}) \leftarrow First^m(\mathbf{x})$, where $\mathbf{t}$ represents the position π, and similarly the facts $WP[0, 0]$ and $ST_{s_0}[0]$. The remaining initialization facts $CC_\#[0, \pi]$, $|I| \leq \pi \leq N$, are translated to the rule

$$CC_\#(\mathbf{x}, \mathbf{y}) \leftarrow First^m(\mathbf{x}), \leq^m(\mathbf{t}, \mathbf{y})$$

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

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

$$CC_{\alpha'}[\tau + 1, \pi] \leftarrow ST_s[\tau], CC_\alpha[\tau, \pi], WP[\tau, \pi]$$

is translated into

$$CC_{\alpha'}(\mathbf{x}', \mathbf{y}) \leftarrow ST_s(\mathbf{x}), CC_\alpha(\mathbf{x}, \mathbf{y}), WP(\mathbf{x}, \mathbf{y}), Succ^m(\mathbf{x}, \mathbf{x}').$$

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 $\leq N$ steps. It is easy to see that P' can be constructed in logarithmic workspace from T and I. Hence, datalog has **DEXPTIME**-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 [108, 14]. We briefly outline this in the rest of this section.

This technique utilizes a conversion lemma [14] 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 [14] for details). The strongest form of the conversion lemma appears in [129], where X is PLT and Y is monotone projection reducibility [77]. The conversion lemma gives rise to an upgrading theorem [14, 54, 66, 129], stated here in the strongest form of [129]:

Theorem 3.6 *Let* $\mathbf{C_1}$ *and* $\mathbf{C_2}$ *be complexity classes s.t.* $long(\mathbf{C_1}) \subseteq \mathbf{C_2}$. *If* Π *is hard for* $\mathbf{C_2}$ *under PLT-reductions, then* $s(\Pi)$ *is hard for* $\mathbf{C_1}$ *under projection reductions.*

Here $long(\mathbf{C_1}) = \{long(A) \mid A \in \mathbf{C_1}\}$, where $long(A) = \bigcup_{n \in 1A} \{0, 1\}^n$.

From the observations in Section 3.2, we then obtain that $s(LP(2))$ is **DEXPTIME**-hard under projection reductions, where each program P is stored in the database $D(P)$, which is represented by a binary string in the standard way.

$s(LP(2))$ 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. E.g., 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 $g_i(\mathbf{x}) \leftarrow g_k(\mathbf{x})$.

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 logspace, from which **DEXPTIME**-hard program complexity of datalog follows. See [54, 55, 66] for details.

3.4. Complexity of LP 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 the query answering in full LP. 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 LP follows from undecidability for diophantine equations [103]. Moreover, this reduction shows r.e.-completeness of LP.

Theorem 3.7 ([5, 121]) *Full LP is r.e.-complete.*

Of course, this theorem may as well be proved by a simple encoding of Turing machines similar as in the proof of Theorem 3.5 (use terms $f^n(c)$, $n \geq 0$, for representing cell positions and time instants). Theorem 3.7 was generalized in [130] for more expressive S-semantics and C-semantics [59].

A natural decidable fragment of LP with functions are non-recursive programs, in which intuitively no predicate depends syntactically on itself (see Section 4.1 for a definition). Their complexity is characterized by the following theorem.

Theorem 3.8 ([37]) *Non-recursive LP 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 LP with function symbols are known to be decidable. For example, the following result was established in [120], by using a simulation of alternating Turing machines by logic programs and vice versa.

Theorem 3.9 ([120]) *LP 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.*

For further investigations of decidability of subclasses of logic programs, see [40]. See also [20, 60] for further material on recursion-theoretic issues related to LP.

4. Complexity of LP with negation

4.1. Stratified negation

A *literal* L is either an atom A (called *positive*) or a negated atom $\neg A$ (called *negative*). 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

$$A \leftarrow L_1, \ldots, L_m \qquad (m \geq 0) \tag{2}$$

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. [9]). However, there is general consensus for stratified normal logic programs.

A normal logic program P is *stratified* [8], 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.

The *reduct* of a normal logic program P by a Herbrand interpretation I [64], denoted P^I, is 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 [8]. Note that in the propositional case $\mathcal{M}_{str}(P)$ is polynomially computable.

Theorem 4.1 (implicit in [8]) *Stratified propositional LP is* **P**-*complete. Stratified datalog is data complete in* **P** *and program complete in* **DEXPTIME**.

For full LP, stratified negation yields the arithmetical hierarchy.

Theorem 4.2 ([7]) *Full LP with n levels of stratified negation is Σ^0_{n+1}-complete.*

See [21, 107] for further complexity results on stratification.

A particular case of stratified negation are non-recursive logic programs. A stratified program is *non-recursive*, if it has a stratification such that each predicate p occurs in its defining stratum $P_{=str(p)}$ only in the heads of rules. E.g., the logic program produced by the DTM encoding from above is non-recursive.

Theorem 4.3 (implicit [77, 127]) *Non-recursive propositional LP is* **P**-*complete. Non-recursive datalog has logtime uniform AC^0 data complexity* [77] *and is program complete in* **PSPACE**.

4.2. Well-founded negation

Roughly, the well-founded semantics [125] (WFS) assigns value "unknown" to atom A, if it is defined by unstratified negation. Briefly, WFS can be defined as follows [16]. 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

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

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

Theorem 4.4 (implicit in [124, 125]) *LP under WFS is* **P**-*complete. Datalog under WFS is data complete in* **P** *and program complete in* **DEXPTIME**.

Whether deciding $P \models_{wfs} A$ can be done in linear-time is open [19]. For full LP, the following is known.

Theorem 4.5 ([119]) *Full LP under WFS is $\mathbf{\Pi}^1_1$-complete.*

4.3. LP under the stable model semantics

An interpretation I of a normal logic program P is a *stable model* of P [64], 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 4 Let P be the following program:

$$\begin{aligned} sleep &\leftarrow \neg work \\ work &\leftarrow \neg sleep \end{aligned}$$

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

Denote by $\mathrm{SM}(P)$ the set of stable models of P. The meaning 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 4.6 ([99]) *Given a propositional logic program P, deciding whether $\mathrm{SM}(P) \neq \emptyset$ is* **NP**-*complete.*

Proof. a) *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.

b) *Hardness*. Modify the DTM encoding in Section 3 for a nondeterministic Turing machine (NTM) T as follows. For each state s and symbol α, introduce atoms $B_{s,\alpha,1}[\tau]$,..., $B_{s,\alpha,k}[\tau]$ for all $1 \leq \tau < N$ and transitions $\langle s, \alpha, \alpha'_i, d_i, s_i\rangle$, $1 \leq i \leq k$. Add $B_{s,\alpha,i}[\tau]$ in the bodies of the transition rules for $\langle s, \alpha, \alpha'_i, d_i, s_i\rangle$ and the rule

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

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

$$ACCEPT \leftarrow \neg ACCEPT.$$

It ensures $ACCEPT$ 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,

Theorem 4.7 ([99, 119]; cf. also [85]) *LP under SMS is* co-**NP**-*complete. Datalog under SMS is data complete in* co-**NP** *and program complete in* co-**NEXPTIME**.

For full LP, SMS has the same complexity as WFS.

Theorem 4.8 ([119, 98]) *Full LP under SMS is $\mathbf{\Pi}^1_1$-complete.*

Further results on stable models of recursive (rather than only finite) logic programs can be found in [97].

4.4. Inflationary and noninflationary semantics

The inflationary semantics (INFS) [3, 2] is inspired by inflationary fixpoint logic [71]. In place of T_P^∞, it uses the limit $\widetilde{T}_P^\infty$ of the sequence $\widetilde{T}_P^0 = \emptyset$, $\widetilde{T}_P^{i+1} = \widehat{T}_P(\widetilde{T}_P^i)$, $i \geq 0$, 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 4.9 ([3]; implicit in [71]) *LP under INFS is* **P**-*complete. Datalog under INFS is data complete in* **P** *and program complete in* **DEXPTIME**.

The noninflationary semantics (NINFS) [3], in the version of [4, page 373], uses in place of T_P^∞ the limit $\widehat{T}_P^\infty$ of the sequence $\widehat{T}_P^0 = \emptyset$, $\widehat{T}_P^{i+1} = \widehat{T}_P(\widehat{T}_P^i)$, $i \geq 0$, 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 described earlier in [28, 127]. In particular, datalog under NIFNS is equivalent to partial fixpoint logic [3, 2].

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

Theorem 4.10 ([3, 2]) *LP under NINFS is* **PSPACE**-*complete. Datalog under NINFS is data complete in* **PSPACE** *and program complete in* **EXPSPACE**.

4.5. Further semantics of negation

A number of interesting further semantics, e.g. partial (maximal) stable models, regular models, perfect models, 2- and 3-valued completion semantics, fixpoint models, must remain undiscussed here; see e.g. [119, 115, 85] for more details and complexity results.

5. Disjunctive logic programming

Informally, disjunctive logic programming (DLP) extends LP by adding disjunction in the rule heads, in order to allow more suitable knowledge representation and to increase expressiveness. E.g.,

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

naturally represents that any person is either male or female.

A *disjunctive (general) logic program* is a set of clauses

$$A_1 \vee \cdots \vee A_k \leftarrow L_1, \ldots, L_m \qquad (k \geq 1, m \geq 0). \quad (3)$$

For a background, see [95] and the more recent [105]. The semantics of $\neg$-free disjunctive logic programs is based on *minimal* Herbrand models, as the least (unique minimal) model does not exist in general.

Example 5 $P = \{p \vee q \leftarrow\}$ 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 [104] (GCWA) for negation-free P amounts to the meaning $\mathcal{M}_{GCWA}(P) = \{L \mid \mathrm{MM}(P) \models L\}$.

Example 6 Consider the following 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:

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

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

Theorem 5.1 ([48]) (i) *Deciding* $P \models_{GCWA} A$ *is* co-**NP**-*complete, and* (ii) *deciding* $P \models_{GCWA} \neg A$ *is* $\mathbf{\Pi}_2^p$-*complete.*

Proof. It is easy to see that for an atom A, it holds $P \models_{GCWA} A$ if and only if $P \models_{PC} A$, where $\models_{PC}$ is classical logical consequence. Hence, by the well-known **NP**-completeness of SAT, part (i) is obvious.

Let us thus consider part (ii).

a) *Membership*. It holds $P \not\models_{GCWA} \neg A$, 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 $\mathbf{\Pi}_2^p$ follows.

b) *Hardness*. (Sketch) We show $\mathbf{\Pi}_2^p$-hardness by an encoding of alternating Turing machines (ATM) [30]. 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 (resp. $\forall$-state) s in a run, then the input is accepted if the computation continued in some (resp. all) of the possible successor configurations is accepting.

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 $\mathbf{\Pi}_2^p$ [30].

By adapting the construction in the proof of Theorem 4.6, we show how any such machine T on input I can be simulated by a disjunctive logic program P under GCWA. W.l.o.g., we assume that each run of T is polynomial-time bounded [13].

We start from the clauses constructed for the NTM T on input I in the proof of Theorem 4.6, from which we drop the clause $ACCEPT \leftarrow \neg ACCEPT$ and replace the clauses

$$B_{s,\alpha,i}[\tau] \leftarrow \neg B_{s,\alpha,1}[\tau], \ldots, \neg B_{s,\alpha,i-1}[\tau], \\ \neg B_{s,\alpha,i+1}[\tau], \ldots, \neg B_{s,\alpha,k}[\tau].$$

by the logically equivalent disjunctive clauses

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

Intuitively, in a minimal model precisely one of the atoms $B_{s,\alpha,i}[\tau]$ 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 iff 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, $\tau \le \tau' \le N$, $0 \le \pi \le N$, and $1 \le i \le k$:

$$\begin{aligned} ST_{s'}[\tau'] &\leftarrow NACCEPT, ST_s[\tau], \\ CC_\alpha[\tau', \pi] &\leftarrow NACCEPT, ST_s[\tau], \\ WP[\tau', \pi] &\leftarrow NACCEPT, ST_s[\tau], \\ B_{s,\alpha,i}[\tau'] &\leftarrow NACCEPT, ST_s[\tau]. \end{aligned}$$

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 $\tau' \ge \tau$ are true. This way, nonaccepting runs are tilted. Finally, we set up for each nonaccepting terminal $\exists$-state s the clauses

$$NACCEPT \leftarrow ST_s[\tau], \; 0 < \tau \le N.$$

Intuitively, these clauses state that $NACCEPT$ is true if the run ends in a nonaccepting state.

Let the resulting program be P^+. 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, $P^+ \models_{GCWA} \neg NACCEPT$, 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 using logarithmic workspace. Consequently, deciding $P \models_{GCWA} \neg A$ is $\mathbf{\Pi}_2^p$-hard under logspace reductions. ■

Notice that many problems in the field of nonmonotonic reasoning are $\mathbf{\Pi}_2^p$-complete, e.g. [65, 47, 50].

Stable negation naturally extends to disjunctive logic programs, by adopting that I is a stable model of a disjunctive logic program P iff $I \in \mathrm{MM}(P^I)$ [111]; it subsumes disjunctive stratified semantics. For well-founded semantics, no such natural extension is known. Clearly, P^I is easily computed, and $P^I = P$ if P is negation-free. Thus,

Theorem 5.2 ([49, 54, 55]) *DLP under SMS is $\mathbf{\Pi}_2^p$ complete. Disjunctive datalog under SMS is data complete in $\mathbf{\Pi}_2^p$ and program complete in* co-$\mathbf{NEXPTIME}^{\mathbf{NP}}$.

The latter result was derived by utilizing complexity upgrading techniques as described above in Section 3.3.

In the case with functions, we have:

Theorem 5.3 ([31]) *Full DLP under GCWA is $\mathbf{\Pi}_2^0$-complete.*

Theorem 5.4 ([49]) *Full DLP under SMS is $\mathbf{\Pi}_1^1$-complete.*

Thus, disjunction adds complexity under GCWA and under SMS in finite Herbrand universes (unless co-$\mathbf{NP} = \mathbf{\Pi}_2^p$), but not in infinite ones. This is intuitively explained by the fact that DLP under SMS corresponds to a weak fragment of $\mathbf{\Pi}_2^1$ which can be recursively translated to $\mathbf{\Pi}_1^1$.

Many other semantics for DLP have been analyzed, some having lower complexity than SMS, e.g., the possible model semantics [27, 116] and the causal model semantics [42], and others higher, e.g. the regular model semantics [57]. However, typically they are $\mathbf{\Pi}_2^p$-complete in the propositional case. (cf. [49, 100]).

6. Expressive power of logic programming

The *expressive power* of query languages such as datalog is a topic common to database theory [2] and finite model theory [46] that has attracted much attention by both communities.

By the *expressive power* of a 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 [46]).

In general, a query q defines a mapping $\mathcal{M}_q$ that to each suitable input database D_{in} (over a fixed input schema) assigns a result database $D_{out} = \mathcal{M}_q(D_{in})$ (over a fixed output schema); more logically speaking, a query defines global relations [70]. For reasons of representation independence, a query should, in addition, be *generic*, i.e., invariant under automorphisms. This means that if τ is an automorphism of the input database, permuting elements of the universe, i.e., names of constants, 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 language Q.

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 $Q_1 \subset Q_2$, which means 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 $\mathbf{C}$. In this case, we say that Q *captures* $\mathbf{C}$ and write simply $Q = \mathbf{C}$. 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* [127, 70].

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

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* [77, 90, 78]. An early foundational result in this field was Fagin's Theorem [58] stating that existential second-order logic captures $\mathbf{NP}$. 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 $\mathbf{C}$, one usually shows that each $\mathbf{C}$-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 $\mathbf{NP}$ 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 $\mathbf{NP}$, generating a linear order is not possible. 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 6.1 (a fortiori from [28]) *Datalog*$^+ \subset \mathbf{P}$.

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. ∎

Theorem 6.2 ([109, 67]; implicit in [127, 76]) *On ordered databases, datalog*$^+$ *captures* $\mathbf{P}$.

Proof. (Sketch) By Theorem 4.1, query answering for a fixed datalog$^+$ program is in $\mathbf{P}$. 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 a graph $\langle U, G\rangle$.

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

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

The simulation is akin to the simulation used in the proofs of Theorems 3.2 and 3.5.

Recall the framework of Section 3.1. 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 $int(\mathbf{x}) = \sum_{i=1}^{k} x_i \times n^{k-i}$.

The simulation starts at time point 0, where the worktape of T contains an encoding of the input graph. Recall that in Section 3.1, this was reflected by the initialization facts

$$CC_\alpha[0,\pi] \leftarrow \qquad 0 \leq \pi < |I|, \textit{ where } I_\pi = \alpha.$$

Before translating this rules into appropriate datalog rules, we shall spend a word about how input graphs are usually represented as binary strings. A graph $\langle U, G\rangle$ is encoded as binary string $enc(U, G)$ of length $|U|^2$. If $G(i, j)$ is true for $i, j \in U = [0, n-1]$, then 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 $\langle 0^{k-2}, x, y\rangle$ such that $x, y \in U$. Moreover, the bit-position $int(\langle 0^{k-2}, x, y\rangle)$ of $enc(U, G)$ is 1 iff $G(x, y)$ is true in the input database and 0 otherwise.

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

$$\begin{array}{l} CC_1(0^k, 0^{k-2}, x, y) \leftarrow G(x,y) \\ CC_0(0^k, 0^{k-2}, x, y) \leftarrow \neg G(x,y) \end{array}$$

Intuitively, the first rule says that if $G(x, y)$ is true, then at time point $0 = int(0^k)$, bit number $int(\langle 0^{k-2}, x, y\rangle)$ of the worktape 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. *This is the only rule in the entire datalog$^+$ program using 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 3.1 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 3.5), 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 3.5.

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., $P^+ \cup D \models ACCEPT$ true, iff T accepts $enc(U, G)$.

The generalization to a setting where the simplifying assumptions 1–4 are not made is rather straightforward and is omitted. ■

Let us now state somewhat more succinctly 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 $\varphi = \varphi(\mathbf{x}; S)$ and $x_1, \ldots, x_k$ are free variables in φ; intuitively, it returns the least fixpoint of the operator Γ defined by $\Gamma(S) = \{\mathbf{a} \mid D \models \varphi(\mathbf{a}; S)\}$. See [28, 2, 46] for details.

As shown in [28], FPL expresses a proper subset of the queries in $\mathbf{P}$. Datalog$^+$ relates to FPL as follows.

Theorem 6.3 ([29]) *Datalog$^+$ = $FPL^+(\exists)$, i.e., coincides with the fragment of FPL having negation restricted to database relations and only existential quantifiers.*

Theorem 6.4 ([84]; implicit in [36]) *Stratified datalog $\subset$ FPL.*

The previous theorem is not obvious. In fact, for some time coincidence of the two languages was assumed, based on [29]. The non-recursive fragment of datalog coincides with well-known database query languages.

Theorem 6.5 (cf. [2]) *Non-recursive datalog = relational algebra = SQL = relational calculus.*

Unstratified negation yields higher expressive power.

Theorem 6.6 ([124];[3], using [71]) *Datalog under WFS = FPL; FPL = datalog under INFS.*

As recently shown, the previous result holds also for total WFS (i.e., the well-founded model is always total) [61].

On ordered databases, Theorem 6.2 and the theorems in Section 4 imply

Theorem 6.7 *On ordered databases, stratified datalog, datalog under INFS, and datalog under WFS capture* $\mathbf{P}$.

Syntactical restrictions allow to capture classes within $\mathbf{P}$. 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 6.8 ([67, 128]) *On ordered databases, datalog*$^+(1)$ *captures* $\mathbf{NL}$ *and datalog*$^+(1, d)$ *captures* $\mathbf{L}$.

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

Theorem 6.9 ([119]) *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's Theorem [58], this implies that datalog under SMS is equivalent to the existential fragment of second-order logic over finite structures.

Theorem 6.10 ([3]) *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 [3]: datalog under INFS = datalog on 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.

Finally, we briefly address the expressive power of disjunctive logic programs and full logic programs. In the latter case, the input databases are arbitrary (not necessary recursive) relations on the genuine (infinite) Herbrand universe of the program.

Theorem 6.11 ([54, 55]) *Disjunctive datalog under SMS captures* $\mathbf{\Pi}_2^p$.

Theorem 6.12 ([119, 51]) *Full LP under WFS, full LP under SMS, and full DLP under SMS all express* $\mathbf{\Pi}_1^1$.

For further expressiveness results, see e.g. [119, 114, 115, 57]. In particular, further classes of the polynomial hierarchy can be captured by variants of stable models [115, 114, 57, 23] as well as through modular logic programming [56].

7. 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 LP 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 decision problem: given terms s and t, are they unifiable?

Robinson described in [113] 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. [101, 110]).

Theorem 7.1 ([44, 131, 45]) *The unification problem is* $\mathbf{P}$-*complete under logspace reductions.*

$\mathbf{P}$-hardness of the unification problem was proved by reductions from some versions of the circuit value problem in [44, 131, 45]. (Article [91] stated that unifiability is complete in co-**NL**, however, [44] gives a counterexample to the proof in [91].)

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 main unification algorithms in [12]). Here we mention only Martelli and Montanari's algorithm [102] based on ideas going back to famous Herbrand's work [73]. 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 (thus, $A^{-1}(n)$ grows very slowly).

8. 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. [1]. Most of these formalisms can be expressed in terms of LP with equality [62, 63, 74, 72, 39] and constraint logic programming considered in Section 9.

8.1. 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 LP. Their semantics is defined as a generalization of the semantics of LP so that terms are identified if they are equal in E.

Example 7 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 need not be 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 non-empty 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

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

has solutions in the equational theory of sets, for example $X = 1, Y_1 = \{3,5\}, Y_2 = \{7,4,1\}$.

Note that if we represent sets by lists in plain LP 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.

8.2. Complexity of E-unification

It is practically impossible to overview all results on the complexity of E-unification because any result on solving equation systems can be viewed as a result on E-unification (solving equations is a traditional subject of all mathematics). Therefore, we restrict this survey to only few cases closely connected with LP. The general theory of E-unification may be found e.g. in [12].

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 8.1 ([96, 11, 17, 86]) *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's algorithm for word equations [96]. The 3-**NEXPTIME** upper bound is obtained in [86].

The following theorem characterizes other popular kinds of equational theories.

Theorem 8.2 ([82, 83]) *Let E be an equational theory defining some symbols as AC-symbols or ACI-symbols or AC1-symbol or ACI1-symbols (there can be one or more of these kinds of symbols). The theory E is assumed to contain no other equations. Then the E-unification problem is* **NP***-complete.*

8.3. Complexity of non-recursive 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 systems is undecidable [106].

Simultaneous E-unification determines decidability of non-recursive LP over E.

Theorem 8.3 ([38]) *Let E be an equational theory. Non-recursive LP 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 [38].

Theorem 8.4 ([37, 38]) *Non-recursive LP 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 non-recursive LP over E is also* **NEXPTIME**-*complete.*

9. Constraint logic programming

Informally, constraint logic programming (CLP) extends LP 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 8 We illustrate CLP by the standard example. Suppose that we would like to solve the following puzzle:

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

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.

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

The query $\mathit{find}(S, E, N, D,\ \ M, O, R, E,\ \ M, O, N, E, Y)$ 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 $=$ in the set N of natural numbers. This structure is denoted by $(N, =, s, 0)$. 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 $n \cdot x$ to denote unary functions of multiplication by particular numbers (of the corresponding domain); x^n 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. Semantics of CLP is defined as a natural generalization of semantics of LP (e.g. [79]). 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 LP. Otherwise, CLP over S can be regarded as an extension of Datalog by constraints.

9.1. Complexity of constraint logic programming

There are two sources of complexity in CLP: complexity of solving systems of constraints and complexity coming from the LP 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 **DEXPTIME**-complete) with linear arithmetic constraints (whose satisfiability problem is in **NP** for integers and in **P** for rational numbers and reals) is undecidable.

Theorem 9.1 ([35]) *CLP over $(N, =, s, 0)$ is r.e.-complete. The same holds for any of Z, Q, R and C instead of N.*

The proof uses the fact that CLP over $(N, =, S, 0, 1)$ 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 9.2 ([34]) *CLP over $(Z, =, <, 0, \pm 1, \pm 2, \ldots)$ is* **DEXPTIME**-*complete. The same holds for any of 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 9.3 ([35]) *Conservative CLP is* **DEXPTIME**-*complete over any of the following structures:*

> $(Q, =, \leq, <, +, -, n \cdot x, 0, 1, \ldots)$, *i.e. linear inequations over the rational numbers;*
>
> $(R, =, \leq, <, +, -, n \cdot x, 0, 1, \ldots)$, *i.e. linear inequations over the reals;*
>
> $(R, =, \leq, <, +, -, \times, /, x^n, 0, 1, \ldots)$, *i.e. polynomial inequations over the reals;*
>
> $(C, =, +, -, \times, /, x^n, 0, 1, \ldots)$, *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 [25, 112, 69, 75]. If we allow non-ground queries, **DEXPTIME**-completeness should be replaced by **NEXPTIME**-completeness.

10. 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. To extend the notion of expressive power to logic programming with complex values, we have 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. The comparative expressive power of such formalisms is studied, for example, in [1]. This paper introduces 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.

The absolute expressive power of such languages (in terms of complexity classes) is studied for example in [117, 92, 93] which, in particular, show how the expressive power depends on the way of representing complex values. For a natural representation of hereditarily finite sets by graphs, there is a logical query language (called Bounded Set Theory) that captures **P**. Some other versions of Bounded Set Theory are shown to capture **L** and **NL**.

Other interesting results on the expressive power of different forms of LP with constraints can be found e.g. in [33, 81, 18, 126].

Unlike research on the expressive power of datalog, there is no mainstream in research on the expressive power of LP with complex values. The latter research yielded so far a number of ad hoc results and approaches. This can be explained by several reasons. One reason is that different kinds of complex values require different computational models. Another reason is that the same kind of complex values admits many different definitions of the input and output.

Extension of declarative query languages by complex values is one of the main problems of database theory and practice. More research is required to develop unifying paradigms for understanding their expressive power.

References

[1] S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *VLDB J.*, 4:727–794, 1995.

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

[3] S. Abiteboul and V. Vianu. Datalog Extensions for Database Updates and Queries. *J. Computer and System Sciences*, 43:62–124, 1991.

[4] S. Abiteboul and V. Vianu. Computing with First-Order Logic. *J. Computer and System Sciences*, 50:309–335, 1995. Preliminary version in STOC 1991.

[5] Andréka and Németi. A generalized completeness of Horn clause logic seen as a programming language. *Acta Cybernetica*, 4:3–10, 1978.

[6] K. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10, pp. 493–574. Elsevier Science, 1990.

[7] K. Apt and H. Blair. Arithmetic Classification of Perfect Models of Stratified Programs. In R. Kowalski and K. Bouwen, editors, *Proc. Fifth JICSLP-88*, pp. 766–779. MIT Press, 1988.

[8] K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Morgan Kaufman, Washington DC, 1988.

[9] K. Apt and R. Bol. Logic Programming and Negation: A Survey. *J. Logic Programming*, 19/20:9–71, 1994.

[10] K. Apt and M. van Emden. Contributions to the Theory of Logic Programming. *JACM*, 29(3):841–862, 1982.

[11] F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In D. Kapur, editor, *11th CADE*, LNCS/LNAI 607, pp. 50–65, 1992.

[12] F. Baader and J. Siekmann. Unification theory. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, 1994.

[13] J. Balcázar, J. Diaz, and J. Gabarró. *Structural Complexity I+II*. Springer, 1988 + 1990.

[14] J. Balcázar, A. Lozano, and J. Torán. The Complexity of Algorithmic Problems on Succinct Instances. In R. Baeta-Yates and U. Manber, editors, *Computer Science*, pp. 351–377. Plenum Press, New York, 1992.

[15] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *J. Logic Programming*, 19/20:73–148, 1994.

[16] C. Baral and V. Subrahmanian. Dualities Between Alternative Semantics for Logic Programming and Nonmonotonic Reasoning. *J. Automated Reasoning*, 10:399–420, 1993.

[17] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *J. Symbolic Computation*, 3:203–216, 1987.

[18] M. Benedikt, G. Dong, L. Libkin, and L. Wong. Expressive Power of Relational Constraint Query Languages. In *Proc. PODS-96*, pp. 5–13, 1996.

[19] K. Berman, J. Schlipf, and J. Franco. Computing Well-Founded Semantics Faster. In W. Marek, A. Nerode, and M. Truszczyński, editors, *Proc. LPNMR-95*, LNCS/LNAI 982, pp. 113–126. Springer, 1995.

[20] H. Blair. The Recursion-Theoretic Complexity of the Semantics of Predicate Logic as a Programming Language. *Information and Control*, 54(1/2):25–47, July/August 1982.

[21] H. Blair and C. Cholak. The Complexity of Local Stratification. *Fundamenta Informaticae*, 21:333–344, 1994.

[22] A. Bonner. Hypothetical Datalog: Complexity and Expressibility. *Theoretical Computer Science*, 76:3–51, 1990.

[23] F. Buccafurri, S. Greco, and D. Saccá. The Expressive Power of Unique Total Stable Model Semantics. In *Proc. ICALP 97*, 1997. To appear.

[24] M. Cadoli and M. Schaerf. A Survey of Complexity Results for Non-monotonic Logics. *J. Logic Programming*, 17:127–160, 1993.

[25] J. Canny. Some algebraic and geometric computations in PSPACE. In *Proc. 20th Annual ACM STOC*, pp. 460–467, Chicago, Illinois, 1988.

[26] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990.

[27] E. Chan. A Possible Worlds Semantics for Disjunctive Databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):282–292, 1993.

[28] A. Chandra and D. Harel. Structure and Complexity of Relational Queries. *J. Computer and System Sciences*, 25:99–128, 1982.

[29] A. Chandra and D. Harel. Horn Clause Queries and Generalizations. *J. Logic Programming*, 2:1–15, 1985.

[30] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *JACM*, 28:114–133, 1981.

[31] J. Chomicki and V. Subrahmanian. Generalized Closed World Assumption is Π^0_2-Complete. *Information Processing Letters*, 34:289–291, 1990.

[32] A. Colmerauer, H. Kanoui, P. Roussel, and R. Passero. Un systeme de communication homme-machine en Francais. Technical report, Groupe de Recherche en Intelligence Artificielle, Universite d' Aix-Marseille, 1973.

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

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

[35] J. Cox, K. McAloon, and C. Tretkoff. Computational complexity and constraint logic programming languages. extended abstract. In S. Debray and M. Hermenegildo, editors, *Proc. NACLP'90*, pp. 401–415. The MIT Press, 1990.

[36] E. Dahlhaus. Skolem Normal Forms Concerning the Least Fixpoint. In E. Börger, editor, *Computation Theory and Logic*, LNCS 270, pp. 101–106. Springer, 1987.

[37] E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex data. In S. Adian and A. Nerode, editors, *LFCS'97*, LNCS, 1997. To appear.

[38] E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex data. UPMAIL technical report, Uppsala University, Computing Science Department, 1997.

[39] A. Degtyarev and A. Voronkov. A note on semantics of logics programs with equality based on complete sets of E-unifiers. *J. Logic Programming*, 28(3):207–216, Sept. 1996.

[40] P. Devienne, P. Lebegue, J.-C. Routier, and J. Würtz. Smallest Horn Clause Programs. *J. Logic Programming*, 27:227–267, 1996.

[41] A. Dikovsky. On Computational Complexity of Prolog Programs. *Theoretical Computer Science*, 119:63–102, 1993.

[42] J. Dix, G. Gottlob, and W. Marek. Reducing Disjunctive to Non-Disjunctive Semantics by Shift-Operations. *Fundamenta Informaticae*, 28:87–100, 1996.

[43] W. Dowling and J. H. Gallier. Linear-time Algorithms for Testing the Satisfiability of Propositional Horn Theories. *J. Logic Programming*, 3:267–284, 1984.

[44] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *J. Logic Programming*, 1:35–50, 1984.

[45] C. Dwork, P. Kanellakis, and L. Stockmeyer. Parallel algorithms for term matching. *SIAM J. Computing*, 17(4):711–731, 1988.

[46] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Perspectives in Mathematical Logic. Springer, 1995.

[47] T. Eiter and G. Gottlob. On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals. *Artificial Intelligence*, 57(2–3):227–270, 1992.

[48] T. Eiter and G. Gottlob. Propositional Circumscription and Extended Closed World Reasoning are $\mathbf{\Pi}_2^P$-complete. *Theoretical Computer Science*, 114(2):231–245, 1993. Addendum 118:315.

[49] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.

[50] T. Eiter and G. Gottlob. The Complexity of Logic-Based Abduction. *JACM*, 42(1):3–42, January 1995.

[51] T. Eiter and G. Gottlob. Expressiveness of Stable Model Semantics for Disjunctive Logic Programs with Functions. Technical Report CD-TR 96/103, Christian Doppler Laboratory for Expert Systems, TU Vienna, Austria, July 1996. *J. Logic Programming*, to appear.

[52] T. Eiter, G. Gottlob, and N. Leone. Complexity Results for Abductive Logic Programming. In W. Marek, A. Nerode, and M. Truszczyński, editors, *Proc. LPNMR-95*, LNCS/LNAI 928, pp. 1–14. Springer, 1995.

[53] T. Eiter, G. Gottlob, and N. Leone. Abduction From Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 1997. to appear.

[54] T. Eiter, G. Gottlob, and H. Mannila. Adding Disjunction to Datalog. In *Proc. 13th ACM Symposium on Principles of Database Systems (PODS-94)*, pp. 267–278, May 1994.

[55] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Trans. on Database Syst.*, September 1997. To appear.

[56] T. Eiter, G. Gottlob, and H. Veith. Modular Logic Programming and Generalized Quantifiers. In *Proc. LPNMR-97*, 1997. To appear. Extended paper CD-TR 97/111, Information Systems Department, TU Vienna, 1997.

[57] T. Eiter, N. Leone, and D. Sacca. Expressive Power of Partial Models for Disjunctive Deductive Databases. In *Proc. International Workshop on Logic in Databases (LID '96)*, LNCS 1154, pp. 245–264. Springer, 1996. *TCS*, to appear.

[58] R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In R. M. Karp, editor, *Complexity of Computation*, pp. 43–74. AMS, 1974.

[59] M. Falashi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[60] M. Fitting. *Computability Theory, Semantics, and Logic Programming*. Oxford University Press, 1987.

[61] J. Flum, M. Kubierschky, and B. Ludäscher. Total and Partial Well-founded Datalog Coincide. In F. Afrati and P. Kolaitis, editors, *Proc. 6th Intl. Conference on Database Theory (ICDT '97)*, LNCS 1186, pp. 113–124, January 1997.

[62] J. Gallier and S. Raatz. Extending SLD-resolution methods for Horn clauses with equality based on E-unification. In *Symposium on Logic Programming*, pp. 168–179, 1986.

[63] J. Gallier and S. Raatz. Extending SLD-resolution to equational Horn clauses using E-unification. *J. Logic Programming*, 6(3):3–44, 1989.

[64] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proc. Fifth Intl Conference and Symposium*, pp. 1070–1080, Cambridge, Mass., 1988. MIT Press.

[65] G. Gottlob. Complexity Results for Nonmonotonic Logics. *J. Logic and Computation*, 2(3):397–425, June 1992.

[66] G. Gottlob, N. Leone, and H. Veith. Second-Order Logic and the Weak Exponential Hierarchies. In J. Wiedermann and P. Hajek, editors, *Proc. 20th MFCS*, LNCS 969, pp. 66–81, Prague, 1995. Full paper available as CD-TR 95/80, Information Systems Department, TU Wien.

[67] E. Grädel. Capturing Complexity Classes with Fragments of Second Order Logic. *Theoretical Computer Science*, 101:35–57, 1992.

[68] C. Green. *The Application of Theorem Proving to Question-Answering Systems*. PhD thesis, Computer Science Department, Stanford University, June 1969.

[69] D. Grigoryev and N. J. Vorobjov. Solving systems of polynomial inequalities in subexponential time. *J. Symbolic Computation*, 5(1,2):37–64, 1988.

[70] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Trends in Theoretical Computer Science*, chapter 1. Computer Science Press, 1988.

[71] Y. Gurevich and S. Shelah. Fixpoint Extensions of First-Order Logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.

[72] M. Hanus. The integration of functions into logic programming: from theory to practice. *J. Logic Programming*, 19,20:583–628, 1994.

[73] J. Herbrand. *Logical Writings*. Harvard University Press, 1972.

[74] S. Hölldobler. *Foundations of Equational Logic Programming*, LNCS/LNAI 353. Springer Verlag, 1989.

[75] D. Ierardi. Quantifier elimination in the theory of an algebraically-closed field. In *Proc. 21st Annual ACM STOC*, pp. 138–147, Seattle, Washington, May 1989.

[76] N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86–104, 1986.

[77] N. Immerman. Languages that Capture Complexity Classes. *SIAM J. Comp.*, 16:760–778, 1987.

[78] N. Immerman. *Descriptive Complexity*. Springer, 1997. To appear.

[79] J. Jaffar and M. Maher. Constraint logic programming: a survey. *J. Logic Programming*, 19,20:503–581, 1994.

[80] N. Jones and W. Laaser. Complete Problems in Deterministic Polynomial Time. *Theoretical Computer Science*, 3:105–117, 1977.

[81] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *J. Computer and System Sciences*, 51:26–52, 1995.

[82] D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In J. Siekmann, editor, *Proc. 8th CADE*, LNCS 230, pp. 489–495, 1986.

[83] D. Kapur and P. Narendran. Complexity of unification problems with associative-commutative operators. *J. Automated Reasoning*, 9(2):261–288, 1992.

[84] P. Kolaitis. The Expressive Power of Stratified Logic Programs. *Information and Computation*, 90:50–66, 1991.

[85] P. Kolaitis and C. H. Papadimitriou. Why Not Negation By Fixpoint ? *J. Computer and System Sciences*, 43:125–144, 1991.

[86] A. Kościelski and L. Pacholski. Complexity of Makanin's algorithm. *JACM*, 43(4):670–684, 1996.

[87] R. Kowalski. Predicate Logic as a Programming Language. In *Information Processing '74*, pp. 569–574. North-Holland, Amsterdam, 1974.

[88] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[89] A. Leitsch. *The Resolution Calculus*. Springer, 1997.

[90] D. Leivant. Descriptive Characterizations of Computational Complexity. *J. Computer and System Sciences*, 39:51–83, 1989.

[91] H. Lewis and R. Statman. Unifiability is complete for co-NLOGSPACE. *Information Processing Letters*, 15:220–223, 1982.

[92] A. Lisitsa and V. Sazonov. Delta-languages for sets and sub-PTIME graph transformers. In G. Gottlob and M. Vardi, editors, *Data Base Theory — ICDT'95*, LNCS 893, pp. 125–138, Prague, 1995. Springer Verlag.

[93] A. Lisitsa and V. Sazonov. Delta-languages for sets and LOGSPACE computable graph transformers. *Theoretical Computer Science*, 175(1):183–222, 1997.

[94] J. Lloyd. *Foundations of Logic Programming (2nd edition)*. Springer Verlag, 1987.

[95] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT Press, 1992.

[96] G. Makanin. The problem of solvability of equations in free semigroups. *Mat. Sbornik (in Russian)*, 103(2):147–236, 1977. English Translation in American Mathematical Soc. Translations (2), vol. 117, 1981.

[97] W. Marek, A. Nerode, and J. Remmel. How Complicated is the Set of Stable Models of a Recursive Logic Program? *Annals of Pure and Applied Logic*, 56:119–135, 1992.

[98] W. Marek, A. Nerode, and J. Remmel. The Stable Models of a Predicate Logic Program. *J. Logic Programming*, 21(3), 1994.

[99] W. Marek and M. Truszczyński. Autoepistemic Logic. *JACM*, 38(3):588–619, 1991.

[100] W. Marek, M. Truszczyński, and A. Rajasekar. Complexity of Extended Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 15(3/4), 1995.

[101] A. Martelli and U. Montanari. Unification in linear time and space: a structured presentation. Technical Report B 76-16, University of Pisa, 1976.

[102] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[103] Y. Matiyasevic. The diophantiness of recursively enumerable sets (in Russian). *Soviet Mathematical Doklady*, pp. 279–282, 1970.

[104] J. Minker. On Indefinite Data Bases and the Closed World Assumption. In D. Loveland, editor, *Proc. 6th CADE*, LNCS 138, pp. 292–308, New York, 1982. Springer.

[105] J. Minker. Overview of Disjunctive Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994.

[106] P. Narendran and F. Otto. Some results on equational unification. In M. Stickel, editor, *Proc. 10th CADE*, volume 449 of *LNAI*, pp. 276–291, 1990.

[107] L. Palopoli. Testing Logic Programs for Local Stratification. *Theoretical Computer Science*, 103:205–234, 1992.

[108] C. Papadimitriou and M. Yannakakis. A Note on Succinct Representations of Graphs. *Information and Computation*, 71:181–185, 1985.

[109] C. H. Papadimitriou. A Note on the Expressive Power of Datalog. *Bulletin of the EATCS*, 26:21–23, 1985.

[110] M. Paterson and M. Wegman. Linear unification. *J. Computer and System Sciences*, 16:158–167, 1978.

[111] T. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[112] J. Renegar. A faster PSPACE algorithm for deciding the existential theory of the reals. In *Proc. 29th IEEE FOCS*, pp. 291–295, White Plains, New York, Oct. 1988. IEEE.

[113] J. Robinson. A Machine-Oriented Logic Based on the Resolution-Principle. *JACM*, 12(1):23–41, 1965.

[114] D. Saccá. The Expressive Powers of Stable Models for Bound and Unbound DATALOG Queries. *J. Computer and System Sciences*. To appear.

[115] D. Saccá. Deterministic and Nondeterministic Stable Model Semantics for Unbound DATALOG Queries. In *Proc. 5th Intl. Conference on Database Theory (ICDT-95)*, LNCS 893, pp. 353–367, January 1995.

[116] C. Sakama and K. Inoue. An Alternative Approach to the Semantics of Disjunctive Logic Programs and Deductive Databases. *J. Automated Reasoning*, 13:145–172, 1994.

[117] V. Sazonov. Hereditarily-finite sets, data bases and polynomial-time computability. *Theoretical Computer Science*, 119:187–214, 1993.

[118] J. Schlipf. Complexity and Undecidability Results in Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):257–288, 1995.

[119] J. Schlipf. The Expressive Powers of Logic Programming Semantics. *J. Computer and System Sciences*, 51(1):64–86, 1995. Abstract in Proc. PODS 90, pp. 196–204.

[120] E. Shapiro. Alternation and the Computational Complexity of Logic Programs. *J. Logic Programming*, 1:19–33, 1984.

[121] S.-A. Tärnlund. Horn clause computability. *BIT*, 17:215–216, 1977.

[122] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[123] M. H. van Emden and R. Kowalski. The Semantics of Predicate Logic as a Programming Language. *JACM*, 23:733–742, 1976.

[124] A. Van Gelder. The Alternating Fixpoint of Logic Programs With Negation. In *Proc. PODS-89*, pp. 1–10, 1989.

[125] A. van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *JACM*, 38(3):620–650, 1991.

[126] L. Vandeurzen, M. Gyssens, and D. Van Gucht. Expressive Power of Relational Constraint Query Languages. In *Proc. 2nd Intl. Conference on Principles and Practice of Constraint Programming*, LNCS 1118, pp. 468–481, 1996.

[127] M. Vardi. Complexity of relational query languages. In *Proc. 14th STOC*, pp. 137–146, San Francisco, 1982.

[128] H. Veith. Logical Reducibilities in Finite Model Theory. Master's thesis, Information Systems Department, TU Vienna, Austria, September 1994.

[129] H. Veith. Succinct Representation and Leaf Languages. In *11th IEEE Conf. on Comput. Complexity*, pp. 118–126, Philadelphia, PA, May 1996. Full version in *Electronic Colloquium on Computational Complexity, Report TR95-048, also CD-TR 95-81*.

[130] A. Voronkov. On computability by logic programs. *Annals of Mathematics and Artificial Intelligence*, 15(3,4):437–456, 1995.

[131] H. Yasuura. On parallel computational complexity of unification. In *Proc. Conference on Fifth Generation Computer Systems*, pp. 235–243. ICOT, 1984.