A nondeterministic polynomial-time unification algorithm for bags, sets and trees

Evgeny Dantsin[1] and Andrei Voronkov[2]

[1] Steklov Institute of Mathematics at St.Petersburg
[2] Computing Science Department, Uppsala University

Abstract. Unification in logic programming deals with tree-like data represented by terms. Some applications, including deductive databases, require handling more complex values, for example finite sets or bags (finite multisets). We extend unification to the combined domain of bags, sets and trees in which bags and sets are generated by constructors similar to the list constructor. Our unification algorithm is presented as a nondeterministic polynomial-time algorithm that solves equality constraints in the spirit of the Martelli and Montanari algorithm.

1 Introduction

Logic programming languages deal with tree-like data represented by terms. Some applications require to handle other kinds of data such as finite sets or bags (finite multisets). For example, this problem arises in databases: relational query languages typically deal with tuples of atomic values and extension to more complex values is required. Various kinds of complex values in databases and logic programming have been considered in many papers, including [2, 3, 31, 16, 46, 43, 32, 15, 1, 27, 26, 22, 20, 48].

In this paper we extend unification, the core mechanism of logic programming, to handle bags and sets. When bags and sets are represented with using the union operations, bag and set unification is a particular case of AC1- and ACI1-unification, which has been extensively studied [44, 25, 24, 10, 33, 18]. Recently, a number of unification algorithms have been introduced for various domains of bags and sets built with the bag and set constructors similar to the list constructor used in functional and logic programming [21, 22, 45, 9].

We contribute to this area by introducing a new unification algorithm for the *combined* domain of bags, sets and trees. The main novelty of our algorithm is that it is a *nondeterministic polynomial-time* algorithm. The algorithm is formalized as a nondeterministic algorithm that solves systems of equations in the spirit of the Martelli and Montanari unification algorithm [38], i.e. it uses a collection of rules that transform systems into equivalent ones. The algorithm is don't-care nondeterministic with respect to the choice of applicable rules. Nondeterministic branches lead to unifiers represented by systems in solved form (triangle form). These unifiers form a complete set of unifiers of the input system, its cardinality is at most $2^{O(n \log n)}$ where n is the input size. Thus, we obtain a single-exponential upper bound on the cardinality of a minimal complete set of unifiers in our domain (cf. a double-exponential lower bound for AC unification in the domain with the bag union [30]).

The paper is organized as follows. In Section 2 we extend the Herbrand universe by adding terms that represent bags and sets. Thus, the extended domain (denoted by $\mathcal{HU}^+$) contains bags, sets and trees. Values in this domain are untyped, for example one can construct a bag whose members are sets and trees. The semantics of logic programs over $\mathcal{HU}^+$ is defined in Section 2 too. We show that logic programming over $\mathcal{HU}^+$ is powerful enough to represent all computable predicates on this domain.

Section 3 is the main section of this paper. It contains a description of our algorithm and theorems asserting that the algorithm is sound, complete and runs in nondeterministic polynomial time. Since NP-hardness is proved easily [28], we obtain that unification over $\mathcal{HU}^+$ is NP-complete. It follows from these theorems that the algorithm yields complete sets of unifiers and their cardinalities are at most $2^{O(n \log n)}$. This bound is tight. In this section we also describe a number of important special cases in which our algorithm is optimal, i.e. it gives minimal complete sets of unifiers. Due to space reasons, we do not include proofs in this paper, they can be found in our technical report [19]. Also, this report contains many examples that illustrate the algorithm as well as some related notions.

In Section 4 we briefly sketch some related results and directions of further research. In particular, we discuss bag and set unification in the context of AC and ACI unification. We also compare our algorithm with other known algorithms. Some extensions and applications of our results are discussed too.

There are several aspects of handling complex values that are beyond the scope of this paper. We do not consider the introduction of the object structure on $\mathcal{HU}^+$ and we do not discuss questions like object identity. We do not consider semantics of negation. Our algorithm can be optimized in some ways but we do not discuss such optimizations here.

2 The combined domain of bags, sets and trees

There are several possibilities to define data models that deal with bags, finite sets and trees. In this section we choose a particular data model, some variations are considered in [19]. For brevity we say "set" instead of "finite set" in the context of this data model. "Bag" is a synonym for "finite multiset".

The Herbrand universe with bags and sets. We extend the Herbrand universe by adding the bag and set constructors. The definition is parametrized by a set $\mathcal{F}$ of *function symbols.* As usual, symbols in $\mathcal{F}$ have non-negative arities. *Constants* are function symbols of arity 0. Intuitively, constants represent atomic values, like integers or strings, Function symbols of arity ≥ 1 are viewed as *tree constructors* that are used to construct complex values from existing ones.

More precisely, given a set $\mathcal{F}$ of function symbols, the *Herbrand universe with bags and sets*, denoted $\mathcal{HU}^+$, is defined inductively as follows.

1. Any constant in $\mathcal{F}$ belongs to $\mathcal{HU}^+$. These constants are called *atomic values*.
2. If $v_1, \dots, v_n \in \mathcal{HU}^+$, where $n \geq 0$, then the bag consisting of $v_1, \dots, v_n$ belongs to $\mathcal{HU}^+$. This bag is denoted by $\{\!|v_1, \dots, v_n|\!\}$ and called a *bag value*.
3. If $v_1, \dots, v_n \in \mathcal{HU}^+$, where $n \geq 0$, then the set $\{v_1, \dots, v_n\}$ belongs to $\mathcal{HU}^+$. This set is called a *set value*.
4. If $f \in \mathcal{F}$ has arity $n \geq 1$ and $v_1, \dots, v_n \in \mathcal{HU}^+$, then the expression $f(v_1, \dots, v_n)$ belongs to $\mathcal{HU}^+$. This expression represents the tree whose root has n children; the root is labeled by f and the children are $v_1, \dots, v_n$. The expression $f(v_1, \dots, v_n)$ is called a *tree value*.

Thus, $\mathcal{HU}^+$ consists of untyped values. For example, the set $\{\{\!|1, 1|\!\}, \{2\}, 3, f(4)\}$ contains a bag, a set, an atomic value and a tree.

Equality on $\mathcal{HU}^+$ is defined inductively as follows.

1. Two atomic values are equal if they coincide.
2. Bag values $\{\!|u_1, \dots, u_m|\!\}$ and $\{\!|v_1, \dots, v_n|\!\}$ are equal if $m = n$ and there is a permutation ρ of the sequence $1, \dots, m$ such that u_i is equal to $v_{\rho(i)}$ for all i.
3. Set values $\{u_1, \dots, u_m\}$ and $\{v_1, \dots, v_n\}$ are equal if each u_i is equal to some v_j, and vice versa, each v_j is equal to some u_i.
4. Tree values $f(u_1, \dots, u_m)$ and $g(v_1, \dots, v_n)$ are equal if f coincides with g and u_i is equal to v_i for all i.
5. No other equalities hold on $\mathcal{HU}^+$.

Terms and their sorts. In order to represent elements of $\mathcal{HU}^+$, we introduce the corresponding notion of a term. We assume that $\{\!||\!\}$ and $\{\}$ are two constants foreign to $\mathcal{F}$. They represent the empty bag and the empty set respectively. Two binary function symbols $\{\!|\,|\,|\!\}$ and $\{\,|\,\}$ are assumed to be foreign to $\mathcal{F}$ too. They are used to construct bags and sets. Namely, if a term s represents an arbitrary element of $\mathcal{HU}^+$ and a term t represents a bag then the term $\{\!|s\,|\,t|\!\}$ represents the bag formed by appending the element corresponding to s to the bag corresponding to t. Similarly, a term $\{s\,|\,t\}$ represents the set formed by adding the element represented by s to the set represented by t.

All terms will be divided into three sorts: sort of bags $\mathfrak{b}$, sort of sets $\mathfrak{s}$ and the universal sort $\mathfrak{u}$. We assume that there are three kinds of variables called *bag variables*, *set variables* and *universal variables*. Terms and their sorts are defined as follows.

1. A bag variable is a term of sort $\mathfrak{b}$. A set variable is a term of sort $\mathfrak{s}$. A universal variable is a term of sort $\mathfrak{u}$.
2. The constant $\{\!||\!\}$ is a term of sort $\mathfrak{b}$. The constant $\{\}$ is a term of sort $\mathfrak{s}$. Any constant in $\mathcal{F}$ is a term of sort $\mathfrak{u}$.
3. If s is an arbitrary term and t is a term of sort $\mathfrak{b}$ then $\{\!|s\,|\,t|\!\}$ is a term of sort $\mathfrak{b}$. If s is an arbitrary term and t is a term of sort $\mathfrak{s}$ then $\{s\,|\,t\}$ is a term of sort $\mathfrak{s}$. Any expression of the form $f(t_1, \dots, t_n)$, where $f \in \mathcal{F}$ and $t_1, \dots, t_n$ are arbitrary terms, is a term of sort $\mathfrak{u}$.
4. No other terms can be formed.

We also introduce the partial order $\sqsubseteq$ on the terms. For any terms s, t, we write $s \sqsubseteq t$ if s and t have the same sort or t is of sort $\mathfrak{u}$.

Equations and solutions. A mapping ν from the set of all variables to $\mathcal{HU}^+$ is called a *valuation* if ν maps every bag variable to a bag value and every set variable to a set value. Universal variables can be mapped to any values. We extend valuations from variables to terms as follows. Let ν be a valuation and r be a non-variable term. The value $\nu(r)$ is defined inductively:

1. $\nu(\{\!|\,|\!\})$ is the bag value $\{\!|\,|\!\}$. If r is $\{\!| s \mid t |\!\}$ and $\nu(t)$ is a bag value $\{\!| v_1, \ldots, v_n |\!\}$, then $\nu(r)$ is the bag value $\{\!| \nu(s), v_1, \ldots, v_n |\!\}$.
2. $\nu(\{\})$ is the set value $\{\}$. If r is $\{s \mid t\}$ and $\nu(t)$ is a set value $\{v_1, \ldots, v_n\}$, then $\nu(r)$ is the set value $\{\nu(s), v_1, \ldots, v_n\}$.
3. If r is $f(v_1, \ldots, v_n)$, where $n \geq 0$, then $\nu(r)$ is $f(\nu(v_1), \ldots, \nu(v_n))$.

By this definition, the value $\nu(r)$ for a ground term r remains the same for all ν. Hence, elements of $\mathcal{HU}^+$ can be alternatively defined as equivalence classes of ground terms by the equality relation.

By an *equation* we mean any expression $s = t$, where s, t are terms. A *solution* to an equation is a valuation ν such that $\nu(s)$ is equal to $\nu(t)$. A finite set of equations is also called a *system (of equations)* or an *equality constraint*. A valuation ν is a *solution to a system* if ν is a solution to every equation in the system. We fix some system that has no solution and denote this system by $\perp$.

Logic programs over $\mathcal{HU}^+$ and their semantics. The notion of a *Horn clause* is defined as usual. We assume that atoms in Horn clauses are not equalities. A *logic program over $\mathcal{HU}^+$* is a finite set of Horn clauses. A *Herbrand model of a logic program L over $\mathcal{HU}^+$* is any model $\mathfrak{M}$ of L such that (i) the carrier set of $\mathfrak{M}$ is $\mathcal{HU}^+$ and (ii) each ground term is interpreted in $\mathfrak{M}$ by itself. As usual, a Herbrand model can be identified with the set of ground atoms true in this model. Thus we can redefine a Herbrand model $\mathfrak{M}$ as a set of ground non-equality atoms (meaning the set of ground non-equality atoms true in $\mathfrak{M}$).

For two Herbrand models $\mathfrak{M}_1$ and $\mathfrak{M}_2$, we write $\mathfrak{M}_1 \subseteq \mathfrak{M}_2$ if the set of all ground atoms true in $\mathfrak{M}_1$ is a subset of all ground atoms true in $\mathfrak{M}_2$. It is not difficult to prove that any logic program over $\mathcal{HU}^+$ has the least Herbrand model (with respect to $\subseteq$). This statement is basically a straightforward generalization of the standard facts of logic programming theory [35, 8].

We say that a logic program L over $\mathcal{HU}^+$ *defines a relation R on $\mathcal{HU}^+$* if for some predicate P in L, the predicate P is interpreted as R in the least Herbrand model of L.

There are several ways of defining a procedural semantics of logic programs over $\mathcal{HU}^+$. The standard way is to modify SLD-resolution, namely, instead of SLD-resolution using substitutions, we can use *constraint SLD-resolution* defined similar to [36, 37]. In particular, constraint SLD-resolution over $\mathcal{HU}^+$ uses unification over $\mathcal{HU}^+$, i.e. solving equality constraints over $\mathcal{HU}^+$.

Logic programs and computability on $\mathcal{HU}^+$. Since elements of $\mathcal{HU}^+$ can be obviously represented as strings of symbols, we can speak of computable (aka recursively enumerable) relations on $\mathcal{HU}^+$. The following theorem shows that our constructors are enough to represent any computable relation on $\mathcal{HU}^+$.

Theorem 1. *A relation R on $\mathcal{HU}^+$ is recursively enumerable if and only if there exists a logic program L that defines R.*

The proof can be easily obtained by adapting the proof of [49] or other similar proofs (e.g., [6]).

3 Unification algorithm

In this section we introduce a unification algorithm for $\mathcal{HU}^+$. The algorithm is presented as a nondeterministic algorithm that transforms an input system of equations into an output system in *solved form* (defined below). The work of the algorithm is based on repeated applications of *transformation rules*. The output systems (all nondeterministic results) are equivalent to the input systems in the sense described in Theorem 3.

3.1 Some definitions and notation

Notation for bags and sets. Like the logic programming notation for lists, we write $\{\!|s_1,\dots,s_n\,|\,t|\!\}$ and $\{\!|s_1,\dots,s_n|\!\}$ for terms $\{\!|s_1\,|\,\dots\{\!|s_n\,|\,t|\!\}\dots|\!\}$ and $\{\!|s_1\,|\,\dots\{\!|s_n\,|\,\{\!||\!\}|\!\}\dots|\!\}$ respectively, and similar for sets. Letters x,y,z,u,v,w with or without indices are used to denote variables. Capital letters $X,Y,\dots$ (possibly with indices) stand for sequences of variables. For example, if X and Y stand for $x_1,\dots,x_k$ and $y_1,\dots,y_m$ respectively then $\{X,Y\,|\,u\}$ and $\{X,Y\}$ denote $\{x_1,\dots,x_k,y_1,\dots,y_m\,|\,u\}$ and $\{x_1,\dots,x_k,y_1,\dots,y_m\}$. When we use notation $\{\!|X\,|\,s|\!\}$ and $\{X\,|\,s\}$, where s is a variable, we assume that X is non-empty. The *length* of a sequence X, i.e. the number of its members, is denoted by $|X|$. If X and Y are $x_1,\dots,x_n$ and $y_1,\dots,y_n$ respectively, $X=Y$ stands for $x_1=y_1,\ \dots,\ x_n=y_n$. Sometimes, we shall use capital letters joined by the set union $\cup$, for example $X\cup Y$. In this case, we mean by such an expression the sequence obtained by appending the sequence Y to the sequence X and removing duplicates. For example, if X is x,x,y and Y is y,y,z then $X\cup Y$ denotes the sequence x,y,z.

Reduction using $x=y$. Let x and y be variables appearing in a system S. The following transformation of S is called *reduction using $x=y$*. First, remove $x=y$ and $y=x$ (if any) from S. Then do the following:

1. If x and y are the same variable, do nothing.
2. Otherwise, if one of x,y is of sort $\mathfrak{b}$ and the other is of sort $\mathfrak{s}$, transform S into $\perp$.
3. Otherwise, if $y\sqsubseteq x$, replace x by y in all equations in S and add $x=y$ to S.
4. Otherwise (in this case $x\sqsubseteq y$), replace y by x in all equations in S and add $y=x$ to S.

Isolated equations. Let S be a system containing an equation $x = t$. This equation is called *isolated* in S if (i) $t \sqsubseteq x$ and (ii) x has exactly one occurrence in S (namely, the occurrence in the left-hand side of $x = t$). Note that if $x = y$ belongs to S and is not isolated in S, then reduction using $x = y$ changes $x = y$ to an isolated equation $x = y$ or $y = x$.

Simple equations. We define a *bag equation* as an equation $x = \{\!|Y \mid z|\!\}$, where x and z are bag variables and Y is a sequence of variables. Similarly, $x = \{Y \mid z\}$, where x and z are set variables and Y is a sequence of variables, is called a *set equation*. We define a *simple equation* as any of the following equations: (i) a bag equation, (ii) a set equation, (iii) an equation $x = f(x_1, \ldots, x_n)$, where $n \geq 0$ and $x, x_1, \ldots, x_n$ are variables.

Systems without duplication. Let S be a system consisting of only simple or isolated equations. We call S a *system of simple or isolated equations without duplication* if S contains no pair of equations $x = t$ and $y = t$ such that t is not a variable.

Lemma 1. *Any system S can be transformed in polynomial time to a system S' that satisfies the following conditions:*

1. *S' is a system of simple or isolated equations without duplication.*
2. *Any solution to S' is also a solution to S.*
3. *For any solution ν to S, there is a solution ν' to S' such that ν and ν' coincide on all variables of S.*

Proof. We shall only sketch the transformation.

1. Get rid of non-variable terms in left sides of equations by introducing new variables. An equation $s = t$ is replaced by two equations $x = s$ and $x = t$, where x is a new universal variable.

2. Variable abstraction. Get rid of non-simple equations $x = t$, where t is not a variable, by introducing new variables. For example, the equation $x = \{\{\}, f(a), b\}$ is replaced by five equations $x = \{y, z, v \mid y\}$, $y = \{\}$, $z = f(u)$, $u = a$ and $v = b$, where z, u, v are new universal variables, y is a new set variable.

3. To get rid of duplications, for any pair of equations $x = t$ and $y = t$ remove $x = t$ and apply reduction using $x = y$.

4. Now every non-simple equation is an equation between two variables $x = y$. If this equation is not isolated, apply reduction using $x = y$.

From now on we deal only with systems of simple or isolated equations without duplication.

Graphs associated with systems. To describe the algorithm, we associate with any system S a directed graph denoted by $\mathcal{G}_S$ and called *the graph of S*. The nodes of this graph are all bag and set variables occurring in S. If S contains a bag equation $x = \{\!|Y \mid z|\!\}$ or a set equation $x = \{Y \mid z\}$, then the graph $\mathcal{G}_S$ contains an edge from x to z labeled by Y.

Final variables. A bag variable x is said to be *final in* S if $\mathcal{G}_S$ contains no edge coming from x. A set variable x is called *final in* S if whenever $\mathcal{G}_S$ contains a path from x to another node y, it also contains a path from y to x.

Bag and set cycles. We define a *bag cycle* as a bag equation $x = \{\!|\, Y \mid x \,|\!\}$ or a sequence of bag equations

$$x_1 = \{\!|\, Y_1 \mid x_2 \,|\!\},\ x_2 = \{\!|\, Y_2 \mid x_3 \,|\!\},\ \ldots,\ x_m = \{\!|\, Y_m \mid x_1 \,|\!\},$$

where $m \geq 2$ and $x_1, \ldots, x_m$ are pairwise different variables. A *set cycle* is defined similarly.

Bag and set extensions. Assume that a system S contains a sequence of bag equations

$$y_1 = \{\!|\, X_1 \mid y_2 \,|\!\},\ y_2 = \{\!|\, X_2 \mid y_3 \,|\!\},\quad \ldots,\quad y_m = \{\!|\, X_m \mid y_{m+1} \,|\!\}$$

where $m \geq 1$ and y_{m+1} is a final variable. We say that the bag equation

$$y_1 = \{\!|\, X_1, \ldots, X_m \mid y_{m+1} \,|\!\}$$

is an *extension* of the equation $y_1 = \{\!|\, X_1 \mid y_2 \,|\!\}$.

Similarly, if S contains a sequence of set equations

$$y_1 = \{X_1 \mid y_2\},\ y_2 = \{X_2 \mid y_3\},\quad \ldots,\quad y_m = \{X_m \mid y_{m+1}\},$$

where $m \geq 1$ and y_{m+1} is a final variable, then an *extension* of the set equation $y_1 = \{X_1 \mid y_2\}$ is defined as the equation

$$y_1 = \{X_1 \cup \ldots \cup X_m \mid y_{m+1}\}.$$

Note that if a bag equation has no extension then S has a bag cycle. Every set equation has an extension.

Reduction using $X = Y$. Let X and Y be sequences $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ of variables respectively. Informally, reduction using $X = Y$ is successive reductions using $x_1 = y_1, \ldots, x_n = y_n$. More precisely, denote the system S by S_0 and denote the system $\{X = Y\}$ by E_0. Define S_i and E_i for $1 \leq i \leq n$ as follows. Let $u = v$ be any equation in E_{i-1}. Then S_i is obtained from S_{i-1} by reduction using $u = v$ and E_i is obtained from E_{i-1} by reduction using $u = v$ and removing $u = v$ or $v = u$. *Reduction using* $X = Y$ is defined as the transformation replacing S by S_n.

Correspondences between sequences. Let X and Y be sequences of variables. We define *correspondences* between X and Y, namely correspondences of two kinds called *bag correspondences* and *set correspondences.* A *bag correspondence* between X and Y is defined as an equivalence relation on $X \cup Y$ such that for every equivalence class R, the variables of R satisfy the following condition:

The total number of their occurrences in X is equal to the total number of their occurrences in Y.

For example, let X and Y be x, x, y, z and x, y, y, u respectively. Then there are three bag correspondences between X and Y. The first consists of two equivalence classes $\{x, y\}$ and $\{u, z\}$, the second consists of $\{x, u\}$ and $\{y, z\}$, and the third consists of one class $\{x, y, z, u\}$. Obviously, there exists a bag correspondence between X and Y if and only if X and Y have the same length.

A bag correspondence is called *minimal* if no equivalence class can be split into proper disjoint subsets such that the resulting equivalence relation remains a bag correspondence. In the above example, the first and second correspondences are minimal and the third one is not minimal. Note that the minimality can be checked in polynomial time by reducing this problem to the unary version of the knapsack problem, i.e. the version in which weights and values are given in unary notation (see e.g. [41]). Details of the reduction will be given in a full version of the paper.

A *set correspondence* between X and Y is an equivalence relation on $X \cup Y$ such that for every equivalence class R, we have

R contains at least one variable of X and at least one variable of Y.

Like the case of bag correspondences, a set correspondence is called *minimal* if no equivalence class can be split into proper disjoint subsets such that the resulting equivalence relation remains a set correspondence. In this case, it means that each equivalence class contains either exactly one variable of X or exactly one variable of Y.

For example, let X and Y be x, x, y, z and x, u, v, w respectively. Then the equivalence relation consisting of classes $\{x, u\}$, $\{y, v\}$ and $\{z, w\}$ is a minimal set correspondence between X and Y. The relation consisting of $\{x, y, u\}$ and $\{z, v, w\}$ is a set correspondence but it is not minimal.

Let E be any set of equations $x_i = y_j$ where x_i is in X and y_j is in Y. By $\sim_E$ we denote the smallest equivalence relation on $X \cup Y$ containing all pairs (x, y) such that $(x = y) \in E$. We say that E is a (minimal) bag or set correspondence between X and Y if such is $\sim_E$. For example, the set $\{x = u, y = v, z = w\}$ is a minimal set correspondence between X and Y that denote x, x, y, z and x, u, v, w respectively.

3.2 Rules

Rule 1 (Tree Decomposition). This rule can be applied to a system S if S contains two distinct equations $z = f(x_1, \ldots, x_n)$ and $z = f(y_1, \ldots, y_n)$, where $f \in \mathcal{F}$ and $n \geq 0$. Remove the equation $z = f(x_1, \ldots, x_n)$ from S and reduce S using $x_1 = y_1, \ldots, x_n = y_n$. (This rule is close to the term decomposition rule of [38]).

Rule 2 (Bag Decomposition). The rule can be applied to a system S if the system contains bag equations $z = s$ and $z = t$, where s and t are different. If at least one of $z = s$ and $z = t$ has no extension then transform S into $\perp$ (the system contains a bag cycle). Otherwise, choose any extensions of $z = s$ and $z = t$. Remove $z = s$ and $z = t$ from S and consider two possible cases.

1. *The extensions have the forms* $z = \{\!|X \,|\, u|\!\}$ *and* $z = \{\!|Y \,|\, u|\!\}$. If $|X| \neq |Y|$ then transform S into $\perp$. Otherwise, add to S the equation $z = \{\!|Y \,|\, u|\!\}$. Don't-know nondeterministically generate a minimal bag correspondence E between X and Y and reduce S using E.
2. *The extensions have the forms* $z = \{\!|X \,|\, u|\!\}$ *and* $z = \{\!|Y \,|\, v|\!\}$, *where the variables* u *and* v *are different.* Don't-know nondeterministically divide X into disjoint parts X_1 and X_2 (one of them may be empty). Similarly, Don't-know nondeterministically divide Y into Y_1 and Y_2. Informal comment: X_1 and Y_1 are intended to coincide as bags, X_2 and Y_2 are intended to have no common elements. The division is required to satisfy the following conditions. First, the parts X_1 and Y_1 have the same length, i.e. $|X_1| = |Y_1|$. Second, X_2 and Y_2 contain no common variables. Then S is transformed as follows.
   (a) Add the equation $z = \{\!|X_2, Y_1, Y_2 \,|\, w|\!\}$, where w is a new bag variable.
   (b) If X_2 is non-empty, add the equation $v = \{\!|X_2 \,|\, w|\!\}$. Otherwise, reduce S using $v = w$.
   (c) If Y_2 is non-empty, add the equation $u = \{\!|Y_2 \,|\, w|\!\}$. Otherwise, reduce S using $u = w$.
   (d) If X_1 and Y_1 are non-empty, don't-know nondeterministically generate a minimal bag correspondence E between X_1 and Y_1. Reduce S using E.

Rule 3 (Set Decomposition). The rule can be applied to a system S if the system contains set equations $z = s$ and $z = t$, where s and t are different. If z is a final variable, do the following. Suppose that s is $\{X \,|\, u\}$ and t is $\{Y \,|\, v\}$. Replace $z = s$ and $z = t$ by the equation $z = \{X \cup Y \,|\, z\}$ and reduce the system using $z = u, z = v$. Otherwise, choose any extensions of $z = s$ and $z = t$. Remove $z = s$ and $z = t$ from S and consider two possible cases.

1. *The extensions have the forms* $z = \{X \,|\, u\}$ *and* $z = \{Y \,|\, u\}$. Don't-know nondeterministically divide X into disjoint parts X_1 and X_2. Similarly, divide Y into Y_1 and Y_2. Informally: X_1 and Y_1 coincide as sets, X_2 and Y_2 have no common members. If one of the parts X_1 and Y_1 is empty then the other is required to be empty too. In addition, like the case of bags, X_2 and Y_2 are required to contain no common variables. Consider two cases.
   (a) *Both* X_1 *and* Y_1 *are empty.* Add the equation $u = \{X_2 \cup Y_2 \,|\, u\}$ and reduce S using $z = u$.
   (b) *Both* X_1 *and* Y_1 *are non-empty.* Don't-know nondeterministically generate a minimal set correspondence E between X_1 and Y_1. Then transform S as follows.
      i. Add the equation $z = \{Y_1 \,|\, u\}$.
      ii. If at least one of X_2 and Y_2 is non-empty, add $u = \{X_2 \cup Y_2 \,|\, u\}$.

iii. Reduce S using E.

2. *The extensions have the forms $z = \{X \mid u\}$ and $z = \{Y \mid v\}$, where the variables u and v are different.* Don't-know nondeterministically divide X into disjoint parts X_1 and X_2 and divide Y into Y_1 and Y_2. As above, the division is required to satisfy the following conditions. First, if one of the parts X_1 and Y_1 is empty then the other is empty too. Second, X_2 and Y_2 contain no common variables. Consider two cases.
   (a) *Both X_1 and Y_1 are empty.* Add the equations $z = \{X_2 \cup Y_2 \mid w\}$, $u = \{Y_2 \mid w\}$ and $v = \{X_2 \mid w\}$, where w is a new variable.
   (b) *Both X_1 and Y_1 are non-empty.* Transform S as follows.
      i. Add equation $z = \{X_2 \cup Y_1 \cup Y_2 \mid w\}$, where w is a new set variable.
      ii. If X_2 is non-empty, add the equation $v = \{X_2 \mid w\}$. Otherwise reduce S using $v = w$.
      iii. If Y_2 is non-empty, add the equation $u = \{Y_2 \mid w\}$. Otherwise reduce S using $u = w$.
      iv. Don't-know nondeterministically generate a minimal set correspondence E between X_1 and Y_1 and reduce S using E.

Rule 4 (Function Failure). If S contains equations $x = f(x_1, \ldots, x_m)$ and $x = g(y_1, \ldots, y_n)$ where $m, n \geq 0$ and f and g are distinct function symbols, transform S into $\bot$.

Rule 5 (Type Failure). If the system S contains an equation $x = t$ such that $t \not\sqsubseteq x$, transform S into $\bot$.

3.3 Algorithms

Solved form. A system S of equations is said to be *in solved form* if no rule is applicable to S. In particular, $\bot$ is in solved form. It is easy to see that a system S is in solved form if and only if (i) S does not contain two different equations $x = s$ and $x = t$, and (ii) for any equation $x = t$ in S, we have $t \sqsubseteq x$.

Transformation algorithm. *The transformation algorithm* don't-care nondeterministically chooses Rules 1–5 and applies them to an input system until no rule is applicable. Thus, the transformation algorithm is a nondeterministic algorithm that transforms any input system into a system in solved form.

Variable dependency graph. We introduce one more graph associated with a system S. The *variable dependency graph* of S is the graph whose nodes are variables occurring in S and whose edges are defined as follows. There is an edge coming from x to y if S contains at least one of the following equations:

$x = f(y_1, \ldots, y_n)$, where $f \in \mathcal{F}$ and y is one of $y_1, \ldots, y_n$;
$x = \{\!|y_1, \ldots, y_n \mid z|\!\}$ such that y is one of $y_1, \ldots, y_n, z$;
$x = \{y_1, \ldots, y_n \mid z\}$ such that y is one of $y_1, \ldots, y_n$.

Occur-check algorithm. The occur-check algorithm is applied to a system S in solved form. If the variable dependency graph of S has a cycle, S is transformed into $\perp$. Otherwise, the algorithm does not change S.

Unification algorithm. The unification algorithm is the composition of the transformation algorithm and the occur-check algorithm.

Theorem 2 (running time). *The unification algorithm runs in nondeterministic polynomial time.*

Theorem 3 (soundness and completeness). *Let $S_1, \dots, S_n$ be all nondeterministic results of the application of the unification algorithm to a system S. Then*

1. *Any solution to S_i is also a solution to S.*
2. *For any solution ν to S, there is a solution ν_i to some S_i such that ν and ν_i coincide on all variables of S.*

It follows from these theorems and NP-hardness [28] that the unifiability problem for $\mathcal{HU}^+$ is NP-complete (as we note in Section 4 below, this fact also follows from results on AC and ACI unification).

Usually, unification problems are formulated in terms of finding *unifiers*, i.e. substitutions that make two terms equal. A set Σ of unifiers of a system S is said to be *complete* if for every unifier of S, the set Σ contains a more general unifier of S. It is straightforward to extract unifiers from our algorithm. Namely, every nondeterministic branch leads to either $\perp$ or a system in solved form. Such a system represents a unifier called a *resulting unifier* for S, for details see [19]. The completeness of our algorithm provides that all resulting unifiers for S form a complete set of unifiers of S. Since every branch gives us at most one unifier, we obtain an upper bound on the minimal cardinality of a complete set of unifiers.

Theorem 4 (upper bound). *For any system S, the set of all resulting unifiers for S is a complete set of unifiers. An upper bound on its cardinality is $2^{O(n \log n)}$ where n is the size of S.*

It follows from [9] and [19] that $2^{O(n \log n)}$ is also a lower bound. To establish this lower bound, it is enough to consider unification for flat sets. Thus, $2^{O(n \log n)}$ is a tight bound.

Minimality. We say that a complete set Σ of unifiers is *minimal* if for every pair of unifiers in Σ, none of them is more general than the other. A unification algorithm is said to be *optimal for a system S* if the algorithm yields a minimal complete set of unifiers of S. Our algorithm (as well as all other known algorithms) is not optimal in general, but it is optimal for a number of important special cases. First, consider the following equations on sets:

$$\{s_1, \dots, s_m \,|\, x\} = \{t_1, \dots, t_n \,|\, x\} \tag{1}$$

$$\{s_1, \dots, s_m \,|\, x\} = \{t_1, \dots, t_n \,|\, y\} \tag{2}$$

$$\{s_1, \dots, s_m \,|\, x\} = \{t_1, \dots, t_n\} \tag{3}$$

$$\{s_1, \dots, s_m\} = \{t_1, \dots, t_n\} \tag{4}$$

where x, y are different variables not occurring in $s_1, \ldots, s_m, t_1, \ldots, t_n$, and $s_1, \ldots, s_m, t_1, \ldots, t_n$ are either variables or ground terms without occurrences of bag or set constructors, not necessarily different. Systems (1)–(4)are not, in general, systems of simple or isolated equations without duplication, and there are several ways of transforming them into such systems.

Assuming that equations (1)–(4) are preprocessed using the algorithm presented in Lemma 1, we obtain

Theorem 5. *The unification algorithm is optimal for any system consisting of one equation of the form* (1)–(4)*.*

The proof is not difficult and is based on properties of minimal set correspondences. Note that (1)–(4) contain all equations considered in [9], for which optimality of a different algorithm is proved.

Similarly, using properties of minimal bag correspondences we can prove optimality for special cases of bag equation. Consider the following equations on bags:

$$\{\!|s_1, \ldots, s_m \,|\, x|\!\} = \{\!|t_1, \ldots, t_n \,|\, x|\!\} \tag{5}$$

$$\{\!|s_1, \ldots, s_m \,|\, x|\!\} = \{\!|t_1, \ldots, t_n \,|\, y|\!\} \tag{6}$$

$$\{\!|s_1, \ldots, s_m \,|\, x|\!\} = \{\!|t_1, \ldots, t_n|\!\} \tag{7}$$

$$\{\!|s_1, \ldots, s_m|\!\} = \{\!|t_1, \ldots, t_n|\!\} \tag{8}$$

with the same conditions as for (1)–(4).

Assuming that equations (5)–(8) are preprocessed using the algorithm presented in Lemma 1, we obtain

Theorem 6. *The unification algorithm is optimal for any system consisting of one equation of the form* (5)–(8)*.*

Note that having systems without duplication is essential for the optimality. For example, our algorithm is optimal when the equation $\{\!|x, c|\!\} = \{\!|y, c|\!\}$ is translated into $\{\!|x, z|\!\} = \{\!|y, z|\!\}, z = c$ but is not optimal when this equation is translated into $\{\!|x, z|\!\} = \{\!|y, u|\!\}, z = c, u = c$.

4 Related results

AC and ACI unification. AC and ACI unification is more general than bag and set unification. First results relevant to bag and set unification appeared in the automated deduction community as results on AC- and ACI-unification algorithms [44, 28, 25, 10, 33, 18]. These algorithms deal with the first three of the following equality axioms:

$$\begin{array}{ll}
(A) & (x \cup y) \cup z = x \cup (y \cup z) \\
(C) & x \cup y = y \cup x \\
(I) & x \cup x = x \\
(1) & x \cup \{\} = x
\end{array}$$

All the four axioms give an axiomatization of finite sets in the signature consisting of the set union $\cup$ and the empty set $\{\}$. By removing (I), we obtain an axiomatization of bags in the signature consisting of the bag union and the empty bag. Both axiomatizations are complete in the sense that every valid equation on sets or bags is a logical consequence of these axiomatizations. It is known that both AC1-unifiability with constants is NP-complete: NP-hardness for a very simple special case is proved in [28] and inclusion in NP is proved in [10]. ACI1-unifiability with constant can be solved in polynomial time [29]. It follows from the results on combination of unification algorithms [11, 42, 17] that for the theory combining AC1 and ACI1, the unifiability problem is in NP [12, Theorem 5.2].

The influence of results on AC1 and ACI1-unification on unification problems with bag and set constructors was largely ignored (for which we are also to blame, see the preliminary version of this paper [19]). For example, one can read in [9]:

> "by dealing with nested sets we can solve set-unification problems that cannot be expressed using ACI unification; for instance $\{x, \{y, \{\emptyset, z\}\}\} = \{\{z\}, w\}$."

However, it is not hard to see that the set constructor can be defined from the union $\cup$ by using the additional singleton set constructor $\{\dots\}$. Indeed, we have $\{x \,|\, y\} = \{x\} \cup y$. Thus, the unification problem for $\mathcal{HU}^+$ (as well as for the domains of [22, 45, 21]) can be implemented as unification in the theory combining AC1 and ACI1. This fact has been noted for example in [7] and in [43]. In particular, by [12, Theorem 5.2], the unifiability in the domain combining bags, sets and trees is in NP.

There are various motivations for using the signature with the bag and set constructors instead of the union. Our motivation is explained by Theorem 1: bag and set constructors are enough to express any computable function on the universe with bags and sets. In addition, known AC1- and ACI1-unification algorithms adapted for $\mathcal{HU}^+$ are too complex compared to our algorithm (for example, they use solutions to systems of Diophantine equations and one should also count the complexity added by the techniques of combining unification algorithms). In the signature with the union, there is a double-exponential lower bound on cardinalities of minimal complete sets of unifiers for bag equations [30]. Using bag and set constructors instead of the union, our algorithm gives a single-exponential upper bound even for the combined domain. This bound is new and does not follows from other results in the literature.

Other domains for bags and finite sets. Our unification algorithm can be modified in a straightforward way to deal with other data models for bags and sets. In particular, [19] defines a *typed universe* (in the spirit of [1] or [34]) and a *universe of colored bags and colored sets* (similar to [22]). The corresponding modifications of the algorithm are sketched too.

Complexity of nonrecursive logic programs with bags and sets. As it is shown in [20], if solvability of equations over a domain is in NP, then the query evaluation problem for nonrecursive Horn clause logic programs over this domain is in NEXP (see [20] for precise definitions). Therefore, the query evaluation problem for such programs over bags and/or sets and/or trees is in NEXP (as it follows from another result of [20], this problem is also NEXP-hard). This bound does not hold for nonrecursive logic programs with negation: in this case the query evaluation problem may be nonelementary or even undecidable already for domains with sets [47].

Comparison with other algorithms. Our approach is close to the approach of [22] where a set unification algorithm has been proposed. However, incorporating bags in a logic programming language is stated as an open problem in [22, page 34]. Also, there is much in common with other known unification algorithms for bags and sets built with the bag and set constructors [21–23, 45, 9]. It is natural to compare them in important special cases of flat bags and sets, for example when solving equations of the form

$$\{x_1, \dots, x_n \,|\, x\} = \{y_1, \dots, y_m \,|\, y\},$$

where $x_1, \dots, x_n, x, y_1, \dots, y_m, y$ are variables. As it was mentioned, in this case the minimal cardinality of complete sets of unifiers may be exponential.

Set unification in [22] is not optimal in this case. In [45] the special case of flat sets is treated in detail. The algorithm of [45] tries to take care of information about unifiability of elements of sets. This idea is interesting and natural, but unfortunately the algorithm of [45] does not work for embedded sets, despite the claim to do so. For the flat case, the algorithm of [45] may be better than all known algoritms in the number of computed unifiers, but it is not clear how to modify it for embedded sets (for example, because it checks unifiability of subterms and uses *most specific generalizations* that do not exist for sets). The set unification algorithm of [9] also uses optimizations for the flat case. It is proved that the algorithm computes minimal complete sets of unifiers for a number of special cases of flat sets. All these special cases are covered by cases (1)–(4). We achieve minimality by using minimal bag and set correspondences which is a new idea and allows us to get an optimal algorithm both for cases (1)–(4) of sets and (5)–(8) of bags. Note that the algorithms of [45, 9] apply substitutions explicitly and thus use exponential space, though we guess they can be modified into nondeterministic polynomial-time ones.

It seems that optimal algorithms for flat bags have not been considered in other papers. For example, the algorithm of [23] is not optimal for the bag equation $\{\!|x, x \,|\, y|\!\} = \{\!|x \,|\, z|\!\}$, while our algorithm is optimal for such equations (case (6) of Theorem 6). Although [23] asserts that

> "The axiomatizations presented can easily be combined in order to obtain axiomatic theories capable to deal with any subset of the collection of proposed structures. Moreover, the unification algorithms presented in the next section can easily be merged to solve the unification problem relative to such combined context",

no details are given.

Directions of further research. It is interesting to consider constraint logic programming over bags and finite sets which uses more powerful primitives than just the bag and set constructors. For examples, what is the complexity of constraint satisfaction when we also have primitives like $\cup$ or $\subseteq$? Set constraints have recently received a considerable attention in connection with program verification, but mostly for infinite sets and constraints that are less relevant to databases or logic programs, see e.g. [5, 14, 39, 13] and the survey [40]).

Also, it is interesting to consider a suitable representation of graphs (up to isomorphism) and the complexity of unification over graphs. This may be useful for dealing with semistructured data [4].

Acknowledgments

The first author was supported by grants from the Swedish Royal Academy of Sciences, the Swedish Institute, INTAS, and RFBR. The second author was supported by a TFR grant. We thank Franz Baader and Klaus Schulz for their comments on the combination of unification algorithms.

References

1. S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4:727–794, 1995.
2. S. Abiteboul and S. Grumbach. Col: A logic-based language for complex objects. In J. Schmidt, S. Ceri, and M. Missikoff, editors, *Advances in Database Technology - EDBT'88. Proceedings of the International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 271–293, Venice, Italy, March 1988. Springer Verlag.
3. S. Abiteboul and S. Grumbach. A rule-based language with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, 1991.
4. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1), 1996.
5. A. Aiken. Set constraints: Results, applications and future directions. In A. Borning, editor, *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 326–335. Springer Verlag, 1994.
6. Andréka and Németi. A generalized completeness of Horn clause logic seen as a programming language. *Acta Cybernetica*, 4:3–10, 1978.
7. V.M. Antimirov, A. Degtyarev, V.S. Procenko, A. Voronkov, and M.V. Zakharjashchev. Mathematical logic in programming (in Russian). In Yu.I. Yanov and M.V. Zakharjashchev, editors, *Mathematical Logic in Programming*, pages 331–407. Mir, Moscow, 1991.
8. K.R. Apt. Logic programming. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 10, pages 493–574. Elsevier Science, Amsterdam, 1990.
9. P. Arenas-Sánchez and A. Dovier. A minimality study for set unification. *Journal of Functional and Logic Programming*, 1997(7), December 1997.

10. F. Baader and W. Büttner. Unification in commutative and idempotent monoids. *Theoretical Computer Science*, 56:345–352, 1988.
11. F. Baader and K.U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 50–65, Saratoga Springs, NY, USA, June 1992.
12. F. Baader and K.U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *Journal of Symbolic Computations*, 21:211–243, 1996.
13. L. Bachmair and H. Ganzinger. Set constraints with intersection. In G. Winskel, editor, *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 362–372. IEEE Computer Society Press, 1997.
14. L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 75–83. IEEE Computer Society Press, 1993.
15. C. Beeri and Y. Kornatzky. A logical query language for hypermedia systems. *Information Sciences*, 77:1–38, 1994.
16. C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set constructors in a logic database language. *Journal of Logic Programming*, 10:181–232, 1991.
17. A. Boudet. Combining unification algorithms. *Journal of Symbolic Computations*, 16:597–626, 1993.
18. A. Boudet, E. Contejean, and C. Marcé. AC-complete unification and its application to theorem proving. In H. Ganzinger, editor, *7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 18–32. Springer Verlag, 1996.
19. E. Dantsin and A. Voronkov. Bag and set unification. UPMAIL Technical Report 150, Uppsala University, Computing Science Department, November 1997.
20. E. Dantsin and A. Voronkov. Complexity of query answering in logic databases with complex values. In S. Adian and A. Nerode, editors, *Logical Foundations of Computer Science. 4th International Symposium, LFCS'97*, volume 1234 of *Lecture Notes in Computer Science*, pages 56–66, Yaroslavl, Russia, July 1997.
21. A. Dovier. *Computable Set Theory and Logic Programming*. PhD thesis, Universita' degli Studi di Pisa, dip. di Informatica, March 1996.
22. A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. {log}: A language for programming in logic with finite sets. *Journal of Logic Programming*, 28(1):1–44, 1996.
23. A. Dovier, A. Policriti, and G. Rossi. Integrating lists, multisets and sets in a logic programming framework. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems*, pages 303–321. Kluwer, 1996.
24. F. Fages. Associative-commutative unification. *Journal of Symbolic Computations*, 3(3), 1987.
25. Fortenbacher. An algebraic approach to unification under associativity and commutativity. *Journal of Symbolic Computations*, 3(3):217–229, 1987.
26. S. Grumbach and V. Vianu. Tractable query languages for complex object databases. *Journal of Computer and System Sciences*, 51(2):149–167, 1995.
27. P. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51:26–52, 1995.
28. D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In J. Siekmann, editor, *Proc. 8th CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 489–495, 1986.

29. D. Kapur and P. Narendran. Complexity of unification problems with associative-commutative operators. *Journal of Automated Reasoning*, 9(2):261–288, 1992.
30. D. Kapur and P. Narendran. Double-exponential complexity of computing a complete set of *AC*-unifiers. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*. IEEE Computer Society Press, 1992.
31. G.M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41:44–64, 1990.
32. N. Leone and P. Rullo. Ordered logic programming with sets. *Journal of Logic and Computation*, 3(6):621–642, 1993.
33. P. Lincoln and J. Christian. Adventures in associative-commutative unification (a summary). *Journal of Logic and Computation*, 8(1/2):217–240, 1989.
34. M. Liu. Relationlog: a typed extension to datalog with sets and tuples. *Journal of Logic Programming*, 35(1):1–30, 1998.
35. J.W. Lloyd. *Foundations of Logic Programming (2nd edition)*. Springer Verlag, 1987.
36. M.J. Maher. A CLP view of logic programming. In *Proc. Conf. on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 364–383, October 1992.
37. M.J. Maher. A logic programming view of CLP. In *International Conference on Logic Programming*, pages 737–753, 1993.
38. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
39. D.A. McAllister, R. Givan, C. Witty, and D. Kozen. Tarskian set constraints. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 138–147, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
40. L. Pacholski and A. Podelski. Set constraints: a pearl in research on constraints. In G. Smolka, editor, *Proceedings of the Third International Conference on Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
41. C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
42. M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *Journal of Symbolic Computations*, 1990. Special issue on Unification.
43. O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of set terms in the logic data language (LDL). *Journal of Logic Programming*, 12(1):89–119, 1992.
44. M.E. Stickel. A complete unification algorithm for associative-commutative functions. *Journal of the Association for Computing Machinery*, 28(3):423–434, 1981.
45. F. Stolzenburg. Membership constraints and complexity in logic programming with sets. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems*, pages 285–302. Kluwer, 1996.
46. K. Vadaparty. On the power of rule-based languages with sets. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 26–36, 1991.
47. S. Vorobyov and A. Voronkov. Complexity of nonrecursive logic programs with complex values. In *PODS'98*, pages 244–253, Seattle, Washington, 1998. ACM Press.
48. A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, *Logic Programming*, volume 592 of *Lecture Notes in Artificial Intelligence*, pages 486–514. Springer Verlag, 1992.
49. A. Voronkov. On computability by logic programs. *Annals of Mathematics and Artificial Intelligence*, 15(3,4):437–456, 1995.