Aggregate Functions in DLV*

Tina Dell'Armi[1], Wolfgang Faber[2], Giuseppe Ielpa[1],
Nicola Leone[1], and Gerald Pfeifer[2]

[1] Department of Mathematics, University of Calabria
87030 Rende (CS), Italy
{dellarmi,ielpa,leone}@unical.it

[2] Institut für Informationssysteme 184/3, TU Wien
Favoritenstraße 9-11, A-1040 Wien, Austria
faber@kr.tuwien.ac.at, gerald@pfeifer.com

Abstract. Disjunctive Logic Programming (DLP) is a very expressive formalism: it allows for expressing every property of finite structures that is decidable in the complexity class $\Sigma_2^P (=\mathrm{NP}^{\mathrm{NP}})$. Despite this high expressiveness, there are some simple properties, often arising in real-world applications, which cannot be encoded in a simple and natural manner. Especially properties that require the use of arithmetic operators (like sum, count, or min) on a set of elements satisfying some conditions, cannot be naturally expressed in classic DLP.
To overcome this deficiency, we extend DLP by aggregate functions. In contrast to a previous proposal, we also consider the case of unstratified aggregates. We formally define the semantics of the new language (called DLP$^{\mathcal{A}}$) by means of a generalization of the Gelfond-Lifschitz transformation, and illustrate the use of the new constructs on relevant knowledge-based problems. We analyze the computational complexity of DLP$^{\mathcal{A}}$, showing that the addition of aggregates does not bring a higher cost in that respect. And we provide an implementation of DLP$^{\mathcal{A}}$ in DLV– the state-of-the-art DLP system – and report on experiments which confirm the usefulness of the proposed extension also for the efficiency of the computation.

1 Introduction

Disjunctive Logic Programs (DLP) are logic programs where (nonmonotonic) negation may occur in the bodies, and disjunction may occur in the heads of rules. This language is very expressive in a precise mathematical sense: under the answer set semantics [GL91] it allows to express every property of finite structures that is decidable in the complexity class $\Sigma_2^P (= \mathrm{NP}^{\mathrm{NP}})$. Therefore, under widely believed assumptions, DLP is strictly more expressive than *normal* (*disjunction-free*) logic programming, whose expressiveness is limited to properties decidable in NP, and it can express problems which cannot be translated to satisfiability of CNF formulas in polynomial time. Importantly, besides enlarging the class of applications which can be encoded in the language, disjunction often allows for representing problems of lower complexity in a simpler and more natural fashion (see [EFLP00]).

* This work was supported by the European Commission under projects IST-2002-33570 INFOMIX, IST-2001-32429 ICONS, and FET-2001-37004 WASP. A preliminary version was presented at IJCAI'03.

The problem. Despite this high expressiveness, there are some simple properties, often arising in real-world applications, which cannot be encoded in DLP in a simple and natural manner. Among these are properties which require the application of some arithmetic operators (e.g., sum, times, count) on a set of elements satisfying some conditions. Suppose, for instance, that you want to know if the sum of the salaries of the employees working in a team exceeds a given budget (see Team Building in Section 3). To this end, you should first order the employees defining a successor relation. You should then define a *sum* predicate, in a recursive way, which computes the sum of all salaries, and compare its result with the given budget. This approach has two drawbacks: (1) It is bad from the KR perspective, as the encoding is not natural at all; (2) it is inefficient, as the (instantiation of the) program is quadratic (in the cardinality of the input set of employees). Thus, there is a clear need to enrich DLP with suitable constructs for the natural representation of such properties and to provide means for an efficient evaluation.

Contribution. We overcome the above deficiency of DLP. Instead of inventing new constructs from scratch, we extend the language with a sort of aggregate functions, first studied in the context of deductive databases, and implement them in DLV [EFLP00] – the state-of-the-art Disjunctive Logic Programming system. The main contributions of this paper are the following:

- We extend Disjunctive Logic Programming by aggregate functions and formally define the semantics of the resulting language, named DLP$^{\mathcal{A}}$.
- We address knowledge representation issues, showing the impact of the new constructs on relevant problems.
- We analyze the computational complexity of DLP$^{\mathcal{A}}$. Importantly, it turns out that the addition of aggregates does not increase the computational complexity, which remains the same as for reasoning on DLP programs.
- We provide an implementation of DLP$^{\mathcal{A}}$ in the DLV system, deriving new algorithms and optimization techniques for efficient evaluation.
- We report on experimentation, evaluating the impact of the proposed language extension on efficiency. The experiments confirm that, besides providing relevant advantages from the knowledge representation point of view, aggregate functions can bring significant computational gains.
- We compare DLP$^{\mathcal{A}}$ with related work.

A previous version of DLP$^{\mathcal{A}}$ [DFI$^+$03] required aggregates to honor stratification, that is, predicates defined by means of aggregates could not mutually depend on one another. Here, we lift this restriction and allow for arbitrarily recursive definitions. To that end we provide a novel approach to defining the semantics of DLP$^{\mathcal{A}}$ programs which naturally extends the original definition of answer sets. We also describe suitable enhancements to the implementation in order to deal with unstratified aggregates.

2 The DLP$^{\mathcal{A}}$ Language

In this section, we provide a formal definition of the syntax and semantics of the DLP$^{\mathcal{A}}$ language – an extension of DLP by set-oriented (or aggregate) functions. We assume that the reader is familiar with standard DLP; we refer to atoms, literals, rules, and programs of DLP

as *standard atoms*, *standard literals*, *standard rules*, and *standard programs*, respectively. For further background, see [GL91,EFLP00].

2.1 Syntax

A (DLP$^{\mathcal{A}}$) *set* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{\, Vars : Conj\}$, where $Vars$ is a list of variables and $Conj$ is a conjunction of standard literals.[3] A *ground set* is a set of pairs of the form $\langle \overline{t} : Conj \rangle$, where $\overline{t}$ is a list of constants and $Conj$ is a ground (variable free) conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where S is a set, and f is a *function name* among $\#\texttt{count}$, $\#\texttt{min}$, $\#\texttt{max}$, $\#\texttt{sum}$, $\#\texttt{times}$. An *aggregate atom* is $Lg \prec_1 f(S) \prec_2 Rg$, where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$, and Lg and Rg (called *left guard*, and *right guard*, respectively) are terms. One of "$Lg \prec_1$" and "$\prec_2 Rg$" can be omitted. An *atom* is either a standard DLP atom or an aggregate atom. A *literal* L is an atom A or an atom A preceded by the default negation symbol *not*; if A is an aggregate atom, L is an *aggregate literal*.

A *(*DLP$^{\mathcal{A}}$*) rule* r is a construct

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ \texttt{:-} \ b_1, \cdots, b_k, \ \texttt{not} \ b_{k+1}, \cdots, \ \texttt{not} \ b_m.$$

where $a_1, \cdots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are atoms, and $n \geq 0$, $m \geq k \geq 0$, $m+n \geq 1$. The disjunction $a_1 \ \vee \ \cdots \ \vee \ a_n$ is the *head* of r while the conjunction $b_1, \ldots, \texttt{not} \ b_m$ is the *body* of r. We define $H(r) = \{a_1, \ldots, a_n\}$ and $B(r) = \{b_1, \ldots, \texttt{not} \ b_m\}$. A rule with an empty body (i.e. $k = m = 0$) is called a fact, and we usually omit the " :- " sign. A *(*DLP$^{\mathcal{A}}$*) program* is a set of DLP$^{\mathcal{A}}$ rules.

Syntactic Restrictions and Notation.

For simplicity, and without loss of generality, we assume that the body of each rule contains at most one aggregate atom. A *global* variable of a rule r is a variable appearing in some standard atom of r; a *local* variable of r is a variable appearing solely in an aggregate function in r.

Safety. A rule r is *safe* if the following conditions hold: (i) each global variable of r appears in a positive standard literal in the body of r; (ii) each local variable of r that appears in a symbolic set $\{\, Vars : Conj\}$ also appears in a positive literal in $Conj$; (iii) each guard of an aggregate atom of r is either a constant or a global variable. Finally, a program is safe if all of its rules are safe.

Example 1. Consider the following rules:

$$p(X) \texttt{:-} \ q(X,Y,V), Y < \#\texttt{max}\{Z : r(Z), \texttt{not} \ a(Z,V)\}.$$
$$p(X) \texttt{:-} \ q(X,Y,V), Y < \#\texttt{sum}\{Z : \texttt{not} \ a(Z,S)\}.$$
$$p(X) \texttt{:-} \ q(X,Y,V), T < \#\texttt{min}\{Z : r(Z), \texttt{not} \ a(Z,V)\}.$$

The first rule is safe, while the second is not, since both local variables Z and S violate condition (ii). The third rule is not safe either, since the guard T is not a global variable. ∎

Stratification. This is a concept originally introduced for the use of negation in logic programming. In our context, it ensures that two predicates defined by means of aggregates do

[3] Intuitively, a symbolic set $\{X : a(X,Y), \texttt{not} \ p(Y)\}$ stands for the set of X-values making the conjunction $a(X,Y), \texttt{not} \ p(Y)$ true, i.e., $\{X : \exists Y \, s.t. \ a(X,Y) \wedge \texttt{not} \ p(Y) \ is \ true\}$.

not mutually depend on one another. Our original definition and implementation of $\mathrm{DLP}^{\mathcal{A}}$ in [DFI+03] imposes this syntactic restriction, which we now lift in the present paper, though we still differentiate stratified programs as they allow for a more efficient computation of the instantiation of aggregate atoms.

A $\mathrm{DLP}^{\mathcal{A}}$ program $\mathcal{P}$ is *aggregate-stratified* if there exists a function $||\ ||$, called *level mapping*, from the set of (standard) predicates of $\mathcal{P}$ to ordinals, such that for each pair a and b of (standard) predicates of $\mathcal{P}$, and for each rule $r \in \mathcal{P}$ the following hold: (i) If a appears in the head of r, and b appears in an aggregate atom in the body of r, then $||b|| < ||a||$; and (ii) if a appears in the head of r, and b occurs in a standard atom in the body of r, then $||b|| \leq ||a||$.

Example 2. Consider the program consisting of a set of facts for predicates a and b, plus the following two rules:

$$q(X) \text{ :- } p(X), \#\mathtt{count}\{Y : a(Y,X), b(X)\} \leq 2.$$
$$p(X) \text{ :- } q(X), b(X).$$

The program is aggregate-stratified, as the level mapping $||a|| = ||b|| = 1 \quad ||p|| = ||q|| = 2$ satisfies the required conditions. If we add the rule $b(X)$:- $p(X)$, then no level-mapping exists and the program becomes aggregate-unstratified. ∎

Intuitively, aggregate-stratification forbids recursion through aggregates. Stratified aggregates are computationally somewhat easier, as they can be evaluated step by step, as the truth value of an aggregate atom cannot change after it has been fixed once. On the other hand, using unstratified aggregates, one can encode unstratified negation, and hence the situation becomes less clear in this case.

Consider, for instance, the (aggregate-unstratified) program consisting only of the rule $r : p(a)$:- $\#\mathtt{count}\{X : p(X)\} = 0$. Neither $\{p(a)\}$ nor $\emptyset$ is an intuitive meaning for the program, and neither of the two is an answer set. Indeed, the rule r corresponds to $p(a)$:- `not` pp. and pp :- $p(X)$. which does not admit any answer set, either.

2.2 Semantics

Given a $\mathrm{DLP}^{\mathcal{A}}$ program $\mathcal{P}$, let $U_{\mathcal{P}}$ denote the set of constants appearing in $\mathcal{P}$, $U_{\mathcal{P}}^{\mathcal{N}} \subseteq U_{\mathcal{P}}$ the set of the natural numbers occurring in $U_{\mathcal{P}}$, and $B_{\mathcal{P}}$ the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$. Furthermore, given a set X, $\overline{2}^{X}$ denotes the set of all multisets over elements from X. Let us now describe the domains and the meanings of the aggregate functions we consider:

$\#\mathbf{count}$: defined over $\overline{2}^{U_{\mathcal{P}}}$, the number of elements in the set.
$\#\mathbf{sum}$: defined over $\overline{2}^{U_{\mathcal{P}}^{\mathcal{N}}}$, the sum of the numbers in the set; 0 in case of the empty set.
$\#\mathbf{times}$: over $\overline{2}^{U_{\mathcal{P}}^{\mathcal{N}}}$, the product of the numbers in the set; 1 for the empty set. $\#\mathbf{min}$, $\#\mathbf{max}$: defined over $\overline{2}^{U_{\mathcal{P}}} - \{\emptyset\}$, the minimum/maximum element in the set; if the set contains also strings, the lexicographic ordering is considered.[4]

If the argument of an aggregate function does not belong to its domain, the aggregate evaluates to false (denoted as $\bot$) and our implementation issues a warning.

[4] `#min` and `#max` over strings are not yet supported in the current implementation.

A *substitution* is a mapping from a set of variables to the set $U_\mathcal{P}$ of the constants in $\mathcal{P}$. A substitution from the set of global variables of a rule r (to $U_\mathcal{P}$) is a *global substitution for r*; a substitution from the set of local variables of a symbolic set S (to $U_\mathcal{P}$) is a *local substitution for* S. Given a symbolic set without global variables $S = \{\, Vars : Conj\}$, the *instantiation* of S is the following ground set of pairs $inst(S)$: $\{\langle \gamma(\,Vars) : \gamma(Conj)\rangle \mid \gamma$ *is a local substitution for* $S\}$.[5]
A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is applied over r; and (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Example 3. Consider the following program $\mathcal{P}_1$:

$$q(1) \vee p(2,2). \qquad q(2) \vee p(2,1).$$
$$t(X) \text{ :- } q(X), \#\mathtt{sum}\{Y : p(X,Y)\} > 1.$$

The instantiation $Ground(\mathcal{P}_1)$ is the following:

$$q(1) \vee p(2,2). \qquad q(2) \vee p(2,1).$$
$$t(1) \text{ :- } q(1), \#\mathtt{sum}\{\langle 1 : p(1,1)\rangle, \langle 2 : p(1,2)\rangle\} > 1.$$
$$t(2) \text{ :- } q(2), \#\mathtt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1.$$

■

Interpretations and models. An *interpretation* for a DLP$^\mathcal{A}$ program $\mathcal{P}$ is a set of standard ground atoms $I \subseteq B_\mathcal{P}$. The truth valuation $I(A)$, where A is a standard ground literal or a standard ground conjunction, is defined in the usual way. Besides assigning truth values to standard ground literals, an interpretation provides meaning also to (ground) sets, aggregate functions and aggregate literals; the meaning of a set, an aggregate function, and an aggregate atom under an interpretation, is a multiset, a value, and a truth value, respectively. Let $f(S)$ be a an aggregate function. The valuation $I(S)$ of the set S w.r.t. I is the multiset of the constants appearing in the first position of the first components of the elements in S whose conjunctions are true w.r.t. I. More precisely, let $S_I = \{\langle t_1, ..., t_n\rangle \mid \langle t_1, ..., t_n : Conj\rangle \in S \wedge Conj \text{ is true w.r.t. } I\}$, then $I(S)$ is the multiset $[\, t_1 \mid \langle t_1, ..., t_n\rangle \in S_I\,]$. The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of the function f on $I(S)$. If the multiset $I(S)$ is not in the domain of *f*, $I(f(S)) = \bot$.

An aggregate atom $A = Lg \prec_1 f(S) \prec_2 Rg$ is *true w.r.t.* I if: (i) $I(f(S)) \neq \bot$, and, (ii) the relationships $Lg \prec_1 I(f(S))$ and $I(f(S)) \prec_2 Rg$ hold whenever they are present; otherwise, A is false.

A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in Ground(\mathcal{P})$ is true w.r.t. M. A model M for $\mathcal{P}$ is (subset) *minimal* if no model N for $\mathcal{P}$ exists such that N is a proper subset of M.

Example 4. Consider the aggregate atom $A = \#\mathtt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1$ from Example 3. Let S be the ground set appearing in A. For the interpretation $I = \{q(2), p(2,2), t(2)\}$, $I(S) = [2]$, the application of $\#\mathtt{sum}$ over $[2]$ yields 2, and A is therefore true w.r.t. I, since $2 > 1$. I is a minimal model of the program of Example 3. ■

[5] Given a substitution σ and a DLP$^\mathcal{A}$ object O (rule, conjunction, set, etc.), with a little abuse of notation, we denote by $\sigma(O)$ the object obtained by replacing each variable X in O by $\sigma(X)$.

Answer Sets. First we define the answer sets of positive programs without aggregates, then we give a reduction of disjunctive datalog programs containing negation as failure and aggregates to positive programs without aggregates and use that to define answer sets of arbitrary disjunctive datalog programs (possibly containing negation as failure and aggregates).

An interpretation $X \subseteq B_{\mathcal{P}}$ is called *closed under* a positive disjunctive datalog program without aggregates $\mathcal{P}$, if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap X \neq \emptyset$ whenever $B(r) \subseteq X$. An interpretation $X \subseteq B_{\mathcal{P}}$ is an *answer set* for a positive $\mathrm{DLP}^{\mathcal{A}}$ program without aggregates $\mathcal{P}$, if it is minimal (under set inclusion) among all interpretations that are closed under $\mathcal{P}$.[6]

Example 5. The positive program $\mathcal{P}_1 = \{a \vee b \vee c.\}$ has the answer sets $\{a\}$, $\{b\}$, and $\{c\}$. Its extension $\mathcal{P}_2 = \{a \vee b \vee c.\ ,\ \text{:- } a.\}$ has the answer sets $\{b\}$ and $\{c\}$. Finally, the positive program $\mathcal{P}_3 = \{a \vee b \vee c.\ ,\ \text{:- } a.\ ,\ b \text{:- } c.\ ,\ c \text{:- } b.\}$ has the single answer set $\{b, c\}$. ∎

The *reduct* or *Gelfond-Lifschitz transform* of a ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^X$ obtained from $\mathcal{P}$ by

- deleting all rules $r \in \mathcal{P}$ for which a negative literal in $B(r)$ is false w.r.t. X or an aggregate literal is false w.r.t. X; and
- deleting the aggregate literals and the negative literals from the remaining rules.

An answer set of a program $\mathcal{P}$ is a set $X \subseteq B_{\mathcal{P}}$ such that X is an answer set of $Ground(\mathcal{P})^X$.

Example 6. Given the following program with negation and stratified aggregates $\mathcal{P}_4 =$

$$\begin{array}{l}\{d(1).\ ,\ a \vee b \text{:- } c.\ ,\\ b \text{:- } \texttt{not}\ a, \texttt{not}\ c, \#\texttt{count}\{Y : d(Y)\} > 0.\ ,\\ a \vee c \text{:- } \texttt{not}\ b, \#\texttt{sum}\{Y : d(Y)\} > 1.\}\end{array}$$

and $I = \{b, d(1)\}$, the reduct $\mathcal{P}_4^I$ is $\{d(1).\ ,\ a \vee b \text{:- } c.\ ,\ b.\}$. It is easy to see that I is an answer set of $\mathcal{P}_4^I$, and for this reason it is also an answer set of $\mathcal{P}_4$.

Now consider $J = \{a, d(1)\}$. The reduct $\mathcal{P}_4^J$ is $\{d(1).\ ,\ a \vee b \text{:- } c.\}$ and it can be easily verified that J is an answer set of $\mathcal{P}_4^J$, so it is also an answer set of $\mathcal{P}_4$.

For $K = \{c, d(1)\}$, on the other hand, the reduct $\mathcal{P}_4^K$ is equal to $\mathcal{P}_4^J$, but K is not an answer set of $\mathcal{P}_4^K$: for the rule $r : a \vee b \text{:- } c$, $B(r) \subseteq K$ holds, but $H(r) \cap K \neq \emptyset$ does not. Indeed, it can be verified that I and J are the only answer sets of $\mathcal{P}_4$. ∎

Example 7. Given the following program with unstratified aggregates

$$\mathcal{P}_5 = \{p(1) \text{:- } \#\texttt{count}\{X : p(X)\} < 2.\ ,\ p(2) \vee q(2).\}$$

and $I = \{p(1), p(2)\}$, the reduct $\mathcal{P}_5^I$ is $\{p(2) \vee q(2).\}$. It is easy to see that I is a model but not a minimal model of $\mathcal{P}_5^I$ and thus not an answer set of $\mathcal{P}_5^I$ nor $\mathcal{P}_5$. Now consider $J = \{p(1), q(2)\}$. The reduct $\mathcal{P}_5^J$ is $\{p(1).\ ,\ p(2) \vee q(2).\}$ and it can be easily verified that J is an answer set of $\mathcal{P}_5^J$, so it is also an answer set of $\mathcal{P}_5$. If, on the other hand, we take $K = \{p(2)\}$, the reduct $\mathcal{P}_5^K$ is equal to $\mathcal{P}_5^J$, but K is not a model of $\mathcal{P}_5^K$ as for the rule $r : p(1)$. We see that $B(r) \subseteq K$ trivially holds, but $H(r) \cap K \neq \emptyset$ obviously does not. Indeed, it can be verified that J is the only answer sets of $\mathcal{P}_5$. ∎

[6] Note that we only consider *consistent answer sets*, while in [GL91] also the inconsistent set of all possible literals can be a valid answer set.

3 Knowledge Representation in $\mathrm{DLP}^{\mathcal{A}}$

In this section, we show how aggregate functions can be used to encode several relevant problems: Team Building, Seating, and Products Control.

Team Building. A project team has to be built from a set of employees according to the following specifications:

- (p_1) The team consists of a certain number of employees.
- (p_2) At least a given number of different skills must be present in the team.
- (p_3) The sum of the salaries of the employees working in the team must not exceed the given budget.
- (p_4) The salary of each individual employee is within a specified limit.
- (p_5) The number of women working in the team has to reach at least a given number.

Suppose that our employees are provided by a number of facts of the form $emp(EmpId, Sex, Skill, Salary)$; the size of the team, the minimum number of different skills, the budget, the maximum salary, and the minimum number of women are specified by the facts $nEmp(N)$, $nSkill(N)$, $budget(B)$, $maxSal(M)$, and $women(W)$. We then encode each property p_i above by an aggregate atom A_i, and enforce it by an integrity constraint containing `not` A_i.

$in(I) \vee out(I)$:- $emp(I, Sx, Sk, Sa)$.
:- $nEmp(N)$, `not` $\#\mathtt{count}\{I : in(I)\} = N$.
:- $nSkill(M)$, `not` $\#\mathtt{count}\{Sk : emp(I, Sx, Sk, Sa), in(I)\} \geq M$.
:- $budget(B)$, `not` $\#\mathtt{sum}\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\} \leq B$.
:- $maxSal(M)$, `not` $\#\mathtt{max}\{Sa : emp(I, Sx, Sk, Sa), in(I)\} \leq M$.
:- $women(W)$, `not` $\#\mathtt{count}\{I : emp(I, f, Sk, Sa), in(I)\} \geq W$.

Intuitively, the disjunctive rule "guesses" whether an employee is included in the team or not, while the five constraints correspond one-to-one to the five requirements p_1-p_5. Thanks to the aggregates the translation of the specifications is surprisingly straightforward. The example highlights the usefulness of representing both sets and multisets in our language; the latter can be obtained by specifying more than one variable in $Vars$ of a symbolic set $\{Vars : Conj\}$). For instance, the encoding of p_2 requires a set, as we want to count *different* skills; two employees in the team having the same skill, should count once w.r.t. p_2. On the contrary, p_3 requires to sum the elements of a multiset; if two employees have the same salary, *both* salaries should be summed up for p_3. This is obtained by adding the variable I, which uniquely identifies every employee, to $Vars$. The valuation of $\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\}$ yields the set $S = \{\langle Sa, I\rangle : Sa \textit{ is the salary of employee } I \textit{ in the team}\}$. The sum function is then applied on the multiset of the first components Sa of the tuples $\langle Sa, I\rangle$ in S (see Section 2.2).

Seating. We have to generate a sitting arrangement for a number of guests, with m tables and n chairs per table. Guests who like each other should sit at the same table; guests who dislike each other should not sit at the same table.

Suppose that the number of chairs per table is specified by $nChairs(X)$ and that $person(P)$ and $table(T)$ represent the guests and the available tables, respectively. Then, we can gener-

ate a seating arrangement by the following program:

% Guess whether person P sits at table T or not.
$at(P,T) \vee not\_at(P,T)$:- $person(P), table(T)$.
% The persons sitting at a table cannot exceed the chairs.
:- $table(T), nChairs(C),$ `not` $\#\texttt{count}\{P : at(P,T)\} \leq C$.
% A person is seated at precisely one table; this is equivalent
% to :- $person(P), at(P,T), at(P,U), T <> U$.
:- $person(P),$ `not` $\#\texttt{count}\{T : at(P,T)\} = 1$.
% People who like each other should sit at the same table...
:- $like(P1,P2), at(P1,T),$ `not` $at(P2,T)$.
% ...while people who dislike each other should not.
:- $dislike(P1,P2), at(P1,T), at(P2,T)$.

Products Control. Given a set of desired products, a set of companies and a fixed budget, the problem consists of buying financial shares in this set of companies, within the given budget, such that the controlled companies produce all the desired products. The specifications are the following:

- (p_1) A product A is produced by us if it is produced by a company under our control.
- (p_2) A company C is under our direct control, if we bought more than 50% of its shares.
- (p_3) A company C is under our (indirect) control, if companies under our control (together) own more than 50% of C.
- (p_4) The majority of the shares of C can be reached by summing up the C shares we bought directly with the shares owned by the companies under our control.
- (p_5) Each desired product has to be produced.
- (p_6) The budget must not be exceeded.

Suppose that desired products and companies are provided by a number of facts of the form $desired(P)$ and $company(C)$, respectively, and suppose that the budget is specified by the fact $budget(B)$. Furthermore, let the relations between products P and producing companies C be given by facts $producedBy(P,C)$ and the fact that N% shares of company C are for sale at price P by $forSale(C,N,P)$. (For simplicity we assume that only one package of a fixed amount of shares is for sale per company.) Finally if company C_1 owns N% shares of company C_2, let a fact $shares(C_2,C_1,N)$ be defined.

Given this information, we encode the problem specified by properties p_1 to p_6 in the following way:

$bought(C,N) \vee notBought(C,N)$:- $company(C), forSale(C,N,Price)$.
$produced(A)$:- $producedBy(A,C), controlled(C)$.
$controlled(C)$:- $bought(C,N), N > 50$.
$controlled(C)$:- $company(C), \#\texttt{sum}\{N,C1 : shares(C,C1,N), controlled(C1)\} > 50$.
$controlled(C)$:- $bought(C,N), N \leq 50,$
$\#\texttt{sum}\{N,C1 : shares(C,C1,N), controlled(C1)\} > K, 50 = K + N$.
:- $desired(P),$ `not` $produced(P)$.
:- $budget(B), \#\texttt{sum}\{Price,C : forSale(C,N,Price), bought(C,N)\} > B$.

Intuitively, the disjunctive rule "guesses" whether a given number of shares of a company C has been bought or not. The other rules define when a product is produced by us and the

different conditions to control a company. The last two constraints correspond to requirements p_5 and p_6. Note that in all aggregates in this program the sums are computed over multisets, as different companies may hold equal percentages of shares and different shares for sale may cost equally much.

4 Computational Complexity of $\mathrm{DLP}^{\mathcal{A}}$

As for the classical nonmonotonic formalisms [MT91], two important decision problems, corresponding to two different reasoning tasks, arise in $\mathrm{DLP}^{\mathcal{A}}$:

Brave Reasoning: Given a $\mathrm{DLP}^{\mathcal{A}}$ program $\mathcal{P}$ and a ground literal L, is L true in some answer set of $\mathcal{P}$?
Cautious Reasoning: Given a $\mathrm{DLP}^{\mathcal{A}}$ program $\mathcal{P}$ and a ground literal L, is L true in all answer sets of $\mathcal{P}$?

The following theorems report on the complexity of the above reasoning tasks for propositional (i.e., variable-free) $\mathrm{DLP}^{\mathcal{A}}$ programs that respect the safety restrictions imposed in Section 2. Importantly, it turns out that reasoning in $\mathrm{DLP}^{\mathcal{A}}$ does not bring an increase in computational complexity, which remains exactly the same as for standard DLP.

Lemma 1. *Deciding whether an interpretation M is an answer set for a ground program $\mathcal{P}$ is in co-NP.*

Proof. We check in NP that M is not an answer set of $\mathcal{P}$ as follows. Guess a subset I of M, and verify that: (1) M is not a model for $\mathcal{P}$, or (2) $I \subset M$ and I is a model of the Gelfond-Lifschitz transform of $\mathcal{P}$ w.r.t. M.

The only difference w.r.t. the corresponding tasks of (1) and (2) in standard DLP, is the computation of the truth valuations of the aggregate atoms, which in turn require to compute the valuations of aggregate functions and sets. Computing the valuation of a ground set T requires scanning each element $\langle t_1, ..., t_n : Conj \rangle$ of T and adding t_1 to the result multiset if $Conj$ is true w.r.t. I. This is evidently polynomial, as is the application of the aggregate operators (#count, #min, #max, #sum, #times) on a multiset; the comparison of the guards with its result, finally, is straightforward.

Therefore, the tasks (1) and (2) are tractable as in standard DLP. Deciding whether M is not an answer set for $\mathcal{P}$ thus is in NP; consequently, deciding whether M is an answer set for $\mathcal{P}$ is in co-NP. □

Based on this lemma, we can identify the computational complexity of the main decision problems, brave and cautious reasoning.

Theorem 1. *Brave Reasoning on ground $\mathrm{DLP}^{\mathcal{A}}$ programs is Σ_2^P-complete.*

Proof. We verify that a ground literal L is a brave consequence of a $\mathrm{DLP}^{\mathcal{A}}$ program $\mathcal{P}$ as follows: Guess a set $M \subseteq B_{\mathcal{P}}$ of ground literals, check that (1) M is an answer set for $\mathcal{P}$, and (2) L is true w.r.t. M. Task (2) is clearly polynomial; while (1) is in co-NP by virtue of Lemma 1. The problem therefore lies in Σ_2^P.

Σ_2^P-hardness follows from the Σ_2^P-hardness of DLP [EGM97], since $\mathrm{DLP}^{\mathcal{A}}$ is a superset of DLP. □

The complexity of cautious reasoning follows by similar arguments as above.

Theorem 2. *Cautious Reasoning on ground* DLP$^{\mathcal{A}}$ *programs is* Π_2^P*-complete.*

Proof. We verify that a ground literal L is *not* a cautious consequence of a DLP$^{\mathcal{A}}$ program $\mathcal{P}$ as follows: Guess a set $M \subseteq B_{\mathcal{P}}$ of ground literals, check that (1) M is an answer set for $\mathcal{P}$, and (2) L is not true w.r.t. M. Task (2) is clearly polynomial; while (1) is in co-NP, by virtue of Lemma 1. Therefore, the complement of cautious reasoning is in Σ_2^P, and cautious reasoning is in Π_2^P.

Π_2^P-hardness again follows from [EG95], since DLP$^{\mathcal{A}}$ is a superset of DLP. □

5 Implementation Issues

The implementation of DLP$^{\mathcal{A}}$ required changes to all modules of DLV. Apart from a preliminary standardization phase, most of the effort concentrated on the Instantiation and Model Generator modules.

Standardization. After parsing, each aggregate A is transformed such that both guards are present and both $\prec_1$ and $\prec_2$ are set to $\leq$. The conjunction $Conj$ of the symbolic set of A is replaced by a single, new atom Aux and a rule Aux :- $Conj$ is added to the program (the arguments of Aux being the distinct variables of $Conj$).

Instantiation. The goal of the instantiator is to generate a ground program that has precisely the same answer sets as the theoretical instantiation $Ground(\mathcal{P})$, but is sensibly smaller. The instantiation proceeds bottom-up following the dependencies induced by the rules, and, in particular, respecting the ordering imposed by aggregate-stratification where applicable.

For aggregate-stratified components of the input program we proceed as follows. Let "H :- $B, aggr$." be a rule r, where H is the head of the rule, B is the conjunction of the standard body literals in r, and $aggr$ is an aggregate literal over a symbolic set $\{ Vars{:}Aux\}$. First we compute an instantiation $\overline{B}$ for the literals in B; this also binds the global variables appearing in Aux. The (partially bound) atom $\overline{Aux}$ is then matched against its extension (which is already available for aggregate-stratified rules since the bottom-up instantiation respects the stratification), all matching facts are computed, and a set of pairs $\{\langle\theta_1(Vars) : \theta_1(\overline{Aux})\rangle,...,\langle\theta_n(Vars) : \theta_n(\overline{Aux})\rangle\}$ is generated, where θ_i is a substitution for the local variables in $\overline{Aux}$ such that $\theta_i(\overline{Aux})$ is an admissible instance of $\overline{Aux}$. (Recall that DLV's instantiator produces only those instances of a predicate which can potentially become true [FLMP99,LPS01], where a ground atom A can potentially become true only if we have generated a ground instance of a rule with A in the head.) Also, we only store those elements of the symbolic set whose truth value cannot be determined yet and process the others dynamically, (partially) evaluating the aggregate already during instantiation. For instance, if Aux is defined by a disjunction-free stratified subprogram (where the truth values of all atoms are already determined during grounding), the aggregate is completely evaluated during the instantiation, its truth valuation is determined, and it is removed from the body of the rule instance. The same process is then repeated for all further instantiations of the literals in B.

Example 8. Consider the rule r: $p(X)$:- $q(X), 1 < \#\texttt{count}\{Y : a(X,Y), \texttt{not } b(Y)\}$. The standardization rewrites r to:

$$p(X) \text{ :- } q(X), 2 \leq \#\texttt{count}\{Y : aux(X,Y)\} \leq \infty.$$
$$aux(X,Y) \text{ :- } a(X,Y), \texttt{not } b(Y).$$

Suppose that the instantiation of the rule for aux generates 3 potentially true facts $aux(1,a)$, $aux(1,b)$, and $aux(2,c)$. If the potentially true facts for q are $q(1)$ and $q(2)$, the following ground instances are generated:

$$p(1) \text{ :- } q(1), 2 \leq \#\texttt{count}\{\langle a : aux(1,a)\rangle, \langle b : aux(1,b)\rangle\} \leq \infty.$$
$$p(2) \text{ :- } q(2), 2 \leq \#\texttt{count}\{\langle c : aux(2,c)\rangle\} \leq \infty.$$

Note that a ground set contains only those aux atoms which are potentially true. ∎

For a rule r with unstratified aggregates, we proceed similarly to the stratified case, but do not instantiate the symbolic set $\{Vars{:}Aux\}$ yet, as we are not guaranteed that the entire extension of $\overline{Aux}$ is already available (since $\overline{Aux}$ is recursive with the head, its extension is being generated during the evaluation of the component at hand, and new facts for $\overline{Aux}$ could still be produced). Thus, as in the stratified case we compute an instantiation $\overline{B}$ for the literals in B, but we do not instantiate the symbolic set in r. We only "prepare" the ground instance of r, storing a "container" for the set appearing in the aggregate function in a temporary memory location. Then, once the instantiation of the current component has been completed and we are sure that no further instance for $\overline{Aux}$ will be generated, we resume the instantiation process for r, compute all facts matching $\overline{Aux}$, and complete the generation of the set.

Duplicate Sets Recognition. To optimize the evaluation during instantiation and especially afterwards, we have designed a hashing technique which recognizes multiple occurrences of the same set in the program, even in different rules, and stores them only once. This saves memory (sets may be very large), and also implies a significant performance gain, especially in the model generation where sets are frequently manipulated during the backtracking process.

Example 9. Consider the following two constraints:

$$c_1 : \quad \text{:-} \quad 10 \leq \#\texttt{max}\{V : d(V,X)\}.$$
$$c_2 : \quad \text{:-} \quad \#\texttt{min}\{Y : d(Y,Z)\} \leq 5.$$

Our technique recognizes that the two sets are equal, and generates only one instance which is shared by c_1 and c_2.

Now assume that both constraints additionally contain a standard literal $p(T)$. In this case, c_1 and c_2 have n instances each, where n is the number of facts for $p(T)$. By means of our technique, each pair of instances of c_1 and c_2 shares a common set, reducing the number of instantiated sets by half. ∎

Note that also the program for the example Products Control in Section 3 contains two equal symbolic sets (appearing in the bodies of the second and the third rule with *controlled(C)* in the head), which would generate, once they are instantiated, several pairs of identical (ground) sets (one for each company C). Thanks to our hashing-based technique, the generation of these duplicates is prevented with a relevant efficiency gain.

Model Generation. We have designed an extension of the Deterministic Consequences operator of the DLV system [FLP99] for DLP$^{\mathcal{A}}$ programs. The new operator makes both forward and backward inferences on aggregate atoms, resulting in an effective pruning of the search space. We have then extended the Dowling and Gallier algorithm [DG84] to compute a fixpoint of this operator in linear time using a multi-linked data structure of pointers. Given a ground set T, say, $\{\langle t_1^1, ..., t_n^1 : Aux^1\rangle, ..., \langle t_1^m, ..., t_n^m : Aux^m\rangle\}$, this structure allows to access T in $O(1)$ whenever some Aux^i changes its truth value (supporting fast forward propagation); on the other hand, it provides direct access from T to each Aux^i atom (supporting fast backward propagation).

6 Experiments and Benchmarks

To assess the usefulness of the proposed DLP extension and evaluate its implementation, we compare the following two methods for solving a given problem:

- DLV$^{\mathcal{A}}$ Encode the problem in DLP$^{\mathcal{A}}$ and solve it by using our extension of DLV with aggregates.
- DLV Encode the problem in standard DLP and solve it by using standard DLV. To generate DLP encodings from DLP$^{\mathcal{A}}$ encodings, suitable logic definitions of the aggregate functions are employed (which are recursive for #count, #sum, and #times).

We compare these methods on two benchmark problems: **Time Tabling** is a classical planning problem. In particular, we consider the problem of planning the timetable of lectures which some groups of students have to take. We consider a number of real-world instances University of Calabria where instance k deals with k groups.

Seating is the problem described in Section 3. We consider 4 (for small instances) or 5 (for larger instances) seats per table, with increasing numbers of tables and persons (with $numPersons = numSeats * numTables$). For each problem size (i.e., seats/tables configuration), we consider classes with different numbers of like and dislike constraints, where the percentages are relative to the maximum numbers of like and dislike constraints, resp. such that the problem is not over-constrained.[7]

In particular, we consider the following classes: (-) no like/dislike constraints at all; (-) 25% like constraints; (-) 25% like and 25% dislike constraints; (-) 50% like constraints; (-) 50% like and 50% dislike constraints. For each problem size, we randomly generated 10 instances for each of these classes.

For Seating we use the DLP$^{\mathcal{A}}$ encoding reported in Section 3; all encodings and benchmark data are available at `http://www.dlvsystem.com/examples/ijcai03.zip`.

We ran the benchmarks on AMD Athlon 1.2 machines with 512MB of memory, using FreeBSD 4.7 and GCC 2.95. We allowed a maximum running time of 1800 seconds per instance and a maximum memory usage of 256MB. Cumulated results are provided in Figure 1. In particular, for Timetabling we report the execution time and the size of the residual ground instantiation (the total number of atoms occurring in the instantiation, where multiple occurrences of the same atom are counted).[8] For Seating, the execution time is the average

[7] Beyond these maxima there is trivially no solution.

[8] Note that also atoms occurring in the sets of the aggregates are counted for the instantiation size.

Number of Groups	Exec. Time DLV	Exec. Time DLV$^{\mathcal{A}}$	Instantiation Size DLV	Instantiation Size DLV$^{\mathcal{A}}$
1	10.95	0.55	91217	6972
2	36.79	2.05	178533	13986
3	79.84	4.68	264938	20888
4	147.53	7.86	367014	29029
5	224.46	12.30	436544	36043
6	321.85	17.18	518950	42767
7	437.94	25.36	606361	49993
8	618.23	37.78	761429	61916
9	-	57.00	-	74027

Number of Persons	Exec. Time DLV	Exec. Time DLV$^{\mathcal{A}}$	Instantiation Size DLV	Instantiation Size DLV$^{\mathcal{A}}$
8	0.010	0.010	320	101
12	0.034	0.010	996	248
16	26.872	0.011	2272	490
25	-	0.024	6643	1346
50	-	0.307	50029	7559
75	-	1.883	165442	22049
100	-	7.082	387886	47946
125	-	64.293	752769	88781
150	-	152.450	1294977	147567

Fig. 1. Experimental Results for Timetabling and Seating

running time over the instances of the same size. A "-" symbol in the tables indicates that the corresponding instance (some of the instances of that size, for Seating) was not solved within the allowed time and memory limits.

On both problems, DLV$^{\mathcal{A}}$ clearly outperforms DLV. On Timetabling, the execution time of DLV$^{\mathcal{A}}$ is one order of magnitude lower than that of DLV on all problem instances, and DLV could not solve the last instances within the allowed memory and time limits. On Seating, the difference becomes even more significant. DLV could solve only the instances of small size (up to 16 persons - 4 tables, 4 seats), while DLV$^{\mathcal{A}}$ could solve significantly larger instances in a reasonable time. The information about the instantiation sizes provides an explanation for such a big difference between the execution times of DLV and DLV$^{\mathcal{A}}$. Thanks to the aggregates, the DLP$^{\mathcal{A}}$ encodings are more succinct than the corresponding encodings in standard DLP. This succinctness is also reflected in the ground instantiations of the programs. Since the evaluation algorithms are then exponential in the size of the instantiation (in the worst case), the execution times of DLV$^{\mathcal{A}}$ turn out to be much shorter than those of DLV.

7 Related Works

Aggregate functions in logic programming languages appeared already in the 1980s, when their need emerged in deductive databases like LDL [CGK^{+}90] and were studied in detail, cf. [RS97,KR98]. However, the first implementation in Answer Set Programming, in the Smodels system, is recent [SNS02].

Comparing DLP$^{\mathcal{A}}$ to the language of Smodels, we observe a strong similarity between cardinality constraints there and $\#\texttt{count}$. Also $\#\texttt{sum}$ and the weight constraints of Smodels are similar in spirit. Indeed, the DLP$^{\mathcal{A}}$ encodings of both Team Building and Seating can be easily translated to Smodels' language. However, there are some relevant differences. For instance, in DLP$^{\mathcal{A}}$ aggregate atoms can be negated, while cardinality and weight constraint literals in Smodels cannot.

Negated aggregates are useful for a more direct knowledge representation, and allow to express, for instance, that some value should be external to a given range. For example, $\texttt{not}\ 3 \leq \#\texttt{count}\{X : p(X)\} \leq 7$ is true if the number of true facts for p is in $[0, 3[\cup]7, \infty[$; for expressing the same property in Smodels one needs to use two cardinality constraints.

Smodels, on the other hand, allows for weight constraints in the heads of rules, while DLP$^{\mathcal{A}}$ aggregates cannot occur in heads. (The presence of weight constraints in heads is a

powerful KR feature, which allows, for instance, to "guess" an arbitrary subset of a given set; however, it causes the loss of some semantic property of nonmonotonic languages [MR02].)

DLP$^{\mathcal{A}}$ aggregates like #min, #max, and #times do not have a counterpart in Smodels. Moreover, DLP$^{\mathcal{A}}$ provides a general framework where further aggregates can be easily accommodated (e.g., #any and #avg are already under development). Furthermore, note that symbolic sets of DLP$^{\mathcal{A}}$ directly represent pure (mathematical) sets, and can also represent multisets rather naturally (see the discussion on Team Building in Section 3). Smodels weight constraints, on the other hand, work on multisets, and additional rules are needed to encode pure sets; for instance, Condition p_2 of Team Building cannot be directly encoded in a constraint, but needs the definition of an extra predicate. Thanks to stricter safety conditions (all variables are to be restricted by *domain predicates*), like DLP$^{\mathcal{A}}$, Smodels is able to deal with recursion through aggregates even though its instantiator is less sophisticated.

Finally, note that DLP$^{\mathcal{A}}$ deals with sets of terms, while Smodels deals with sets of atoms. As far as the implementation is concerned, Smodels, too, is endowed with advanced pruning operators for weight constraints, which are efficiently implemented. We are not aware, though, of techniques for the automatic recognition of duplicate sets in Smodels.

DLP$^{\mathcal{A}}$ also seems to be very similar to a special case of the semantics for aggregates discussed in [Gel02], which we are currently investigating.

Another interesting research line uses 4-valued logics and approximating operators to define the semantics of aggregate functions in logic-based languages [DPB01,DMT02,Pel02]. These approaches are founded on very solid theoretical grounds, and appear very promising, as they could provide a clean formalization of a very general framework for arbitrary aggregates in logic programming and nonmonotonic reasoning, where aggregate atoms can also "produce" new values (currently, both DLP$^{\mathcal{A}}$ and Smodels require the guards of aggregates to be bound to some value). These approaches sometimes amount to a higher computational complexity [Pel02] and have not been implemented so far. However, on the common fragment we believe our language and semantics to be in sync with those defined in [Pel02].

8 Conclusion

We have proposed DLP$^{\mathcal{A}}$, an extension of DLP by aggregate functions, and have implemented it in the DLV system. On the one hand, we have demonstrated that the aggregate functions increase the knowledge modeling power of DLP, supporting a more natural and concise knowledge representation. On the other hand, we have shown that aggregate functions do not increase the complexity of the main reasoning tasks. Moreover, experiments have confirmed that the succinctness of the encodings employing aggregates has a strong positive impact on the efficiency of the computation.

Future work will concern the introduction of further aggregate operators like #any ("Is there any matching element in the set?") and #avg, as well as the design of further optimization techniques and heuristics to improve the efficiency of the computation.

References

[CGK+90] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *IEEE TKDE*, 2(1), 1990.

[DFI+03] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *IJCAI 2003*, pp. 847–852, Acapulco, Mexico, August 2003. Morgan Kaufmann.

[DG84] W. F. Dowling and J. H. Gallier. Linear-time Algorithms for Testing the Satisfability of Propositional Horn Formulae. *JLP*, 3:267–284, 1984.

[DMT02] M. Denecker, V. Marek, and M. Truszczyński. Ultimate Approximations in Monotonic Knowledge Representation Systems. In *KR-2002*, pp. 177–188.

[DPB01] M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate Well-Founded and Stable Model Semantics for Logic Programs with Aggregates. In *ICLP-2001*, pp. 212–226. Springer, 2001.

[EFLP00] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In *Logic-Based Artificial Intelligence*, pp. 79–103. Kluwer, 2000.

[EG95] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *AMAI*, 15(3/4):289–323, 1995.

[EGM97] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM TODS*, 22(3):364–418, September 1997.

[FLMP99] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In *DDLP'99*, pp. 135–139.

[FLP99] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. In *LPNMR'99*, pp. 177–191. Springer.

[Gel02] M. Gelfond. Representing Knowledge in A-Prolog. In *Computational Logic. Logic Programming and Beyond*, LNCS 2408, pp. 413–451. Springer, 2002.

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[KR98] D. B. Kemp and K. Ramamohanarao. Efficient Recursive Aggregation and Negation in Deductive Databases. *IEEE TKDE*, 10:727–745, 1998.

[LPS01] N. Leone, S. Perri, and F. Scarcello. Improving ASP Instantiators by Join-Ordering Methods. In *LPNMR'01*, LNAI 2173. Springer, September 2001.

[MR02] V.W. Marek and J.B. Remmel. On Logic Programs with Cardinality Constraints. In *NMR'2002*, pp. 219–228, April 2002.

[MT91] V.W. Marek and M. Truszczyński. Autoepistemic Logic. *JACM*, 38(3):588–619, 1991.

[Pel02] N. Pelov. Non-monotone Semantics for Logic Programs with Aggregates. `http://www.cs.kuleuven.ac.be/~pelov/papers/nma.ps.gz`., October 2002.

[RS97] K. A. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *JCSS*, 54(1):79–97, February 1997.

[SNS02] P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.