TLP **8** (5 & 6): 545–580, 2008. © 2008 Cambridge University Press
doi:10.1017/S1471068408003323 First published online 8 May 2008 Printed in the United Kingdom

*Design and implementation of aggregate functions in the DLV system**

WOLFGANG FABER, GERALD PFEIFER, NICOLA LEONE,
TINA DELL'ARMI and GIUSEPPE IELPA

Department of Mathematics, University of Calabria
87036 Rende (CS), Italy
(*e-mail:* {faber,gerald,leone,dellarmi,ielpa}@mat.unical.it)

submitted 30 June 2007; revised 28 December 2007; accepted 1 February 2008

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, times, or count) on a set or multiset of elements, which satisfy some conditions, cannot be naturally expressed in classic DLP. To overcome this deficiency, we extend DLP by aggregate functions in a conservative way. In particular, we avoid the introduction of constructs with disputed semantics, by requiring aggregates to be stratified. We formally define the semantics of the extended language (called $\mathrm{DLP}^{\mathcal{A}}$), and illustrate how it can be profitably used for representing knowledge. Furthermore, we analyze the computational complexity of $\mathrm{DLP}^{\mathcal{A}}$, showing that the addition of aggregates does not bring a higher cost in that respect. Finally, we provide an implementation of $\mathrm{DLP}^{\mathcal{A}}$ in DLV—a state-of-the-art DLP system—and report on experiments which confirm the usefulness of the proposed extension also for the efficiency of computation.

KEYWORDS: disjunctive logic programming, answer set programming, aggregates, knowledge representation, implementation

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 Minker (1982). This language is very expressive in a precise mathematical sense: under the answer set semantics Gelfond and Lifschitz (1991), it allows to express every property of finite structures that is decidable in the complexity class $\Sigma_2^P (= \mathrm{NP}^{\mathrm{NP}})$ Eiter *et al.* (1997). Therefore, under widely believed assumptions, DLP is strictly more expressive than *normal* (*disjunction-free*) logic programming, whose expressiveness is limited to

* A preliminary version of this work appeared in the Proceedings of IJCAI-03.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

properties decidable in NP, and it can express problems which cannot be translated to satisfiability of Conjunctive Normal Form (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 arguably more natural fashion (cf. Eiter *et al.* 2000).

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 arithmetic operators such as count, sum, or min 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). Using standard DLP, one first has to define an order over the employees, yielding a successor relation. Then, one has to define a *sum* predicate in a recursive way using this successor relation, computing the sum of all salaries, and compare its result with the given budget. This approach has two drawbacks: (1) It is bad from the Knowledge Representation (KR) perspective, as the encoding is not immediate and not natural at all. In particular, an ordering or successor relation often is not available and has to be provided in an explicit manner; (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 outlined deficiency of DLP. Instead of inventing new constructs from scratch, as in some approaches in the literature (e.g., Simons *et al.* 2002), we extend the language with aggregate functions, like those studied in the context of databases, and implement them in DLV Leone *et al.* (2006)—a state-of-the-art disjunctive logic programming system. The main advantages of this approach are that extensibility of the language (both syntactically and semantically) is straightforward, that aggregate functions are widely used, for instance in database query languages, and that many issues arising from the use of aggregates are well understood.

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$^{\mathcal{A}}$. Actually, we introduce aggregates in the full DLV language, that is, DLP$^{\mathcal{A}}$ includes also weak constraints Buccafurri *et al.* (2000).
- We address knowledge representation issues, showing the impact of the new constructs and describing ways how they can be employed profitably on relevant problems. We also highlight the usefulness of *assignment aggregates*, a new feature of DLP$^{\mathcal{A}}$, which is not supported by other answer set programming (ASP) systems with aggregates.
- We analyze the computational complexity of DLP$^{\mathcal{A}}$. We consider DLP$^{\mathcal{A}}$ programs with and without weak constraints. Importantly, it turns out that in both cases the addition of (stratified) aggregates does not increase the

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

computational complexity, which remains the same as for reasoning on aggregate-free 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 proposed in the literature.

The result of this work is a concrete and powerful tool for knowledge representation and reasoning, enhancing the modeling features of standard DLP and ASP systems.

DLP$^{\mathcal{A}}$, as described in this paper, requires aggregates to be stratified, that is, predicates defined by means of aggregates are not allowed to mutually depend on each other. The reason is that the set of stratified aggregate programs is the largest class on which all major semantics proposed in the literature coincide. Moreover, the introduction of unstratified aggregates causes a computational overhead in some cases, while the computational complexity of the reasoning tasks remains the same if stratified aggregates are introduced. (See Section 7.1 for a discussion about this issue.)

It is worthwhile noting that, compared with other implementations of aggregates in DLP and ASP, the language of our system supports some extra features which turn out to be very useful in practice for KR applications. For instance, the *Fastfood problem*, described in Section 3, is represented naturally and compactly in our language, while its encoding in the language of other DLP and ASP systems seems to be more involved causing computation to be dramatically less efficient, due to their more severe safety restrictions (domain predicates), and also due to the lack of the "min" aggregate function (see Section 7.2).

This paper is organized as follows: Section 2 illustrates the DLP$^{\mathcal{A}}$ language, providing a formal specification of both the syntax and the semantics of our extension of DLP with aggregates. Section 3 addresses knowledge representation issues, showing the profitable employment of aggregate functions in a couple of examples. Section 4 analyzes the computational complexity of the DLP$^{\mathcal{A}}$ language. Section 5 addresses some implementation issues. Section 6 reports on the results of the experimentation activity. Section 7 discusses related works. Finally, in Section 8 we draw our conclusions.

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. DLP$^{\mathcal{A}}$ is an extension of the language of the DLV system by set-oriented (or aggregate) functions. Specifically, DLP$^{\mathcal{A}}$ includes disjunction, default (or nonmonotonic) negation, integrity and weak constraints, and aggregates.[1] For

[1] We do not treat strong negation explicitly. DLV supports this by a simple rewriting technique, adding a constraint $:- a, -a$ for each strongly negated atom $-a$, where a also occurs in the program.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

further background, we refer to Gelfond and Lifschitz (1991), Baral (2003), and Leone *et al.* (2006).

2.1 Syntax

We assume sets of variables, constants, and predicates to be given. Similar to Prolog, we assume *variables* to be strings starting with uppercase letters and *constants* to be nonnegative integers or strings starting with lowercase letters. *Predicates* are strings starting with lowercase letters or symbols such as $=$, $<$, and $>$ (so-called built-in predicates that have a fixed meaning). An *arity* (nonnegative integer) is associated with each predicate.

Standard atoms and literals. A *term* is either a variable or a constant. A *standard atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A *standard literal* L is either a standard atom A (in this case, it is *positive*) or a standard atom A preceded by the default negation symbol *not* (in this case, it is *negative*). A conjunction of standard literals is of the form $L_1, \ldots, L_k$, where each L_i $(1 <= i <= k)$ is a standard literal.

A structure (e.g., standard atom, standard literal, and conjunction) is ground, if neither the structure itself nor any substructures contain any variables.

Sets. 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 comma-separated list of variables and *Conj* is a conjunction of standard literals. Intuitively, a symbolic set $\{X : a(X, Y), not\ p(Y)\}$ stands for the set of X-values making the conjunction $a(X, Y), not\ p(Y)$ true, i.e., $\{X : \exists Y\ such\ that\ a(X, Y) \wedge not\ p(Y)\ is\ true\}$; see Section 2.3 for details.

A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj \rangle$, where $\bar{t}$ is a list of constants and *Conj* is a ground conjunction of standard literals.

Aggregate functions and aggregate atoms. An *aggregate function* is of the form $f(S)$, where S is a set and f is a *function name* among `#count`, `#min`, `#max`, `#sum`, and `#times`. An *aggregate atom* is

$$Lg \prec_1 f(S) \prec_2 Rg,$$

where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leqslant, >, \geqslant\}$, 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. In this case, "$0 \leqslant$" and "$\leqslant +\infty$" are assumed, respectively. If both $\prec_1$ and $\prec_2$ are present, we assume for simplicity that $\prec_1 \in \{<, \leqslant\}$ if and only if $\prec_2 \in \{<, \leqslant\}$ and that both $\prec_1$ and $\prec_2$ are different from $=$.[2]

Example 1
The following are two aggregate atoms. The latter contains a ground set and can be a ground instance of the former.

$$\#\mathtt{max}\{Z : r(Z), a(Z, V)\} > Y$$
$$\#\mathtt{max}\{\langle 2 : r(2), a(2, x)\rangle, \langle 2 : r(2), a(2, y)\rangle\} > 1$$

[2] The aggregates not considered are of limited importance, as they impose two upper or two lower guards, of which one will be redundant.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

(General) Atoms, literals, and rules. An *atom* is either a standard atom or an aggregate atom.

A *literal* L is an atom A (positive literal) or an atom A preceded by the default negation symbol *not* (negative literal). If A is an aggregate atom, L is an *aggregate literal*.

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

$$a_1 \vee \cdots \vee a_n \text{ :- } b_1, \ldots, b_k, b_{k+1}, \ldots, b_m,$$

where $a_1, \ldots, a_n$ are standard atoms, $b_1, \ldots, b_k$ are positive literals, and $b_{k+1}, \ldots, b_m$ are negative literals, and $n \geqslant 0$, $m \geqslant k \geqslant 0$, $m+n \geqslant 1$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_m$ is the *body* of r, $b_1, \ldots, b_k$ being the *positive body* and $b_{k+1}, \ldots, b_m$ the *negative body*. We define $H(r) = \{a_1, \ldots, a_n\}$, $B(r) = \{b_1, \ldots, b_m\}$, $B^+(r) = \{b_1, \ldots, b_k\}$, and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. A rule without head literals (i.e., $n = 0$) is usually referred to as an *integrity constraint*. A rule with an empty body (i.e., $m = 0$) is called a fact, and we usually omit the " :- " sign in this case.

Weak constraints. The language of DLV, which we enhance by aggregates in this paper, extends disjunctive Datalog by another construct called *weak constraint* Buccafurri *et al.* (2000). The $\mathrm{DLP}^{\mathcal{A}}$ language allows for a general form of weak constraints also including aggregate literals.

We define weak constraints as a variant of integrity constraints. In order to differentiate these two, weak constraints use the symbol ":∼" instead of " :- ". In addition, a weight and a priority level inducing a partial order among weak constraints are specified.

Formally, a weak constraint wc is an expression of the form

$$:\sim\ b_1, \ldots, b_k, b_{k+1}, \ldots, b_m.\ [w : l],$$

where $b_1, \ldots, b_k$ are positive literals, $b_{k+1}, \ldots, b_m$ are negative literals, and w (the *weight*) and l (the *level*, or *layer*) are positive integer constants or variables. For convenience, w, l, or both can be omitted and set to 1 by default in this case.

$\mathrm{DLP}^{\mathcal{A}}$ ***programs.*** A *(*$\mathrm{DLP}^{\mathcal{A}}$*) program* $\mathcal{P}$ (*program*, for short) is a set of $\mathrm{DLP}^{\mathcal{A}}$ rules (possibly including integrity constraints) and weak constraints. For a program $\mathcal{P}$, let $Rules(\mathcal{P})$ denote the set of rules (including integrity constraints), and let $WC(\mathcal{P})$ denote the set of weak constraints in $\mathcal{P}$. A program is *positive* if it does not contain any negative literal.

2.2 Syntactic restrictions and notation

We begin with two notions of stratification, which make use of the concept of a level mapping. Functions $||\ ||$ from predicates in a program $\mathcal{P}$ to finite ordinals are called *level mappings* of $\mathcal{P}$.

Negation-stratification. A program $\mathcal{P}$ is called *negation-stratified* Apt *et al.* (1988); Przymusinski (1988) if there is a level mapping $||\ ||_n$ of $\mathcal{P}$ such that for each pair of p and p' of predicates of $\mathcal{P}$ and every rule r of $\mathcal{P}$,

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

1. if p occurs in $B^+(r)$ and p' occurs in $H(r)$, then $||p||_n \leqslant ||p'||_n$;
2. if p occurs in $B^-(r)$ and p' occurs in $H(r)$, then $||p||_n < ||p'||_n$; and
3. if p and p' occur in $H(r)$, then $||p||_n = ||p'||_n$.

Aggregate-stratification. The idea of aggregate-stratification is that two predicates defined by means of aggregates do not mutually depend on one another.

A DLP$^{\mathscr{A}}$ program $\mathscr{P}$ is *aggregate-stratified* if there exists a level mapping $||\ ||_a$ such that for each pair of p and p' of predicates of $\mathscr{P}$, and for each rule $r \in \mathscr{P}$,

1. if p occurs in a standard atom in $B(r)$ and p' occurs in $H(r)$, then $||p||_a \leqslant ||p'||_a$;
2. if p occurs in an aggregate atom in $B(r)$, and p' occurs in $H(r)$, then $||p||_a < ||p'||_a$; and
3. if p and p' occur in $H(r)$, then $||p||_a = ||p'||_a$.

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

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

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 $b(X) \text{ :- } p(X)$, no such level mapping exists and the program becomes aggregate-unstratified, as in this case a level mapping would have to satisfy $||q|| > ||b|| \geqslant ||p|| \geqslant ||q||$, hence $||q|| > ||q||$. ■

Intuitively, aggregate-stratification forbids recursion through aggregates. It guarantees that the semantics of aggregates is agreed upon and coherent with the intuition, while the semantics of aggregate-unstratified programs is debatable, and some semantic properties (e.g., existence of answer sets for positive programs) are usually lost. For a more detailed discussion, see Section 7.1.

Local and global variables—Safety. For simplicity, and without loss of generality, we assume that the body of each rule and weak constraint contains at most one aggregate atom.[3] A *local* variable of a rule r is a variable appearing solely in an aggregate function in r; a variable of r which is not local is called *global*. A *nested* atom of r is an atom appearing in an aggregate atom of r; an atom of r which is not nested is called *unnested*.

A rule or weak constraint r is *safe* if the following conditions hold: (i) each global variable of r appears in a positive unnested standard literal of 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 $Conj$. Finally, a program is safe if all of its rules and weak constraints are safe.[4]

Condition (i) is the standard safety condition adopted in Datalog to guarantee that the variables are range-restricted Ullman (1989), while condition (ii) is specific to aggregates.

[3] Note that we do this only to simplify the definitions; our implementation can deal with multiple aggregates in one rule.
[4] Note that the safety restrictions apply also to aggregate-free rules and constraints.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

Example 3
Consider the following rules:

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

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 global variable T violates condition (i). ■

We assume in the following that DLP$^{\mathcal{A}}$ programs are safe and aggregate-stratified, unless explicitly stated otherwise.

2.3 Semantics

Let us first define some notation which is to be used subsequently. Given a DLP$^{\mathcal{A}}$ program $\mathscr{P}$, let $U_{\mathscr{P}}$ denote the set of constants appearing in $\mathscr{P}$, $U_{\mathscr{P}}^{\mathcal{N}} \subseteq U_{\mathscr{P}}$ the set of the natural numbers occurring in $U_{\mathscr{P}}$, and $B_{\mathscr{P}}$ the set of standard atoms constructible from the (standard) predicates of $\mathscr{P}$ with constants in $U_{\mathscr{P}}$. Given a set X, let $\overline{2}^X$ denote the set of all multisets over elements from X.

Let us next describe the domains and the meanings of the aggregate functions considered in this work:

#count: defined over $\overline{2}^{U_{\mathscr{P}}}$, the number of elements in the set.
#sum: defined over $\overline{2}^{U_{\mathscr{P}}^{\mathcal{N}}}$, the sum of the numbers in the set; 0 in case of the empty set.
#times: over $\overline{2}^{U_{\mathscr{P}}^{\mathcal{N}}}$, the product of the numbers in the set; 1 for the empty set.
#min, **#max**: defined over $\overline{2}^{U_{\mathscr{P}}^{\mathcal{N}}} - \{\emptyset\}$, the minimum/maximum element in the set.[5]

If the argument of an aggregate function does not belong to its domain, the aggregate evaluates to false (denoted as $\perp$).

Instantiation. A *substitution* is a mapping from a set of variables to the set $U_{\mathscr{P}}$ of the constants in $\mathscr{P}$. A substitution from the set of global variables of a rule r (to $U_{\mathscr{P}}$) is a *global substitution for* r; a substitution from the set of local variables of a symbolic set S (to $U_{\mathscr{P}}$) 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 $inst(S)$: $\{\langle\gamma(Vars) : \gamma(Conj)\rangle \mid \gamma \text{ is a local substitution for } S\}$.[6]

A *ground instance* of a rule or a weak constraint r is obtained in two steps: (1) a global substitution σ for r is applied to r and (2) every symbolic set S in $\sigma(r)$ is then replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathscr{P})$ of a program $\mathscr{P}$ is the set of all possible instances of the rules and the weak constraints of $\mathscr{P}$.

[5] Note that #min and #max can be easily extended to the domain of the strings by considering the lexicographic ordering.
[6] Given a substitution σ and a DLP$^{\mathcal{A}}$ object O (rule, conjunction, set, etc.), we denote by $\sigma(O)$ the object obtained by replacing each variable X in O by $\sigma(X)$.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

Example 4
Consider the following program $\mathscr{P}_1$:

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

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

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

For any program $\mathscr{P}$, $Ground(\mathscr{P})$ denotes the set $GroundRules(\mathscr{P}) \cup GroundWC(\mathscr{P})$, where $GroundRules(\mathscr{P}) = \bigcup_{r \in Rules(\mathscr{P})} Ground(r)$ and $GroundWC(\mathscr{P}) = \bigcup_{w \in WC(\mathscr{P})} Ground(w)$. Note that for propositional programs, $\mathscr{P} = Ground(\mathscr{P})$ holds.

Interpretations and models. An *interpretation* of a DLP$^{\mathscr{A}}$ program $\mathscr{P}$ is a set of standard ground atoms $I \subseteq B_{\mathscr{P}}$. 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 ground aggregate function. The valuation of the (ground) set S w.r.t. I is the multiset $I(S)$ defined as follows: 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 obtained as the projection of the tuples of S_I on their first constant, that is $I(S) = [t_1 \mid \langle t_1, ..., t_n\rangle \in S_I]$.

The valuation $I(f(S))$ of a ground aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \bot$ (where $\bot$ is a fixed symbol not occurring in $\mathscr{P}$).

A ground 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[7]; otherwise, A is false.

Example 5
Let I be the interpretation $\{f(1), g(1,2), g(1,3), g(1,4), g(2,4), h(2), h(3), h(4)\}$. With respect to the interpretation I, and assuming that all variables are local, we have that:

- $\#\mathtt{count}\{X : g(X,Y)\} > 2$ is false, because S_I for the corresponding ground set is $\{\langle 1\rangle, \langle 2\rangle\}$, so $I(S) = [1,2]$ and $\#\mathtt{count}([1,2]) = 2$.
- $\#\mathtt{count}\{X, Y : g(X,Y)\} > 2$ is true, because $S_I = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle\}$, $I(S) = [1,1,1,2]$ and $\#\mathtt{count}([1,1,1,2]) = 4$.
- $23 < \#\mathtt{times}\{Y : f(X), g(X,Y)\} \leqslant 24$ is true; in this case $S_I = \{\langle 2\rangle, \langle 3\rangle, \langle 4\rangle\}$, $I(S) = [2,3,4]$ and $\#\mathtt{times}([2,3,4]) = 24$.

[7] Note that in the implemented system (cf. Section 5) an error will be produced if Lg or Rg are not in $U_{\mathscr{P}}^{\mathscr{N}}$.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

- $\#\texttt{sum}\{A : g(A,B), h(B)\} \leqslant 3$ is true, as we have that $S_I = \{\langle 1\rangle, \langle 2\rangle\}$, $I(S) = [1,2]$ and $\#\texttt{sum}([1,2]) = 3$.
- $\#\texttt{sum}\{A,B : g(A,B), h(B)\} \leqslant 3$ is false, since $S_I = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle\}$, $I(S) = [1,1,1,2]$ and $\#\texttt{sum}([1,1,1,2]) = 5$.
- $\#\texttt{min}\{X : f(X), g(X)\} \geqslant 2$ is false because the evaluation of (the instantiation of) $\{X : f(X), g(X)\}$ w.r.t. I yields the empty set, which does not belong to the domain of $\#\texttt{min}$ (we have that $I(\#\texttt{min}\{\}) = \bot$). ■

A ground rule $r \in \mathit{GroundRules}(\mathscr{P})$ is *satisfied w.r.t. I* if some head atom is true w.r.t. I whenever all body literals are true w.r.t. I. (If r is an integrity constraint, r is satisfied iff its body is false.) A ground weak constraint $w \in \mathit{GroundWC}(\mathscr{P})$ is *satisfied w.r.t. I* if some body literal of w is false w.r.t. I.

A *model* of $\mathscr{P}$ is an interpretation M of $\mathscr{P}$ such that every rule $r \in \mathit{GroundRules}(\mathscr{P})$ is satisfied w.r.t. M. A model M of $\mathscr{P}$ is (subset) *minimal* if no model N of $\mathscr{P}$ exists such that N is a proper subset of M.[8]

Example 6
Consider the aggregate atom $A = \#\texttt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1$ from Example 4. 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 $\#\texttt{sum}$ over $[2]$ yields 2, and A is therefore true w.r.t. I, since $2 > 1$. Indeed, one can verify that I is a minimal model of the program of Example 4. ■

Answer sets. We define the answer sets of DLP$^{\mathscr{A}}$ programs in three steps, using their ground instantiation. First we define the answer sets of standard positive programs (i.e., programs without aggregates and without weak constraints), then we give a reduction of DLP$^{\mathscr{A}}$ programs containing aggregates and negation as failure to standard positive ones and use it to define answer sets of arbitrary sets of rules, possibly containing aggregates and negation as failure. Finally, we specify how weak constraints affect the semantics, arriving at the semantics of general DLP$^{\mathscr{A}}$ programs (with negation, aggregates, and weak constraints).

Step 1 An interpretation $X \subseteq B_{\mathscr{P}}$ is an *answer set* of a standard positive DLP$^{\mathscr{A}}$ program (without aggregates and weak constraints) $\mathscr{P}$, if it is a minimal model of $\mathscr{P}$.

Example 7
The positive program $\mathscr{P}_1 = \{a \vee b \vee c.\}$ has the answer sets $\{a\}$, $\{b\}$, and $\{c\}$. Its extension $\mathscr{P}_2 = \{a \vee b \vee c.,\ \ :\!-\, a.\}$ has the answer sets $\{b\}$ and $\{c\}$. Finally, the positive program $\mathscr{P}_3 = \{a \vee b \vee c.,\ \ :\!-\, a.,\ b :\!-\, c.,\ c :\!-\, b.\}$ has the single answer set $\{b,c\}$. ■

Step 2 The *reduct* or *Gelfond–Lifschitz transform* of a DLP$^{\mathscr{A}}$ program $\mathscr{P}$ w.r.t. a set $X \subseteq B_{\mathscr{P}}$ is the standard positive ground program $\mathscr{P}^X$ obtained from $\mathit{GroundRules}(\mathscr{P})$ by

[8] Note that a model can violate weak constraints.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

- deleting all rules $r \in \mathit{GroundRules}(\mathscr{P})$ 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 all negative literals and aggregate literals from the remaining rules.

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

Example 8
Given the following aggregate-stratified program with negation $\mathscr{P}_4 =$

$$\{d(1).,\ a \vee b \coloneq c.,\\ b \coloneq \mathit{not}\ a, \mathit{not}\ c, \#\mathtt{count}\{Y : d(Y)\} > 0.,\\ a \vee c \coloneq \mathit{not}\ b, \#\mathtt{sum}\{Y : d(Y)\} > 1.\}$$

and $I = \{b, d(1)\}$, the reduct $\mathscr{P}_4^I$ is $\{d(1).,\ a \vee b \coloneq c.,\ b.\}$. It is easy to see that I is an answer set of $\mathscr{P}_4^I$, and thus an answer set of $\mathscr{P}_4$ as well.

Now consider $J = \{a, d(1)\}$. The reduct $\mathscr{P}_4^J$ is $\{d(1).,\ a \vee b \coloneq c.\}$. It can be easily verified that J is a model of $\mathscr{P}_4^J$. However, also $J' = \{d(1)\} \subset J$ is a model of $\mathscr{P}_4^J$, so J is not an answer set of $\mathscr{P}_4^J$ and thus J is not an answer set of $\mathscr{P}_4$.

For $K = \{c, d(1)\}$, on the other hand, the reduct $\mathscr{P}_4^K$ is equal to $\mathscr{P}_4^J$, but K is not an answer set of $\mathscr{P}_4^K$: for the rule $r : a \vee b \coloneq c$, $B(r) \subseteq K$ 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 $\mathscr{P}_4$. ■

Step 3 Given a ground program $\mathscr{P}$ with weak constraints $WC(\mathscr{P})$, we are interested in the answer sets of $\mathit{Rules}(\mathscr{P})$ which minimize the sum of weights of the violated (unsatisfied) weak constraints in the highest priority level,[9] and among them those which minimize the sum of weights of the violated weak constraints in the next lower level, etc. Formally, this is expressed by an objective function $H^{\mathscr{P}}(A)$ for $\mathscr{P}$ and an answer set A as follows, using an auxiliary function $f_{\mathscr{P}}$ which maps leveled weights to weights without levels:

$$\begin{aligned} f_{\mathscr{P}}(1) &= 1, \\ f_{\mathscr{P}}(n) &= f_{\mathscr{P}}(n-1) \cdot |WC(\mathscr{P})| \cdot w^{\mathscr{P}}_{max} + 1, \quad n > 1, \\ H^{\mathscr{P}}(A) &= \sum_{i=1}^{l^{\mathscr{P}}_{max}} (f_{\mathscr{P}}(i) \cdot \sum_{w \in N_i^{\mathscr{P}}(A)} \mathit{weight}(w)), \end{aligned}$$

where $w^{\mathscr{P}}_{max}$ and $l^{\mathscr{P}}_{max}$ denote the maximum weight and maximum level over the weak constraints in $\mathscr{P}$, respectively, $N_i^{\mathscr{P}}(A)$ denotes the set of the weak constraints in level i that are violated by A, and $\mathit{weight}(w)$ denotes the weight of the weak constraint w. Note that $|WC(\mathscr{P})| \cdot w^{\mathscr{P}}_{max} + 1$ is greater than the sum of all weights in the program, and therefore guaranteed to be greater than the sum of weights of any single level.

Intuitively, the function $f_{\mathscr{P}}$ handles priority levels. It guarantees that the violation of a single constraint of priority level i is more "expensive" than the violation of *all* weak constraints of the lower levels (i.e., all levels $< i$).

[9] Higher values for weights and priority levels mark weak constraints of higher importance. The most important constraints are those having the highest weight among those with the highest priority level.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

For a DLP$^{\mathcal{A}}$ program $\mathscr{P}$ (possibly with weak constraints), a set A is an *(optimal) answer set* of $\mathscr{P}$ if and only if (1) A is an answer set of $Rules(\mathscr{P})$ and (2) $H^{\mathscr{P}}(A)$ is minimal over all the answer sets of $Rules(\mathscr{P})$.

Example 9
Consider the following program $\mathscr{P}_5$, which has three weak constraints:

$a \vee b.$
$b \vee c.$
$d \vee nd \ :\!-\ a, c.$
$:\!\sim \#\mathtt{sum}\{\langle 4 : b\rangle\} > 3.\ [1:2]$
$:\!\sim a, nd.\ [4:1]$
$:\!\sim c, d.\ [3:1]$

$Rules(\mathscr{P}_5)$ admits three answer sets: $A_1 = \{a, c, d\}$, $A_2 = \{a, c, nd\}$, and $A_3 = \{b\}$. We have $H^{\mathscr{P}_5}(A_1) = 3$, $H^{\mathscr{P}_5}(A_2) = 4$, and $H^{\mathscr{P}_5}(A_3) = 13$. Thus, the unique (optimal) answer set is $\{a, c, d\}$ with weight 3 in level 1 and weight 0 in level 2.

2.4 Computing new values from aggregates

Due to the definition of safety in Section 2.2, we could define the semantics of aggregates using the standard Herbrand Base and Herbrand Universe. The values returned by aggregate functions do not extend the Herbrand Universe.

This restriction, which is also imposed in the language of the Lparse system Syrjänen (2002) (see also Section 7.2) appears to be severe and limits the expressiveness of the language. Suppose, for instance, that the employees of a company are stored by a number of facts of the form $employee(Id, Name, Salary)$. If the boss wants to know the sum of the salaries she pays, a rule

$$total(T) \text{ :- } T = \#\mathtt{sum}\{S, I : employee(I, N, S)\}$$

would be most intuitive.[10]

However, the above rule is unsafe because of the variable T. Our language thus fails to naturally express a simple query which can be easily stated in SQL.[11] To overcome this problem, we introduce the notion of *assignment aggregate* and make appropriate adjustments to the notion of safety and semantics.

Assignment aggregate. We denote by $def^r(p)$ the set of *defining rules* of a predicate p, that is, those rules r in which p occurs in the head. Moreover, the *defining program* of a predicate p, denoted by $def^P(p)$, consists of $def^r(p)$ and the defining programs of all predicates which occur in the bodies of rules in $def^r(p)$.

An aggregate atom is an *assignment aggregate* if it is of the form $X = f(S)$, $f(S) = X$, or $X = f(S) = X$, where X is a variable and for each predicate p in S, $def^P(p)$ is negation-stratified and nondisjunctive.

[10] We aggregate over I (in addition to S), as otherwise two employees having the same salary would count only once in the total. This is also why we allow for multisets.
[11] Note that also the language of Lparse cannot express this query (cf. Section 7.2).

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

The intuition of the restriction on the definition of the nested predicates is to ensure that these predicates are deterministically computable.

Relaxed safety. We slightly relax the notion of safety as defined in Section 2.2, changing only condition (i):

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

To adapt the formal semantics to this extension, we enrich the Universe $U_{\mathscr{P}}$ of the program by the set of positive integers which results from the evaluation of an aggregate function, with a consequent enlargement of $B_{\mathscr{P}}$. Note that the (relaxed) safety criterion guarantees domain independence of rules and weak constraints, which—together with aggregate stratification—guarantees a simple (and finite) evaluation. None of the remaining semantic notions needs further adaptations.

3 Knowledge representation in $\mathrm{DLP}^{\mathscr{A}}$

In this section, we show how aggregate functions can be used to encode several relevant problems: Team building, Seating, and a logistics problem, called Fastfood. Moreover, we show how some properties of the input relations (e.g., the cardinality) can be simply computed by using aggregates, and we describe the encoding of a variant of the Fastfood problem.

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.

Information on our employees is 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 in the team, 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 $women(W)$, respectively. We then encode each property p_i stated above by an aggregate atom A_i, and enforce it by an integrity constraint containing $not\ A_i$.

% Guess whether to take an employee or not.
$in(I) \vee out(I) \text{ :- } emp(I, Sx, Sk, Sa).$
% The team consists of exactly N employees. (p_1)
$\text{ :- } nEmp(N), not\ \#\mathtt{count}\{I : in(I)\} = N.$

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

% Overall, employees need to have at least M different skills. (p_2)
$:\!-\ nSkill(M), not\ \#\texttt{count}\{Sk : emp(I, Sx, Sk, Sa), in(I)\} \geqslant M.$
% The sum of the individual salaries must not exceed the budget B. (p_3)
$:\!-\ budget(B), not\ \#\texttt{sum}\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\} \leqslant B.$
% The max. salary in the team must not exceed the max. allowed salary M. (p_4)
$:\!-\ maxSal(M), not\ \#\texttt{max}\{Sa : emp(I, Sx, Sk, Sa), in(I)\} \leqslant M.$
% We have at least W women in the team. (p_5)
$:\!-\ women(W), not\ \#\texttt{count}\{I : emp(I, f, Sk, Sa), in(I)\} \geqslant W.$

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 specification 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 the *Vars* part 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 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 $\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\}$ gives rise to the set $S = \{\langle Sa, I\rangle : Sa\ is\ the\ salary\ of\ employee\ I\ in\ the\ team\}$. The sum function is then applied on the multiset of the first component Sa of all the tuples $\langle Sa, I\rangle$ in S (see Section 2.3).

Seating. We have to generate a seating arrangement for k 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 sit at different tables.

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 generate 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 number of chairs there.
$:\!-\ table(T), nChairs(C), not\ \#\texttt{count}\{P : at(P, T)\} \leqslant C.$
% A person is to be seated at precisely one table.
$:\!-\ person(P), not\ \#\texttt{count}\{T : at(P, T)\} = 1.$
% 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.
$:\!-\ dislike(P1, P2), at(P1, T), at(P2, T).$

This encoding does not make as massive a use of aggregates as Team building, but it is useful to highlight a readability issue, which also has impact on efficiency,

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

as discussed in Section 6. The last aggregate atom above could be replaced by

```
% A person cannot sit at two different tables...
:- person(P), at(P, T), at(P, T1), T != T1.
% ...and has to sit at one table at least.
seated(P) :- at(P, T).
:- person(P), not seated(P).
```

This is less concise and arguably less readable. Moreover, the number of ground rules and constraints necessary for expressing the same statement would grow from $k * m$ to $k * m * (m-1) + k * m + k$, where k is the number of guests and m the number of tables.

Fastfood. The "Fastfood" problem, number 662 of volume VI of the ACM programming contests problem set archive (`http://acm.uva.es/p/v6/662.html`), is specified as follows:

> The fastfood chain McBurger owns several restaurants along a highway. Recently, they have decided to build several depots along the highway, each one located at a restaurant and supplying several of the restaurants with the needed ingredients. Naturally, these depots should be placed so that the average distance between a restaurant and its assigned depot is minimized. You are to write a program that computes the optimal positions and assignments of the depots.
>
> To make this more precise, the management of McBurger has issued the following specification: You will be given the positions of n restaurants along the highway as n integers $d_1 < d_2 < \ldots < d_n$ (these are the distances measured from the company's headquarter, which happens to be at the same highway). Furthermore, a number k ($k \leqslant n$) will be given, the number of depots to be built.
>
> The k depots will be built at the locations of k different restaurants. Each restaurant will be assigned to the closest depot, from which it will then receive its supplies. To minimize shipping costs, the total distance sum, defined as
>
> $$\sum_{i=1}^{n} | d_i - (\text{position of depot serving restaurant } i) |$$
>
> must be as small as possible.

We assume that instances are given as facts of the form $restaurant(res, d)$ representing a restaurant uniquely named res at kilometer d of the highway. Moreover, a fact $nDepots(k)$ is included which specifies k, the number of depots to be built.

```
% A restaurant can be a depot or not.
depot(Res, D) v notdepot(Res, D) :- restaurant(Res, D).
% The number of depots must be as specified.
:- nDepots(K), not #count{Dep, D : depot(Dep, D)} = K.
% Determine the serving depot for each restaurant.
serves(Dep, Res, D) :- restaurant(Res, ResD), depot(Dep, DepD),
    distance(ResD, DepD, D),
```

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

$\#\texttt{min}\{Y : depot(Dep1, DepD1), distance(DepD1, ResD, Y)\} = D.$
% Minimize the serving distances.
$:\sim serves(Dep, Res, D).[D :]$
% Auxiliary predicate.
$distance(X, Y, D) \coloneq restaurant(Res1, X), restaurant(Res2, Y), X > Y, X = Y + D.$
$distance(X, Y, D) \coloneq restaurant(Res1, X), restaurant(Res2, X), X <= Y, Y = X + D.$

In the definition for *distance*, we have used atoms involving built-in predicates $>$, $<=$, and $+$, which are defined on a bounded set of integer constants. That is, these predicates define greater than, less than or equal, and sum, respectively, on the finite set of integers $[0, n]$. For this example domain, each instance implies an upper bound for the integers that can occur, and we assume that the maximum integer n is chosen appropriately for each instance. Note that atoms like $X = Y + D$ are quite different from assignment aggregates: For the former, an admissible value range has to be specified explicitly (n for the integer range on which the predicate is defined), while for the latter a value range is not necessary.

Note that this example involves minimization in two different ways. On the one hand, the serving distance for a restaurant is the minimum distance to a depot. On the other hand, we look for a solution which minimizes the sum of serving distances. It is important to note that the first minimum (choosing the closest depot for every restaurant) refers to a fixed depot assignment, whereas the second (minimizing the sum of serving distances) is to be determined with respect to all possible depot assignments. It is therefore not possible to merge the two criteria, and indeed we use different constructs (an aggregate and a weak constraint) for representing them.

Input cardinality. In several problems, it is important to determine the cardinality of input relations. Doing so is simple using an assignment aggregate: If the input predicate is p and has arity n, we can write

$$cardinality_p(C) \coloneq \#\texttt{count}\{X1, \ldots, Xn : p(X1, \ldots, Xn)\} = C.$$

Note that in general this cannot be achieved without assignment aggregates as defined in Section 2.4. Without these, one could add some kind of domain predicate restricting the range of the variable C:

$$cardinality_p(C) \coloneq \#\texttt{count}\{X1, \ldots, Xn : p(X1, \ldots, Xn)\} = C,\ domain(C).$$

However, since the maximum cardinality of p is not known in advance, the size of *domain* would have to be countably infinite, which is not feasible.

In a similar way, again by assignment aggregates, one may compute the sum of the values of an attribute of an input relation (e.g., compute the sum of the salaries of the employees).

Fastfood solution checking. Consider a slight variation of the Fastfood problem introduced above: Instead of computing the optimal solutions, one has to check

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

whether a given depot assignment is optimal and compute a witness (a depot assignment with smaller distance sum) if it is not. This problem features in the first answer set programming system competition[12] Gebser *et al.* (2007).

Here, in addition to facts $restaurant(res, d)$ (as in the Fastfood problem input), also facts $depot(dep, d)$ will be in the input, representing the depot assignment to be checked for optimality. $nDepots(k)$ is no longer part of the input.

The encoding is an elaboration of the encoding for Fastfood. Here we define a predicate *altdepot*, which represents an alternative depot assignment. Such an assignment is a witness if its distance sum is less than the distance sum of the input depot assignment.

```
% A restaurant can be an alternative depot or not.
altdepot(Res, D) ∨ notaltdepot(Res, D) :- restaurant(Res, D).
% The number of alternative depots must be equal to the number of depots.
:- #count{Dep, D : depot(Dep, D)} = N,
   not #count{Dep, D : altdepot(Dep, D)} = N.
% Determine the serving input depot for each restaurant.
serves(Dep, Res, D) :- restaurant(Res, ResD), depot(Dep, DepD),
   distance(ResD, DepD, D),
   #min{Y : depot(Dep1, DepD1), distance(DepD1, ResD, Y)} = D.
% Determine the serving alternative depot for each restaurant.
altserves(Dep, Res, D) :- restaurant(Res, ResD), altdepot(Dep, DepD),
   distance(ResD, DepD, D),
   #min{Y : altdepot(Dep1, DepD1), distance(DepD1, ResD, Y)} = D.
% Accept an alternative solution only if its supply costs are not greater or
% equal than the supply costs for the input candidate.
:- #sum{D, Res : serves(Dep, Res, D)} = Cost,
   #sum{D, Res : altserves(Dep, Res, D)} ⩾ Cost.
% Auxiliary predicate.
distance(X, Y, D) :- restaurant(Res1, X), restaurant(Res2, Y), X > Y, X = Y + D.
distance(X, Y, D) :- restaurant(Res1, X), restaurant(Res2, X), X <= Y, Y = X + D.
```

It should be noted that this encoding relies heavily on assignment aggregates. The first constraint determines the cardinality of the input predicate *depot* using an assignment aggregate and makes sure that any alternative assignment has the same cardinality. The final constraint also employs an assignment aggregate, in this case not directly involving an input predicate, but a predicate which has a deterministic definition (*serves*) and which involves yet another aggregate. In fact, it is unclear if and how this constraint could be encoded without an assignment aggregate, as the range for *Cost* is not known or bounded *a priori*.

[12] See `http://asparagus.cs.uni-potsdam.de/contest/`.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

4 Computational complexity of DLP$^{\mathcal{A}}$

As for the classical nonmonotonic formalisms Marek and Truszczyński (1991), two important decision problems, corresponding to two different reasoning tasks, arise in DLP$^{\mathcal{A}}$:

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

The following theorems report on the complexity of the above reasoning tasks for propositional (i.e., variable-free) DLP$^{\mathcal{A}}$ programs that respect the safety restrictions imposed in Section 2. Importantly, it turns out that reasoning in DLP$^{\mathcal{A}}$ does not bring an increase in computational complexity, which remains exactly the same as for standard DLP. We begin with programs without weak constraints, and then discuss the complexity of full DLP$^{\mathcal{A}}$ programs including weak constraints.

Lemma 1
Deciding whether an interpretation M is an answer set of a ground program $\mathscr{P}$ without weak constraints is in co-NP.

Proof
We check in NP that M is *not* an answer set of $\mathscr{P}$ as follows. Guess a subset I of M, and verify that (1) M is not a model of $\mathscr{P}$, or (2) $I \subset M$ and I is a model of $\mathscr{P}^M$, the Gelfond–Lifschitz transform of $\mathscr{P}$ 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 requires 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`, and `#times`) on a multiset. The comparison of this result against the guards, finally, is straightforward.

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

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

Theorem 10
Brave reasoning on ground DLP$^{\mathcal{A}}$ programs without weak constraints is Σ_2^P-complete.

Proof
We verify that a ground literal L is a brave consequence of a DLP$^{\mathcal{A}}$ program $\mathscr{P}$ as follows: Guess a set $M \subseteq B_{\mathscr{P}}$ of ground atoms and check that (1) M is an answer

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

set of $\mathscr{P}$ 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 Eiter *et al.* (1997), since DLP$^{\mathscr{A}}$ is a superset of DLP. □

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

Theorem 11
Cautious reasoning on ground DLP$^{\mathscr{A}}$ programs without weak constraints is Π_2^P-complete.

Proof
We verify that a ground literal L is *not* a cautious consequence of a DLP$^{\mathscr{A}}$ program $\mathscr{P}$ as follows: Guess a set $M \subseteq B_{\mathscr{P}}$ of ground atoms and check that (1) M is an answer set of $\mathscr{P}$ 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 Eiter and Gottlob (1995), since DLP$^{\mathscr{A}}$ is a superset of DLP. □

From these results we can derive the results for DLP$^{\mathscr{A}}$ with weak constraints.

Theorem 12
For a ground DLP$^{\mathscr{A}}$ program $\mathscr{P}$, deciding whether an interpretation M is an answer set is Π_2^P-complete, while brave and cautious reasoning are both Δ_3^P-complete.

Proof
The key to this proof is that one can rewrite each DLP$^{\mathscr{A}}$ program $\mathscr{P}$ to another DLP$^{\mathscr{A}}$ program $\mathscr{W}(\mathscr{P})$ in which no aggregates occur in weak constraints, by replacing each aggregate literal that occurs in a weak constraint by a new standard atom, and adding a rule with the aggregate literal in the body and the new atom in the head.

Hardness for the Π_2^P result follows directly from item (3) of Theorem 4.14 in Leone *et al.* (2006). For membership, we show that deciding whether an interpretation M is not an answer set is Σ_2^P. We consider $\mathscr{W}(\mathscr{P})$ and M', which is obtained from M by adding those new atoms that replaced aggregate literals that are true w.r.t. M. We first test whether M' is an answer set of $Rules(\mathscr{W}(\mathscr{P}))$, which is in co-NP by Lemma 1. If M' is not an answer set, we stop and return yes. Otherwise we determine the cost c of M' in polynomial time, and guess an $M'' \subseteq B_{\mathscr{P}}$. We check that M'' is an answer set of $Rules(\mathscr{W}(\mathscr{P}))$ by a single call to an NP oracle, and check that the cost of M'' is less than c in polynomial time.

For the Δ_3^P results, hardness is an immediate consequence of Theorem 4.8 in Leone *et al.* (2006). Membership can be shown exactly as in the proof of Theorem 4.8 in Leone *et al.* (2006), using $\mathscr{W}(\mathscr{P})$ and the fact that the necessary oracle for determining whether an interpretation is an answer set of $\mathscr{P}$, the cost of which is less than a fixed bound, is Σ_2^P also in this case, as argued above. □

The above theorems confirm that our addition of aggregates to disjunctive logic programming does not cause any increase in the computational complexity of the language, and the same holds even if weak constraints are allowed.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

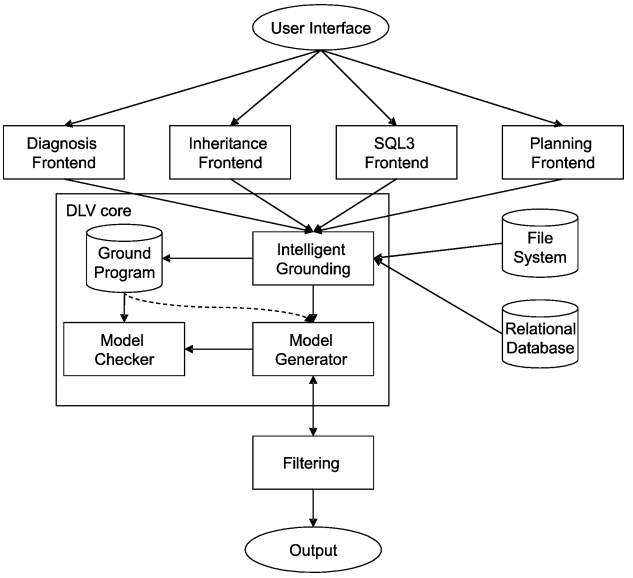


Fig. 1. The system architecture of DLV.

We end this section by discussing the complexity of nonground programs. The problems with respect to data-complexity for DLP$^{\mathscr{A}}$ programs (i.e., a program $\mathscr{P}$ is fixed, while the input consists of a set of facts) have the same complexity as for propositional programs. Concerning program complexity (i.e., a program $\mathscr{P}$ is given as input), complexity rises in a similar manner as for aggregate-free programs. A nonground program $\mathscr{P}$ can be reduced, by naive instantiation, to a ground instance of the problem, the size of which is single exponential in the size of $\mathscr{P}$. Informally, the complexity results thus increase accordingly by one exponential, from co-NP to co-NEXPTIME, Σ_2^P to NEXPTIME$^{\mathrm{NP}}$, Π_2^P to co-NEXPTIME$^{\mathrm{NP}}$, and Δ_3^P to EXPTIME$^{\Sigma_2^P}$. These results can be derived using complexity upgrading techniques as presented in Eiter *et al.* (1997); Gottlob *et al.* (1999).

5 Implementation issues

In this section, we illustrate the design of the implementation of aggregates in the DLV system. We first briefly describe the overall architecture of DLV, and we then discuss the impact of the implementation of aggregates in the system.

5.1 DLV *architecture*

An outline of the general architecture of the DLV system is depicted in Figure 1. It includes four front-ends for solving domain-oriented tasks; these are implemented on top of the DLV core by means of suitable rewriting techniques to DLP. Clearly, the implementation of aggregates does not affect these front-ends, even if the availability of the aggregates will allow to enhance the front-ends and improve the expressiveness of their languages.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

Instead, the implementation of aggregates heavily affects the DLV core, which we will describe next. Input data can be supplied by regular files, and also by relational databases. The DLV core then produces answer sets one at a time, and each time an answer set is found, the "Filtering" module is invoked, which performs postprocessing (dependent on the active front-ends) and controls continuation or abortion of the computation.

The DLV core consists of three major components: the "Intelligent Grounding" (IG), the "Model Generator" (MG), and the "Model Checker" (MC) modules that share a principal data structure, the "Ground Program." The "Ground Program" is created by the IG using differential (and other advanced) database techniques together with suitable data structures, and used by the MG and the MC. The Ground Program is guaranteed to have exactly the same answer sets as the original program. For some syntactically restricted classes of programs (e.g., stratified programs), the IG module already computes the corresponding answer sets.

For harder problems, most of the computation is performed by the MG and the MC. Roughly, the former produces some candidate answer sets (models) (Faber *et al.* 1991, 2001), the stability and minimality of which are subsequently verified by the latter.

The MC verifies whether the model at hand is an answer set. This task is very hard in general, because checking the stability of a model is known to be co-NP-complete. However, MC exploits the fact that minimal model checking—the hardest part—can be efficiently performed for the relevant class of *head-cycle-free* (HCF) programs Ben-Eliyahu and Dechter (1994); Leone *et al.* (1997).

5.2 *Implementation of aggregates in* DLV

Implementing aggregates in the DLV system has had a strong impact on DLV requiring many changes to the modules of the DLV core, and, especially, to the IG and to the MG modules. We next describe the main changes carried out in the modules of DLV core to implement aggregates.

5.2.1 *Intelligent grounding*

The changes performed in the IG module to implement aggregates in DLV can be summarized in three main activities: Standardization, Instantiation procedure (the main task), and Duplicate sets recognition.

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

Instantiation procedure. The goal of the instantiator is to generate a ground program that has precisely the same answer sets as the theoretical instantiation $Ground(\mathscr{P})$, but is as small as possible. The instantiation of standard DLV proceeds bottom-up following the dependencies induced by the rules, and, in particular, respecting the

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

ordering imposed by negation-stratification where this is possible. DLV's instantiator produces only those instances of a predicate which can potentially become true (Faber *et al.* 1991; Leone *et al.* 2001), where a ground atom A can potentially become true only if we have generated or may generate a ground instance of a rule with A in the head. Ground atoms, which have determined to be true or false in any answer set, are instead partially evaluated, that is if a literal it occurs in is true, that literal is omitted from the ground rule to be generated; if that literal is false, the ground rule it would occur in will not be generated.

For programs containing stratified aggregates, strategy is extended such that the order of processing respects aggregate-stratification. In this way, any truth-values (true, false, or potentially true) of nested atoms, which can be determined during grounding, have already been determined before the aggregate atom itself is instantiated.

When processing a rule containing an aggregate atom we proceed as follows. Assume that "$H \text{ :- } B, aggr.$" is the rule r which is to be processed, where H is the head of the rule, B is the conjunction of the standard body literals in r, and $aggr$ is a standardized 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 as the computation follows aggregate-stratification as discussed above), all matching facts are collected, 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 a potentially true instance of $\overline{Aux}$. For all $\sigma(\overline{Aux})$ which are true or false instances of $\overline{Aux}$, the aggregate is partially evaluated, which is done by methods that depend on the aggregate function involved. Note that in this way aggregates will only ground atoms the truth-value of which cannot be determined already by the instantiator.

Note that for several classes of programs, the instantiator guarantees complete evaluation. If a predicate is defined by a subprogram of such a class, no ground atom of it will be generated. In particular, if the predicate Aux of a standardized aggregate is defined by such a program, the aggregate function can be fully evaluated by the instantiator. One notable class, for which this is possible, is nondisjunctive negation-stratified programs. Therefore, by the definition of assignment aggregates, the value of the aggregate function inside an assignment aggregate can always be determined by the instantiator, thus providing a binding for the assigned variable (or no binding if the function evaluates to $\perp$). An assignment aggregate thus is treated like a unary positive atom which has at most one true matching instance.

If a nonassignment aggregate literal can be fully evaluated by the instantiator, its truth-value will be determined by computing the value of the aggregate function and comparing it to the guards. If it evaluates to true, it is removed from the ground rule; if it evaluates to false, the ground rule is simply discarded, thus partially evaluating the ground rule on the aggregate literal.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

The same process is then repeated for all further instantiations of the literals in B.

Example 13
Consider the rule r:

$$p(X) \text{ :- } q(X), 1 < \#\texttt{count}\{Y : a(X,Y), not\ b(Y)\}.$$

The standardization rewrites r to:

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

Suppose that the instantiation of the rule for *aux* generates three 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:

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

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

Duplicate sets recognition. To optimize the evaluation during instantiation and especially afterward, 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 implies a significant performance gain, especially during model generation where sets are frequently manipulated by the backtracking process.

Example 14
Consider the following two constraints:

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

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

To see the impact of this technique, consider a situation in which the two constraints additionally contain another standard literal $p(T)$:

$$c_3: \quad \text{:-} \quad p(T), 10 \leqslant \#\texttt{max}\{V : d(V,X)\}.$$
$$c_4: \quad \text{:-} \quad p(T), \#\texttt{min}\{Y : d(Y,Z)\} \leqslant 5.$$

Here, c_3 and c_4 have n instances each, where n is the number of potentially true atoms matching $p(T)$. By means of our technique, all instances of the constraint atoms in c_3 and c_4 share one common set, reducing the number of instantiated sets from $2 * n$ to 1. ■

5.2.2 Model generator

In our implementation, an aggregate atom will be assigned a truth-value just like a standard atom. However, different from a standard atom, its truth-value also

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

depends on the valuation of the aggregate function and thus on the truth-value of the nested predicates. Therefore, an aggregate atom adds an implicit constraint on models and answer sets: The truth-value assigned to the aggregate atom must correspond to the truth-value obtained by the valuation.

We have designed an extension of the Deterministic Consequences operator of the DLV system (Faber *et al.* 1999; Faber 2002) for DLP$^{\mathscr{A}}$ programs which account for these additional implicit constraints. As for rules, we differentiate between "forward propagation" (when an aggregate atom is assigned a truth-value because of the valuation of its aggregate function) and "backward propagation" (when a nested atom is derived in order to make the valuation of the aggregate atom compliant with its assigned truth-value).

We have extended the Dowling and Gallier algorithm Dowling and Gallier (1984) (in the version of Minoux (1988)) to deal with aggregates, and we compute the fixpoint of the enhanced Deterministic Consequences operator in linear time. To achieve this, we have endowed aggregate atoms with data structures similar to those used in rules. In particular, all aggregate atoms have a lower and upper bound holding the minimum and maximum value of the aggregate function w.r.t. the interpretation at hand to efficiently determine whenever an aggregate atom becomes true or false. `#min` and `#max` hold additional values for differentiating between undefined and true nested atoms. Moreover, for each standard atom we keep an index of aggregate sets in which it occurs to update these counters in an efficient way.

Forward propagation can then be achieved comparatively easily: whenever a standard atom is assigned a truth-value (other than undefined), the bounds and additional data of all aggregate functions it occurs in are updated. Where the bound range is fully covered by the guard range, the aggregate atom is derived as true. If the bound range and the guard range do not intersect, it is derived as false. For the backward propagation, whenever an aggregate atom gets a truth-value other than undefined or a non-undefined aggregate atom has an update of its bounds, several checks for inferences are performed, dependent on the type of aggregate function. For example, if there exists a tuple $\langle k, \ldots : a \rangle$ in a ground `#sum` aggregate which is true, such that a is undefined and the lower bound plus k is greater than the upper guard, then a can be derived as false. In order to make these checks efficient, the set of entries in the ground aggregate set is stored in a structure which is ordered on the projected term.

Example 15

Let us consider some of the propagations that are done for the following ground program.

$$a(1) \vee b(1). \qquad a(2) \vee b(2).$$
$$\coloneq \#\mathtt{sum}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle\} < 3.$$
$$cs \coloneq \#\mathtt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle\} \geqslant 2.$$
$$c(1) \coloneq cs. \qquad c(2) \vee c(3) \coloneq cs. \qquad \coloneq c(1), d(1).$$
$$d(2) \coloneq \#\mathtt{min}\{\langle 1 : c(1)\rangle, \langle 2 : c(2)\rangle, \langle 3 : c(3)\rangle\} < 2.$$
$$d(1) \coloneq \#\mathtt{max}\{\langle 1 : c(1)\rangle, \langle 2 : c(2)\rangle, \langle 3 : c(3)\rangle\} \geqslant 3.$$

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

At the very beginning, the internal data structures of the aggregate atoms are initialized. $\#\texttt{sum}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle\} < 3$ gets bounds $[0, 3]$ and guards $[0, 2]$ (the guards are normalized to be inclusive). In a similar way, $\#\texttt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle\} \geqslant 2$ gets bounds $[0, 2]$ and guards $[2, \infty]$. $\#\texttt{min}\{\langle 1 : c(1)\rangle, \langle 2 : c(2)\rangle, \langle 3 : c(3)\rangle\} < 2$ is initialized with bounds $[-\infty, +\infty]$ (because the value of the aggregate function may become undefined) and guards $[0, 1]$, and in addition $\texttt{minTrue} = \infty$ and $\texttt{minUndef} = 1$ for keeping track of possible minima. In a similar way, $\#\texttt{max}\{\langle 1 : c(1)\rangle, \langle 2 : c(2)\rangle, \langle 3 : c(3)\rangle\} \geqslant 3$ is initialized with bounds $[-\infty, +\infty]$, guards $[3, \infty]$, and special values $\texttt{maxTrue} = -\infty$ and $\texttt{maxUndef} = 3$.

In the first step, $\#\texttt{sum}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle\} < 3$ is derived false in order to satisfy the first constraint. In order to look for possibilities for backward propagation, the elements of the multiset are examined in a descending order, beginning with the largest one. For each element, we check whether the bound minus the element value is less than or equal to the upper guard, as the condition of these elements must become true. So for $\langle 2 : a(2)\rangle$, we obtain $3 - 2 \leqslant 2$ and we make a derivation establishing the fact that $a(2)$ must be true. In a similar manner, we obtain that $a(1)$ must be true. Since both $a(1)$ and $a(2)$ each occur in a single rule head, they are derived as definitely true, being supported by the respective rule, which in turn causes $b(1)$ and $b(2)$ to be derived as false. Moreover, the truth of $a(2)$ causes the bounds of $\#\texttt{count}\{\langle 1 : a(1)\rangle, \langle 2 : a(2)\rangle\} \geqslant 2$ to become $[1, 2]$, which due to the truth of $a(1)$ then become $[2, 2]$, causing the aggregate atom to become true.

As a consequence, also cs and $c(1)$ become true, while $c(2)$ and $c(3)$ remain undefined. So in $\#\texttt{min}\{\langle 1 : c(1)\rangle, \langle 2 : c(2)\rangle, \langle 3 : c(3)\rangle\} < 2$, $\texttt{minTrue}$ becomes 1, while $\texttt{minUndef}$ becomes 2, so its bounds become $[1, 1]$, and the aggregate atom becomes true, causing also $d(2)$ to become true. For $\#\texttt{max}\{\langle 1 : c(1)\rangle, \langle 2 : c(2)\rangle, \langle 3 : c(3)\rangle\} \geqslant 3$, $\texttt{maxTrue}$ becomes 1 and $\texttt{maxUndef}$ becomes 3, causing the bounds to become $[1, 3]$.

Moreover, $d(1)$ becomes false because of the constraint $\coloneq c(1), d(1)$. Therefore, the aggregate atom $\#\texttt{max}\{\langle 1 : c(1)\rangle, \langle 2 : c(2)\rangle, \langle 3 : c(3)\rangle\} \geqslant 3$ is derived as false. We then examine the elements of the multiset, starting with the greatest. If a condition of the element is undefined and its value is between the guards (inclusively), that condition must become false. In our example, for $\langle 3 : c(3)\rangle$ this holds (the guards are $[3, \infty]$) and so we derive the falsity of $c(3)$. For $\langle 2 : c(2)\rangle$ we cannot do this, as 2 is not within the guards. Indeed, $c(2)$ is eventually derived true in order to satisfy the rule $c(2) \vee c(3) \coloneq cs$.

In this example, the Deterministic Consequence operator has thus already determined the answer set $\{a(1), a(2), cs, c(1), c(2), d(2)\}$, as no undefined atoms are left.

5.2.3 Model checker

The stratification constraint that we have imposed on DLP$^{\mathscr{A}}$ aggregates, allows us to treat aggregate literals as negative literals in the reduct (see Section 2.3), and minimize the impact of aggregates on answer set checking.

The MC receives a model M in input, and checks whether M is an answer set of the instantiated program $\mathscr{P}$ (see Subsection 5.1). To this end, it first computes

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

the reduct $\mathscr{P}^M$, by (i) deleting the rules having a false aggregate literal or a false negative literal (w.r.t. M) in their bodies, and (ii) removing the aggregates literals and the negative literals from the bodies of the remaining rules. Since the resulting program is aggregate-free, the standard DLV techniques can then be applied to check whether $\mathscr{P}^M$ is an answer set. Thus, no further change is needed in MC, after the modification of the procedure computing the reduct.

6 Experiments and benchmarks

6.1 Compared methods, problems, and data

To assess the usefulness of the proposed DLP language extension and evaluate its implementation, we compare the following two methods on some relevant benchmark problems:

DLV$^{\mathscr{A}}$ Encode each problem in DLP$^{\mathscr{A}}$ and solve it using our extension of DLV with aggregates.

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

We compare these methods on three benchmark problems: Timetabling, Seating, and Fastfood. *Timetabling* is a classical scheduling problem. In particular, we consider the problem of scheduling the timetable of lectures which some groups of students have to take using a number of real-world instances from the University of Calabria where instance k deals with k groups of students.

Seating is the problem described in Section 3. We consider four (for small instances with at most four tables) or five (for larger instances with at least five tables) seats per table, with increasing numbers of tables and persons (and $numPersons = numSeats * numTables$).

For each problem size (i.e., seats per tables/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, respectively, such that the problem is not over-constrained.[13] 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.

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

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

Table 1. *Experimental results for Timetabling*

Number of groups	Execution time		Instantiation size	
	DLV	DLV$^{\mathscr{A}}$	DLV	DLV$^{\mathscr{A}}$
1	3.45	0.22	91,337	7,092
2	12.40	0.77	178,756	14,209
3	32.63	1.57	265,250	21,200
4	59.49	2.73	367,362	29,377
5	90.93	4.18	437,018	36,517
6	129.44	5.76	519,568	43,385
7	153.30	7.98	607,099	50,731
8	216.12	11.70	762,026	62,513
9	-	16.51	944,396	74,772

For each problem size, we randomly generated 10 instances for each of these classes, 50 instances for each problem size overall.

We use the DLP$^{\mathscr{A}}$ encoding reported in Section 3. All encodings and benchmark data are available on the web at `http://www.dlvsystem.com/examples/` in the files aggregates-timetabling.zip, aggregates-seating.zip, and aggregates-fastfood.zip.

Fastfood is the problem described in Section 3. The concrete instances consist of service station data of the company "Tank&Rast" which runs the majority of service stations on German motorways. This data has been obtained from the company website `http://www.rast.de/standorte/`.

The instances are grouped by motorway and vary over the number of depots to be built, ranging from zero to the total number of restaurants along the motorway. The maximum length of any motorway is 910 km, the maximum number of restaurants per motorway is 49.

6.2 Results and discussion

We ran all benchmarks on an Intel dual Xeon 3 GHz machine, using Debian GNU/Linux sarge with kernel version 2.4.27 and DLV release 2006-07-14. We allowed a maximum running time of 1800 s per instance and a maximum memory usage of 256 MB.

Cumulated results for Timetabling and Seating are provided in Tables 1 and 2, respectively. 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 separately and atoms occurring in the sets of the aggregates are considered, too). For Seating, the execution time is the average running time over all 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.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

Table 2. *Experimental results for Seating*

Number of persons	Execution time		Instantiation size	
	DLV	DLV$^{\mathcal{A}}$	DLV	DLV$^{\mathcal{A}}$
8	0.01	0.01	228	72
12	0.0155	0.01	710	176
16	10.294	0.01	1,621	348
25	–	0.01	4,744	960
50	–	0.0505	35,779	5,443
75	–	0.1869	118,167	15,744
100	–	0.5371	277,035	34,221
125	–	1.2619	537,635	63,358
150	–	2.6204	925,055	105,476
175	–	4.854	1,464,260	162,773

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 became even more significant. DLV could solve only instances of small size (up to 16 persons—four tables, four seats per table), while DLV$^{\mathcal{A}}$ could solve significantly larger instances in reasonable time.

The data on the instantiation sizes provide an explanation for the large difference between the execution times of DLV and DLV$^{\mathcal{A}}$. Thanks to aggregates, the DLP$^{\mathcal{A}}$ encodings of Timetabling and Seating are far more succinct than the corresponding encodings in standard DLP. This also reflects in the ground instantiations of the programs. Since the evaluation algorithms are 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.

For Fastfood, we report only on motorways yielding hard instances in Figure 2. We have omitted the graph for A7, as it is very similar to that of A3. In each graph, the horizontal axis represents the number of depots to be built, while the vertical axis stands for execution time. For all motorways, we can observe an interesting easy–hard–easy pattern with increasing number of depots. This is expected, as most possibilities for placing depots exist when the number of depots to be built is about half of the number of restaurants. We also observe that the average execution times strongly depend on the number of restaurants.

It is easy to see that the encoding greatly benefits from the use of aggregates: Whenever there are instances that cannot be solved within the time limit, the version with aggregates manages to solve strictly more instances without timing out. Also when looking at the amount of time needed, the version with aggregates is always faster, and the advantage becomes more pronounced with rising difficulty of the instances, yielding speedups of up to 3.5. The computational benefits for this problem are not as dramatic as for Timetabling and Seating, but still quite notable.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

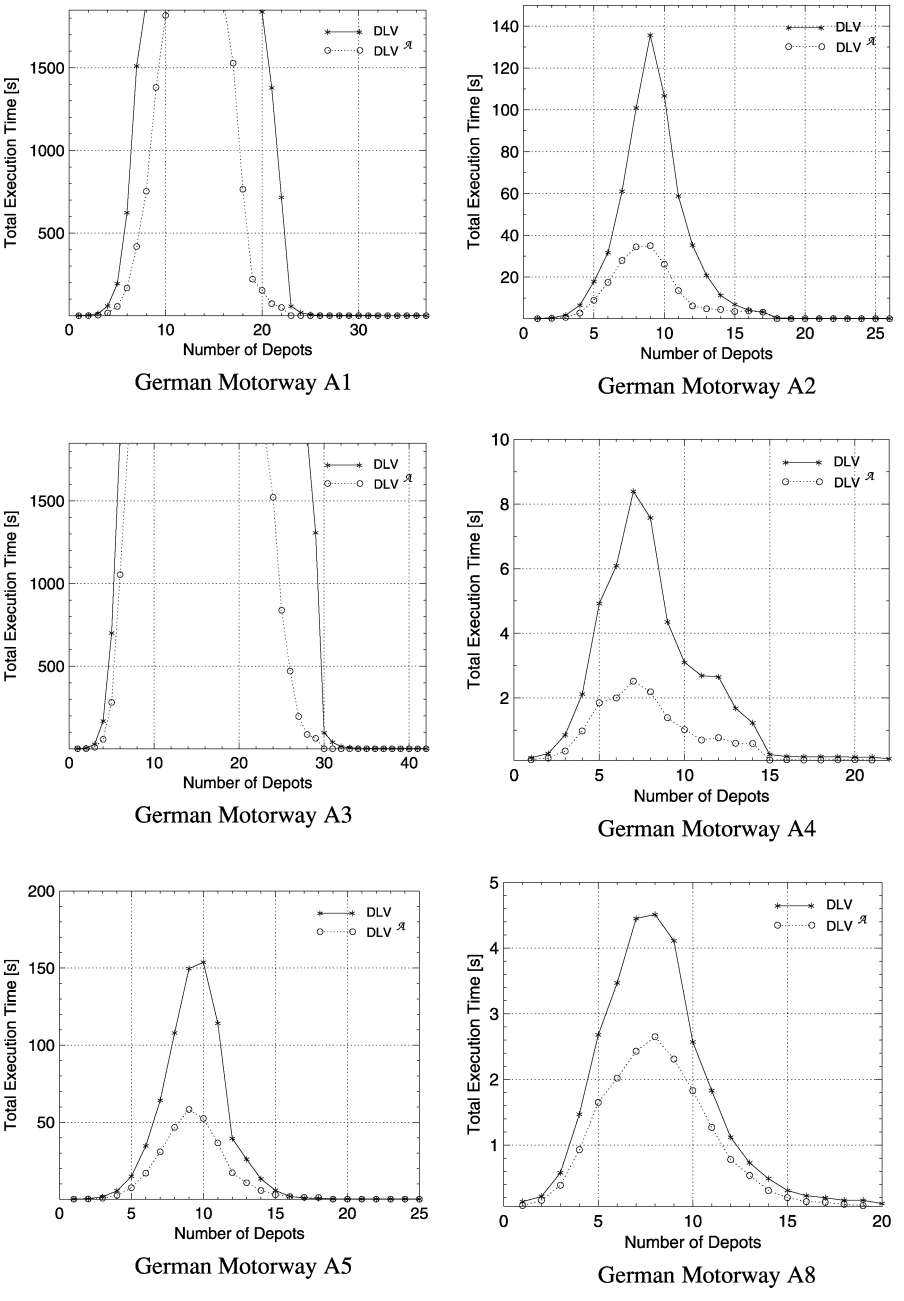


Fig. 2. Results for Fastfood on German Motorways A1–A5 and A8.

7 Related work

Aggregates have been studied fairly extensively in the context of databases and logics for databases; see Hella *et al.* (2001) for a summary. The logics studied in this setting are typically first-order logic endowed with some sort of aggregation operators, which are used to express queries. In such logics there is no concept

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

of recursive definitions, and the aggregations therefore occur in a stratified way. Moreover, as shown in Hella *et al.* (2001), the expressivity of these languages suffers from similar limitations as standard first-order logics for query answering.

Aggregate functions in logic programming languages appeared already in the 1980s, when their need emerged in deductive databases like LDL Chimenti *et al.* (1990) and were studied in detail, cf. Ross and Sagiv (1997); Kemp and Ramamohanarao (1998). However, the first implementation in ASP, in the Smodels system, has been fairly recent Simons *et al.* (2002).

7.1 Aggregate-stratification

The discussion on the "right" semantics for aggregate-unstratified programs is still going on in the DLP and ASP communities. Several proposals have been made in the literature, which can roughly be grouped as follows: In Eiter *et al.* (1997b), Gelfond (2002) and Dell'Armi *et al.* (2003), aggregate atoms are basically treated like negative literals. In Niemelä *et al.* (1999), only aggregates involving cardinality and sum are considered; as argued in Ferraris and Lifschitz (2005) this semantics is not intuitive for aggregates which are not monotonic, such as sum aggregates involving negative summands. In Pelov (2004) and Pelov *et al.* (2004), a family of semantics, which extend completion, stable and well-founded semantics, is defined by means of operator fixpoints, approximations and transformations; a very similar approach has been given by Son *et al.* (2005) and Son and Pontelli (2007). In Faber *et al.* (2004), a semantics based on a modified program reduct has been defined, for which alternative characterizations have been provided by Ferraris (2005), Calimeri *et al.* (2005), Faber (2005). All of these four groups of semantics differ on certain language fragments; but they coincide on aggregate-stratified programs. Finally, in Marek *et al.* (2004) and Liu and Truszczyński (2005), semantically restricted aggregates are considered, on which the newer proposals coincide; but still the first group of semantics Eiter *et al.* (1997); Gelfond (2002); Dell'Armi *et al.* (2003) differs even on these programs. To illustrate the difficulties with unstratified aggregates, we look at a simple example:

Example 16
Consider the (aggregate-unstratified) program consisting only of the rule

$$p(a) \coloneq \#\mathtt{count}\{X : p(X)\} = 0.$$

As neither $\{p(a)\}$ nor $\emptyset$ is an intuitive meaning for the program, one would expect that this program admits no answer sets. In this case, the role of the aggregate literal is similar to a negative literal. And indeed, approaches by Dell'Armi *et al.* (2003), Gelfond (2002), Eiter *et al.* (1997b) treat aggregates like negative literals.

However, consider a slight modification of this program, containing only the rule

$$p(a) \coloneq \#\mathtt{count}\{X : p(X)\} > 0.$$

If the aggregate is treated like a negative literal, this program allows for two answer sets $\{p(a)\}$ and $\emptyset$. Other approaches Pelov (2004); Faber *et al.* (2004); Marek *et al.* (2004); Liu and Truszczyński (2005) try to maintain subset minimality and therefore

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

differ on this program with respect to those Dell'Armi *et al.* (2003), Gelfond (2002), Eiter *et al.* (1997b). We conclude that this program does not have a semantics which is generally agreed upon.

Our policy, in the development of DLV, is to keep the system language as much agreed-upon as possible, and to try to guarantee a clear and intuitive semantics for the newly introduced constructs. Thus, we disregard programs which are not aggregate-stratified, leaving their introduction in DLV to future work.[14]

In addition, we observe that unstratified aggregates may cause a computational overhead. For instance, the complexity of brave and cautious reasoning on normal programs without weak constraints jumps from NP and co-NP to Σ_2^P and Π_2^P, respectively, if unstratified aggregates are allowed Ferraris (2005); Calimeri *et al.* (2005), while it remains in NP and co-NP if aggregates are stratified.

7.2 Comparison to the language of Lparse

Very related to DLP$^{\mathcal{A}}$ is without doubt the language of Lparse (Syrjänen 2002), which serves as a grounding front-end to systems like Smodels (Simons *et al.* 2002), Cmodels (Lierler 2005), or clasp (Gebser *et al.* 2007), which deal with aggregates. We observe a strong similarity between cardinality constraints and `#count`, as well as weight constraints and `#sum`, respectively. Indeed, the DLP$^{\mathcal{A}}$ encodings of both Team building and Seating can be easily translated to the language of Lparse. However, there are several relevant differences.

DLP$^{\mathcal{A}}$ aggregates like `#min`, `#max`, and `#times` do not have a counterpart in the language of Lparse. Moreover, DLP$^{\mathcal{A}}$ provides a general syntactic framework into which further aggregates can be easily included.

In DLP$^{\mathcal{A}}$ aggregate atoms can be negated, while cardinality and weight constraints in the language of Lparse 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, *not* $3 \leqslant \#\texttt{count}\{X : p(X)\} \leqslant 7$ 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 has to write two cardinality constraints.

Furthermore, note that symbolic sets of DLP$^{\mathcal{A}}$ directly represent pure sets of term tuples, and by means of projection DLP$^{\mathcal{A}}$ can also represent multisets naturally (see the discussion on Team building in Section 3). In contrast, cardinality constraints work on sets of ground atoms rather than on multisets of terms. For instance, condition p_2 of Team building in Section 3 cannot be directly encoded in a constraint, but needs the definition of an auxiliary predicate. Weight constraints, on the other hand, work exclusively on multisets of numbers, and additional rules are needed to encode pure sets.

The language of Lparse requires that each variable has to occur in a positive atom formed by a *domain predicate* which must not be recursive with a head atom—by

[14] Note that the limitation to aggregate-stratified programs is justified also from philosophical perspectives. For instance, defining a class q before defining subsets of q has been recommended by Zermelo, but we will not go into details of this aspect.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

default a domain predicate must be defined by an aggregate-free program. It follows that the language of Lparse has no equivalent to assignment aggregates, which prohibits the definition of simple concepts such as determining the cardinality of input relations, as discussed in Section 3.

The language of Lparse does however allow for cardinality and weight constraints in the heads of rules, while DLP$^{\mathcal{A}}$ aggregates may only occur in rule bodies. The presence of weight constraints in heads is an interesting feature, which allows, for instance, to "guess" an arbitrary subset of a given set. But it causes the loss of some semantic properties of nonmonotonic languages (see Marek and Remmel (2002)). Lparse rules having cardinality and weight constraints in the head can however be expressed in DLP$^{\mathcal{A}}$ in the following way: The atom to be aggregated over is put into a disjunctive head, which also contains a copy of this atom in which the predicate symbol is replaced by a fresh one, keeping the body of the original rule augmented by the "domain atom" of the constraint atom. Moreover, an integrity constraint is generated, which contains the negated constraint atom (transformed into a corresponding DLP$^{\mathcal{A}}$ aggregate atom) from the original rule head and the body of the original rule. Transforming a Lparse program in this way to a DLP$^{\mathcal{A}}$ program (replacing also cardinality and weight constraint atoms in rule bodies by corresponding DLP$^{\mathcal{A}}$ aggregate atoms), the answer sets of the resulting DLP$^{\mathcal{A}}$ program without atoms containing the fresh predicates are precisely the answer sets of the original Lparse program.

Moreover, the language of Lparse does allow for aggregate-unstratified programs, with the intended semantics of Niemelä *et al.* (1999). As discussed earlier, there is currently no consensus about the semantics of aggregate-unstratified programs, and indeed the semantics of Niemelä *et al.* (1999) has been criticized to yield unintuitive results when weight constraints over signed integers are present Ferraris and Lifschitz (2005).

7.3 *Comparison to the language of* SMODELS^{A}

More recently, the system SMODELS^{A} has been described by Elkabani *et al.* (2005). Its language is an extension of the language of Lparse which allows for aggregates (possibly not aggregate-stratified) under the semantic described by Son *et al.* (2005), which coincides with the semantics of DLP$^{\mathcal{A}}$ on aggregate-stratified programs.

The syntax of the additional aggregate constructs allowed in SMODELS^{A} is more similar to the one of DLP$^{\mathcal{A}}$ (compared to those of Lparse, which are also available in SMODELS^{A}), and allows for *sum*, *count*, *min*, *max*, and also *avg*, while *times* is currently not supported. In this sense, SMODELS^{A} can be considered as the system most similar to DLP.

There is, however, one rather crucial difference between the aggregate syntax of these two systems: SMODELS^{A} only allows for one term to be aggregated over. This means that, for example, the following DLP$^{\mathcal{A}}$ rule has no counterpart in terms of the new aggregate constructs in SMODELS^{A}.

$$tooexpensive \text{ :- } \#\mathtt{sum}\{Cost, Item : order(Item, Cost)\} > 100.$$

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

The intended meaning of this rule is that *tooexpensive* is derived when the sum of the costs of all ordered items exceeds a threshold of 100. Note that we specify two terms to be aggregated over, where the sum will be computed over the first one. This is important, as different items may incur the same cost. For instance if $order(valve, 60)$ and $order(pipe, 60)$ hold, then *tooexpensive* should be derived. One may try to write the following variant in the syntax of SMODELS^A:

$$tooexpensive \text{ :- } sum(Cost, order(Item, Cost)) > 100.$$

However, when we have both $order(valve, 60)$ and $order(pipe, 60)$, *tooexpensive* will not be derived as 60 is considered only once. We are not aware of any way of circumventing this problem with the aggregates introduced by SMODELS^A.

Actually, there is a second problem with the rule mentioned above in the current version of SMODELS^A. The way the preprocessing is done requires that each variable in the aggregate atom is domain restricted by an atom outside the aggregate. In our example this condition is not met, but it is also not possible to add an atom outside the aggregate to domain restrict *Item* without changing the semantics of the rule. In many cases this restriction can be overcome by writing an equivalent weight constraint in the language of Lparse, which is also available in SMODELS^A.

Other differences between the language of SMODELS^A and DLP$^\mathcal{A}$ are that aggregate atoms may not occur negated, that all variables must be domain-restricted, that each rule may contain only one aggregate, and that assignment aggregates are not permitted. Moreover, there is currently no possibility to specify a conjunction of literals (rather than a single atom) inside an aggregate atom in SMODELS^A though one can fairly and easily circumvent this limitation by replacing the conjunction by a new atom which is then defined by an appropriate rule. These differences are similar to the differences between DLP$^\mathcal{A}$ and the language of Lparse. Moreover, given that SMODELS^A relies on the Smodels engine, it also does not support disjunctive rules under the semantics of DLP$^\mathcal{A}$.

On the system level, the architecture of SMODELS^A considerably differs from the one of the DLV$^\mathcal{A}$. It first preprocesses the input using an algorithm implemented in Prolog, yielding an intermediate program. This program is then submitted to Lparse. The output of Lparse is subsequently processed by a transformation algorithm (also implemented in Prolog), whose output is then submitted to Lparse another time. Finally, Smodels is called on the output of the second Lparse invocation to compute the answer sets. The key idea of the system is to compile away the aggregates, creating new rules or constraints which emulate the aggregate atoms. In contrast, in the implementation of DLV$^\mathcal{A}$, aggregates are first-class citizens and all the internal algorithms of DLV have been extended in order to deal with aggregates.

8 Conclusion

We have proposed DLP$^\mathcal{A}$, an extension of DLP by aggregate functions count, sum, times, min, and max, and have implemented this 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.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

On the other hand, we have shown that aggregate functions do not increase the complexity of the main reasoning tasks. In fact, 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`, investigations of a general framework that will allow adding further aggregates much more easily, extending semantics to classes of programs which are not aggregate-stratified, as well as the design of further optimization techniques and heuristics to improve the efficiency of the computation.

Acknowledgements

This work has greatly benefited from interesting discussions with and comments by Paolo Ferraris, Michael Gelfond, Vladimir Lifschitz, Nikolay Pelov, and from the comments and suggestions by the anonymous referees. It was partially supported by M.U.R. under the PRIN project "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva" and by M.I.U.R. under the internationalization project "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione." Wolfgang Faber's work was funded by an APART grant of the Austrian Academy of Sciences.

References

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

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, UK.

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

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

Calimeri, F., Faber, W., Leone, N. and Perri, S. 2005. Declarative and computational properties of logic programs with aggregates. In *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*. 406–411.

Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S. A., Tsur, S. and Zaniolo, C. 1990. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering 2,* 1, 76–90.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N. and Pfeifer, G. 2003. Aggregate functions in DLV. In *Proceedings ASP03—Answer Set Programming: Advances in Theory and Implementation*, M. de Vos and A. Provetti, Eds. Messina, Italy, 274–288. Online at `http://CEUR-WS.org/Vol-78/`.

Dowling, W. F. and Gallier, J. H. 1984. Linear-time algorithms for testing the satisfability of propositional horn formulae. *Journal of Logic Programming 3*, 267–284.

Eiter, T., Faber, W., Leone, N. and Pfeifer, G. 2000. Declarative problem-solving using the DLV system. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic Publishers, Norwell, MA, USA, 79–103.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

Eiter, T. and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: propositional case. *Annals of Mathematics and Artificial Intelligence 15,* 3/4, 289–323.

Eiter, T., Gottlob, G. and Mannila, H. 1997a. Disjunctive datalog. *ACM Transactions on Database Systems 22,* 3 (Sept.), 364–418.

Eiter, T., Gottlob, G. and Veith, H. 1997b. Modular logic programming and generalized quantifiers. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-97)*, J. Dix, U. Furbach, and A. Nerode, Eds. Lecture Notes in Computer Science, vol. 1265. Springer, Berlin, Germany, 290–309.

Elkabani, I., Pontelli, E. and Son, T. C. 2005. Smodels^{A}—a system for computing answer sets of logic programs. In *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05*, Diamante, Italy, September 2005, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. Lecture Notes in Computer Science, vol. 3662. Springer-Verlag, Berlin, Germany, 427–431.

Faber, W. 2002. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems.* Ph.D. thesis, Institut für Informationssysteme, Technische Universität Wien.

Faber, W. 2005. Unfounded sets for disjunctive logic programs with arbitrary aggregates. In *Logic Programming and Nonmonotonic Reasoning—8th International Conference, LPNMR'05*, Diamante, Italy, September 2005, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. Lecture Notes in Computer Science, vol. 3662. Springer-Verlag, Berlin, Germany, 40–52.

Faber, W., Leone, N., Mateis, C. and Pfeifer, G. 1999. Using database optimization techniques for nonmonotonic reasoning. In *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, INAP Organizing Committee, Ed. Prolog Association of Japan, 135–139.

Faber, W., Leone, N. and Pfeifer, G. 1999. Pushing goal derivation in DLP computations. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, M. Gelfond, N. Leone, and G. Pfeifer, Eds. Lecture Notes in AI (LNAI), vol. 1730. Springer Verlag, El Paso, TX, 177–191.

Faber, W., Leone, N. and Pfeifer, G. 2001. Experimenting with heuristics for answer set programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*. Morgan Kaufmann Publishers, Seattle, WA, USA, 635–640.

Faber, W., Leone, N. and Pfeifer, G. 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)*, J. J. Alferes and J. Leite, Eds. Lecture Notes in AI (LNAI), vol. 3229. Springer-Verlag, Berlin, Germany, 200–212.

Ferraris, P. 2005. Answer sets for propositional theories. In *Logic Programming and Nonmonotonic Reasoning—8th International Conference, LPNMR'05*, Diamante, Italy, September 2005, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. Lecture Notes in Computer Science, vol. 3662. Springer-Verlag, Berlin, Germany, 119–131.

Ferraris, P. and Lifschitz, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5*, 1/2, 45–74.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 386–392.

Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T. and Truszczyński, M. 2007. The first answer set programming system competition. In *Logic Programming and Nonmonotonic Reasoning—9th International Conference, LPNMR'07*, C. Baral, G. Brewka, and J. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer-Verlag, Tempe, AZ, 3–17.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

Gelfond, M. 2002. Representing knowledge in A-prolog. In *Computational Logic. Logic Programming and Beyond*, A. C. Kakas and F. Sadri, Eds. Lecture Notes in Computer Science, vol. 2408. Springer, Berlin, Germany, 413–451.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

Gottlob, G., Leone, N. and Veith, H. 1999. Succinctness as a source of expression complexity. *Annals of Pure and Applied Logic 97,* 1–3, 231–260.

Hella, L., Libkin, L., Nurmonen, J. and Wong, L. 2001. Logics with aggregate operators. *Journal of the ACM 48,* 4, 880–907.

Kemp, D. B. and Ramamohanarao, K. 1998. Efficient recursive aggregation and negation in deductive databases. *IEEE Transactions on Knowledge and Data Engineering 10*, 727–745.

Leone, N., Perri, S. and Scarcello, F. 2001. Improving ASP instantiators by join-ordering methods. In *Logic Programming and Nonmonotonic Reasoning—6th International Conference, LPNMR'01, Vienna, Austria*, T. Eiter, W. Faber, and M. Truszczyński, Eds. Lecture Notes in AI (LNAI), vol. 2173. Springer-Verlag, Berlin, Germany, 280–294.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3 (July), 499–562.

Leone, N., Rullo, P. and Scarcello, F. 1997. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation 135,* 2 (June), 69–112.

Lierler, Y. 2005. Cmodels for tight disjunctive logic programs. In *W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany*. Ulmer Informatik-Berichte. Universität Ulm, Germany, 163–166.

Liu, L. and Truszczyński, M. 2005. Properties of programs with monotone and convex constraints. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'05)*, M. M. Veloso and S. Kambhampati, Eds. The MIT Press, Cambridge, MA, USA, 701–706.

Marek, V. W., Niemelä, I. and Truszczyński, M. 2004. Logic programming with monotone cardinality atom. In *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, V. Lifschitz and I. Niemelä, Eds. Lecture Notes in AI(LNAI), vol. 2923. Springer, Berlin, Germany, 154–166.

Marek, V. W. and Remmel, J. B. 2002. On logic programs with cardinality constraints. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, S. Ben-ferhat and E. Giunchiglia, Eds. Toulouse, France, 219–228.

Marek, V. W. and Truszczyński, M. 1991. Autoepistemic logic. *Journal of the ACM 38,* 3, 588–619.

Minker, J. 1982. On indefinite data bases and the closed world assumption. In *Proceedings 6th Conference on Automated Deduction (CADE '82)*, D. W. Loveland, Ed. Lecture Notes in Computer Science, vol. 138. Springer, New York, 292–308.

Minoux, M. 1988. LTUR: A simplified linear-time unit resolution algorithm for horn formulae and computer implementation. *Information Processing Letters 29*, 1–12.

Niemelä, I., Simons, P. and Soininen, T. 1999. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, M. Gelfond, N. Leone, and G. Pfeifer, Eds. Lecture Notes in AI (LNAI), vol. 1730. Springer Verlag, El Paso, TX, 107–116.

Pelov, N. 2004. *Semantics of Logic Programs with Aggregates*. Ph.D. thesis, Katholieke Universiteit Leuven, Leuven, Belgium.

Pelov, N., Denecker, M. and Bruynooghe, M. 2004. Partial stable models for logic programs with aggregates. In *Proceedings of the 7th International Conference on Logic*

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press

Programming and Non-Monotonic Reasoning (LPNMR-7). Lecture Notes in AI (LNAI), vol. 2923. Springer, Berlin, Germany, 207–219.

Przymusinski, T. C. 1988. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann Publishers, San Francisco, CA, USA, 193–216.

Ross, K. A. and Sagiv, Y. 1997. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences 54,* 1 (February), 79–97.

Simons, P., Niemelä, I. and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 181–234.

Son, T. C. and Pontelli, E. 2007. A constructive semantic characterization of aggregates in ASP. *Theory and Practice of Logic Programming 7*, 355–375.

Son, T. C., Pontelli, E. and Elkabani, I. 2005. On logic programming with aggregates. Technical Report NMSU-CS-2005-006, New Mexico State University.

Syrjänen, T. 2002. Lparse 1.0 User's Manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

Ullman, J. D. 1989. *Principles of Database and Knowledge Base Systems*. Computer Science Press. New York, NY, USA.

https://doi.org/10.1017/S1471068408003323 Published online by Cambridge University Press