Well-Founded Semantics for Logic Programs with Aggregates: Implementation and Experimentation

Mario Alviano, Wolfgang Faber, and Nicola Leone

Department of Mathematics
University of Calabria
87030 Rende (CS), Italy
{alviano,faber,leone}@mat.unical.it

Abstract

Aggregate constructs are one of the major linguistic extensions to logic programming. In this paper, we focus on logic programming with monotone and antimonotone aggregate literals with the well-founded semantics defined in [1], which allows for aggregates occurring in recursive definitions. We formally show that computing this semantics is tractable and present a prototype system obtained by modifying DLV. To our knowledge, this is the only system supporting well-founded semantics for logic programs with recursive aggregates. While aggregates yield an obvious representational improvement, we also present experiments involving our prototype system and XSB, showing that aggregates are also beneficial from a computational point of view.

1 Introduction

Logic Programming (LP) is a formalism widely used in various areas. In LP, problems are solved providing a declarative representation of the requirements to be achieved, instead of defining an ad-hoc algorithm. Several semantics for LP have been proposed, the well-founded semantics [2] is one of them, associating a three-valued model to every logic program.

Even if LP is a declarative programming language, standard LP does not allow for representing properties over sets of data in a natural way, a relevant aspect in several application domains. To overcome this lack, several syntactical extensions to LP have been proposed, the most important of which is the introduction of aggregate functions ($\text{LP}^{\mathcal{A}}$) [3, 4, 5, 6]. Among them, recursive definitions involving aggregate functions (i.e., aggregation in which aggregated data depend on the evaluation of the aggregate itself) are particularly relevant [7].

In this paper we focus on the fragment of $\text{LP}^{\mathcal{A}}$ allowing for monotone and antimonotone aggregate literals ($\text{LP}^{\mathcal{A}}_{m,a}$) [1]. $\text{LP}^{\mathcal{A}}_{m,a}$ programs have many interesting properties. Among them, we highlight similarities between monotone aggregate literals and positive standard literals, and between antimonotone aggregate literals and negative standard literals. In particular, this aspect is exploited for defining unfounded sets and well-founded semantics of $\text{LP}^{\mathcal{A}}_{m,a}$ programs. Another interesting property of $\text{LP}^{\mathcal{A}}_{m,a}$ programs is the tractability of literal evaluation w.r.t. partial interpretations.

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

*Proceedings of the 17th International RCRA workshop (RCRA 2010): Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Bologna, Italy, June 10–11, 2010*

The main contributions of the paper are as follows.

- We work with unfounded sets and the well-founded operator $\mathcal{W}_{\mathcal{P}}$ for $\mathrm{LP}^{\mathcal{A}}_{m,a}$ programs defined in [1], and formally prove that each fixpoint of $\mathcal{W}_{\mathcal{P}}$ is a (partial) model, that $\mathcal{W}_{\mathcal{P}}$ admits a least fixpoint (the well-founded model), that the well-founded model of a ground $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program is polynomial-time computable. For non-ground programs, the data-complexity remains polynomial, while the program complexity rises from P to EXPTIME as for aggregate-free programs.

- We implement a prototype system supporting the well-founded semantics for $\mathrm{LP}^{\mathcal{A}}_{m,a}$ programs. The prototype extends DLV and is the first system implementing a well-founded semantics for unrestricted $\mathrm{LP}^{\mathcal{A}}_{m,a}$ programs.

- We report on the results of the experimentation with the implemented prototype. Specifically, we define a problem having a natural representation in $\mathrm{LP}^{\mathcal{A}}_{m,a}$: *Attacks*, a variant of the classical *Win-Lose* problem.

The sequel of the paper is organized as follows. In Section 2 we recall the $\mathrm{LP}^{\mathcal{A}}$ language and its fragment $\mathrm{LP}^{\mathcal{A}}_{m,a}$, for which in Section 3 we recall the well-founded semantics and prove some of its properties, followed by a complexity analysis in Section 4. In Section 5 we discuss how some particular aggregates can be compiled away, which we used for some of the experiments reported in Section 6, in which we also overview the implemented prototype system. In Section 7, we discuss the literature, and draw our conclusions in Section 8.

2 The $\mathrm{LP}^{\mathcal{A}}$ Language

In this section, we present the $\mathrm{LP}^{\mathcal{A}}$ language – an extension of Logic Programming (LP) by set-oriented functions (also called aggregate functions). Subsequently, we introduce the $\mathrm{LP}^{\mathcal{A}}_{m,a}$ fragment, the language analyzed in this paper. For further background on LP, we refer to [8, 9].

2.1 Syntax, Instantiations, Interpretations, and Models

We assume sets of *variables*, *constants*, and *predicates* to be given. 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 $n \geq 0$ and $t_1, \ldots, t_n$ are terms.

Aggregate Atoms. An *aggregate function* is of the form $f(S)$, where f is an aggregate function symbol and S is a set term; a set term is a pair $\{\mathit{Terms} : \mathit{Conj}\}$, where Terms is a list of terms (variables or constants) and Conj is a conjunction of standard atoms. Finally, an *aggregate atom* is a structure of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq\}$ is a comparison operator, and T is a term (variable or constant).

Literals. A *literal* is either (i) a standard atom, or (ii) a standard atom preceded by the *negation as failure* symbol `not` , or (iii) an aggregate atom. Two standard literals are complementary if they are of the form a and `not` a, for some standard atom a. For a standard literal ℓ, we denote by $\neg.\ell$ the complement of ℓ. Abusing of notation, if L is a set of standard literals, then $\neg.L$ denotes the set $\{\neg.\ell \mid \ell \in L\}$.

Programs. A *program* is a set of *rules* r of the form $a \text{ :- } \ell_1, \ \ldots, \ \ell_m.$, where a is a standard atom, $\ell_1, \ \ldots, \ \ell_m$ are literals, and $m \geq 0$. The atom a is referred to as the *head* of r, denoted $H(r)$, while the conjunction $\ell_1, \ \ldots, \ \ell_m$ is the *body* of r, denoted $B(r)$. A structure (atom, literal, rule, or program) without variables is *ground*.

Safety. A *local* variable of a rule r is a variable appearing solely in sets terms of r; a variable of r which is not local is called *global*. A rule r is *safe* if both the following conditions hold: (i) each global variable appears in some positive standard literal of $B(r)$; (ii) each local variable appearing in a set term $\{Terms : Conj\}$ also appears in $Conj$. Finally, a program is safe if all its rules are safe.

Instantiation. The *universe* of an LP$^{\mathcal{A}}$ program $\mathcal{P}$, denoted $U_{\mathcal{P}}$, is the set of constants appearing in $\mathcal{P}$. The *base* of $\mathcal{P}$, denoted $B_{\mathcal{P}}$, is the set of standard atoms constructible from predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$.

A *substitution* is a mapping from a set of variables to $U_{\mathcal{P}}$. Given a substitution σ and an LP$^{\mathcal{A}}$ object obj (literal, rule, etc.), we denote by $obj\,\sigma$ the object obtained by replacing each variable X in obj by $\sigma(X)$. A *ground instance* of a rule r is obtained in two steps: First, a substitution σ for the global variables of r is applied, and then every set term S in $r\sigma$ is replaced by its instantiation $inst(S) = \{\langle Terms\,\sigma : Conj\,\sigma\rangle \mid \sigma \text{ is a local substitution for } S\}$. The set of all instances of rules in a program $\mathcal{P}$ is denoted $Ground(\mathcal{P})$.

Example 1. Consider the following program $\mathcal{P}_1$:

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

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

$$\begin{array}{ll} q(1) \ \vee \ p(2,2). & t(1) \text{ :- } q(1), \ \#\mathtt{sum}\{\langle 1 : p(1,1)\rangle, \ \langle 2 : p(1,2)\rangle\} > 1. \\ q(2) \ \vee \ p(2,1). & t(2) \text{ :- } q(2), \ \#\mathtt{sum}\{\langle 1 : p(2,1)\rangle, \ \langle 2 : p(2,2)\rangle\} > 1. \end{array}$$

Interpretation. An *interpretation* I for an LP$^{\mathcal{A}}$ program $\mathcal{P}$ is a consistent set of standard ground literals, that is, $I \subseteq B_{\mathcal{P}} \cup \neg.B_{\mathcal{P}}$ such that $I \cap \neg.I = \emptyset$. The set of all the interpretations of $\mathcal{P}$ is denoted by $\mathcal{I}_{\mathcal{P}}$.

Given an interpretation I, a standard literal ℓ is either (i) true if $\ell \in I$, or (ii) false if $\neg.\ell \in I$, or (iii) undefined otherwise. We denote by I^+ and I^- the set of standard positive and negative literals occurring in I, respectively. An interpretation I is *total* if there are no undefined literals w.r.t. I, otherwise I is *partial*.

An interpretation also provides a meaning to set terms, aggregate functions and aggregate literals, namely a multiset, a value, and a truth value, respectively. We first consider a total interpretation I. The evaluation $I(S)$ of a set term 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 and all the atoms in $Conj$ are 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 evaluation $I(f(S))$ of an 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 $\mathcal{P}$). A ground aggregate atom $f(S) \prec k$ is true w.r.t. I if both (i) $I(f(S)) \neq \bot$, and (ii) $I(f(S)) \prec k$ holds; otherwise, $f(S) \prec k$ is false.

We now consider a *partial* interpretation I and define *extension* of I an interpretation J such that $I \subseteq J$. If a ground aggregate literal ℓ is true (resp. false) w.r.t. *each total* interpretation J extending I, then ℓ is true (resp. false) w.r.t. I; otherwise ℓ is undefined.

Example 2. Let S_1 be the set term in the literal $\ell_1 = \#\texttt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1$, and consider a partial interpretation $I_1 = \{p(2,2)\}$. Since each total interpretation extending I_1 contains either $p(2,1)$ or $\texttt{not}\ p(2,1)$, we have either $I_1(S_1) = [2]$ or $I_1(S_1) = [1,2]$. Thus, the application of $\#\texttt{sum}$ yields either $2 > 1$ or $3 > 1$, and so ℓ_1 is true w.r.t. I_1.

Remark 1. Observe that our definitions of interpretation and truth values preserve "knowledge monotonicity": If an interpretation J extends I (i.e., $I \subseteq J$), then each literal which is true w.r.t. I is true w.r.t. J, and each literal which is false w.r.t. I is false w.r.t. J as well.

Model. Given an interpretation I, a rule r is *satisfied w.r.t.* I if either (i) the head atom is true w.r.t. I, or (ii) some body literal is false w.r.t. I, or (iii) both the head atom and some body literal are undefined w.r.t. I. An interpretation M is a *model* of an $\text{LP}^{\mathcal{A}}$ program $\mathcal{P}$ if all the rules in $Ground(\mathcal{P})$ are satisfied w.r.t. M.

2.2 The $\text{LP}^{\mathcal{A}}_{m,a}$ Language

In this section, we present the $\text{LP}^{\mathcal{A}}_{m,a}$ fragment of $\text{LP}^{\mathcal{A}}$.

Monotonicity. Given two interpretations I and J, we say that $I \leq J$ if $I^+ \subseteq J^+$ and $I^- \supseteq J^-$. A ground literal ℓ is *monotone* if for all interpretations I, J, such that $I \leq J$, we have that: (i) ℓ true w.r.t. I implies ℓ true w.r.t. J, and (ii) ℓ false w.r.t. J implies ℓ false w.r.t. I. A ground literal ℓ is *antimonotone* if the opposite happens, that is, for all interpretations I, J, such that $I \leq J$, we have that: (i) ℓ true w.r.t. J implies ℓ true w.r.t. I, and (ii) ℓ false w.r.t. I implies ℓ false w.r.t. J. A ground literal ℓ is *nonmonotone* if ℓ is neither monotone nor antimonotone.

Positive standard literals are monotone, while negative standard literals are antimonotone. Aggregate literals may be monotone, antimonotone or nonmonotone.

We denote by $\text{LP}^{\mathcal{A}}_{m,a}$ the fragment allowing only monotone and antimonotone aggregates. For an $\text{LP}^{\mathcal{A}}_{m,a}$ rule r, the set of its monotone and antimonotone body literals are denoted by $B^m(r)$ and $B^a(r)$, respectively.

3 Unfounded Sets and Well-Founded Semantics

In this section, we recall the notion of unfounded set for $\mathrm{LP}^{\mathcal{A}}_{m,a}$ programs defined in [1]. We then exploit unfounded sets for extending the well-founded semantics defined in [2] for aggregate-free programs to the $\mathrm{LP}^{\mathcal{A}}_{m,a}$ framework. A complexity analysis of the well-founded semantics herein defined will be presented in Section 4.

In the following we deal with ground programs, so we will usually denote by $\mathcal{P}$ a ground program. We will also exploit the notation $L \dot{\cup} \neg.L'$ for the set $(L \setminus L') \cup \neg.L'$, where L and L' are sets of standard ground literals.

Definition 1 (Unfounded Set [1])**.** A set $X \subseteq B_{\mathcal{P}}$ of ground atoms is an unfounded set for an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program $\mathcal{P}$ w.r.t. a (partial) interpretation I if and only if, for each rule $r \in \mathcal{P}$ having $H(r) \in X$, either (1) some (antimonotone) literal in $B^a(r)$ is false w.r.t. I, or (2) some (monotone) literal in $B^m(r)$ is false w.r.t. $I \dot{\cup} \neg.X$.

Intuitively, each rule having the head atom belonging to some unfounded set is either already satisfied w.r.t. I (in case condition (1) holds), or satisfiable by taking as false all the atoms in the unfounded set (in case condition (2) holds).

Example 3. Consider an interpretation $I_2 = \{a(1), a(2), a(3)\}$ for the following program $\mathcal{P}_2$:

$$r_1: \quad a(1) \coloneq \ell_2. \qquad\qquad r_2: \quad a(2). \qquad\qquad r_3: \quad a(3) \coloneq \ell_2.$$

where $\ell_2 = \#\mathtt{count}\{\langle 1\!:\!a(1)\rangle, \langle 2\!:\!a(2)\rangle, \langle 3\!:\!a(3)\rangle\} > 2$. Then $X_1 = \{a(1)\}$ is an unfounded set for $\mathcal{P}_2$ w.r.t. I_2, since condition (2) of Definition 1 holds for r_1 (the only rule with head $a(1)$). Indeed, the (monotone) literal appearing in $B^m(r_1)$ is false w.r.t. $I_2 \dot{\cup} \neg.X_1 = \{\mathtt{not}\ \ a(1), a(2), a(3)\}$. Similarly, we can check that $\{a(3)\}$ and $\{a(1), a(3)\}$ are unfounded sets for $\mathcal{P}_2$ w.r.t. I_2, and clearly also $\emptyset$ is. No other set of atoms is an unfounded set for $\mathcal{P}_2$ w.r.t. I_2.

The union of all the unfounded sets for a program $\mathcal{P}$ w.r.t. an interpretation I is an unfounded set as well; it is called the *greatest unfounded set* for $\mathcal{P}$ w.r.t. I and denoted $GUS_{\mathcal{P}}(I)$, cf. [1].

Definition 2 ([1])**.** Let $\mathcal{P}$ be an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program. The *immediate logical consequence operator* $\mathcal{T}_{\mathcal{P}} : \mathcal{I}_{\mathcal{P}} \to 2^{B_{\mathcal{P}}}$ and the *well-founded operator* $\mathcal{W}_{\mathcal{P}} : \mathcal{I}_{\mathcal{P}} \to 2^{B_{\mathcal{P}} \cup \neg.B_{\mathcal{P}}}$ are defined as follows:

$$\begin{aligned}
\mathcal{T}_{\mathcal{P}}(I) &= \{\ell \in B_{\mathcal{P}} \mid \exists r \in \mathcal{P} \text{ such that } H(r) = \ell \\
&\qquad\qquad\qquad \text{and all the literals in } B(r) \text{ are true w.r.t. } I\} \\
\mathcal{W}_{\mathcal{P}}(I) &= \mathcal{T}_{\mathcal{P}}(I) \cup \neg.GUS_{\mathcal{P}}(I).
\end{aligned}$$

Both $\mathcal{T}_{\mathcal{P}}$ and $GUS_{\mathcal{P}}$ are monotone operators, and so $\mathcal{W}_{\mathcal{P}}$ is monotone as well. Moreover, on aggregate-free programs, the well-founded operator of Definition 2 exactly coincides with the well-founded operator defined in [2].

We next show that a fixpoint of $\mathcal{W}_{\mathcal{P}}$ is a (partial) model.

Theorem 1. *Let $\mathcal{P}$ be an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program and M a (partial) interpretation. If M is a fixpoint of $\mathcal{W}_{\mathcal{P}}$, then M is a (partial) model of $\mathcal{P}$.*

Proof. Let assume $\mathcal{W}_{\mathcal{P}}(M) = M$ and consider a rule $r \in \mathcal{P}$. If all the literals in $B(r)$ are true w.r.t. M, then $H(r) \in \mathcal{T}_{\mathcal{P}}(M) \subseteq M$. If $H(r)$ is false w.r.t. M, then $H(r) \in GUS_{\mathcal{P}}(M)$. Since $GUS_{\mathcal{P}}(M)$ is an unfounded set for $\mathcal{P}$ w.r.t. M, either some literal in $B^a(r)$ is false w.r.t. M, or some literal in $B^m(r)$ is false w.r.t. $M \mathbin{\dot\cup} \neg.GUS_{\mathcal{P}}(M) = M$. We can then conclude that r is satisfied by M. $\square$

We can then prove that the sequence $W_0 = \emptyset$, $W_{n+1} = \mathcal{W}_{\mathcal{P}}(W_n)$ is well-defined, that is, each element of the sequence is an interpretation.

Theorem 2. Let $\mathcal{P}$ be an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program. The sequence $W_0 = \emptyset$, $W_{n+1} = \mathcal{W}_{\mathcal{P}}(W_n)$ is well-defined.

Proof. We use strong induction. The base case is trivial, since $W_0 = \emptyset$. In order to prove the consistency of W_{n+1}, we assume the consistency of every W_m such that $m \leq n$. We have to show that $GUS_{\mathcal{P}}(W_n) \cap W_{n+1} = \emptyset$. To this end, we show that if a set X of atoms is such that $X \cap W_{n+1} \neq \emptyset$, then X is not an unfounded set for $\mathcal{P}$ w.r.t. W_n. Let W_{m+1} be the first element of the sequence such that $X \cap W_{m+1} \neq \emptyset$ (note that $m \leq n$). Consider any atom $\ell \in X \cap W_{m+1}$. By definition of $\mathcal{T}_{\mathcal{P}}$, there is a rule $r \in \mathcal{P}$ having $H(r) = \ell$ and such that all the literals in $B(r)$ are true w.r.t. W_m. Note that no atom in W_m can belong to X (for the way W_{m+1} has been chosen). Thus, by Remark 1, all the literals in $B(r)$ are true w.r.t. both W_n and $W_n \mathbin{\dot\cup} \neg.X$ (we recall that $W_n \supseteq W_m$ because $\mathcal{W}_{\mathcal{P}}$ is monotone). This ends the proof, as neither condition (1) nor (2) of Definition 1 hold for ℓ. $\square$

The theorem above and the monotonicity of $\mathcal{W}_{\mathcal{P}}$ imply that $\mathcal{W}_{\mathcal{P}}$ admits a least fixpoint $\mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset)$ (as a consequence of Tarski's fixed point theorem [10]). The well-founded semantics of an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program $\mathcal{P}$ is exactly given by the least fixpoint of $\mathcal{W}_{\mathcal{P}}$, called the *well-founded model* of $\mathcal{P}$.

4 The Complexity of the Well-Founded Semantics

For the complexity analysis carried out in this section, we consider ground programs and polynomial-time computable aggregate functions (note that all sample aggregate functions appearing in this paper fall into this class).

We start by stating an important property of monotone and antimonotone aggregates, from which the tractability of the evaluation of $\mathrm{LP}^{\mathcal{A}}_{m,a}$ literals w.r.t. partial interpretations immediately follows.

Lemma 3. Let I be a partial interpretation for an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program $\mathcal{P}$, A a ground aggregate literal occurring in $\mathcal{P}$, I_{min} and I_{max} two total interpretations such that $I_{min} = I \cup \neg.(B_{\mathcal{P}} \setminus I)$ and $I_{max} = I \cup (B_{\mathcal{P}} \setminus \neg.I)$.

1. If A is a monotone literal, then A is true (resp. false) w.r.t. I if and only if A is true w.r.t. I_{min} (resp. false w.r.t. I_{max}).

2. If A is an antimonotone literal, then A is true (resp. false) w.r.t. I if and only if A is true w.r.t. I_{max} (resp. false w.r.t. I_{min}).

Proof. We start by noting that I_{min} (resp. I_{max}) is a total interpretation extending I and such that all the standard atoms which are undefined w.r.t. I are false w.r.t. I_{min} (resp. true w.r.t. I_{max}). Thus, we have $(*)$ $I_{min} \leq I \leq I_{max}$. If A is monotone and true w.r.t. I_{min} (resp. false w.r.t. I_{max}), then A is true (resp. false) w.r.t. I because of $(*)$. If A is antimonotone and true w.r.t. I_{max} (resp. false w.r.t. I_{min}), then A is true (resp. false) w.r.t. I because of $(*)$. We end the proof by observing that if A is true (resp. false) w.r.t. I, then A is true (resp. false) w.r.t. I_{min} and I_{max} by definition. □

We now define an operator which can be used for computing $B_{\mathcal{P}} \setminus GUS_{\mathcal{P}}(I)$, and thus $GUS_{\mathcal{P}}(I)$ itself.

Definition 3. *Let I be a partial interpretation for an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program $\mathcal{P}$, and $Y \subseteq B_{\mathcal{P}}$, let*

$$\begin{aligned}\phi_I(Y) = \{\ell \in B_{\mathcal{P}} \mid\ & \exists\, r \in \mathcal{P} \text{ with } H(r) = \ell \text{ such that}\\ & \text{no (antimonotone) literal in } B^a(r) \text{ is false w.r.t. } I, \text{ and}\\ & \text{all the (monotone) literals in } B^m(r) \text{ are true w.r.t. } Y \setminus \neg.I^-\}\end{aligned}$$

Proposition 1. *For a partial interpretation I for an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program $\mathcal{P}$, the least fixpoint $\phi^{\omega}_I(\emptyset)$ coincides with the set $B_{\mathcal{P}} \setminus GUS_{\mathcal{P}}(I)$.*

We can now prove the tractability of the well-founded semantics.

Theorem 4. Given an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program $\mathcal{P}$, $\mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset)$ is polynomial-time computable.

Proof. Both $\phi^{\omega}_I(\emptyset)$ and $\mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset)$ are obtainable by at most $|B_{\mathcal{P}}|$ application of ϕ and $\mathcal{W}_{\mathcal{P}}$, respectively. □

Well-founded semantics is also hard for polynomial-time. In particular, deciding whether a (ground) atom is true w.r.t. the well-founded semantics is P-complete, as this task is P-hard even for aggregate-free programs [11].

5 Compilation into Standard Logic Programming

We now briefly present a strategy for representing #`count`, #`sum` and #`times`[1] with standard constructs. The compilation is in the spirit of the one introduced for #`min` and #`max` in [12] and defines a subprogram computing the value of a (recursive) aggregate exploiting monotone/antimonotone peculiarities of the specific aggregate function. For these reasons, the compilation is referred to as *monotone/antimonotone encoding* (*mae* encoding).

[1]Since we are considering only monotone and antimonotone aggregate literals, the domains of #`sum` and #`times` are assumed to be $\mathbb{N}$ and $\mathbb{N}^+$, respectively.

The monotone/antimonotone encoding of an $\mathrm{LP}^{\mathcal{A}}_{m,a}$ program $\mathcal{P}$ is obtained by replacing each aggregate literal $A = f(S) \prec T$ by a new predicate symbol $f_\prec$ whose definition is a compilation of A into standard LP. In the compilation we exploit a total order $<$ on the elements of $U_\mathcal{P} \cup \{\bot\}$, where $\bot$ is a symbol not occurring in $\mathcal{P}$ and such that $\bot < u$ for each $u \in U_\mathcal{P}$. We further assume the presence of a "built-in" relation $<(\overline{Y}, \overline{Y'})$, where $\overline{Y} = Y_1, \ldots, Y_n$ and $\overline{Y'} = Y'_1, \ldots, Y'_n$ are lists of terms, defining those pairs of tuples $\overline{y}, \overline{y'}$ such that $\overline{y}$ precedes $\overline{y'}$ in the lexicographical order induced by $<$.

For simplicity, we consider an aggregate literal $f(\{\overline{Y} : p(\overline{Y}, \overline{Z})\}) \prec k$ having only local variables. We introduce a new predicate symbol f_{aux} of arity $|\overline{Y}| + 1$. Intuitively, an atom $f_{aux}(\overline{y}, s)$ is intended for representing that there are at least s tuples $\overline{y'}$ such that $\overline{y}$ does not precede $\overline{y'}$ and $p(\overline{y'}, \overline{z})$ is true, for some tuple $\overline{z}$. For this purpose, we first evaluate the aggregate function on the empty set, and then opportunely increase its evaluation for greater sets. For guaranteeing that each element in the set term is never taken twice, we exploit the lexicographical order induced by $<$. The two rules below encode this monotonically increasing evaluation:

$$\begin{array}{l} f_{aux}(\overline{\bot}, \alpha). \\ f_{aux}(\overline{Y'}, \beta) :- f_{aux}(\overline{Y}, S), \\ \qquad p(\overline{Y'}, \overline{Z}),\ <(\overline{Y}, \overline{Y'}). \end{array} \quad \text{where} \left\{ \begin{array}{ll} \alpha = 0,\ \beta = S + 1, & \text{if } f = \#\texttt{count} \\ \alpha = 0,\ \beta = S + Y'_1, & \text{if } f = \#\texttt{sum} \\ \alpha = 1,\ \beta = S \times Y'_1, & \text{if } f = \#\texttt{times} \end{array} \right.$$

In case $\prec \in \{\geq, >\}$, the aggregate is definitely true if *some* $f_{aux}(\overline{y}, s)$ with $s \prec k$ is true:

$$f_\geq :- f_{aux}(\overline{Y}, S),\ S \geq k. \qquad\qquad f_> :- f_{aux}(\overline{Y}, S),\ S > k.$$

For $\prec \in \{\leq, <\}$, to conclude the truth of the aggregate we have to be sure that no $f_{aux}(\overline{y}, s)$ with $s \not\prec k$ is true. We can model this aspect by means of the following rules:

$$f_\leq :- \texttt{not}\ f_>. \qquad\qquad f_< :- \texttt{not}\ f_\geq.$$

Extending the technique to aggregate literals with global variables is quite simple: Global variables are added to the arguments of all the atoms used in the compilation, and a new predicate $f_{group-by}$ is used for collecting their possible substitutions.

6 Implementation and Experimental Results

The well-founded semantics for $\mathrm{LP}^{\mathcal{A}}_{m,a}$ programs has been implemented in the DLV [13] system. In this section, we give a very rough description of the implementation and discuss the results of our experimentation aimed at assessing the efficiency of the prototype.

6.1 System Architecture and Usage

We have developed a prototype system by implementing our well-founded operator in the core of DLV. Both the *intelligent grounding* module and the *model generator* module of DLV have been modified for the implementation of the well-founded semantics for $\mathrm{LP}^{\mathcal{A}}_{m,a}$ programs. In particular, in the grounding module, we extended the technique for aggregate literal instantiation to correctly deal with aggregate atoms occurring recursively, while in the model generator we implemented a polynomial time algorithm for computing the least fixpoint of the operator ϕ introduced in Section 4.

In our prototype, the well-founded semantics is adopted when the user invokes DLV with `-wf` or `--well-founded`. If none of these two options are specified, then the stable model semantics is adopted as usual. In both cases, the system exploits the well-founded operator $\mathcal{W}_\mathcal{P}$ introduced in Section 3. For the stable model semantics, the well-founded model is profitably used for search space pruning after each non-deterministic choice. For the well-founded semantics, instead, the well-founded model is yielded immediately after the least fixpoint of the well-founded operator is computed; in this case, the system outputs two sets, representing true and undefined (standard) atoms w.r.t. the well-founded model.

An executable of the DLV system supporting well-founded semantics for $\mathrm{LP}^{\mathcal{A}}_{m,a}$ programs is available at `http://www.dlvsystem.com/dlvRecAggr/`.

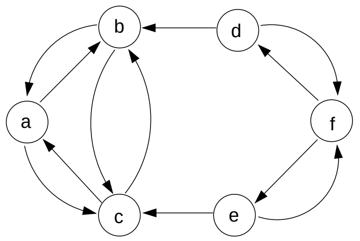
6.2 Experimental Results

To our knowledge, the implemented prototype is currently the only system supporting a well-founded semantics for logic programs with recursive aggregates. In particular, one of the major systems supporting the well-founded semantics, XSB, has some support for aggregates, but (apart from #`min` and #`max`) not when they occur in recursive definitions. Therefore, our experiments have been designed for evidencing the computational advantages of aggregate constructs w.r.t. equivalent encodings without aggregates.

The experiments have been performed on a 3GHz Intel® Xeon® processor system with 4GB RAM under the Debian 4.0 operating system with GNU/Linux 2.6.23 kernel. The tested systems have been compiled with GCC 4.4.1. For every instance, we have allowed a maximum running time of 600 seconds (10 minutes) and a maximum memory usage of 3GB.

For the benchmark, we have defined the *Attacks* problem, a problem similar to the classic *Win-Lose* problem often used as an example for the well-founded semantics of standard logic programs (see for instance [9]). In the Attacks problem, a set of p players, each one attacking n other players, and a positive integer m are given. A player wins if no more than m winners attack it.

Example 4. An instance of the Attacks problem in which $p = 6$, $n = 2$ and $m = 1$ could be the one represented by the following directed graph:



Since d is attacked only by f, we can conclude that d is a winner. Similarly for e. Therefore, f is a loser because f is attacked by d and e, which are winners. For the other players, namely a, b and c, it is not possible to determine whether they are winners or losers.

The encodings used in our experiments are reported below, where max, $player$ and $attacks$ are EDB predicates representing the parameter m, the set of players and the attacks done by the players, respectively.

Aggregate-Based Encoding:

$$win(X) \text{ :- } max(M),\ player(X),\ \#\texttt{count}\{Y : attacks(Y,X),\ win(Y)\} \leq M.$$

This encoding exploits aggregate constructs and is a natural representation of the Attacks problem.

Join-Based Encoding:

$$\begin{aligned}
&win(X) \text{ :- } player(X),\ \texttt{not}\ \ lose(X).\\
&lose(X) \text{ :- } max(1),\ attacks(Y_1,X),\ win(Y_1),\\
&\qquad\qquad attacks(Y_2,X),\ win(Y_2),\ Y_1 < Y_2.\\
&lose(X) \text{ :- } max(2),\ attacks(Y_1,X),\ win(Y_1),\\
&\qquad\qquad attacks(Y_2,X),\ win(Y_2),\ Y_1 < Y_2,\\
&\qquad\qquad attacks(Y_3,X),\ win(Y_3),\ Y_1 < Y_3,\ Y_2 < Y_3.\\
&lose(X) \text{ :- } max(3),\ \ldots
\end{aligned}$$

In this encoding there is a rule for each possible value of the m parameter. However, the presence of the predicate max in the body of these rules assures that the solvers considered in our experiments automatically disregard rule instances that do not match the given max.

Mae-Based Encoding *(from monotone/antimonotone encoding)*:

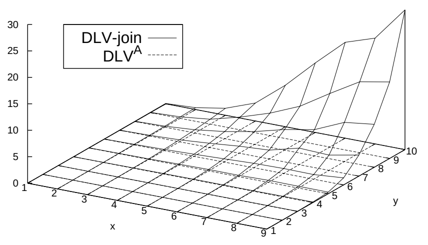
$$\begin{aligned}
&win(X) \text{ :- } player(X),\ \texttt{not}\ \ lose(X).\\
&lose(X) \text{ :- } count(X,Y,S),\ max(M),\ S > M.\\
&count(X,Y,1) \text{ :- } aux(X,Y).\\
&count(X,Y',S+1) \text{ :- } count(X,Y,S),\ aux(X,Y'),\ Y < Y'.\\
&aux(X,Y) \text{ :- } attacks(Y,X),\ win(Y).
\end{aligned}$$

This is an encoding in the spirit of [12] substantially obtained by applying the compilation presented in Section 5 (with some minor simplifications). Intuitively, an atom $count(x,y,s)$ stands for "there are at least s constants y' such that $y' \leq y$ and $attacks(y',x),\ win(y')$ is true". The definition of $count$ exploits the natural order of integers for guaranteeing that no y' is counted twice.

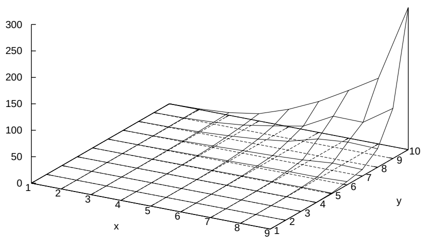
Example 5. The EDB representing the instance in Example 4 is the following:

$player(a).$	$player(d).$	$attacks(a,b).$	$attacks(c,a).$	$attacks(e,c).$
$player(b).$	$player(e).$	$attacks(a,c).$	$attacks(c,b).$	$attacks(e,f).$
$player(c).$	$player(f).$	$attacks(b,a).$	$attacks(d,b).$	$attacks(f,d).$
$max(1).$		$attacks(b,c).$	$attacks(d,f).$	$attacks(f,e).$

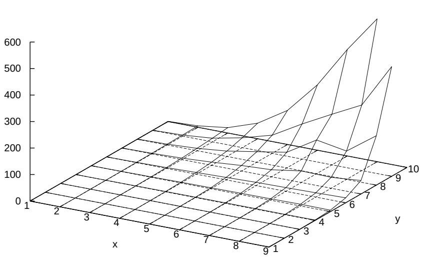
For all the encodings, the well-founded model restricted to the win predicate is $\{win(d),\ win(e),\ \texttt{not}\ \ win(f)\}$. Note that $win(a)$, $win(b)$ and $win(c)$ are undefined.



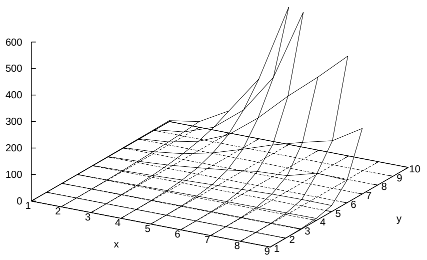
(a) 100 players



(b) 200 players



(c) 400 players



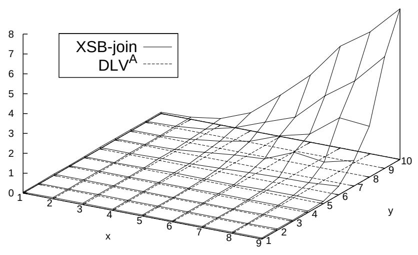
(d) 800 players

Figure 1: *Attacks:* Average execution time of DLV running the aggregate-based encoding and DLV running the join-based encoding.

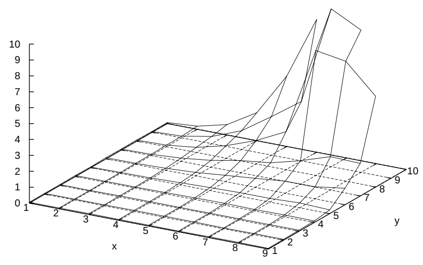
We performed an intensive experimentation for the *Attacks* problem, varying the parameters p, m and n. For each combination of these parameters, we measured the average execution time of DLV and XSB on 3 randomly generated instances. The results of our experimentation are reported in figures 1–4. In the graphs, $\mathrm{DLV}^{\mathrm{A}}$ is the implemented prototype with the aggregate-based encoding, DLV-join and DLV-mae the implemented prototype with the aggregate-free encodings, XSB-join and XSB-mae the XSB system with the aggregate-free encodings (as mentioned earlier, XSB does not support recursive aggregates). For the XSB system, we explicitly set indexes and tabled predicates for optimizing its computation.

For each graph, the number of players is fixed, while parameters m (x-axis) and n (y-axis) vary. Therefore, the size of the instances grows moving from left to right along the y-axis, while is invariant w.r.t. the x-axis. However, the number of joins required by the join-based encoding depends on the parameter m. As a matter of fact, we can observe in the graphs in figures 1–2 that the average execution time of the join-based encoding increases along both the x- and y-axis (for both DLV and XSB). Instead, for the encoding exploiting aggregates, and for the mae encoding, the average execution time depends only on instance size, as shown in the graphs in Figures 3–4.

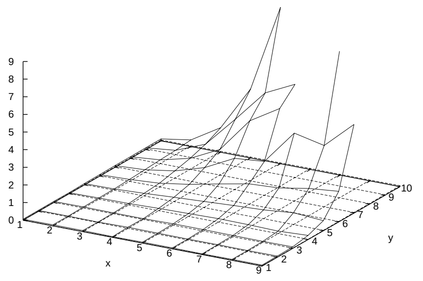
For the join-based encoding, XSB is generally faster than DLV, but consumes much more memory. Indeed, in Figure 2, we can observe that XSB terminates its computation in a few seconds for the smaller instances, but rapidly runs out of memory on slightly larger instances.



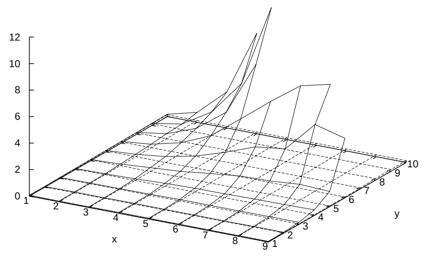
(e) 100 players



(f) 200 players



(g) 400 players



(h) 800 players

Figure 2: *Attacks:* Average execution time of DLV running the aggregate-based encoding and XSB running the join-based encoding.

Considering the mae-based encoding, we can observe significant performance gains for both DLV and XSB (see figures 3–4). Indeed, both systems complete their computation in the allowed time and memory on larger instances. Computational advantages of the mae-based encoding w.r.t. the join-based encoding are particularly evident for XSB, which can solve all instances of the benchmark with this encoding. However, also XSB with the mae-based encoding is outperformed by DLV with native support for aggregate constructs (see Figure 4).

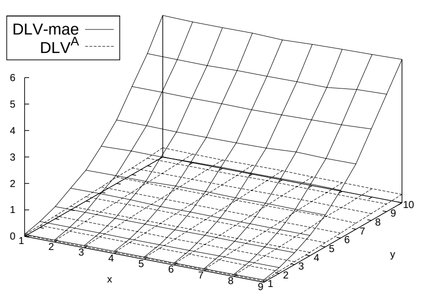
In sum, the experimental results highlight that the presence of aggregate constructs can significantly speed-up the computation. Indeed, the encoding exploiting recursive aggregates outperforms the aggregate-free encodings in all the instances.

7 Related Work

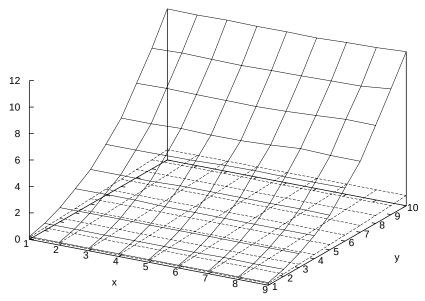
The definition of well-founded semantics for $\text{LP}^{\mathcal{A}}$ has been a challenge of major interest in the last years. The first attempts, not relying on a notion of unfounded set, have been defined on a limited framework. Some of these are discussed in [3].

A first attempt to define a well-founded semantics for $\text{LP}^{\mathcal{A}}$ without restriction has been done in [3]. Even if the semantics defined in [3] has the advantage of being based on a notion of unfounded set, it often leaves too many undefined literals.

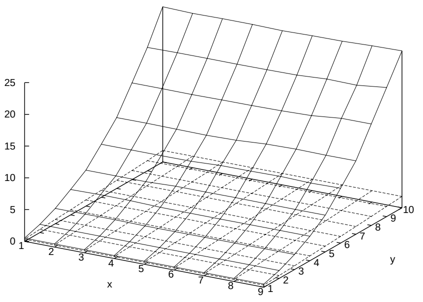
Our work is particularly related to [14], where $\tilde{\mathcal{D}}$-well-founded semantics has been defined. $\tilde{\mathcal{D}}$-well-founded semantics is based on approximating operators, not on unfounded sets, and the semantics depends on the adopted approximating ag-



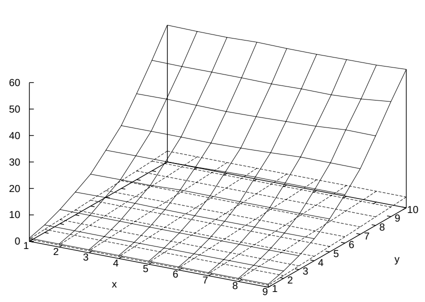
(i) 1600 players



(j) 3200 players



(k) 6400 players



(l) 12800 players

Figure 3: *Attacks:* Average execution time of DLV running the aggregate-based encoding and DLV running the *mae*-based encoding.

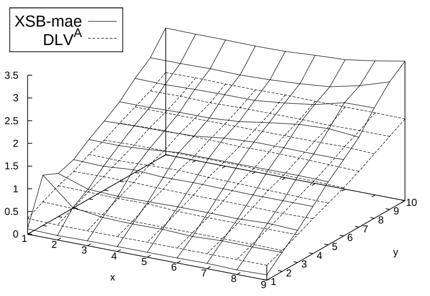
gregate relation; the authors discuss *trivial*, *bound* and *ultimate* approximating aggregate relations. Semantics relying on *trivial* approximating aggregates is very imprecise, but it is still suitable for the class of stratified aggregate programs. Both *trivial* and *bound* approximations have polynomial complexity, while *ultimate* has been proved to be intractable for nonmonotone aggregate functions [4]. For $\mathrm{LP}^{\mathcal{A}}_{m,a}$ programs, the $\tilde{\mathcal{D}}$-well-founded semantics under *ultimate* and *bound* approximations coincide with the well-founded semantics presented in this paper.

Other works attempted to define stronger notions of well-founded semantics (also for programs with aggregates), like the Ultimate Well-Founded Semantics of [5], or WFS^1 and WFS^2 of [6]. Whether a characterization in terms of unfounded sets exists for these semantics is an open problem.

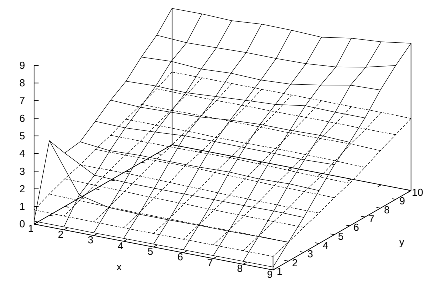
Programs with aggregates are related to *abstract constraint programs* [15], for which no well-founded semantics has been defined to our knowledge. The definitions in this paper can be easily adapted to cover abstract constraints.

8 Conclusion

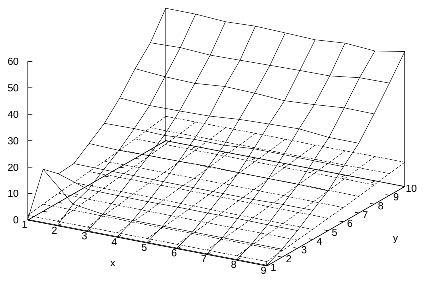
In this paper we analyzed $\mathrm{LP}^{\mathcal{A}}_{m,a}$ programs under well-founded semantics. We showed that computing this semantics is a tractable problem. Indeed, the semantics is given by the least fixpoint of the well-founded operator $\mathcal{W}_{\mathcal{P}}$. The fixpoint is



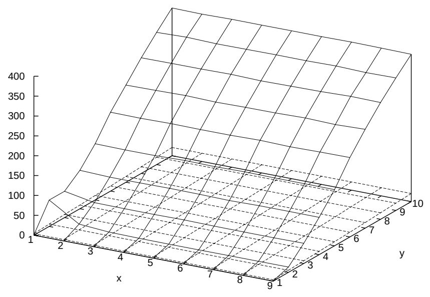
(m) 6400 players



(n) 12800 players



(o) 25600 players



(p) 51200 players

Figure 4: *Attacks:* Average execution time of DLV running the aggregate-based encoding and XSB running the *mae*-based encoding.

reached in a polynomial number of applications of $\mathcal{W}_\mathcal{P}$ (w.r.t. the size of the input program), each of them requiring polynomial time. For showing that an application of $\mathcal{W}_\mathcal{P}$ is polynomial-time feasible, we have proved that evaluating monotone and antimonotone aggregate literals remains polynomial-time doable also for partial interpretations, since in this case only one of the possibly exponential extensions must be checked. For a monotone aggregate literal, this extension is obtained by falsifying each undefined literal, while for an antimonotone aggregate literal, each undefined literal is taken as true in the extension.

Motivated by these positive theoretical results, we have implemented the first system supporting a well-founded semantics for unrestricted $\mathrm{LP}^{\mathcal{A}}_{m,a}$. Allowing for using monotone and antimonotone aggregate literals, the implemented prototype is ready for experimenting with the $\mathrm{LP}^{\mathcal{A}}_{m,a}$ framework. The experiments conducted on the Attacks benchmark highlight the computational gains of a native implementation of aggregate constructs w.r.t. equivalent encodings in standard LP.

Acknowledgments

This research has been partly supported by Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l., and by MIUR under the PRIN project LoDeN.

References

[1] Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). (2005) 406–411

[2] Van Gelder, A., Ross, K.A., Schlipf, J.S.: The Well-Founded Semantics for General Logic Programs. Journal of the ACM **38**(3) (1991) 620–650

[3] Kemp, D.B., Stuckey, P.J.: Semantics of Logic Programs with Aggregates. In Saraswat, V.A., Ueda, K., eds.: Proceedings of the International Symposium on Logic Programming (ISLP'91), MIT Press (1991) 387–401

[4] Pelov, N.: Semantics of Logic Programs with Aggregates. PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium (2004)

[5] Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate Well-Founded and Stable Model Semantics for Logic Programs with Aggregates. In Codognet, P., ed.: Proceedings of the 17th International Conference on Logic Programming, Springer Verlag (2001) 212–226

[6] Dix, J., Osorio, M.: On Well-Behaved Semantics Suitable for Aggregation. In: Proceedings of the International Logic Programming Symposium (ILPS '97), Port Jefferson, N.Y. (1997)

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

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

[9] Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)

[10] Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math **5** (1955) 285–309

[11] Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425

[12] Alviano, M., Faber, W., Leone, N.: Compiling minimum and maximum aggregates into standard ASP. In Formisano, A., ed.: Proceedings of the 23rd Italian Conference on Computational Logic (CILC 2008). (2008)

[13] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562

[14] Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and Stable Semantics of Logic Programs with Aggregates. Theory and Practice of Logic Programming **7**(3) (2007) 301–353

[15] Liu, L., Truszczyński, M.: Properties and applications of programs with monotone and convex constraints. Journal of Artificial Intelligence Research **27** (2006) 299–334