TPLP **20** (2): 294–309, 2020. © Cambridge University Press 2019
doi:10.1017/S1471068419000450 First published online 20 December 2019

ASP-Core-2 Input Language Format

FRANCESCO CALIMERI
DeMaCS, Università della Calabria, Rende, Italy
(*e-mail:* calimeri@mat.unical.it)

WOLFGANG FABER and MARTIN GEBSER
Institut für Angewandte Informatik, Alpen-Adria-Universität, Klagenfurt, Austria
(*e-mails:* Wolfgang.Faber@aau.at, Martin.Gebser@aau.at)

GIOVAMBATTISTA IANNI
DeMaCS, Università della Calabria, Rende, Italy
(*e-mail:* ianni@mat.unical.it)

ROLAND KAMINSKI
Institute of Computer Science, University of Potsdam, Potsdam, Germany
(*e-mail:* kaminski@cs.uni-potsdam.de)

THOMAS KRENNWALLNER
XIMES GmbH, Vienna, Austria
(*e-mail:* tk@postsubmeta.net)

NICOLA LEONE
DeMaCS, Università della Calabria, Rende, Italy
(*e-mail:* leone@mat.unical.it)

MARCO MARATEA
DIBRIS, University of Genova, Genova, Italy
(*e-mail:* marco@dibris.unige.it)

FRANCESCO RICCA
DeMaCS, Università della Calabria, Rende, Italy
(*e-mail:* ricca@mat.unical.it)

TORSTEN SCHAUB
Institute of Computer Science, University of Potsdam, Potsdam, Germany
(*e-mail:* schaub@cs.uni-potsdam.de)

submitted 24 January 2019; revised 13 November 2019; accepted 15 November 2019

Abstract

Standardization of solver input languages has been a main driver for the growth of several areas within knowledge representation and reasoning, fostering the exploitation in actual applications. In this document, we present the ASP-Core-2 standard input language for Answer Set Programming, which has been adopted in ASP Competition events since 2013.

KEYWORDS: Answer Set Programming, standard language, knowledge representation and reasoning, standardization

1 Introduction

The process of standardizing the input languages of solvers for knowledge representation and reasoning research areas has been of utmost importance for the growth of the related research communities: this has been the case for, for example, the CNF-DIMACS format for SAT, then extended to describe input formats for Max-SAT and QBF problems, the OPB format for pseudo-Boolean problems, somehow at the intersection between the CNF-DIMACS format and the LP format for integer linear programming, the XCSP3 format for CP solving, SMT-LIB format for SMT solving, and the STRIPS/PDDL language for automatic planning. The availability of such common input languages has led to the development of efficient solvers in different KR communities, through a series of solver competitions that have pushed the adoption of these standards. The availability of efficient solvers, together with a presence of a common interface language, has helped the exploitation of these methodologies in applications.

The same has happened for answer set programming (ASP) (Brewka *et al.* 2011), a well-known approach to knowledge representation and reasoning with roots in the areas of logic programming and nonmonotonic reasoning (Gelfond and Lifschitz 1991), through the development of the ASP-Core language (Calimeri *et al.* 2011b). The first ASP-Core version was a rule-based language whose syntax stems from plain Datalog and Prolog and was a conservative extension to the nonground case of the `Core` language adopted in the First ASP Competition held in 2002 during the Dagstuhl Seminar "Nonmonotonic Reasoning, Answer Set Programming and Constraints."[1] It featured a restricted set of constructs, that is disjunction in the rule heads, both strong and negation-as-failure negation in rule bodies, as well as nonground rules.

In this document, we present the latest evolution of ASP-Core, namely, ASP-Core-2, which currently constitutes the standard input language of ASP solvers adopted in the ASP Competition series since 2013 (Calimeri *et al.* 2014; Calimeri *et al.* 2016; Gebser *et al.* 2017b; Gebser *et al.* 2017a). ASP-Core-2 substantially extends its predecessor by incorporating many language extensions that became mature and widely adopted over the years in the ASP community, such as aggregates, weak constraints, and function symbols. The ASP Competition series pushed its adoption and significantly contributed both to the availability of efficient solvers for ASP (Lierler *et al.* 2016; Gebser *et al.* 2018a) and to the exploitation of the ASP methodology in academic and in industrial applications (Erdem *et al.* 2016; Leone and Ricca 2015; Gebser *et al.* 2018b). In the following, we first present syntax and semantics for the basic building blocks of the language and then introduce more expressive constructs such as choice rules and aggregates, which helps with obtaining compact problem formulations. Eventually, we present syntactic restrictions for the use of ASP-Core-2 in practice.

2 ASP-Core-2 language syntax

For the sake of readability, the language specification is herein given in the traditional mathematical notation. A lexical matching table from the following notation to the actual raw input format is provided in Section 6.

[1] https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=02381.

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

Terms. Terms are either *constants*, *variables*, *arithmetic terms*, or *functional terms*. Constants can be either *symbolic constants* (strings starting with some lowercase letter), *string constants* (quoted strings), or *integers*. Variables are denoted by strings starting with some uppercase letter. An *arithmetic term* has form $-(t)$ or $(t \diamond u)$ for terms t and u with $\diamond \in \{\text{“+”}, \text{“−”}, \text{“∗”}, \text{“/”}\}$; parentheses can optionally be omitted in which case standard operator precedences apply. Given a *functor* f (the *function name*) and terms $t_1, \ldots, t_n$, the expression $f(t_1, \ldots, t_n)$ is a *functional term* if $n > 0$, whereas $f()$ is a synonym for the symbolic constant f.

Atoms and Naf-Literals. A *predicate atom* has form $p(t_1, \ldots, t_n)$, where p is a *predicate name*, $t_1, \ldots, t_n$ are the terms, and $n \geq 0$ is the arity of the predicate atom; a predicate atom $p()$ of arity 0 is likewise represented by its predicate name p without parentheses. Given a predicate atom q, q and $\neg q$ are *classical atoms*. A *built-in atom* has form $t \prec u$ for terms t and u with $\prec \in \{\text{“<”}, \text{“}\leq\text{”}, \text{“=”}, \text{“}\neq\text{”}, \text{“>”}, \text{“}\geq\text{”}\}$. Built-in atoms a as well as the expressions a and **not** a for a classical atom a are *naf-literals*.

Aggregate Literals. An *aggregate element* has form

$$t_1, \ldots, t_m : l_1, \ldots, l_n$$

where $t_1, \ldots, t_m$ are the terms and $l_1, \ldots, l_n$ are the naf-literals for $m \geq 0$ and $n \geq 0$.

An *aggregate atom* has form

$$\#\text{aggr } E \prec u$$

where $\#\text{aggr} \in \{\text{“\#count”}, \text{“\#sum”}, \text{“\#max”}, \text{“\#min”}\}$ is an *aggregate function name*, $\prec \in \{\text{“<”}, \text{“}\leq\text{”}, \text{“=”}, \text{“}\neq\text{”}, \text{“>”}, \text{“}\geq\text{”}\}$ is an *aggregate relation*, u is a term, and E is a (possibly infinite) collection of aggregate elements, which are syntactically separated by “;”. Given an aggregate atom a, the expressions a and **not** a are *aggregate literals*. In the following, we write *atom* (respectively, *literal*) without further qualification to refer to some classical, built-in, or aggregate atom (respectively, naf- or aggregate literal).

We here allow for infinite collections of aggregate elements because the semantics in Section 3 is based on ground instantiation, which may map some nonground aggregate element to infinitely many ground instances. The semantics of Abstract Gringo (Gebser *et al.* 2015) handles such cases by means of infinitary propositional formulas, while the Abstract Gringo language avoids infinite collections of aggregate elements in the input. As shown in Harrison and Lifschitz (2018), the semantics by ground instantiation or infinitary propositional formulas, respectively, is equivalent on the common subset of Abstract Gringo and ASP-Core-2. Moreover, we note that the restrictions to ASP-Core-2 programs claimed in Section 5 require the existence of a finite equivalent ground instantiation for each input, so that infinite collections of aggregate elements do not show up in practice.

Rules. A *rule* has form

$$h_1 \mid \ldots \mid h_m \leftarrow b_1, \ldots, b_n.$$

where $h_1, \ldots, h_m$ are the classical atoms and $b_1, \ldots, b_n$ are the literals for $m \geq 0$ and $n \geq 0$. When $n = 0$, the rule is called a *fact*. When $m = 0$, the rule is referred to as a *constraint*.

Weak Constraints. A *weak constraint* has form

$$:\sim b_1, \ldots, b_n. [w@l, t_1, \ldots, t_m]$$

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

where $t_1, \ldots, t_m$ are the terms and $b_1, \ldots, b_n$ are the literals for $m \geq 0$ and $n \geq 0$; w and l are the terms standing for a *weight* and a *level*. Writing the part "@l" can optionally be omitted if $l = 0$; that is, a weak constraint has level 0 unless specified otherwise.

Queries. A *query* Q has form $a?$, where a is a classical atom.

Programs. An *ASP-Core-2 program* is a set of rules and weak constraints, possibly accompanied by a (single) query.[2] A program (rule, weak constraint, query, literal, aggregate element, etc.) is *ground* if it contains no variables.

3 Semantics

We herein give the full model-theoretic semantics of ASP-Core-2. As for nonground programs, the semantics extends the traditional notion of Herbrand interpretation, taking care of the fact that *all* integers are part of the Herbrand universe. The semantics of propositional programs is based on Gelfond and Lifschitz (1991), extended to aggregates according to Faber *et al.* (2004), Faber *et al.* (2011). Choice atoms (Simons *et al.* 2002) are treated in terms of the reduction given in Section 4.

We restrict the given semantics to programs containing nonrecursive aggregates (see Section 5 for this and further restrictions to the family of admissible programs), for which the general semantics presented herein is in substantial agreement with a variety of proposals for adding aggregates to ASP (Kemp and Stuckey 1991; Van Gelder 1992; Osorio and Jayaraman 1999; Ross and Sagiv 1997; Denecker *et al.* 2001; Gelfond 2002; Simons *et al.* 2002; Dell'Armi *et al.* 2003; Pelov and Truszczyński 2004; Pelov *et al.* 2004; Ferraris 2005; Pelov *et al.* 2007).

Herbrand Interpretation. Given a program P, the *Herbrand universe* of P, denoted by U_P, consists of all integers and (ground) terms constructible from constants and functors appearing in P. The *Herbrand base* of P, denoted by B_P, is the set of all (ground) classical atoms that can be built by combining predicate names appearing in P with terms from U_P as arguments. A (Herbrand) *interpretation* I for P is a subset of B_P.

Ground Instantiation. A *substitution* σ is a mapping from a set V of variables to the Herbrand universe U_P of a given program P. For some object O (rule, weak constraint, query, literal, aggregate element, etc.), we denote by $O\sigma$ the object obtained by replacing each occurrence of a variable $v \in V$ by $\sigma(v)$ in O.

A variable is *global* in a rule, weak constraint or query r if it appears outside of aggregate elements in r. A substitution from the set of global variables in r is a *global substitution for* r; a substitution from the set of variables in an aggregate element e is a (local) *substitution for* e. A global substitution σ for r (or substitution σ for e) is *well formed* if the arithmetic evaluation, performed in the standard way, of any arithmetic subterm $(-(t)$ or $(t \diamond u)$ with $\diamond \in \{\text{“+”}, \text{“−”}, \text{“∗”}, \text{“/”}\})$ appearing outside of aggregate elements in $r\sigma$ (or appearing in $e\sigma$) is well defined.

[2] Unions of conjunctive queries (and more) can be expressed by including appropriate rules in a program.

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

Given a collection E of aggregate elements, the *instantiation* of E is the following set of aggregate elements:

$$\text{inst}(E) = \bigcup_{e \in E} \{e\sigma \mid \sigma \text{ is a well-formed substitution for } e\}$$

A *ground instance* of a rule, weak constraint or query r is obtained in two steps: (1) a well-formed global substitution σ for r is applied to r; (2) for every aggregate atom $\#\text{aggr}\ E \prec u$ appearing in $r\sigma$, E is replaced by $\text{inst}(E)$.

The *arithmetic evaluation* of a ground instance r of some rule, weak constraint or query is obtained by replacing any maximal arithmetic subterm appearing in r by its integer value, which is calculated in the standard way.[3] The *ground instantiation* of a program P, denoted by $grnd(P)$, is the set of arithmetically evaluated ground instances of rules and weak constraints in P.

Term Ordering and Satisfaction of naf-Literals. A classical atom $a \in B_P$ is *true* with respect to a interpretation $I \subseteq B_P$ if $a \in I$. A naf-literal of the form **not** a, where a is a classical atom, is *true* with respect to I if $a \notin I$, and it is false otherwise.

To determine whether a built-in atom $t \prec u$ (with $\prec \in \{\text{“}<\text{”}, \text{“}\leq\text{”}, \text{“}=\text{”}, \text{“}\neq\text{”}, \text{“}>\text{”}, \text{“}\geq\text{”}\}$) holds, we rely on a total order $\preceq$ on terms in U_P defined as follows:

- $t \preceq u$ for integers t and u if $t \leq u$;
- $t \preceq u$ for any integer t and any symbolic constant u;
- $t \preceq u$ for symbolic constants t and u if t is lexicographically smaller than or equal to u;
- $t \preceq u$ for any symbolic constant t and any string constant u;
- $t \preceq u$ for string constants t and u if t is lexicographically smaller than or equal to u;
- $t \preceq u$ for any string constant t and any functional term u;
- $t \preceq u$ for functional terms $t = f(t_1, \ldots, t_m)$ and $u = g(u_1, \ldots, u_n)$ if
  - $m < n$ (the arity of t is smaller than the arity of u),
  - $m \leq n$ and $g \not\preceq f$ (the functor of t is smaller than the one of u, while arities coincide) or
  - $m \leq n$, $f \preceq g$ and, for any $1 \leq j \leq m$ such that $t_j \not\preceq u_j$, there is some $1 \leq i < j$ such that $u_i \not\preceq t_i$ (the tuple of arguments of t is smaller than or equal to the arguments of u).

Then, $t \prec u$ is *true* with respect to I if $t \preceq u$ for $\prec = \text{“}\leq\text{”}$; $u \preceq t$ for $\prec = \text{“}\geq\text{”}$; $t \preceq u$ and $u \not\preceq t$ for $\prec = \text{“}<\text{”}$; $u \preceq t$ and $t \not\preceq u$ for $\prec = \text{“}>\text{”}$; $t \preceq u$ and $u \preceq t$ for $\prec = \text{“}=\text{”}$; $t \not\preceq u$ or $u \not\preceq t$ for $\prec = \text{“}\neq\text{.”}$ A positive naf-literal a is *true* with respect to I if a is a classical or built-in atom that is *true* with respect to I; otherwise, a is *false* with respect to I. A negative naf-literal **not** a is *true* (or *false*) with respect to I if a is *false* (or *true*) with respect to I.

Satisfaction of Aggregate Literals. An *aggregate function* is a mapping from sets of tuples of terms to terms, $+\infty$ or $-\infty$. The aggregate functions associated with aggregate

[3] Note that the outcomes of arithmetic evaluation are well defined relative to well-formed substitutions.

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

function names introduced in Section 2 map a set T of tuples of terms to a term, $+\infty$ or $-\infty$ as follows:[4]

- $\#\text{count}(T) = \begin{cases} |T| & \text{if } T \text{ is finite} \\ +\infty & \text{if } T \text{ is infinite;} \end{cases}$
- $\#\text{sum}(T) = \begin{cases} \sum_{(t_1,\ldots,t_m) \in T,\ t_1 \text{ is an integer}} t_1 & \text{if } \{(t_1,\ldots,t_m) \in T \mid t_1 \text{ is a nonzero integer}\} \text{ is finite} \\ 0 & \text{if } \{(t_1,\ldots,t_m) \in T \mid t_1 \text{ is a nonzero integer}\} \text{ is infinite;} \end{cases}$
- $\#\text{max}(T) = \begin{cases} \max\{t_1 \mid (t_1,\ldots,t_m) \in T\} & \text{if } T \neq \emptyset \text{ is finite} \\ +\infty & \text{if } T \text{ is infinite} \\ -\infty & \text{if } T = \emptyset; \end{cases}$
- $\#\text{min}(T) = \begin{cases} \min\{t_1 \mid (t_1,\ldots,t_m) \in T\} & \text{if } T \neq \emptyset \text{ is finite} \\ -\infty & \text{if } T \text{ is infinite} \\ +\infty & \text{if } T = \emptyset. \end{cases}$

The terms selected by $\#\text{max}(T)$ and $\#\text{min}(T)$ for finite sets $T \neq \emptyset$ are determined relative to the total order $\preceq$ on terms in U_P. In the special cases that $\#\text{aggr}(T) = +\infty$ or $\#\text{aggr}(T) = -\infty$, we adopt the convention that $-\infty \preceq u$ and $u \preceq +\infty$ for every term $u \in U_P$. An expression $\#\text{aggr}(T) \prec u$ is *true* (or *false*) for $\#\text{aggr} \in \{\text{“\#count”}, \text{“\#sum”}, \text{“\#max”}, \text{“\#min”}\}$, an aggregate relation $\prec \in \{\text{“}<\text{”}, \text{“}\leq\text{”}, \text{“}=\text{”}, \text{“}\neq\text{”}, \text{“}>\text{”}, \text{“}\geq\text{”}\}$, and a term u if $\#\text{aggr}(T) \prec u$ is *true* (or *false*) according to the corresponding definition for built-in atoms, given previously, extended to the values $+\infty$ and $-\infty$ for $\#\text{aggr}(T)$.

An interpretation $I \subseteq B_P$ maps a collection E of aggregate elements to the following set of tuples of terms:

$$\text{eval}(E, I) = \{(t_1,\ldots,t_m) \mid t_1,\ldots,t_m : l_1,\ldots,l_n \text{ occurs in } E \text{ and } l_1,\ldots,l_n \text{ are } \textit{true} \text{ with respect to } I\}$$

A positive aggregate literal $a = \#\text{aggr}\ E \prec u$ is *true* (or *false*) with respect to I if $\#\text{aggr}(\text{eval}(E, I)) \prec u$ is *true* (or *false*) with respect to I; **not** a is *true* (or *false*) with respect to I if a is *false* (or *true*) with respect to I.

Answer Sets. Given a program P and a (consistent) interpretation $I \subseteq B_P$, a rule $h_1 \mid \ldots \mid h_m \leftarrow b_1,\ldots,b_n$ in $grnd(P)$ is *satisfied* with respect to I if some $h \in \{h_1,\ldots,h_m\}$ is *true* with respect to I when $b_1,\ldots,b_n$ is *true* with respect to I; I is a *model* of P if every rule in $grnd(P)$ is satisfied with respect to I. The *reduct* of P with respect to I, denoted by P^I, consists of the rules $h_1 \mid \ldots \mid h_m \leftarrow b_1,\ldots,b_n$ in $grnd(P)$ such that $b_1,\ldots,b_n$ are *true* with respect to I; I is an *answer set* of P if I is a $\subseteq$-minimal model of P^I. In other words, an answer set I of P is a model of P such that no proper subset of I is a model of P^I.

The semantics of P is given by the collection of its answer sets, denoted by $AS(P)$.

[4] The special cases in which $\#\text{aggr}(T) = +\infty$, $\#\text{aggr}(T) = -\infty$ or $\#\text{sum}(T) = 0$ for an infinite set $\{(t_1,\ldots,t_m) \in T \mid t_1 \text{ is a nonzero integer}\}$ are adopted from Abstract Gringo (Gebser *et al.* 2015).

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

Optimal Answer Sets. To select optimal members of $AS(P)$, we map an interpretation I for P to a set of tuples as follows:

$$\begin{aligned}\text{weak}(P,I) = \{(w@l, t_1, \ldots, t_m) \mid \\ :\sim b_1, \ldots, b_n.\,[w@l, t_1, \ldots, t_m] \text{ occurs in } grnd(P) \text{ and } b_1, \ldots, b_n \text{ are } \textit{true} \\ \text{with respect to } I\}\end{aligned}$$

For any integer l, let

$$P_l^I = \begin{cases} \sum_{(w@l, t_1, \ldots, t_m) \in \text{weak}(P,I),\ w \text{ is an integer}} w & \text{if } \{(w@l, t_1, \ldots, t_m) \in \text{weak}(P,I) \mid w \\ & \text{is a nonzero integer}\} \text{ is finite} \\ 0 & \text{if } \{(w@l, t_1, \ldots, t_m) \in \text{weak}(P,I) \mid w \\ & \text{is a nonzero integer}\} \text{ is infinite} \end{cases}$$

denote the sum of integers w over tuples with $w@l$ in $\text{weak}(P,I)$. Then, an answer set $I \in AS(P)$ is *dominated* by $I' \in AS(P)$ if there is some integer l such that $P_l^{I'} < P_l^I$ and $P_{l'}^{I'} = P_{l'}^I$ for all integers $l' > l$. An answer set $I \in AS(P)$ is *optimal* if there is no $I' \in AS(P)$ such that I is dominated by I'. Note that P has some (and possibly more than one) optimal answer sets if $AS(P) \neq \emptyset$.

Queries. Given a ground query $Q = q?$ of a program P, Q is true if $q \in I$ for all $I \in AS(P)$. Otherwise, Q is false. Note that, if $AS(P) = \emptyset$, all queries are true. In the presence of variables, one is interested in substitutions that make the query true. Given the nonground query $Q = q(t_1, \ldots, t_n)?$ of a program P, let $Ans(Q,P)$ be the set of all substitutions σ for Q such that $Q\sigma$ is true. The set $Ans(Q,P)$ constitutes the set of answers to Q. Note that, if $AS(P) = \emptyset$, $Ans(Q,P)$ contains all possible substitutions for Q.

Note that query answering, according to the definitions above, corresponds to cautious (skeptical) reasoning as defined in, for example, Abiteboul *et al.* (1995).

4 Syntactic shortcuts

This section specifies additional constructs by reduction to the language introduced in Section 2.

Anonymous Variables. An *anonymous variable* in a rule, weak constraint or query is denoted by "_" (character underscore). Each occurrence of "_" stands for a fresh variable in the respective context (i.e., different occurrences of anonymous variables represent distinct variables).

Choice Rules. A *choice element* has form

$$a : l_1, \ldots, l_k$$

where a is a classical atom and $l_1, \ldots, l_k$ are the naf-literals for $k \geq 0$.

A *choice atom* has form

$$C \prec u$$

where C is a collection of choice elements, which are syntactically separated by ";", $\prec$ is an aggregate relation (see Section 2) and u is a term. The part "$\prec u$" can optionally be omitted if $\prec$ is "$\geq$" and $u = 0$.

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

A *choice rule* has form

$$C \prec u \leftarrow b_1, \ldots, b_n.$$

where $C \prec u$ is a choice atom and $b_1, \ldots, b_n$ are the literals for $n \geq 0$.

Intuitively, a choice rule means that, if the body of the rule is *true*, an arbitrary subset of the classical atoms a such that $l_1, \ldots, l_k$ are *true* can be chosen as *true* in order to comply with the aggregate relation $\prec$ between C and u. In the following, this intuition is captured by means of a proper mapping of choice rules to rules without choice atoms (in the head).

For any predicate atom $q = p(t_1, \ldots, t_n)$, let $\widehat{q} = \hat{p}(1, t_1, \ldots, t_n)$ and $\widehat{\neg q} = \hat{p}(0, t_1, \ldots, t_n)$, where $\hat{p} \neq p$ is an (arbitrary) predicate and function name that is uniquely associated with p, and the first argument (which can be 1 or 0) indicates the "polarity" q or $\neg q$, respectively.[5]

Then, a choice rule stands for the rules

$$a \mid \widehat{a} \leftarrow b_1, \ldots, b_n, l_1, \ldots, l_k.$$

for each choice element $a : l_1, \ldots, l_k$ in C along with the constraint

$$\leftarrow b_1, \ldots, b_n, \mathbf{not}\ \#\mathrm{count}\{\widehat{a} : a, l_1, \ldots, l_k \mid (a : l_1, \ldots, l_k) \in C\} \prec u.$$

The first group of rules expresses that the classical atom a in a choice element $a : l_1, \ldots, l_k$ can be chosen as *true* (or *false*) if $b_1, \ldots, b_n$ and $l_1, \ldots, l_k$ are *true*. This "generates" all subsets of the atoms in choice elements. On the other hand, the second rule, which is an integrity constraint, requires the condition $C \prec u$ to hold if $b_1, \ldots, b_n$ are *true*.[6]

For illustration, consider the choice rule

$$\{p(a) : q(2); \neg p(a) : q(3)\} \leq 1 \leftarrow q(1).$$

Using the fresh predicate and function name $\hat{p}$, the choice rule is mapped to three rules as follows:

$$\begin{aligned} p(a) \mid \hat{p}(1, a) &\leftarrow q(1), q(2). \\ \neg p(a) \mid \hat{p}(0, a) &\leftarrow q(1), q(3). \\ &\leftarrow q(1), \mathbf{not}\ \#\mathrm{count}\{\hat{p}(1, a) : p(a), q(2); \hat{p}(0, a) : \neg p(a), q(3)\} \leq 1. \end{aligned}$$

Note that the three rules are satisfied with respect to an interpretation I such that $\{q(1), q(2), q(3), \hat{p}(1, a), \hat{p}(0, a)\} \subseteq I$, and $\{p(a), \neg p(a)\} \cap I = \emptyset$. In fact, when $q(1)$, $q(2)$, and $q(3)$ are *true*, the choice of none or one of the atoms $p(a)$ and $\neg p(a)$ complies with the aggregate relation "$\leq$" to 1.

Aggregate Relations. An aggregate or choice atom

$$\#\mathrm{aggr}\ E \prec u \quad \text{or} \quad C \prec u$$

[5] It is assumed that fresh predicate and function names are outside of possible program signatures and cannot be referred to within user input.

[6] In disjunctive heads of rules of the first form, an occurrence of $\widehat{a}$ denotes an (auxiliary) *atom* that is linked to the original atom a. Given the relationship between a and $\widehat{a}$, the latter is reused as a *term* in the body of a rule of the second form. That is, we overload the notation $\widehat{a}$ by letting it stand both for an atom (in disjunctive heads) and a term (in #count aggregates).

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

may be written as

$$u \prec^{-1} \#\text{aggr } E \quad \text{or} \quad u \prec^{-1} C$$

where "$<$"$^{-1}$ = "$>$"; "$\leq$"$^{-1}$ = "$\geq$"; "$=$"$^{-1}$ = "$=$"; "$\neq$"$^{-1}$ = "$\neq$"; "$>$"$^{-1}$ = "$<$"; "$\geq$"$^{-1}$ = "$\leq$".

The left and right notations of aggregate relations may be combined in expressions as follows:

$$u_1 \prec_1 \#\text{aggr } E \prec_2 u_2 \quad \text{or} \quad u_1 \prec_1 C \prec_2 u_2$$

Such expressions are mapped to available constructs according to the following transformations:

◇ $u_1 \prec_1 C \prec_2 u_2 \leftarrow b_1, \ldots, b_n$ stands for

$$u_1 \prec_1 C \leftarrow b_1, \ldots, b_n.$$
$$C \prec_2 u_2 \leftarrow b_1, \ldots, b_n.$$

◇ $h_1 \mid \ldots \mid h_k \leftarrow b_1, \ldots, b_{i-1}, u_1 \prec_1 \#\text{aggr } E \prec_2 u_2, b_{i+1}, \ldots, b_n$ stands for

$$h_1 \mid \ldots \mid h_k \leftarrow b_1, \ldots, b_{i-1}, u_1 \prec_1 \#\text{aggr } E, \#\text{aggr } E \prec_2 u_2, b_{i+1}, \ldots, b_n.$$

◇ $h_1 \mid \ldots \mid h_k \leftarrow b_1, \ldots, b_{i-1}, \textbf{not } u_1 \prec_1 \#\text{aggr } E \prec_2 u_2, b_{i+1}, \ldots, b_n$ stands for

$$h_1 \mid \ldots \mid h_k \leftarrow b_1, \ldots, b_{i-1}, \textbf{not } u_1 \prec_1 \#\text{aggr } E, b_{i+1}, \ldots, b_n.$$
$$h_1 \mid \ldots \mid h_k \leftarrow b_1, \ldots, b_{i-1}, \textbf{not } \#\text{aggr } E \prec_2 u_2, b_{i+1}, \ldots, b_n.$$

◇ $:\sim b_1, \ldots, b_{i-1}, u_1 \prec_1 \#\text{aggr } E \prec_2 u_2, b_{i+1}, \ldots, b_n. [w@l, t_1, \ldots, t_k]$ stands for

$$:\sim b_1, \ldots, b_{i-1}, u_1 \prec_1 \#\text{aggr } E, \#\text{aggr } E \prec_2 u_2, b_{i+1}, \ldots, b_n. [w@l, t_1, \ldots, t_k]$$

◇ $:\sim b_1, \ldots, b_{i-1}, \textbf{not } u_1 \prec_1 \#\text{aggr } E \prec_2 u_2, b_{i+1}, \ldots, b_n. [w@l, t_1, \ldots, t_k]$ stands for

$$:\sim b_1, \ldots, b_{i-1}, \textbf{not } u_1 \prec_1 \#\text{aggr } E, b_{i+1}, \ldots, b_n. [w@l, t_1, \ldots, t_k]$$
$$:\sim b_1, \ldots, b_{i-1}, \textbf{not } \#\text{aggr } E \prec_2 u_2, b_{i+1}, \ldots, b_n. [w@l, t_1, \ldots, t_k]$$

5 Using ASP-Core-2 in practice – restrictions

To promote declarative programming as well as practical system implementation, ASP-Core-2 programs are supposed to comply with the restrictions listed in this section. This particularly applies to input programs starting from the System Track of the 4th ASP Competition (Calimeri *et al.* 2014).

Safety. Any rule, weak constraint or query is required to be safe; to this end, for a set V of variables and literals $b_1, \ldots, b_n$, we inductively (starting from an empty set of bound variables) define $v \in V$ as *bound* by $b_1, \ldots, b_n$ if v occurs outside of arithmetic terms in some b_i for $1 \leq i \leq n$ such that b_i is

- (i) a classical atom,
- (ii) a built-in atom $t = u$ or $u = t$, and any member of V occurring in t is bound by $\{b_1, \ldots, b_n\} \setminus b_i$ or
- (iii) an aggregate atom $\#\text{aggr } E = u$, and any member of V occurring in E is bound by $\{b_1, \ldots, b_n\} \setminus b_i$.

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

The entire set V of variables is *bound* by $b_1, \ldots, b_n$ if each $v \in V$ is bound by $b_1, \ldots, b_n$.

A rule, weak constraint or query r is *safe* if the set V of global variables in r is bound by $b_1, \ldots, b_n$ (taking a query r to be of form $b_1?$), and for each aggregate element $t_1, \ldots, t_k : l_1, \ldots, l_m$ in r with occurring variable set W, the set $W \setminus V$ of local variables is bound by $l_1, \ldots, l_m$. For instance, the rule

$$p(X, Y) \leftarrow q(X), \#\text{sum}\{S, X : r(T, X), S = (2 * T) - X\} = Y.$$

is safe because all variables are bound by $q(X), r(T, X)$, while

$$p(X, Y) \leftarrow q(X), \#\text{sum}\{S, X : r(T, X), S + X = 2 * T\} = Y$$

is not safe because the expression $S + X = 2 * T$ does not respect condition (ii) above.

Finiteness. Pragmatically, ASP programs solving real problems have a finite number of answer sets of finite size. As an example, a program including $p(X + 1) \leftarrow p(X)$ or $p(f(X)) \leftarrow p(X)$ along with a fact like $p(0)$ is not an admissible input in ASP Competitions. There are pragmatic conditions that can be checked to ensure that a program admits finitely many answer sets (e.g., Calimeri *et al.* 2011a); in alternative, finiteness can be witnessed by providing a known maximum integer and maximum function nesting level per problem instance, which correctly limits the absolute values of integers as well as the depths of functional terms occurring as arguments in the atoms of answer sets. The last option is the one adopted in ASP Competitions since 2011.

Aggregates. For the sake of an uncontroversial semantics, we require aggregates to be nonrecursive. To make this precise, for any predicate atom $q = p(t_1, \ldots, t_n)$, let $q^v = p/n$ and $\neg q^v = \neg p/n$. We further define the directed *predicate dependency graph* $D_P = (V, E)$ for a program P by

- the set V of vertices a^v for all classical atoms a appearing in P and
- the set E of edges $(h_i^v, h_1^v), \ldots, (h_i^v, h_m^v)$ and $(h_1^v, a^v), \ldots, (h_m^v, a^v)$ for all rules $h_1 \mid \ldots \mid h_m \leftarrow b_1, \ldots, b_n.$ in P, $1 \leq i \leq m$ and classical atoms a appearing in $b_1, \ldots, b_n$.

The aggregates in P are *nonrecursive* if, for any classical atom a appearing within aggregate elements in a rule $h_1 \mid \ldots \mid h_m \leftarrow b_1, \ldots, b_n$ in P, there is no path from a^v to h_i^v in D_P for $1 \leq i \leq m$.

Predicate Arities. The arity of atoms sharing some predicate name is not assumed to be fixed. However, system implementers are encouraged to issue proper warning messages if an input program includes classical atoms with the same predicate name but different arities.

Undefined Arithmetics. The semantics of ASP-Core-2 requires that substitutions that lead to undefined arithmetic subterms (and are thus not well formed) are excluded by ground instantiation as specified in Section 3. In practice, this condition is not easy to meet and implement for a number of technical reasons; thus, it might cause problems to existing implementations, or even give rise to unexpected behaviors.

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

In order to avoid such complications, we require that a program P shall be invariant under undefined arithmetics; that is, $grnd(P)$ is supposed to be equivalent to any ground program P' obtainable from P by freely replacing arithmetic subterms with undefined outcomes by arbitrary terms from U_P. Intuitively, rules have to be written in such a way that the semantics of a program does not change, no matter the handling of substitutions that are not well formed.

For instance, the program

$$a(0).$$
$$p \leftarrow a(X), \textbf{not } q(X/X).$$

has the (single) answer set $\{a(0)\}$. This program, however, is not invariant under undefined arithmetics. Indeed, a vanilla grounder that skips arithmetic evaluation (in view of no rule with atoms of predicate q in the head) might produce the (simplified) ground rule $p \leftarrow a(0)$, and this would result in the wrong answer set $\{a(0), p\}$.

In contrast to the previous program,

$$a(0).$$
$$p \leftarrow a(X), \textbf{not } q(X/X), X \neq 0$$

is invariant under undefined arithmetics, since substitutions that are not well formed cannot yield applicable ground rules. Hence, a vanilla grounder as considered above may skip the arithmetic evaluation of ground terms obtained from X/X without risking wrong answer sets.

6 EBNF grammar and lexical table

```
<program>            ::= [<statements>] [<query>]

<statements>         ::= [<statements>] <statement>
<query>              ::= <classical_literal> QUERY_MARK

<statement>          ::= CONS [<body>] DOT
                       | <head> [CONS [<body>]] DOT
                       | WCONS [<body>] DOT
                           SQUARE_OPEN <weight_at_level> SQUARE_CLOSE

<head>               ::= <disjunction> | <choice>
<body>               ::= [<body> COMMA]
                           (<naf_literal> | [NAF] <aggregate>)

<disjunction>        ::= [<disjunction> OR] <classical_literal>

<choice>             ::= [<term> <binop>]
                           CURLY_OPEN [<choice_elements>]
```

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

```
                          CURLY_CLOSE [<binop> <term>]
<choice_elements>    ::= [<choice_elements> SEMICOLON]
                          <choice_element>
<choice_element>     ::= <classical_literal> [COLON [<naf_literals>]]
<aggregate>          ::= [<term> <binop>] <aggregate function>
                          CURLY_OPEN  [<aggregate_elements>]
                          CURLY_CLOSE [<binop> <term>]
<aggregate_elements> ::= [<aggregate_elements> SEMICOLON]
                          <aggregate_element>
<aggregate_element>  ::= [<basic_terms>] [COLON [<naf_literals>]]
<aggregate_function> ::= AGGREGATE_COUNT
                       | AGGREGATE_MAX
                       | AGGREGATE_MIN
                       | AGGREGATE_SUM

<weight_at_level>    ::= <term> [AT <term>] [COMMA <terms>]

<naf_literals>       ::= [<naf_literals> COMMA] <naf_literal>
<naf_literal>        ::= [NAF] <classical_literal> | <builtin_atom>

<classical_literal>  ::= [MINUS] ID [PAREN_OPEN [<terms>] PAREN_CLOSE]
<builtin_atom>       ::= <term> <binop> <term>

<binop>              ::= EQUAL
                       | UNEQUAL
                       | LESS
                       | GREATER
                       | LESS_OR_EQ
                       | GREATER_OR_EQ

<terms>              ::= [<terms> COMMA] <term>
<term>               ::= ID [PAREN_OPEN [<terms>] PAREN_CLOSE]
                       | NUMBER
                       | STRING
                       | VARIABLE
                       | ANONYMOUS_VARIABLE
                       | PAREN_OPEN <term> PAREN_CLOSE
                       | MINUS <term>
                       | <term> <arithop> term>

<basic_terms>        ::= [<basic_terms> COMMA] <basic_term>
<basic_term>         ::= <ground_term> |
                         <variable_term>
<ground_term>        ::= SYMBOLIC_CONSTANT |
```

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

```
                          STRING | [MINUS] NUMBER
<variable_term>      ::= VARIABLE |
                          ANONYMOUS_VARIABLE
<arithop>            ::= PLUS
                       | MINUS
                       | TIMES
                       | DIV
```

Token name	Mathematical notation used within this document (exemplified)	Lexical format (Flex notation)
ID	$a, b, anna, \ldots$	`[a-z][A-Za-z0-9_]*`
VARIABLE	$X, Y, Name, \ldots$	`[A-Z][A-Za-z0-9_]*`
STRING	"http://bit.ly/cw6lDS", "Peter", ...	`\"([^\"]|\\\")*\"`
NUMBER	$1, 0, 100000, \ldots$	`"0"|[1-9][0-9]*`
ANONYMOUS_VARIABLE	_	`"_"`
DOT	.	`"."`
COMMA	,	`","`
QUERY_MARK	?	`"?"`
COLON	:	`":"`
SEMICOLON	;	`";"`
OR	|	`"|"`
NAF	**not**	`"not"`
CONS	$\leftarrow$	`":-"`
WCONS	$:\sim$	`":~"`
PLUS	$+$	`"+"`
MINUS	$-$ or $\neg$	`"-"`
TIMES	$*$	`"*"`
DIV	/	`"/"`
AT	@	`"@"`
PAREN_OPEN	(	`"("`
PAREN_CLOSE	)	`")"`
SQUARE_OPEN	[	`"["`
SQUARE_CLOSE	]	`"]"`
CURLY_OPEN	{	`"{"`
CURLY_CLOSE	}	`"}"`
EQUAL	$=$	`"="`
UNEQUAL	$\neq$	`"<>"|"!="`
LESS	$<$	`"<"`
GREATER	$>$	`">"`
LESS_OR_EQ	$\leq$	`"<="`
GREATER_OR_EQ	$\geq$	`">="`
AGGREGATE_COUNT	#count	`"#count"`
AGGREGATE_MAX	#max	`"#max"`
AGGREGATE_MIN	#min	`"#min"`
AGGREGATE_SUM	#sum	`"#sum"`
COMMENT	% this is a comment	`"%"([^*\n][^\n]*)?\n`
MULTI_LINE_COMMENT	%* this is a comment *%	`"%*"([^*]|\*[^%])*"*%"`
BLANK		`[ \t\n]+`

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

Lexical values are given in Flex[7] syntax. The `COMMENT`, `MULTI_LINE_COMMENT`, and `BLANK` tokens can be freely interspersed amidst other tokens and have no syntactic or semantic meaning.

7 Conclusions

In this document, we have presented the ASP-Core-2 standard language that defines syntax and semantics of a standard language to which ASP solvers have to adhere in order to enter the ASP Competitions series, since 2013. The standardization committee is still working on the evolution of the language in order to keep it aligned with the achievements of the ASP research community. Among the features that are currently under consideration, we mention here a semantics for recursive aggregates, for which several proposals are at the moment in place, for example, Pelov (2004), Faber *et al.* (2011), Alviano *et al.* (2011), Gelfond and Zhang (2014), Alviano *et al.* (2015), and a standard for intermediate (Gebser *et al.* 2016) and output (Brain *et al.* 2007; Krennwallner 2013) formats for ASP solvers.

References

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

Alviano, M., Calimeri, F., Faber, W., Leone, N. and Perri, S. 2011. Unfounded sets and well-founded semantics of answer set programs with aggregates. *Journal of Artificial Intelligence Research 42*, 487–527.

Alviano, M., Faber, W. and Gebser, M. 2015. Rewriting recursive aggregates in answer set programming: Back to monotonicity. *Theory and Practice of Logic Programming 15,* 4–5, 559–573.

Brain, M., Faber, W., Maratea, M., Polleres, A., Schaub, T. and Schindlauer, R. 2007. What should an asp solver output? A multiple position paper. In *Proceedings of the First International SEA'07 Workshop*. CEUR Workshop Proceedings, vol. 281.

Brewka, G., Eiter, T. and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

Calimeri, F., Cozza, S., Ianni, G. and Leone, N. 2011a. Finitely recursive programs: Decidability and bottom-up computation. *AI Communications 24,* 4, 311–334.

Calimeri, F., Gebser, M., Maratea, M. and Ricca, F. 2016. Design and results of the Fifth Answer Set Programming Competition. *Artificial Intelligence 231*, 151–181.

Calimeri, F., Ianni, G. and Ricca, F. 2014. The third open answer set programming competition. *TPLP 14,* 1, 117–135.

Calimeri, F., Ianni, G., Ricca, F. and della Calabria Organizing Committee, T. U. 2011b. Third ASP Competition, File and language formats. URL: `http://www.mat.unical.it/aspcomp2011/files/LanguageSpecifications.pdf`.

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*, Messina, Italy, M. de Vos and A. Provetti, Eds, 274–288. URL: `http://CEUR-WS.org/Vol-78/`.

[7] `http://flex.sourceforge.net/`.

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

Denecker, M., Pelov, N. and Bruynooghe, M. 2001. Ultimate well-founded and stable model semantics for logic programs with aggregates. In *Proceedings of the 17th International Conference on Logic Programming*, P. Codognet, Ed. Springer Verlag, 212–226.

Erdem, E., Gelfond, M. and Leone, N. 2016. Applications of answer set programming. *AI Magazine 37,* 3, 53–68.

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, 200–212.

Faber, W., Leone, N. and Pfeifer, G. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175,* 1, 278–298. Special Issue: John McCarthy's Legacy.

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

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V. and Schaub, T. 2015. Abstract Gringo. *Theory and Practice of Logic Programming 15,* 4–5, 449–463.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. and Wanko, P. 2016. Theory solving made easy with clingo 5. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016 TCs)*, M. Carro, A. King, N. Saeedloei and M. D. Vos, Eds. OASICS, vol. 52. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2:1–2:15.

Gebser, M., Leone, N., Maratea, M., Perri, S., Ricca, F. and Schaub, T. 2018a. Evaluation techniques and systems for answer set programming: A survey. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI 2018)*, J. Lang, Ed. ijcai.org, 5450–5456.

Gebser, M., Maratea, M. and Ricca, F. 2017a. The design of the seventh answer set programming competition. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017)*, M. Balduccini and T. Janhunen, Eds. Lecture Notes in Computer Science, vol. 10377. Springer, 3–9.

Gebser, M., Maratea, M. and Ricca, F. 2017b. The sixth answer set programming competition. *Journal of Artificial Intelligence Research 60*, 41–95.

Gebser, M., Obermeier, P., Schaub, T., Ratsch-Heitmann, M. and Runge, M. 2018b. Routing driverless transport vehicles in car assembly with answer set programming. *TPLP 18,* 3–4, 520–534.

Gelfond, M. 2002. Representing knowledge in A-Prolog. In *Computational Logic. Logic Programming and Beyond*, A. C. Kakas and F. Sadri, Eds. LNCS, vol. 2408. Springer, 413–451.

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

Gelfond, M. and Zhang, Y. 2014. Vicious circle principle and logic programs with aggregates. *Theory and Practice of Logic Programming 14,* 4–5, 587–601.

Harrison, A. and Lifschitz, V. 2018. Relating two dialects of answer set programming. In *Proceedings of the 17th International Workshop on Non-monotonic Reasoning (NMR 2018)*, E. Fermé and S. Villata, Eds, 99–108.

Kemp, D. B. and Stuckey, P. J. 1991. Semantics of logic programs with aggregates. In *Proceedings of the International Symposium on Logic Programming (ISLP'91)*, V. A. Saraswat and K. Ueda, Eds. MIT Press, 387–401.

Krennwallner, T. 2013. ASP Competition, output format. URL: https://www.mat.unical.it/aspcomp2013/files/aspoutput.txt.

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press

Leone, N. and Ricca, F. 2015. Answer set programming: A tour from the basics to advanced development tools and industrial applications. In *Web Logic Rules - 11th International Summer School on Reasoning Web, Tutorial Lectures*, W. Faber and A. Paschke, Eds. Lecture Notes in Computer Science, vol. 9203. Springer, 308–326.

Lierler, Y., Maratea, M. and Ricca, F. 2016. Systems, engineering environments, and competitions. *AI Magazine 37,* 3, 45–52.

Osorio, M. and Jayaraman, B. 1999. Aggregation and negation-as-failure. *New Generation Computing 17,* 3, 255–284.

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

Pelov, N., Denecker, M. and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming 7,* 3, 301–353.

Pelov, N. and Truszczyński, M. 2004. Semantics of disjunctive programs with monotone aggregates - An operator-based approach. In *Proceedings of the 10th International Workshop on Non-monotonic Reasoning (NMR 2004), Whistler, BC, Canada*, 327–334.

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

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

Van Gelder, A. 1992. The well-founded semantics of aggregation. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS'92)*. ACM Press, 127–138.

https://doi.org/10.1017/S1471068419000450 Published online by Cambridge University Press