Compilation of Aggregates in ASP

Giuseppe Mazzotta
University of Calabria
Rende,Italy
giuseppe.mazzotta@unical.it

Answer Set Programming (ASP) is a well-known problem-solving formalism in computational logic. Nowadays, ASP is used in many real world scenarios thanks to ASP solvers. Standard evaluation of ASP programs suffers from an intrinsic limitation, knows as Grounding Bottleneck, due to the grounding of some rules that could fit all the available memory. As a result, there exist instances of real world problems that are untractable using the standard Ground & Solve approach. In order to tackle this problem, different strategies have been proposed. Among them we focus on a recent approach based on compilation of problematic constraints as *propagators*, which revealed to be very promising but is currently limited to constraints without aggregates. Since aggregates are widely used in ASP, in this paper we extend such an approach also to constraints containing aggregates. Good results, that proof the effectiveness of the proposed approach, have been reached.

1 Introduction

Answer Set Programming (ASP) [6] is a well-known problem-solving formalism in computational logic that is based on the stable model semantics [18]. ASP systems, such as CLINGO [16] and DLV [1], made possible the development of many real-world applications. As a matter of fact, in the recent years, ASP has been widely used for solving problems of game theory [3], natural language processing [23], natural language understanding [11], robotics [15], scheduling [13], and more [14].

A key role in the development of applications is played by system performances. Since ASP system are involved in many real world applications, efficient ASP systems are needed and then the improvement of those systems is an interesting research topic in computational logic.

Traditional ASP systems are based on the ground & solve approach [19]. Stable model computation starts with, a *grounder* module that transforms the input program (containing variables) in its propositional counterpart by substituting variables with possible constants appearing in the program. Afterward, a *solver* module will compute the stable models for the grounded program implementing an extension of the Conflict Driven Clause Learning (CDCL) algorithm [19].

As observed in different contexts [8], the ground & solve approach suffers from an intrinsic limitation: the combinatorial blow-up of the grounding due to some rules, known as *grounding bottleneck*. In particular, the grounding of these rules could consume all the available memory and then the solver module cannot compute stable models. As a result, there are many problems that are not tractable with the standard approach due to grounding bottleneck.

In order to tackle this problem different approaches have been proposed. One of these strategies is based on the lazy grounding of the rules during model computation. In this approach, the grounder and the solver are continuously working together in an iterative way. The idea behind lazy grounding is essentially to ground a rule only when its body is satisfied, then the instantiated rules will be added to the solver and then the model computation process restarts. Lazy grounding turns out to be very efficient in solving grounding bottleneck problem but obtain bad search performance. In order to improve

© G. Mazzotta
This work is licensed under the
Creative Commons Attribution License.

the performance of lazy-grounding approach, the concept of laziness has been relaxed and different strategies have been proposed [25]. Recently, a different solution was proposed in order to solve this issue. The propsed approach is based on the *compilation* of problematic constraints as propagators [8, 9]. Basically, Cuteri et al. [9] proposed to translate (or compile) some non-ground constraints into a dedicated C++ procedure in order to skip the grounding of these constraints and simulate them in the solver with this ad-hoc procedure. This approach reveals to be very promising but, unfortunately, is limited to simple constraints that does not contain aggregates. Since aggregates are widely used in ASP and their grounding could be very large, we decided to extend the benefits of the compilation based approach also to constraints that contain aggregates. At early stage of our research, we focused on the compilation of constraints containing count aggregates only and preliminary results were encouraging. Performances of the solver WASP [2] were improved on tested benchmarks. Then, we decided to push forward this idea and try to extend the compilation also to constraints with sum aggregates and also to simple rules.

2 Background and Existing Solutions

2.1 Answer Set Programming

An ASP program π is a set of rules of the form:

$$h_1 \mid \ldots \mid h_n : -b_1, \ldots, b_m.$$

where $n+m > 0$, $h_1|\ldots|h_n$ is a disjunction of atoms referred to as *head*, and $b_1,\ldots,b_m$ is a conjunction of literals referred to as *body*. If $n = 0$, then the rule is called *constraint*, whereas if $m = 0$ the rule is called *fact*.

An atom a is an expression of the form $p(t_1,\ldots,t_k)$ where p is a predicate of arity k and $t_1,\ldots,t_k$ are *terms*. A term is an alphanumeric string that could be either a *variable* or a *constant*. According to Prolog notation, if a term starts with a capital letter is a *variable* otherwise is *constant*. If $\forall i \in \{1,\ldots,k\}$, t_i is a constant then the atom a is said *ground*. A *literal* is an atom a or its negation $\sim a$ where $\sim$ denotes the *negation as failure*. Given a literal l it is said *positive* if $l = a$, *negative* if $l = \sim a$. Given a positive literal $l = a$, we define the *complement*, $\bar{l} = \sim a$, instead, for a negative literal $l = \sim a$, $\bar{l} = \overline{\sim a} = a$. ASP supports also *aggregate atoms*. An aggregate atom is of the form $f(S) \succ T$, where $f(S)$ is an aggregate function, $\succ \in \{=,<,\leq,>,\geq\}$ is a predefined comparison operator, and T is a term referred to as guard. An aggregate function is of the form $f(S)$, where S is a set term and $f \in \{\#count, \#sum\}$ is an aggregate function symbol. A set term S is a pair that is either a symbolic set or a ground set. A symbolic set is a pair $\{Terms : Conj\}$, where *Terms* is a list of variables and *Conj* is a conjunction of *standard* atoms, that is, *Conj* does not contain aggregate atoms. A ground set, instead, is a set of pairs of the form $(\bar{t} : conj)$, where $\bar{t}$ is a list of constants and *conj* is a conjunction of ground atoms. Given a program π we define the positive dependency graph $G_\pi =< V,E >$ as a directed graph where $V = \{p : p$ is a predicate term appearing in $\pi\}$ and $E = \{(u,v) : u \in V, v \in V$ and $\exists r \in \pi$ s.t. $\exists$ a positive literal l and an atom a s.t. l is of the form $u(t_1,...,t_k)$ and l appears in the body of r, and a is of the form $v(t_1,...,t_n)$ and a appears in the head of $r\}$. π is said to be recursive if G_π is a cyclic graph. Given a program π, we define U_π, the *Herbrand Universe*, as the set of all constants appearing in π and B_π, the *Herbrand Base*, as the set of all possible ground atoms that can be built using predicate in π and constants in U_π. $\mathscr{B}$ denotes $B_\pi \cup \overline{B_\pi}$. Given a rule r and the Herbrand Universe U_π, we define $ground(r)$ as the set of all possible instantiations of r that can be built by assigning variables in r to constant in U_π. Given a program π, instead, $ground(\pi) = \bigcup_{r\in\pi} ground(r)$. An interpretation I is a set of literals. In particular, I is total if

$\forall a \in B_\pi (a \in I \vee \sim a \in I) \wedge (a \in I \rightarrow \sim a \notin I)$. A literal l is true w.r.t I if $l \in I$, otherwise it is false. A ground conjunction *conj* of atoms is true w.r.t I if all atoms in *conj* are true, otherwise, if at least one atom is false then *conj* is false w.r.t. I. Let $I(S)$ denote the multiset $[t_1 | (t_1, \dots, t_n) : conj \in S \wedge conj$ is true w.r.t. $I]$. The evaluation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$.

Example 2.1 *Let A be an aggregate atom* $A = \#count\{(1 : p(1,1)), (2 : p(2,1)), (3 : p(3,1))\} > 1$ *and let* $I = \{p(1,1), p(2,1), \sim p(3,1)\}$. $I(S) = [1,2]$, $I(f(S)) = 2$ *since* $f = \#count$ *so the aggregate atom A is true w.r.t. I.*

An interpretation I is a *model* for π if $\forall r \in ground(\pi)(\forall l \in body(r), l \in I) \rightarrow (\exists a \in head(r) : a \in I)$. The *FLP-reduct* of π, denoted by π^I, is the set of rules obtained from π by deleting those rules whose body is false w.r.t I. Let I be a model for π, I is a *stable model* for π if there is no $I' \subset I$ such that I' is a model for π^I. Given a program π, π is *coherent* if it admits at least one stable model otherwise is *incoherent*.

2.2 Existing Literature

In order to solve the grounding bottleneck problem, several attempts have been made [17], including language extensions (such as constraint programming [22, 4], difference logic [16, 24]) and *lazy grounding* techniques [12, 20, 26]. Hybrid formalisms are efficiently evaluated by coupling an ASP system with a solver for the other theory, thus circumventing the grounding bottleneck. Lazy grounding implementations instantiate a rule only when its body is satisfied to prevent the grounding of rules which are unnecessary during the search of an answer set. Albeit lazy grounding techniques obtained good preliminary results, their performance is still not competitive with state-of-the-art systems [17]. Lazy grounding has been also extended to support aggregates [5]. To the best of my knowledge, this normalization strategies is limited to monotone aggregates with a lower bound. This approach turns out to be very promising outperfoming the ground & and solve system CLINGO on benchmarks were the grounding is really hard.

Another existing approach is proposed by Cuteri et al. [8, 9] and is based on the compilation of constraints into a dedicated procedure, called *propagator*, that supports the solver in the model computation process. There exists two different implementations of this approach, namely *lazy* and *eager*. In more details, given an ASP program P and a set of constraint C such that $C \subseteq P$, if the program $P \setminus C$ is unsatisfiable then the original program is also unsatisfiable. Otherwise, if $P \setminus C$ admits at least one model I and constraints in C are satisfied w.r.t. I, then I is a model for the original program. Starting from this description the *lazy* implementation build a procedure that automatically understands if a candidate model I satisfies constraints in C. If some of these constraints are not satisfied then they will be lazily instantiated in the solver and the model computations starts again, otherwise the process stops returning the model I. The *eager* implementation is based on a different approach. Indeed, in this case the propagator procedure is completely involved in the model computation process. Every time that the solver assign a truth value to a literal, either true or false, the propagator is notified and it simulates the instantiation of the constraints in C without storing it in memory, and if it is possible it makes some inferences on the truth values of the literals that have not been assigned yet, in order to prevent constraints failure. Results obtained with this compilation-based approach are very promising but currently the eager approach supports only simple constraints without aggregates.

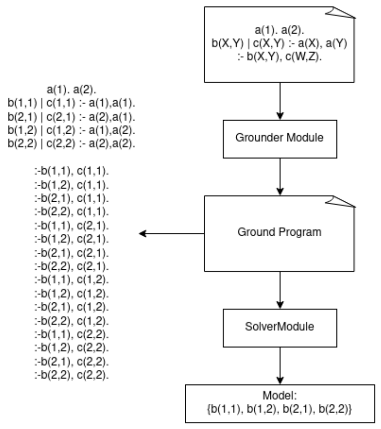


Figure 1: Solving using Ground & Solve approach

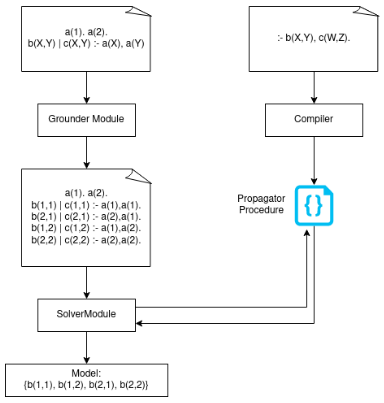

a(1). a(2).
b(X,Y) | c(X,Y) :- a(X), a(Y)
:- b(X,Y), c(W,Z).
Grounder Module
Compiler
Propagator Procedure
a(1). a(2).
b(1,1) | c(1,1) :- a(1),a(1).
b(2,1) | c(2,1) :- a(2),a(1).
b(1,2) | c(1,2) :- a(1),a(2).
b(2,2) | c(2,2) :- a(2),a(2).
SolverModule
Model:
{b(1,1), b(1,2), b(2,1), b(2,2)}

Figure 2: Solving using compilation based approach

3 Research Goal

Approaches based on compilation of constraints revealed to be very promising, outperforming traditional systems in many comparisons. However, a significant number of problems, especially hard combinatorial problems from ASP competitions feature aggregates that are not yet supported by compilation-based approaches. Aggregates are among the standardized knowledge modeling constructs that make ASP effective in representing complex problems. So, the extension of the existing compilation approach is an interesting research topic in order to extend the benefits of the compilation to all those ASP programs that cannot be solved by traditional approached. For this reason, we decided to tackle this problem starting from the compilation of constraints containing aggregates. In particular, a first attempt has been proposed in the paper [21]. In particular, we presented an extension of the eager compilation for constraints containing *#count* aggregates and promising results have been obtained. The next natural step is to extend the compilation approach also to *#sum* aggregate. Moreover, we also consider to extend the compilation approach also to rules, whereas now it is limited to constraints.

Therefore, our research goals are as follows:

- Extend the eager compilation approach also to *#sum* aggregates.
- Support the eager compilation of rules without recursion.
- Support the eager compilation of simple programs containing recursive rules.
- Support, also, the compilation of disjunctive and choice rules in order to provide a complete compilation of ASP program, providing a novel approach to compute stable models of an ASP program.

In principle it can be also combined with the lazy normalization [5]. However this would not be a trivial combination and might be subject of future research once the compilation approach will be extended to a larger class of programs.

4 Current Research Status

Up to now we have extended the compilation to constraints containing *#count* aggregates. Moreover, we currently have a preliminary implementation of a technique that is able to compile constraints containing also *#sum* aggregates, whose performance is still not satisfactory. It turns out that compiling *#sum* aggregates requires several additional optimization techniques and dedicated data structures.

In order to show the working principles of our approach in presence of *#count* aggregates, we present the following example.

Example 4.1 *Let* $:-a(X,Z),c(Z),\#count\{Y : b(X,Y)\} >= 2$ *be a constraint, then there are several possible propagation steps that can be done. Note that a propagation step is done to avoid possible violations of the constraint.*

- ***Aggregate propagation.*** *Let* $I = \{a(1,1),\ c(1)\}$ *be an interpretation and assume that the solver assigns* $b(1,1)$ *to true. Starting from* $b(1,1)$ *it is possible to propagate all the undefined values of the form* $b(1,\_)$ *to false, since if* $b(1,2)$ *becomes true the count returns 2 that is greater than or equal to 2 and thus the constraint is violated.*
- ***Body literal propagation.*** *Let* $I = \{b(1,1),\ b(1,2)\}$ *be an interpretation and assume that the solver assigns* $a(1,1)$ *to true. Note that I satisfies the aggregate, therefore starting from* $a(1,1)$ *it is possible to propagate* $c(1)$ *to false because if* $c(1)$ *becomes true then the constraint is violated.*

In general, the propagator starts building all possible instantiations of the constraint and looks for undefined values that could be propagated. If there is a constraint instantiation such that each body literal is true, then we have to check if the count has reached the aggregate's guard minus one. In this case, we can propagate the aggregate body to false. Note that in this simple case we can just propagate undefined values of b but if the aggregate body is more complex propagation is not so simple. Since literals in the aggregate are in conjunction, in order to propagate a conjunction as false we need only one literal false and then for each possible conjunction we can propagate the last undefined literal of that conjunction. On the other hand, if there is a constraint instantiation such that exactly one literal is undefined and the aggregate is true then the undefined literal could be propagated as false. So, the propagator procedure is a complex and custom procedure that stores partial interpretation and implements optimized join techniques in order to build constraint instantiation.

In Algorithm 1, we report a simplified pseudo code propagator for the constraint described in the example. In particular, here we focus only on one propagation case. The complete propagator is based on several procedures, one for every literal in the body of the constraint or in the aggregate body, whose algorithms are similar to Algorithm 1.

Notation In ordered to better understand Algorithm 1 lets introduce some utility functions. Let l be a ground literal of the form $a(1,2)$ we define:

- *getPredicateName()* returns the predicate name of l, e.g., ”a”
- *getTermAt(integer i)* return the i-th term of l, e.g., l.getTermAt(0) return 1

The goal of the proposed procedure is to simulate the grounding of the constraint after a literal becomes true. In particular, assume that $a(1,2)$ becomes true, then the procedure is invoked passing $a(1,2)$. So X will be assigned to 1 and Z to 2 (lines 2 and 3, respectively). Afterward, true and undefined values of c matching Z are searched. If the procedure finds a true value of c matching Z then it evaluates the aggregate propagation. If the aggregate is also true then a conflict is found. Otherwise, if the aggregate's guard -1 has been reached the procedure makes a propagation to ensure that the aggregate is made false. On the other hand, if the procedure finds an undefined value of c, then it can be propagated as false only when the aggregates is true.

Algorithm 1 Example of propagator for one literal.

```
Input  : A literal $l$ of the form $a(X,Z)$.
Output: List of propagated literals.
if l.getPredicateName() == "a" then
    X = l.getTermAt(0)
    Z = l.getTermAt(1)
    tupleU = $\bot$
    tuplesC = trueLiteralsC.getValuesMatching(Z)
    tuplesUC = []
    if tupleU == $\bot$ then
        tuplesUC = undefinedLiteralsC.getValuesMatching(Z)
    foreach tupleC $\in$ tuplesC $\cup$ tuplesUC do
        if tupleC $\in$ tuplesUC then
            tupleU = tupleC
        if tupleU == $\bot$ then
            tuplesB = trueLiteralsB.getValuesMatching(X)
            if tuplesB.size() $\geq$ 2 then
                conflict detected on propagator
            else if tuplesB.size() == 1 then
                foreach undefinedb $\in$ undefinedLiteralsB.getValuesMatching(X) do
                    propagated undefinedb as False
        else
            tuplesB = trueLiteralsB.getValuesMatching(X)
            if tuplesB.size() $\geq$ 2 then
                propagate tupleU as False
```

5 Preliminary Result

Implementation. We started from the baseline system presented in [10]. The approach has been extended to support the compilation of the propagation of aggregates. In particular, the compiler has been implemented in C++, and its output, that is the propagator procedure itself, is also C++ code compliant to the WASP propagator interface, and is loaded in the ASP solver as a C++ dynamic library. The source code is not available yet, since we are in a development stage and we are working on a more robust and stable version.

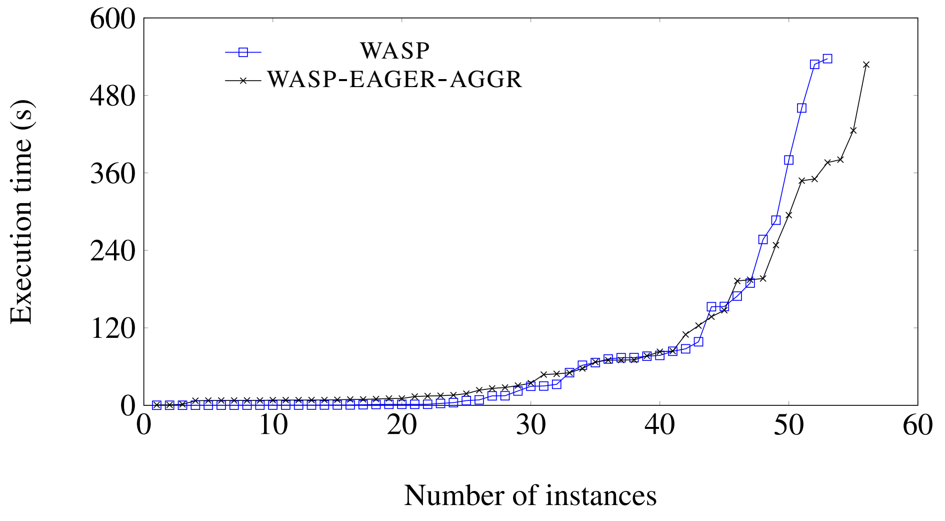
Experimental Settings. We carried our an experimental evaluation to empirically assessed the performance gain of the proposed approach w.r.t. the base solver WASP. Namely, we considered two hard benchmarks of the ASP competitions [7], namely *Combined Configuration* and *Abstract Dialectical Frameworks*, featuring some constraints containing *#count* aggregates.

In *Combined Configuration*, the problem is to configure an artifact by combining several components in order to achieve some goals; whereas in *Abstract Dialectical Frameworks* the problem is to find all statements which are necessarily accepted or rejected in a given abstract argumentation framework. In both benchmarks we compile all constraints with aggregates supported by our implementation (i.e., constraints with exactly one *#count* aggregate). The experiments were run on an Intel Xeon CPU E7-8880 v4 2.20GHz, time and memory were limited to 10 minutes and 4 GB, respectively.

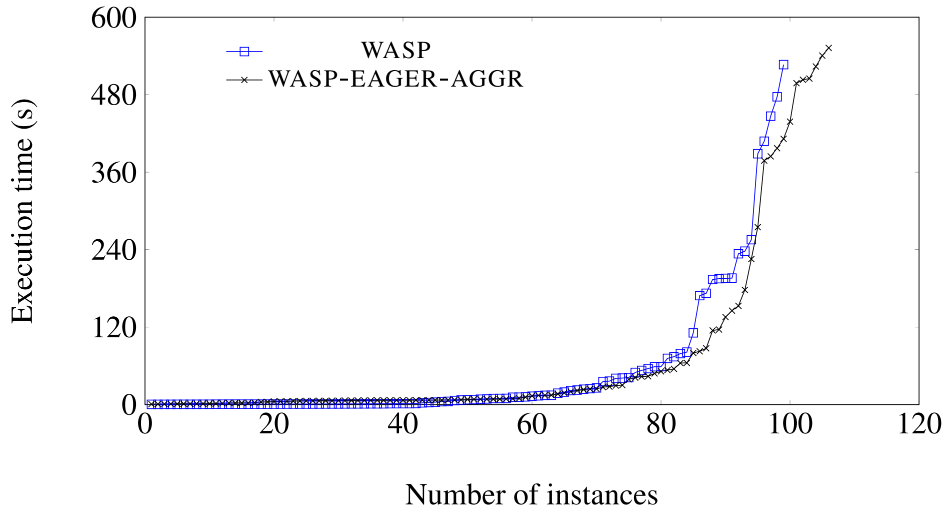
Results. The results are presented in Figure 3a and Figure 3b as two cactus plots. Overall, our approach is able to boost the performance of WASP, with the result of obtaining smaller execution times, on average, and more solved instances (3 more instances for *Combined Configuration* and 7 more for *Abstract Dialectical Frameworks*). The results are very promising, also considering the fact that the benchmarks in the ASP competitions are more oriented towards the evaluation of solving techniques.

6 Conclusion

In this paper we provided an overview of new strategies to deal with grounding bottleneck in presence of aggregates. Our research is currently focused on compilation based approach, in particular, on eager compilation of problematic constraints possibly containing *#count* aggregates. Obtained results were very promising and the extension of this approach to cover other aggregates and rules might lead to an improved of the performance of existing ASP solvers. In particular, we are currently able to compile only simple constraints and also constraint with aggregates but our goal is to move towards the compilation of general rules. A first attempt has been made together with a rewriter that uses rules to rewrite aggregates, but we are in development stage and no experimental analysis has been conducted yet. The crucial point of the overall work is how to generate procedures that have to be smart enough to save as much space as possible, in order to avoid grounding bottleneck, but at the same time they have to be also time efficient. We plan to extend the experimental analysis in order to include different benchmarks and to ensure that our approach is able to solve programs that are currently not solvable by state-of-the-art solvers. Moreover, we also plan to evaluate the impact of our techniques also on problems that are not affected by grounding bottleneck, in order to estimate the overhead of our approach also in this case. Such analysis might lead to a better understanding of the solving procedure and it can lead to the



(a) Combined configuration.



(b) Abstract dialectical frameworks.

Figure 3: Preliminary experimental results on two hard benchmarks taken from ASP competition.

implementation of (automated) systems that are able to reason, in some way, about the sub-program to compile and afterward create a procedure that is strictly customized on that sub-program.

References

[1] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri & Jessica Zangari (2017): *The ASP System DLV2*. In: *LPNMR*, *LNCS* 10377, Springer, pp. 215–221. Available at `https://doi.org/10.1007/978-3-319-61660-5_19`.

[2] Mario Alviano, Carmine Dodaro, Nicola Leone & Francesco Ricca (2015): *Advances in WASP*. In: *LPNMR*, *LNCS* 9345, Springer, pp. 40–54, doi:10.1007/978-3-319-23264-5_5.

[3] Giovanni Amendola, Gianluigi Greco, Nicola Leone & Pierfrancesco Veltri (2016): *Modeling and Reasoning about NTU Games via Answer Set Programming*. In: *IJCAI*, IJCAI/AAAI Press, pp. 38–45, doi:10.1016/j.artint.2016.01.011.

[4] Marcello Balduccini & Yuliya Lierler (2017): *Constraint answer set solver EZCSP and why integration schemas matter*. *TPLP* 17(4), pp. 462–515. Available at `https://doi.org/10.1017/S1471068417000102`.

[5] Jori Bomanson, Tomi Janhunen & Antonius Weinzierl (2019): *Enhancing Lazy Grounding with Lazy Normalization in Answer-Set Programming*. In: *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, AAAI Press, pp. 2694–2702. Available at `https://doi.org/10.1609/aaai.v33i01.33012694`.

[6] Gerhard Brewka, Thomas Eiter & Miroslaw Truszczynski (2011): *Answer set programming at a glance*. *Commun. ACM* 54(12), pp. 92–103, doi:10.1145/2043174.2043195.

[7] Francesco Calimeri, Martin Gebser, Marco Maratea & Francesco Ricca (2016): *Design and results of the Fifth Answer Set Programming Competition*. *Artif. Intell.* 231, pp. 151–181, doi:10.1016/j.artint.2015.09.008.

[8] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca & Peter Schüller (2017): *Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis*. *TPLP* 17(5-6), pp. 780–799, doi:10.1017/S1471068417000254.

[9] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca & Peter Schüller (2019): *Partial Compilation of ASP Programs*. *TPLP* 19(5-6), pp. 857–873. Available at `https://doi.org/10.1017/S1471068419000231`.

[10] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca & Peter Schüller (2020): *Overcoming the Grounding Bottleneck Due to Constraints in ASP Solving: Constraints Become Propagators*. In Christian Bessiere, editor: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, ijcai.org, pp. 1688–1694. Available at `https://doi.org/10.24963/ijcai.2020/234`.

[11] Bernardo Cuteri, Kristian Reale & Francesco Ricca (2019): *A Logic-Based Question Answering System for Cultural Heritage*. In: *JELIA*, *Lecture Notes in Computer Science* 11468, Springer, pp. 526–541, doi:10.1007/978-3-030-19570-0_35.

[12] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli & Gianfranco Rossi (2009): *GASP: Answer Set Programming with Lazy Grounding*. *Fundam. Inform.* 96(3), pp. 297–322, doi:10.3233/FI-2009-180.

[13] Carmine Dodaro & Marco Maratea (2017): *Nurse Scheduling via Answer Set Programming*. In: *LPNMR*, *LNCS* 10377, Springer, pp. 301–307, doi:10.1007/978-3-319-61660-5_27.

[14] Esra Erdem, Michael Gelfond & Nicola Leone (2016): *Applications of Answer Set Programming*. *AI Magazine* 37(3), pp. 53–68, doi:10.1609/aimag.v37i3.2678.

[15] Esra Erdem & Volkan Patoglu (2018): *Applications of ASP in Robotics*. *KI* 32(2-3), pp. 143–149. Available at `https://doi.org/10.1007/s13218-018-0544-x`.

[16] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub & Philipp Wanko (2016): *Theory Solving Made Easy with Clingo 5*. In: *ICLP TCs*, *OASICS* 52, pp. 2:1–2:15, doi:10.4230/OASIcs.ICLP.2016.2.

[17] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca & Torsten Schaub (2018): *Evaluation Techniques and Systems for Answer Set Programming: a Survey*. In: *IJCAI*, ijcai.org, pp. 5450–5456, doi:10.24963/ijcai.2018/769.

[18] Michael Gelfond & Vladimir Lifschitz (1991): *Classical Negation in Logic Programs and Disjunctive Databases*. *New Generation Comput.* 9(3/4), pp. 365–386, doi:10.1007/BF03037169.

[19] Benjamin Kaufmann, Nicola Leone, Simona Perri & Torsten Schaub (2016): *Grounding and Solving in Answer Set Programming*. *AI Magazine* 37(3), pp. 25–32, doi:10.1609/aimag.v37i3.2672.

[20] Claire Lefèvre & Pascal Nicolas (2009): *The First Version of a New ASP Solver: ASPeRiX*. In: *LPNMR*, *LNCS* 5753, Springer, pp. 522–527, doi:10.1007/978-3-642-04238-6_52.

[21] Giuseppe Mazzotta, Bernardo Cuteri, Carmine Dodaro & Francesco Ricca (2020): *Compilation of Aggregates in ASP: Preliminary Results*. In Francesco Calimeri, Simona Perri & Ester Zumpano, editors: *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020*, *CEUR Workshop Proceedings* 2710, CEUR-WS.org, pp. 278–296. Available at `http://ceur-ws.org/Vol-2710/paper18.pdf`.

[22] Max Ostrowski & Torsten Schaub (2012): *ASP modulo CSP: The clingcon system*. *TPLP* 12(4-5), pp. 485–503, doi:10.1017/S1471068412000142.

[23] Peter Schüller (2016): *Modeling Variations of First-Order Horn Abduction in Answer Set Programming*. *Fundam. Inform.* 149(1-2), pp. 159–207, doi:10.3233/FI-2016-1446.

[24] Benjamin Susman & Yuliya Lierler (2016): *SMT-Based Constraint Answer Set Solver EZSMT (System Description)*. In: *ICLP TCs*, *OASICS* 52, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 1:1–1:15, doi:10.4230/OASIcs.ICLP.2016.1.

[25] Richard Taupe, Antonius Weinzierl & Gerhard Friedrich (2019): *Degrees of Laziness in Grounding - Effects of Lazy-Grounding Strategies on ASP Solving*. In Marcello Balduccini, Yuliya Lierler & Stefan Woltran, editors: *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, *Lecture Notes in Computer Science* 11481, Springer, pp. 298–311. Available at `https://doi.org/10.1007/978-3-030-20528-7_22`.

[26] Antonius Weinzierl (2017): *Blending Lazy-Grounding and CDNL Search for Answer-Set Solving*. In: *LPNMR*, *LNCS* 10377, pp. 191–204, doi:10.1007/978-3-319-61660-5_17.