Compilation of ASP programs: Recent developments

Carmine **Dodaro**, Giuseppe **Mazzotta** and Francesco **Ricca**

University of Calabria, Rende CS, 87036, Italy

Abstract
Answer Set Programming (ASP) is a well-known declarative AI formalism developed in the area of logic programming and nonmonotonic reasoning. Modern ASP systems are based on the ground&solve approach. Although effective in industrial and academic applications ground&solve ASP systems are still unable to handle some classes of programs because of the so-called grounding bottleneck problem. Compilation of ASP programs demonstrated to be an effective technique for overcoming the grounding bottleneck. In the paper titled "Compilation of Aggregates in ASP Systems", which we presented in the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022), the first compilation-based approach for ASP programs that contain aggregates has been presented. In this paper, we outline the benefits of compiling ASP programs and mention possible developments in this line of research.

1. Introduction

Answer Set Programming (ASP) [1] is a fully declarative AI formalism. ASP was developed in the Logic Programming and Knowledge Representation communities and represents the most well-known logic formalism that is based on stable model semantics [2]. ASP demonstrated to be very useful for representing and solving many classes of problems. Indeed, ASP features both a standardized first-order language and several efficient systems [3]. ASP has many applications both in academia and in industry such as planning, scheduling, robotics, decision support, natural language understanding, and more (cfr. [4]). ASP is supported by efficient systems, but the improvement of their performance is still an open and challenging research topic. State-of-the-art ASP systems are based on the ground&solve approach [5]. The first-order input program is transformed by the *grounder* module in its propositional counterpart, whose stable models are computed by the solver, implementing a Conflict-Driven Clause Learning (CDCL) algorithm [5]. ASP implementations based on ground&solve, basically, enabled the development of ASP applications. However, there are classes of ASP programs whose evaluation is not efficient (sometimes not feasible) due to the combinatorial blow-up of the program produced by the grounding step. This issue is known under the term *grounding bottleneck* [6, 7]. Many attempts have been done to approach the grounding bottleneck, from language extensions [6, 8, 9, 10, 11, 12, 13] to lazy grounding [14, 15, 16]. These techniques obtained good preliminary results, but lazy grounding systems are still not competitive with ground&solve systems on common problems [3]. Recent research suggests that compilation-based techniques can mitigate the grounding bottleneck problem due to constraints [17, 18].

CEUR Workshop Proceedings (CEUR-WS.org)

Essentially, their idea is to identify the subprograms causing the grounding bottleneck, and subsequently translate them to propagators, which are custom procedures that lazily simulate the ground&solve on the removed subprograms. Problematic constraints are removed from the non-ground input program and the corresponding propagator is dynamically linked to the solver to simulate their presence at running time. Compilation of ASP programs demonstrated to be an effective technique for overcoming the grounding bottleneck. This approach is meant to speed-up computation by avoiding full grounding and exploiting information known at compilation time to create custom procedures that are specific to the program at hand. In the paper titled "Compilation of Aggregates in ASP Systems", that we presented in the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022) [19], the first compilation-based approach for ASP programs that contain aggregates [20] has been presented[1], despite aggregates are among the most relevant and commonly-employed constructs of ASP [21]. In this paper, we propose a compilation-based approach for ASP programs with aggregates. We identify the syntactic conditions under which a program with aggregates can be compiled, thus extending the definition of compilable subprograms of [17]. Then, we implement the approach on top of the state-of-the-art ASP solver wasp [22]. In this paper we overview our compilation technique, outline the benefits of compiling ASP programs, and mention possible developments in this line of research. In the following, we assume the reader familiar with Answer Set Programming syntax and semantics, and refer the reader to introductory and founding papers for details [1, 2, 20].

2. Compilation-based approaches

In recent years, compilation-based approaches have been proposed to overcome the grounding bottleneck problem. Basically, the idea behind these approaches is to compile the grounding-intensive part of an ASP program into a dedicated procedure, named *propagator*, that simulates it into an ASP solver during the solving phase. A first attempt has been proposed in [17] where the authors identified some syntactic conditions that allow the compilation of a subprogram into a lazy propagator. In this approach, the compiled program is omitted in the original program and is simulated by a procedure that, as soon as a candidate stable model, M, of the remaining program is computed, checks whether M is coherent or not with omitted subprogram. If it is the case then the candidate stable model is also a model of the original program otherwise violated constraints are added into the solver and the model computation process restarts. This strategy proved to be very effective on grounding-intensive benchmarks and so it motivated the idea to push forward the concept of propagators. Another compilation-based approach for ASP constraints has been proposed in [18]. This work introduced the translation of constraints of an ASP program into a propagator of the CDCL algorithm that works on partial interpretation. More precisely, constraints are removed from the input program and they are compiled into an eager propagator, that simulates the propagation of the omitted constraints during the solving phase. In this approach solver and propagator works together in order to prevent constraints failure and so, unlike lazy propagators, the candidate stable model produced by the solver will be coherent also with the omitted constraints. Eager compilation of grounding-intensive

---

[1]Recipient of the "Outstanding student paper award honorable mention of AAAI2022"

constraints introduced significant improvements outperforming traditional ASP solvers but, unfortunately, it is limited to simple constraints that do not contain aggregates. Aggregates are among the most used constructs that make ASP effective in representing complex problems in a very compact way and very often their grounding could be a bottleneck. An attempt to extend the compilation also to constraints with aggregate has been proposed in [23]. Basically, this approach provides a compilation strategy for constraints with a count aggregate into an eager propagator. Count aggregates are normalized under a unique comparison operator, $\geq$, and then, following the idea described in [18], they are compiled into a custom procedure that simulates aggregates propagations. Results obtained by this extension highlighted the strength of the contribution solving more instances than state-of-the-art system WASP on hard benchmarks from ASP competitions [7] containing constraints with count aggregate. Starting from the idea proposed in [23] we generalized the compilation of aggregates to rules with sum or count aggregate [19]

2.1. Conditions for Splitting and Compiling Programs

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

$$h \leftarrow b_1, \ldots, b_k, \sim b_{k+1}, \ldots, \sim b_m \tag{1}$$

with $m \geq k \geq 0$, where h is a standard atom referred to as *head* and it can absent, whereas $b_1, \ldots, b_k, \sim b_{k+1}, \ldots, \sim b_m$ is the *body*, $b_1, \ldots, b_k$ are atoms, and $b_{k+1}, \ldots, b_m$ are standard atoms. Moreover, for a rule r, H_r and B_r are two sets containing the head and the body of a rule r, respectively, B_r^+ and B_r^- are two sets containing the positive and the negative body of r, respectively, B_r^a denotes the set of aggregate atoms appearing in B_r, and $Conj^+(B_r^a)$ and $Conj^-(B_r^a)$ denotes the set of positive and negative standard literals appearing in the aggregate atoms of the body, respectively. Given a program π, a sub-program of π is a set of rules $\lambda \subseteq \pi$. In what follows, we denote with $preds(X)$ the set of predicate names appearing in X where X is a structure (literal, conjunction, rule, program, etc). Moreover, given a set of rules λ, let $head(\lambda) = \{a \mid a \in H_r, r \in \lambda\}$.

Definition 1. *Given an ASP program π, the dependency graph of π, denoted DG_π, is a labeled graph (V, E) where V is the set of predicate names appearing in some head of π, and E contains (i) $(v_1, v_2, +)$, if there is $r \in \pi \mid v_1 \in preds(B_r^+) \cup preds(Conj^+(B_r^a)), v_2 \in preds(H_r)$; (ii) $(v_1, v_2, -)$, if there is $r \in \pi \mid v_1 \in preds(B_r^-) \cup preds(Conj^-(B_r^a)), v_2 \in preds(H_r)$.*

Definition 2. *Given an ASP program π, an ASP sub-program $\lambda \subseteq \pi$ is compilable w.r.t. π if (i) DG_λ has no loop in it; (ii) for each $p \in pred(head(\lambda)), p \notin pred(\pi \setminus \lambda)$; (iii) given two rules $r_1, r_2 \in \lambda, r_1 \neq r_2, preds(H_{r_1}) \cap preds(H_{r_2}) = \emptyset$; and (iv) for each $r \in \lambda, |B_r^a| \leq 1$.*

2.2. Normalization of the Input Program

In the following, we describe the main preprocessing steps that are performed to compile the input sub-program. First of all the sub-program λ is analyzed in order to be split into two sub-programs, namely λ_{lazy} and λ_{eager}. This analysis consists of navigating the dependency

graph starting from nodes that have no incoming edges and recursively label predicates that appear in the body of a rule whose head predicate has been already labeled. In this way, the rules whose head predicate has not been labeled could be treated in a lazy way (λ_{lazy}); other rules are in λ_{eager}. For λ_{eager}, we perform a rewriting to obtain a normalized form with rules of a specific format in order to have a uniform treatment of all the rules to compile. Also, for a structure (set, list, conjunction, etc.) of elements X, let $vs(X)$ be the set of all variables appearing in X.

Step 1. Each rule $r \in \lambda_{eager}$ of the form (1), with $|B_r^a| = 1$, $f(\{Vars : Conj\}) \prec T \in B_r^a$, and $\prec \in \{<, \leq, >, \geq\}$, is replaced by the following rules:

1. $as_r(Vars, \rho) \leftarrow Conj$;
2. $bd_r(\rho, T) \leftarrow B_r \setminus B_r^a$;
3. $aggr_r(\rho, T) \leftarrow dm_r(\rho, T), f(\{Vars : as_r(Vars, \rho)\}) \geq G$, where $G = T$ if $\prec \in \{\geq, <\}$, and $G = T + 1$ if $\prec \in \{\leq, >\}$;
4. $h \leftarrow bd_r(\rho, T), aggr_r(\rho, T)$ if $\prec \in \{>, \geq\}$ and
   $h \leftarrow bd_r(\rho, T), {\sim}aggr_r(\rho, T)$ if $\prec \in \{<, \leq\}$;

where ρ is $vs(Conj) \cap vs(B_r \setminus B_r^a)$. Intuitively, ρ is a set of all variables appearing in both aggregate set and body.

Example 1. *Let assume r to be the following rule:*

$$a(X, W) \leftarrow \quad b(X, Y), c(Y, W),$$
$$\#sum\{Z : d(X, Z), {\sim}e(Z)\} \geq W.$$

Then, r is replaced by the following rules:

$$
\begin{aligned}
r_1 : \quad & as_r(Z, X) && \leftarrow d(X, Z), {\sim}e(Z) \\
r_2 : \quad & bd_r(X, W) && \leftarrow b(X, Y), c(Y, W) \\
r_3 : \quad & aggr_r(X, W) && \leftarrow dm_r(X, W), \\
& && \#sum\{Z : as_r(Z, X)\} \geq W \\
r_4 : \quad & a(X, W) && \leftarrow bd_r(X, W), aggr_r(X, W)
\end{aligned}
$$

Step 2. Each rule $r \in \lambda_{eager}$ of the form (1), with $|B_r^a| = 1$, $f(\{Vars : Conj\}) \prec T \in B_r^a$, and $\prec \in \{=\}$, is replaced by the rules *1.*, *2.*, and by the following rules:

5. $aggr_r^1(\rho, T) \leftarrow dm_r(\rho, T), f(\{Vars : as_r(Vars, \rho)\}) \geq T$;
6. $aggr_r^2(\rho, T) \leftarrow dm_r(\rho, T), f(\{Vars : as_r(Vars, \rho)\}) \geq T + 1$;
7. $h \leftarrow bd_r(\rho, T), aggr_r^1(\rho, T), {\sim}aggr_r^2(\rho, T)$.

Step 3. Each rule $r \in \lambda_{eager}$, with $|B_r^a| = 0$, is replaced by the following rules:

8. $h \leftarrow aux_r(vs(B_r^+))$;
9. $\leftarrow aux_r(vs(B_r^+)), \overline{b_i}$ \hfill $\forall i \in \{1, \ldots, m\}$;
10. $\leftarrow B_r, {\sim}aux_r(vs(B_r^+))$.

This step is applied also to rules from steps 1 and 2.

Example 2. *Let assume r to be the following rule:*

$$a(Z, X) \leftarrow d(X, Z), \sim e(Z).$$

Then, r is replaced by the following rules:

$$
\begin{aligned}
r_8 : \quad a(Z, X) \quad &\leftarrow aux_r(X, Z) \\
r_9' : \quad &\leftarrow aux_r(X, Z), \sim d(X, Z) \\
r_9'' : \quad &\leftarrow aux_r(X, Z), e(Z) \\
r_{10} : \quad &\leftarrow d(X, Z), \sim e(Z), \sim aux_r(X, Z).
\end{aligned}
$$

Intuitively, the normalization ensures that aggregate functions are applied to a set of atoms, and rules are subject to a form of completion [24]. After applying the normalization step the program contains only rules of the form:

$$
\begin{aligned}
&(1) \quad h \leftarrow b \\
&(2) \quad h \leftarrow d, \#count(\{Vars : b\}) \geq g \\
&(3) \quad h \leftarrow d, \#sum(\{Vars : b\}) \geq g \\
&(4) \quad \leftarrow c_1, ..., c_n
\end{aligned}
$$

Thus, the compiler will only have to produce propagators simulating the above-mentioned four rule types. Finally, it is important to emphasize that atoms of the form $dm_r(\cdot)$ and $aux_r(\cdot)$ do not appear in the head of any rule in the program, and thus the ASP semantics would make them false in all stable models. Therefore, in our approach, they are treated as *external atoms* [25], whose instantiation and truth values are defined at running time in the propagator when the base $B_{\lambda_{eager}}$ is determined.

2.3. Compilation

The compilation step, basically, follows the baseline described in [18]. In particular, a rewritten subprogram λ will be compiled into a C++ procedure that includes propagator code for each rule $r \in \lambda$. Afterward, the eager propagator procedure is nested into the state-of-the-art system WASP as a dynamic library. In order to better understand the idea behind our approach let us consider the following rules produced by the normalization step:

$$
\begin{aligned}
r_1 : \quad &a(X) :\text{--} aux(X, Y). \\
r_2 : \quad aggr(X) :\text{--} &dm(X), \#count\{Z : as(Z, X)\} \geq 2.
\end{aligned}
$$

Basically, the idea behind generated propagators is reported in Algorithms 1 and 2. For more details on the automatic generation of such propagators we refer the reader to [19].

3. Experiments

Our approach has been evaluated in three different settings: (i) A simple benchmark used to show the limits of ground&solve (cfr. [19]). (ii) All benchmarks from ASP competitions [7] including at least one rule with aggregates that can be compiled under our conditions. (iii) Grounding-intensive benchmarks from the literature: Component Assignment Problem [26], Dynamic In-Degree Counting and Exponential-Save [27].

Algorithm 1 Propagator example for r_1

1: **if** $\exists\, a(X) \in I$ **then**
2: $T = \{Y : aux(X, Y)\, is\, true\}$
3: $U = \{Y : aux(X, Y)\, is\, not\, assigned\}$
4: **if** $\mid T \cup U \mid = 1$ **then**
5:   $I = I \cup \{aux(X, Y) \mid Y \in U\}$
6: **end if**
7: **else**

8: **if** $\exists \sim a(X) \in I$ **then**
9:   $I = I \cup \{aux(X, Y) \mid$
      $aux(X, Y)\, is\, not\, false\}$
10: **else**
11:   **if** $\exists Y \mid a(X, Y) \in I$ **then**
12:    $I = I \cup \{a(X)\}$
13:   **end if**
14: **end if**
15: **end if**

Algorithm 2 Propagator example for r_2

1: **for all** $X : \exists\, dm(X)$ **do**
2: $T = \{Z : as(Z, X)\, is\, true\}$
3: $F = \{Z : as(Z, X)\, is\, false\}$
4: $U = \{Z : as(Z, X)\, is\, not\, assigned\}$
5: **if** $\sim aggr(X) \in I$ **then**
6:   **if** $\mid T \mid = 1$ **then**
7:    $I = I \cup \{\sim a(Z) \mid Z \in U\}$
8:   **end if**
9: **else**
10:   **if** $aggr(X) \in I$ **then**
11:    **if** $\mid T \cup U \mid = 2$ **then**
12:     $I = I \cup \{a(Z) \mid Z \in U\}$
13:    **end if**

14: **else**
15:   **if** $\mid T \cup U \mid < 2$ **then**
16:    $I = I \cup \{\sim aggr(X)\}$
17:   **else**
18:    **if** $\mid T \mid >= 2$ **then**
19:     $I = I \cup \{aggr(X)\}$
20:    **end if**
21:   **end if**
22: **end if**
23: **end if**
24: **end for**

Hardware and Software. In all the experiments, the compilation-based approach, reported as wasp-comp, has been compared with the plain version of wasp [22] v. 169e40d and with the state-of-the-art system clingo v. 5.4.0 [25]. Moreover, for Dynamic In-Degree Counting and Exponential-Save we considered also the system alpha [16], which is based on lazy-grounding techniques. alpha cannot be used for other experiments since it does not support some of the language constructs used in the benchmarks (e.g., *choice rules* with bounds). Experiments were executed on Xeon(R) Gold 5118 CPUs running Ubuntu Linux (kernel 5.4.0-77-generic), time and memory are limited to 2100 seconds and 4GB, respectively.

Table 1
Solvers evaluation on setting (i).

	k	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	20000	30000	40000
CLINGO	t	0.66	3.23	7.68	13.62	22.23	32.73	42.88	-	-	-	-	-	-
	mem	40.2	303.5	709.5	1216.8	1933.3	2871.1	3576.7	-	-	-	-	-	-
WASP	t	1.02	4.36	9.98	18.47	28.80	42.47	58.31	-	-	-	-	-	-
	mem	59.6	286.5	696.3	1168.1	2215.4	2807.2	3402.0	-	-	-	-	-	-
WASP-COMP	t	**0**	**0.1**	**0.24**	**0.53**	**0.93**	**1.19**	**1.44**	**1.96**	**2.89**	**3.87**	**12.52**	**26.43**	**58.88**
	mem	0	18.6	38.5	53.7	74.6	109.8	142.3	152	191.6	254.7	898.5	1888.3	3319.6

Table 2
Solvers evaluation on settings (ii) and (iii).

Benchmark	#	WASP-COMP				WASP			CLINGO		
		sol.	sum t	avg mem	comp. t	sol.	sum t	avg mem	sol.	sum t	avg mem
Abs. Dia. Fram.	200	117	23467.7	118.9	6.47	123	24311.1	117.2	**200**	1244.6	26.4
Bottle Filling	100	**100**	2208.3	773.8	11.98	**100**	545.8	761.8	**100**	**394.0**	213.4
Con. Max. Den.	26	**6**	2166.8	26.4	31.8	**6**	**427.5**	31.4	4	85.4	11.1
Crossing Min.	85	**84**	305.3	16.4	8.97	**84**	257.3	14.5	51	10013.0	20.4
Incr. Sched.	500	329	25960.7	96.6	6.94	317	42282.4	210.9	**345**	18952.8	253.7
Partner Units	112	52	35525.7	160.9	12.51	69	29612.5	247.6	**80**	7147.8	73.8
Solitaire	27	**25**	601.7	27.0	12.03	**25**	**140.2**	18.4	**25**	273.1	9.8
Weighted Seq.	65	**65**	6663.4	36.1	8.46	**65**	4713.5	32.3	**65**	**569.6**	14.5
Comp. Assign.	302	**188**	56137.2	814.4	20.79	70	15325.81	973.3	118	36832.0	1288.6
Dyn. Ind. Cou.	80	**80**	**48.0**	62.0	6.48	**80**	1374.6	619.4	**80**	1282.1	551.8
Exp.-Save	27	**21**	2594.1	527.0	*	6	18.5	369.8	7	24.4	365.4

Results. Concerning the setting (i), we report execution time (t) and used memory (mem) for each instance in Table 1. In particular, we observe that CLINGO and WASP cannot solve instances with $k \geq 8000$, whereas WASP-COMP can efficiently handle instances up to values of $k = 40000$. Concerning the setting (ii) and (iii) we report for each solver the number of solved instances (sol.), the sum of running times (sum t) in seconds and the average used memory (avg mem) in MB. Concerning WASP-COMP we report also the compile time in seconds (comp t) that in general cases is not included in the sum of the time, since the compilation is done only once for each benchmark (with the exception of Exponential-Save) and, thus, can be done offline. Concerning the setting (ii), we observe that CLINGO obtains the best performance overall, and it is faster than WASP and WASP-COMP. This setting is useful to analyze the performance of the proposed technique on benchmarks that do not present a grounding issues and comparing WASP-COMP and WASP, we observe that the former is competitive with the latter in all the benchmarks but Abstract Dialectical Framework and Partner Units, where WASP solves 6 and 17 more instances than WASP-COMP. Nonetheless, WASP-COMP performs better than WASP on the benchmark Incremental Scheduling, solving 12 more instances. Interestingly, on this benchmark, WASP-COMP also uses less memory than WASP and CLINGO. Indeed, if only 512 MB are available (as reasonable in some cases) WASP-COMP solves 57 and 53 instances more than WASP and CLINGO, respectively. Concerning the setting (iii), WASP-COMP outperforms WASP in all the tested benchmarks solving 133 more instances overall. It is important to observe that each instance of Exponential-Save requires to be compiled since aggregates to be compiled are part of the instances. Therefore, in this benchmark, the solving time includes also the compilation time. Concerning Dynamic In-Degree Counting, WASP-COMP and WASP solve the same number of instances, but WASP-COMP is much faster. Moreover, we observe that WASP-COMP solves 84 more instances than CLINGO overall. Finally, we observe that WASP-COMP is competitive also with ALPHA, since the latter solves 80 instances of Dynamic In-Degree Counting in 492.0 seconds using 649.1 MB, and 27 instances of Exponential-Save in 332.9s using 227.8 MB.

4. Discussion

The grounding bottleneck is a limiting factor for ASP systems based on the ground&solve architecture. Compilation-based approaches proposed by [18] offer the possibility to mitigate this issue in presence of constraints while keeping the benefits of state-of-the-art approaches. Compilation of aggregates introduced significant improvements in grounding-intensive domains outperforming state-of-the-art systems. It is worth observing that, compilation techniques currently are applicable to a limited class of programs. Indeed, the conditions for a subprogram to be compilable cover many cases of practical interest (constraint-like subprograms) and they are orthogonal to many available constructs, such as weak constraints, cautious reasoning and qualitative preferences among answer sets [28], but are far from covering the entire range of possible cases. Indeed, the support for the compilation of rules is limited, since the system does not support: unfounded sets propagation, and answer-sets-generator rules (eg., disjunction, choice rules, and unrestricted negation). The missing constructs are not easy to support, thus there is a long promising route to follow to deliver a system able to compile any ASP program that suffers from grounding bottleneck.

5. Acknowledgements

References

[1] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103.

[2] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Gener. Comput. 9 (1991) 365–386.

[3] M. Gebser, N. Leone, M. Maratea, S. Perri, F. Ricca, T. Schaub, Evaluation techniques and systems for answer set programming: a survey, in: IJCAI, ijcai.org, 2018, pp. 5450–5456.

[4] E. Erdem, U. Öztok, Generating explanations for biomedical queries, TPLP 15 (2015) 35–78.

[5] B. Kaufmann, N. Leone, S. Perri, T. Schaub, Grounding and solving in answer set programming, AI Mag. 37 (2016) 25–32.

[6] M. Ostrowski, T. Schaub, ASP modulo CSP: the clingcon system, TPLP 12 (2012) 485–503.

[7] F. Calimeri, M. Gebser, M. Maratea, F. Ricca, Design and results of the fifth answer set programming competition, Artif. Intell. 231 (2016) 151–181.

[8] M. Balduccini, Y. Lierler, Constraint answer set solver EZCSP and why integration schemas matter, TPLP 17 (2017) 462–515.

[9] M. Balduccini, Y. Lierler, Integration schemas for constraint answer set programming: a case study, TPLP 13 (2013).

[10] R. A. Aziz, G. Chu, P. J. Stuckey, Stable model semantics for founded bounds, TPLP 13 (2013) 517–532.

[11] B. D. Cat, M. Denecker, M. Bruynooghe, P. J. Stuckey, Lazy model expansion: Interleaving grounding with search, J. Artif. Intell. Res. 52 (2015) 235–286.

[12] B. Susman, Y. Lierler, Smt-based constraint answer set solver EZSMT (system description), in: ICLP (Technical Communications), volume 52 of *OASICS*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 1:1–1:15.

[13] T. Eiter, C. Redl, P. Schüller, Problem solving using the HEX family, in: Computational Models of Rationality, College Publications, 2016, pp. 150–174.

[14] A. D. Palù, A. Dovier, E. Pontelli, G. Rossi, GASP: answer set programming with lazy grounding, Fundam. Informaticae 96 (2009) 297–322.

[15] C. Lefèvre, P. Nicolas, The first version of a new ASP solver : Asperix, in: LPNMR, volume 5753 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 522–527.

[16] A. Weinzierl, Blending lazy-grounding and CDNL search for answer-set solving, in: LPNMR, volume 10377 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 191–204.

[17] B. Cuteri, C. Dodaro, F. Ricca, P. Schüller, Partial compilation of ASP programs, TPLP 19 (2019) 857–873.

[18] B. Cuteri, C. Dodaro, F. Ricca, P. Schüller, Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators, in: IJCAI, ijcai.org, 2020, pp. 1688–1694.

[19] G. Mazzotta, F. Ricca, C. Dodaro, Compilation of aggregates in ASP systems, in: Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022, AAAI Press, 2022, pp. 5834–5841. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20527.

[20] W. Faber, G. Pfeifer, N. Leone, Semantics and complexity of recursive aggregates in answer set programming, Artif. Intell. 175 (2011) 278–298.

[21] M. Gebser, M. Maratea, F. Ricca, The sixth answer set programming competition, J. Artif. Intell. Res. 60 (2017) 41–95.

[22] M. Alviano, C. Dodaro, N. Leone, F. Ricca, Advances in WASP, in: LPNMR, volume 9345 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 40–54.

[23] G. Mazzotta, B. Cuteri, C. Dodaro, F. Ricca, Compilation of aggregates in ASP: preliminary results, in: F. Calimeri, S. Perri, E. Zumpano (Eds.), Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020, volume 2710 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, pp. 278–296. URL: http://ceur-ws.org/Vol-2710/paper18.pdf.

[24] K. L. Clark, Negation as failure, in: Logic and Data Bases, Advances in Data Base Theory, Plemum Press, New York, 1977, pp. 293–322.

[25] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, in: ICLP (Technical Communications), volume 52 of *OASICS*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:15.

[26] M. Alviano, C. Dodaro, M. Maratea, Shared aggregate sets in answer set programming, TPLP 18 (2018) 301–318.

[27] J. Bomanson, T. Janhunen, A. Weinzierl, Enhancing lazy grounding with lazy normalization in answer-set programming, in: AAAI, AAAI Press, 2019, pp. 2694–2702.

[28] E. Di Rosa, E. Giunchiglia, M. Maratea, A new approach for solving satisfiability problems with qualitative preferences, in: ECAI, volume 178 of *FAIA*, IOS Press, 2008, pp. 510–514.