Compilation of Aggregates in ASP Systems

Giuseppe Mazzotta, Francesco Ricca, Carmine Dodaro

Department of Mathematics and Computer Science, University of Calabria, Rende (CS), Italy
firstname.lastname@unical.it

Abstract

Answer Set Programming (ASP) is a well-known declarative AI formalism for knowledge representation and reasoning. State-of-the-art ASP implementations employ the ground&solve approach, and they were successfully applied to industrial and academic problems. Nonetheless 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. Recent research suggests that compilation-based techniques can mitigate the grounding bottleneck problem. However, no compilation-based technique has been developed for ASP programs that contain aggregates, which are one of the most relevant and commonly-employed constructs of ASP. In this paper, we propose a compilation-based approach for ASP programs with aggregates. We implement it on top of a state-of-the-art ASP system, and evaluate the performance on publicly-available benchmarks. Experiments show our approach is effective on ground-intensive ASP programs.

Introduction

Answer Set Programming (ASP) (Brewka, Eiter, and Truszczynski 2011) is a declarative AI formalism for knowledge representation and reasoning based on the stable model semantics (Gelfond and Lifschitz 1991). ASP is a viable solution for representing and solving many classes of problems thanks to a standardized first-order language that has been implemented in several efficient systems (Gebser et al. 2018). Indeed, ASP has been successfully applied to several academic and industrial AI applications such as planning, scheduling, robotics, decision support, natural language understanding and more (cfr. (Erdem and Öztok 2015)).

Albeit ASP is supported by efficient systems, the improvement of their performance is still an open and interesting research topic. State-of-the-art ASP systems are based on the ground&solve approach (Kaufmann et al. 2016). 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 (Kaufmann et al. 2016).

ASP implementations based on ground&solve, basically, enabled the development of ASP applications. Nonetheless 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* (Ostrowski and Schaub 2012; Calimeri et al. 2016). Many attempts have been done to approach the grounding bottleneck, from language extensions (Ostrowski and Schaub 2012; Balduccini and Lierler 2017, 2013; Aziz, Chu, and Stuckey 2013; Cat et al. 2015; Susman and Lierler 2016; Eiter, Redl, and Schüller 2016) to lazy grounding (Palù et al. 2009; Lefèvre and Nicolas 2009; Weinzierl 2017). These techniques obtained good preliminary results, but lazy grounding systems are still not competitive with ground&solve systems on common problems (Gebser et al. 2018). Recent research suggests that compilation-based techniques can mitigate the grounding bottleneck problem due to constraints (Cuteri et al. 2019, 2020). 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. 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. However, no compilation-based technique has been developed so far for ASP programs with aggregates (Faber, Pfeifer, and Leone 2011), which are among the most relevant and commonly-employed constructs of ASP (Gebser, Maratea, and Ricca 2017).

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 subprogram of (Cuteri et al. 2019). Then, we implement the approach on top of the state-of-the-art ASP solver WASP (Alviano et al. 2015). Propagators are both of the *eager* (called during the propagation phase) and *lazy* (called once an answer set candidate is found) kind (Dodaro and Ricca 2020). Our compiler automatically selects the kind based on the program structure. Experiments show our approach is effective on ground-intensive ASP programs.

Preliminaries

ASP Syntax. Variables are strings starting with uppercase letter and constants are non-negative integers or strings starting with lowercase letters. A *term* is either a variable or a constant. A *standard atom* is an expression of the form $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. A standard atom $p(t_1, \ldots, t_n)$ is ground if $t_1, \ldots, t_n$ are constants. A standard literal is an atom p or its negation $\sim p$, where $\sim$ denotes negation as failure. A *ground set* is a set of pairs of the form $\langle consts : conj \rangle$, where $consts$ is a list of constants and $conj$ is a conjunction of ground standard literals. A *symbolic set* is a set specified syntactically as $\{ Vars : Conj \}$, where $Vars$ is a non-empty list of variables, and $Conj$ is a non-empty conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where S is a symbolic set, and $f \in \{\#count, \#sum\}$ is an *aggregate function symbol*. An *aggregate atom* is of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq, =\}$ is a comparison operator, and T is a term called guard. An aggregate atom $f(S) \prec T$ is ground if T is a constant and S is a ground set. An *atom* is either a standard atom or an aggregate atom. A *literal* is an atom or its negation. The complement of l is denoted $\bar{l}$, and it is $\sim a$, if $l = a$, or a a if $l = \sim a$. This notation is extended also to sets of literals. For a set of literals L, L^+ and L^- denote the set of positive and negative literals appearing in L, resp. An ASP program π is a set of rules:

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

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. A *constraint* is a rule where h is absent, and a *fact* is a rule where $m = 0$. 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, U_π is the *Herbrand Universe*, and represents the set of all constants appearing in π; whereas B_π is the *Herbrand Base*, and the set of all possible ground atoms that can be built using predicate in π and constants in U_π. $\mathcal{B}$ denotes $B_\pi \cup \overline{B_\pi}$. Given an ASP expression (atom, literal, rule, program, etc.) ϵ and the Herbrand Universe U_π, we define $ground(\epsilon)$ as the set of all instantiations of ϵ built by assigning variables to constant in U_π.

ASP Semantics. An interpretation $I \subseteq \mathcal{B}$ is a set of literals. A literal l is true w.r.t I if $l \in I$, l is false w.r.t. I if $\bar{l} \in I$, otherwise it is undefined. In the following, for a program π, I_π^U denotes all literals in $\mathcal{B}$ that are undefined. For a rule, its body is a conjunction of literals. A conjunction $conj$ of literals is true w.r.t I if all literals in $conj$ are true w.r.t. I, and false if there is at least one literal in $conj$ that is false w.r.t. I. Let $I(S)$ denote the multiset $[t_1 | (t_1, \ldots, t_n) : conj \in S, ground(conj)$ 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 $I(S)$. An interpretation I is *total* if all literals in $\mathcal{B}$ are either true or false, otherwise I is *partial*. An interpretation I is *consistent* if for each literal $l \in I$, $\bar{l} \notin I$, otherwise it is *inconsistent*. A total and consistent interpretation I is a *model* for π if for each rule $r \in ground(\pi)$, the head of r is true whenever the body of r is true. Given a program π and an interpretation I, the *FLP-reduct* (Faber, Pfeifer, and Leone 2011) of π, denoted by π^I, is defined as 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 also a *stable model* for π if there is no $I' \subset I$ such that I' is a model for π^I.

Classical CDCL Evaluation. The standard solving approach for ASP uses two components, namely grounder and solver. The grounder takes as input an ASP program π and produces $ground(\pi)$, which is later on processed by the solver to produce stable models. The solver employes an algorithm that extends the CDCL algorithm with *propagators* specific to ASP (Kaufmann et al. 2016). The CDCL is based on a *choose-propagate-learn* pattern, and the idea is to build a stable model step-by-step starting from an empty interpretation I. At each step, a literal is heuristically selected and added to I (*choice*). Then, specific procedures, called (*eager*) *propagators*, are used to extend I with the deterministic consequences of this choice and their *reason*, i.e., literals in I leading to the propagation. In case the propagation leads to an inconsistency in the interpretation I, the algorithm *learns* a new constraint using the reason of each propagated literal, undoes the choices leading to the inconsistency, and restores the consistency of I. This process is repeated until I is a stable model candidate or the consistency of I cannot be restored, showing that no stable model can be found. In the first case I is then analyzed by *lazy propagators*, which perform additional checks on I. If I is consistent under those checks, then the algorithm terminates, otherwise new constraints are added to $ground(\pi)$ and the algorithm restarts. We refer the reader to (Brewka, Eiter, and Truszczynski 2016) for a gentle overview of ASP.

Loop Unrolling and Dead Code Elimination. We recall two optimizations used by compilers, namely *loop unrolling* and *dead code elimination* (Muchnick 1997). Loop unrolling consists of removing loop control instructions by proper replicating the loop body in order to obtain an equivalent code. The following (exemple) portion of code clarifies this:

```
for($i = 0, \ldots, n$)                              for($i = 0, \ldots, n$){
  for($j = 0, \ldots, 3$)                              $a[0] = b[0] + i$;
    if($j < 2$) $a[j] = b[j] + i$;     $\Longrightarrow$      $a[1] = b[1] + i$;
    else        $b[j] = a[j] + i$;                     $b[2] = a[2] + i$;}
```

where the code on the left-hand side is transformed resulting in the code reported on the right-hand side. Note that for each iteration of the outermost statement one has to perform $3 \cdot n$ increments of the variable j and $3 \cdot n$ evaluations of the **if** statement, which are not required after loop unrolling. Clearly, the benefits of loop unrolling can be seen with large values of n. Dead code elimination consists of removing instructions that can never be executed, e.g., the body of statements that cannot be reached. We refer the reader to (Muchnick 1997) for more details.

Compilation of Aggregates

Conditions for Splitting and Compiling Programs

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 $r \in \lambda$, $|B_r^a| \leq 1$.*

Normalization of the Input Program

In the following we describe the main preprocessing steps that are performed to the sub-program to compile. First of all the sub-program λ is analyzed in order to be split in 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 appears 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, and 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:*

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

Then, r is replaced by the following rules:

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

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 $r \in \lambda_{eager}$ of the form (1), with $|B_r^a| = 0$, is replaced by the following rules:

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

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

Intuitively, the normalization ensures that aggregate functions are applied to set of atoms, and rules are subject to a form of completion (Clark 1977). After applying the normalization step the program contains only rules of the form:

$$h \leftarrow b \tag{2}$$
$$h \leftarrow d, \#count(\{Vars : b\}) \geq g \tag{3}$$
$$h \leftarrow d, \#sum(\{Vars : b\}) \geq g \tag{4}$$
$$\leftarrow c_1, ..., c_n \tag{5}$$

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* (Gebser et al. 2016), whose instantiation and truth values are defined at running time in the propagator when the base $B_{\lambda_{eager}}$ is determined.

Compilation

In this section we present our strategy to compile a subprogram into a propagator focusing on the main compilation procedures for space reasons. In order to present our algorithms we followed the baseline and syntactic conventions described in (Cuteri et al. 2020) with some minor differences. In particular, the compiler transforms a logic program in the pseudo-code of the corresponding propagator. In this presentation the code enclosed between $\ll\gg$ is printed by the compiler as it is, but the code enclosed in $[\![ ]\!]_{\sharp}$, is first substituted with its run-time value before being printed. For example, given a rule r of type (2) of the form $a(X) \leftarrow b(X)$, Algorithm 2 at line 3 prints **case** "a" :.

Given a compilable sub-program λ in input, λ is split in two subprograms, namely λ_{lazy} and λ_{eager} as explained in previous section. Then λ_{lazy} and λ_{eager} are compiled in different ways as described in the following.

Algorithm 1 CompileProgram

```
Input  : A normalized ASP program $P$
Output: Prints propagator procedure of $P$ propagating $l$
1  begin
2      ≪  $I_l = \emptyset$≫
3      ≪  switch $pred(l)$≫
4      forall the $r \in P$ do
5          if $r$ is of type (2) then
6              CompileRule(r)
7          else if $r$ is of type (3) then
8              CompileRuleWithCount(r)
9          else if $r$ is of type (4) then
10             CompileRuleWithSum(r)
11         else if $r$ is of type (5) then
12             forall the $c \in B_r$ do
13                 CompileConstraintWithStarter(r,c)
14     ≪return $I_l$≫
```

Algorithm 2 CompileRule

```
Input  : A rule r of type (2), i.e., $h \leftarrow b$
Output: Prints propagator procedure for the rule r
1  begin
2      ≪$\sigma = \epsilon$≫ ≪$u = \bot$≫
3      ≪case  ” $\llbracket pred(h) \rrbracket_{\lightning}$”:≫
4      forall the $i \in 1, ..., |trm(h)|$ do
5          if $trm(h)[i]$ is variable then
6              ≪$\sigma = \sigma \cup \{ \llbracket trm(h)[i] \rrbracket_{\lightning} \mapsto trm(l)[ \llbracket i \rrbracket_{\lightning} ]\}$≫
7      ≪   $T = \{p \in I^+ | \, match(\sigma( \llbracket b \rrbracket_{\lightning} ), p)\}$≫
8      ≪   $U = \{p \in (\mathcal{B} \setminus I)^+ | \, match(\sigma( \llbracket b \rrbracket_{\lightning} ), p)\}$≫
9      ≪   if $|T| == 0 \, \wedge \, |U| == 1 \, \wedge \, l \in I^+$≫
10     ≪       $u = p \, | \, p \in U$≫
11     ≪       $R = \{l\} \cup \{p \in I^- | \, match(\sigma( \llbracket b \rrbracket_{\lightning} ), p)\}$≫
12     ≪       $I_l = I_l \cup (u, R)$≫
13     ≪   else if $|T| = 0 \, \wedge \, l \in I^-$≫
14     ≪       forall $p \in U$ ≫
15     ≪           $u = p$≫
16     ≪           $I_l = I_l \cup (\overline{u}, \{l\})$≫
17     ≪case ” $\llbracket pred(b) \rrbracket_{\lightning}$”:≫
18     forall the $i \in 1, ..., |trm(b)|$ do
19         if $trm(b)[i]$ is variable $\wedge \, trm(b)[i] \in trm(h)$ then
20             ≪$\sigma = \sigma \cup \{ \llbracket trm(b)[i] \rrbracket_{\lightning} \mapsto trm(l)[ \llbracket i \rrbracket_{\lightning} ]\}$≫
21     ≪   $T = \{p \in I^+ | \, match(\sigma( \llbracket b \rrbracket_{\lightning} ), p)\}$≫
22     ≪   $U = \{p \in (\mathcal{B} \setminus I)^+ | \, match(\sigma( \llbracket b \rrbracket_{\lightning} ), p)\}$≫
23     ≪   if $l \in I^+$≫
24     ≪       if $\sigma( \llbracket h \rrbracket_{\lightning} ) \in (\mathcal{B} \setminus I)^+$≫
25     ≪           $u = \sigma( \llbracket h \rrbracket_{\lightning} )$≫
26     ≪           $I_l = I_l \cup (u, \{l\})$≫
27     ≪   else≫
28     ≪       if $|T| = 0 \, \wedge \, |U| = 0 \, \wedge \, \sigma( \llbracket h \rrbracket_{\lightning} ) \in (\mathcal{B} \setminus I)^+$≫
29     ≪           $R = \{p \in I^- | \, match(\sigma( \llbracket b \rrbracket_{\lightning} ), p)\}$≫
30     ≪           $u = \sigma( \llbracket h \rrbracket_{\lightning} )$≫
31     ≪           $I_l = I_l \cup (\overline{u}, R)$≫
```

Eager Propagator. Given a compilable sub-program λ_{eager}, we first transform it in the normalized counterpart P, as described in the previous section. Then, Algorithm 1 compiles P as an eager propagator. Recall that a propagator is called to compute the deterministic consequences of one literal at time, which is given by the solver (Cuteri et al. 2020). Thus, in the algorithms the variable l printed in output by the compiler is used to store the literal to propagate in the propagator pseudo-code. Roughly, the propagator matches ground literals to first order literals in the rules of λ_{eager} (as in the grounding) and tries to temporarily build the substitution that instantiates that rule, thus it determines where some propagation occurs. In case of inconsistency a reason is computed and passed to the solver for enabling learning (Cuteri et al. 2020). In this process loop unrolling and dead code elimination is performed using the syntactic information on the rule structure. More formally, The algorithm 1 first prints the declaration of an empty implication list, namely I_l, where the propagator stores all the propagations caused by the fact that a literal to progagate l is added to the interpretation I. Afterward, according to the predicate name of l, the propagator evaluates the propagations depending on the kind of the rule in which the predicate appears. We concentrate on rules of type (2) and (4), since (3) is analogous to (4), and (5) is the same as in (Cuteri et al. 2020) (full details and examples in supplementary material). In turn, Algorithm 2 prints two different switch cases for checking if l appears in h (line 3) or in b (line 17). In both cases the algorithm also prints the code for substituting variables that are in the head of the rule (lines 4-6 and 18-20, respectively). Moreover, true and undefined atoms matching $\sigma(b)$ (lines 7-8 and 21-22, respectively) are stored in two sets, namely T and U. Note that σ is a variable substitution that replaces variables occurrences with constants that are mapped to, e.g., consider $l = a(X)$, if $\sigma(X) = 1$ then $\sigma(l) = a(1)$. Then, the algorithm prints the code specific for the case when l appears in h. In particular, if l is positive then h is true w.r.t. I and the propagator has to find a body that can support it (lines 9-12). Otherwise, if l is negative, then h is false w.r.t. I, then the propagator has to set as false all possible body instantiations (lines 13-16). Instead, if the predicate of l appears in b and l is positive, then the body of the rule is true w.r.t. I and then the propagator has to set h as true (lines 24-26). Otherwise, if l is negative and all other body instantiations are false w.r.t. I, then h is derived as false (lines 28-31). For rules of type (4), i.e., rules of the form $h \leftarrow d, \#sum(\{V : b\}) \geq g$, Algorithm 3 prints the propagator code. In particular, it prints two different switch cases for checking if the predicate of l appears in the head of the rule (line 3) or in the aggregate atom (line 20). In both cases the propagator needs to build a variable substitution σ to simulate the rule instantiation. This substitution maps the variables occurring in h with the constants in l (lines 4-7 and 21-24). If the predicate of l appears in h, the propagator first collects true (line 8) and undefined (line 9) atoms w.r.t. I matching $\sigma(b)$ in T and U, respectively. Next, the propagator computes actual and possible sums as the sum of the weights associated with literal in T and U, respectively (line 10). Then, the propagator has to distinguish two cases. If l is positive (line 11), thus h is true w.r.t. I, then the aggregate must be true in order to support h. This can be done by checking that the maximal possible sum is always greater than or equal to g. Therefore, if there is a literal, say p, whose falsity would make the maximal possible sum less than g (line 14), then p is derived as true (line 15). Otherwise, if l is negative (line 16), thus h is false w.r.t. I, then the aggregate must be false, i.e., the propagator ensures that the actual sum does not exceed g. Therefore, if there is a literal, say p, whose truth would make the actual sum greater than or equal to g (line 18), then p is derived as false (line 19). If the predicate of l appears in the aggregate, then the code of the propagator is as follows. First, the propagator collects in T (respectively, F) all possible positive (respectively, negative) true w.r.t. I literals matching the variable substitution on the head, and in U all undefined atoms matching the variable substitution on the head (lines 25-27). For each literal, say $h_l \in T \cup F \cup U$, the propagator considers three different types of propagations. In particular, if $h_l \in T$ or $h_l \in F$, then the propagator performs similar operations as in lines 8-19 (lines 33-44). Instead, if $h_l \in U$, then the propagator considers two cases. If the actual sum is greater than or equal to the guard (line 46), therefore the body of the rule is true, and thus the head is derived as true (line 47). If the maximal possible sum is less than the guard (line 48) then the head is derived as false (line 49). Note that the propagator stores in I_l also the literals causing the propagation of a literal, i.e., the so-called *reason* (e.g., line 12 of Algorithm 3). In case of inconsistency, the solver retrieves from I_l the reason for propagation used later on for learning.

Lazy Propagator. Lazy propagator instantiates the body of rules by using literals in the model I, and extends I with the heads obtained from such body instantiations. In particular, for each rule $r \in \lambda_{lazy}$, the propagator calls Algorithm 4 with r as parameter. The algorithm first reorders the body of r in such a way that the positive body is evaluated before the negative one (line 3). Subsequently, the algorithm calls Algorithm 5 with the reordered body (B), the rule r, and the index 0 as parameters. Then, for each positive literal b in B, Algorithm 5 prints the code to iterate over positive and true w.r.t. I literals that match $\sigma(b)$ (lines 5-9). For each negative literal n in B, Algorithm 5 prints the code to check if $\sigma(n)$ is negative and true w.r.t. I (lines 29-30). For each aggregate literal, instead, the algorithm recursively calls itself with the list of literals inside the aggregate, the rule r, and the index j of the aggregate as parameters (line 15). In this way, the algorithm prints nested join loops and if statements that evaluate the aggregate conjunction. After that, the algorithm prints the code that computes the sum of the weights (or count) of the elements in the aggregate set (lines 16-21). Once nested join loops and if statements for the aggregate conjunction are closed, the algorithm prints the code that either checks if the aggregate relation is verified or maps the aggregate guard to the aggregate value. Then, the code returns to the end of Algorithm 4, which prints the code that adds $\sigma(H_r)$ to the model I (line 6), since nested join loops and if statements for each body literal are printed.

Algorithm 3 CompileRuleWithSum

```
   Input  : A rule r of type (4), $h \leftarrow d, \#sum(\{V : b\}) \geq g$
   Output: Prints propagator code starting for rule r
1  begin
2      $k = i$ s.t. $trm(b)[i] = V[0], i \in \{1, ..., |trm(b)|\}$
3      ≪case " $[\![pred(h)]\!]_{\lightning}$ "≫
4      $\ll \sigma = \epsilon \gg$
5      forall the $i \in 1, ..., |trm(h)|$ do
6          if $trm(h)[i]$ is variable then
7              $\ll \sigma = \sigma \cup \{ [\![trm(h)[i]]\!]_{\lightning} \mapsto trm(l)[ [\![i]\!]_{\lightning} ]\} \gg$
8      ≪      $T = \{p \in I^+ | \, match(\sigma( [\![b]\!]_{\lightning} ), p)\}$≫
9      ≪      $U = \{p \in (\mathcal{B} \setminus I)^+ | \, match(\sigma( [\![b]\!]_{\lightning} ), p)\}$≫
10     ≪      $act, pos = getSum(T, U, [\![k]\!]_{\lightning} )$≫
11     ≪      if $l \in I^+$≫
12     ≪          $R = \{l\} \cup \{p \in I^- | \, match(\sigma( [\![b]\!]_{\lightning} ), p)\}$≫
13     ≪          forall $p \in U$ ≫
14     ≪              if $act + pos - trm(p)[ [\![k]\!]_{\lightning} ] < \sigma( [\![g]\!]_{\lightning} )$≫
15     ≪                  $I_l = I_l \cup (p, R)$≫
16     ≪      else≫
17     ≪          forall $p \in U$ ≫
18     ≪              if $act + trm(p)[ [\![k]\!]_{\lightning} ] \geq \sigma( [\![g]\!]_{\lightning} )$≫
19     ≪                  $I_l = I_l \cup (\overline{p}, \{l\} \cup T)$≫
20     ≪case " $[\![pred(b)]\!]_{\lightning}$ "≫
21     ≪      $\sigma = \epsilon$≫
22     forall the $i \in 1, ..., |trm(b)|$ do
23         if $trm(b)[i]$ is variable $\wedge$ $trm(b)[i] \in trm(h)$ then
24             $\ll \sigma = \sigma \cup \{ [\![trm(b)[i]]\!]_{\lightning} \mapsto trm(l)[ [\![i]\!]_{\lightning} ]\} \gg$
25     ≪      $T = \{p \in I^+ | \, match(\sigma( [\![h]\!]_{\lightning} ), p)\}$≫
26     ≪      $F = \{p \in I^- | \, match(\sigma( [\![h]\!]_{\lightning} ), p)\}$≫
27     ≪      $U = \{p \in (\mathcal{B} \setminus I)^+ | \, match(\sigma( [\![h]\!]_{\lightning} ), p)\}$≫
28     ≪      forall $h_l \in T \cup U \cup F$ ≫
29     ≪          $\sigma_h = \sigma$≫
30     forall the $i \in 1, ..., |trm(h)|$ do
31         if $trm(h)[i]$ is variable then
32             $\ll \sigma_h = \sigma_h \cup \{ [\![trm(h)[i]]\!]_{\lightning} \mapsto$
               $trm(h_l)[ [\![i]\!]_{\lightning} ]\} \gg$
33     ≪          $UB = \{p \in (\mathcal{B} \setminus I)^+ | \, match(\sigma_h( [\![b]\!]_{\lightning} ), h)\}$≫
34     ≪          $FB = \{p \in I^- | \, match(\sigma_h( [\![b]\!]_{\lightning} ), p)\}$≫
35     ≪          $TB = \{p \in I^+ | \, match(\sigma_h( [\![b]\!]_{\lightning} ), p)\}$≫
36     ≪          $act, pos = getSum(TB, UB, [\![k]\!]_{\lightning} )$≫
37     ≪          if $h_l \in T$≫
38     ≪            forall $p \in UB$ ≫
39     ≪              if $act + pos - trm(p)[ [\![k]\!]_{\lightning} ] < \sigma_h( [\![g]\!]_{\lightning} )$≫
40     ≪                  $I_l = I_l \cup (p, \{h_l\} \cup FB)$≫
41     ≪          else if $h_l \in F$≫
42     ≪            forall $p \in UB$ ≫
43     ≪              if $act + trm(p)[ [\![k]\!]_{\lightning} ] \geq \sigma_h( [\![g]\!]_{\lightning} )$≫
44     ≪                  $I_l = I_l \cup (\overline{p}, \{h_l\} \cup TB)$≫
45     ≪          else≫
46     ≪              if $act \geq \sigma_h( [\![g]\!]_{\lightning} )$≫
47     ≪                  $I_l = I_l \cup (h_l, TB)$≫
48     ≪              else if $act + pos < \sigma_h( [\![g]\!]_{\lightning} )$≫
49     ≪                  $I_l = I_l \cup (\overline{h_l}, FB)$≫
```

Experiments

In the experiments we considered three different settings.

Setting (i). A simple benchmark that we use as a motivating example to show the limits of ground&solve:

Algorithm 4 CompileLazyRule

```
Input  : A rule r
Output: Prints lazy propagator code for a program $P$
1  begin
2      $h = H_r$ // head of r
3      $B = bodyOrdering(B_r)$
4      $\ll \sigma = \epsilon \gg$
5      $PrintNestedJoin(B, r, 0)$
6      $\ll \qquad I = I \cup \{\sigma(h)\} \gg$
7      forall the $i \in 1, ..., |B|$ do
8          $\ll \sigma = \sigma_{\llbracket i,0 \rrbracket_{\lightning}} \gg$
```

Algorithm 5 PrintNestedJoin

```
Input  : A list of literals B, a rule r, an index i
Output: Prints nested join to instantiate B
1  begin
2      forall the $j \in 1, ..., |B|$ do
3          $\ll \sigma_{\llbracket j,i \rrbracket_{\lightning}} = \sigma \gg$
4          if $B[j]$ is a positive literal then
5              $\ll T_{\llbracket j,i \rrbracket_{\lightning}} = \{p \in$
               $I^{+} | \, match(\sigma(\llbracket B[j] \rrbracket_{\lightning}), p)\} \gg$
6              $\ll$for all $t_{\llbracket j,i \rrbracket_{\lightning}} \in T_{\llbracket j,i \rrbracket_{\lightning}} \gg$
7              forall the $k \in 1, ..., |trm(B[i])|$ do
8                  if $trm(B[j,i])[k]$ is variable then
9                      $\ll \sigma = \sigma \cup \{ \llbracket trm(B[j])[k] \rrbracket_{\lightning} \mapsto$
                       $trm(t_{\llbracket j,i \rrbracket_{\lightning}})[\llbracket k \rrbracket_{\lightning}]\} \gg$
10         else if $B[j]$ is an aggregate literal then
11             $B_a = bodyOrdering(conj(B_r^a))$
12             $V = vars(B_r^a)$
13             $a\_s = \emptyset$
14             $a\_v = 0$
15             $PrintNestedJoin(B_a, r, j)$
16             $\ll \qquad$ if $\sigma(\llbracket V \rrbracket_{\lightning}) \notin a\_s \gg$
17             $\ll \qquad a\_s = a\_s \cup \sigma(\llbracket V \rrbracket_{\lightning}) \gg$
18             if $B[j]$ is sum aggregate then
19                 $\ll \qquad a\_v += \sigma(\llbracket V[0] \rrbracket_{\lightning}) \gg$
20             else
21                 $\ll \qquad a\_v += 1 \gg$
22             for $k \in 1, ..., |conj(B_a)|$ do
23                 $\ll \sigma = \sigma_{\llbracket k,j \rrbracket_{\lightning}} \gg$
24             if $B[j]$ is bound relation then
25                 $\ll$if $a\_v \succ \sigma(\llbracket guard(B_r^a) \rrbracket_{\lightning}) \gg$
26             else
27                 $\ll \sigma = \sigma \cup \{ \llbracket guard(B_r^a) \rrbracket_{\lightning} \mapsto a\_v\} \gg$
28         else
29             $\ll t_{\llbracket j,i \rrbracket_{\lightning}} = \sigma(\llbracket B[j] \rrbracket_{\lightning}) \gg$
30             $\ll$if $t_{\llbracket j,i \rrbracket_{\lightning}} \in I \gg$
```

$$
\begin{aligned}
r_1 : &\quad d(1..k) \leftarrow \\
r_2 : &\quad \{a(X) : d(X); b(Y) : d(Y)\} \leftarrow \\
r_3 : &\quad \leftarrow \#count\{X : a(X)\} > Y, b(Y).
\end{aligned}
$$

r_1 is a shortcut for k facts of the form $d(i) \leftarrow (i = 1, \ldots, k)$; and r_2 contains a *choice rule* (Calimeri et al. 2020);

Setting (ii). All benchmarks from ASP competitions (Calimeri et al. 2016) including at least one rule with aggregates that can be compiled under our conditions. This setting is useful to analyze the performance of our compilation-based technique on benchmarks that do not present a grounding bottleneck problem. The selected benchmarks are the following: Abstract Dialectical Framework, Bottle Filling Problem, Connected Maximum Density-Still Life, Crossing Minimization, Incremental Scheduling, Partner Units, Solitaire, Weighted Sequence Problem;

Setting (iii). Three grounding-intensive benchmarks taken from the literature, in particular Component Assignment Problem proposed by Alviano, Dodaro, and Maratea (2018), Dynamic In-Degree Counting and Exponential-Save proposed by Bomanson, Janhunen, and Weinzierl (2019).

Hardware and Software. In all the experiments, the compilation-based approach, reported as WASP-COMP, has been compared with the plain version of WASP (Alviano et al. 2015) v. 169e40d and with the state-of-the-art system CLINGO v. 5.4.0 (Gebser et al. 2016). All the tested systems use GRINGO (included in the binary of CLINGO) as grounder. Moreover, for the benchmarks Dynamic In-Degree Counting and Exponential-Save we considered also the system ALPHA (Weinzierl 2017), 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). For each benchmark, we selected the rules to be compiled by looking at the potential grounding issues. 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.

Results. Results of experiments in the setting (i) and setting (ii) are reported in Table 1 and Table 2, respectively. In the tables 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. Moreover, concerning WASP-COMP, Table 2 reports an additional col-

Table 1: Comparison with WASP and CLINGO on setting (i).

k	WASP-COMP		WASP		CLINGO	
	t	mem	t	mem	t	mem
1000	**0**	0	1.02	59.6	0.66	40.2
2000	**0.1**	18.6	4.36	286.5	3.23	303.5
3000	**0.24**	38.5	9.98	696.3	7.68	709.5
4000	**0.53**	53.7	18.47	1168.1	13.62	1216.8
5000	**0.93**	74.6	28.8	2215.4	22.23	1933.3
6000	**1.19**	109.8	42.47	2807.2	32.73	2871.1
7000	**1.44**	142.3	58.31	3402	42.88	3576.7
8000	**1.96**	152	-	-	-	-
9000	**2.89**	191.6	-	-	-	-
10000	**3.87**	254.7	-	-	-	-
20000	**12.52**	898.5	-	-	-	-
30000	**26.43**	1888.3	-	-	-	-
40000	**58.88**	3319.6	-	-	-	-
50000	-	-	-	-	-	-

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

Table 2: Comparison of the compilation-based approach with WASP and CLINGO on instances of settings (ii) and (iii).

umn (comp. t), representing the compile time in seconds. The latter 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 (i), we observe that CLINGO and WASP cannot solve instances with $k = 8000$, whereas WASP-COMP can efficiently handle instances up to values of $k = 40000$. Concerning the setting (ii), we observe that CLINGO obtains the best performance overall, and it is faster than WASP and WASP-COMP on the non-ground-intensive benchmarks (above the double line in Table 2). The impact of the proposed technique can be seen by comparing WASP-COMP and WASP. In this case, 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, where the lazy propagator gives a clear advantage. 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 case) 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.

Related Work

The grounding bottleneck is widely recognized as one of the major drawbacks of state-of-the-art ASP systems (Cuteri et al. 2020; Bomanson, Janhunen, and Weinzierl 2019). Different alternative approaches have been proposed to overcome it, from extensions of the ASP language, as Constraints Answer Set Programming (CASP) (Ostrowski and Schaub 2012; Balduccini and Lierler 2017, 2013; Aziz, Chu, and Stuckey 2013; Cat et al. 2015; Susman and Lierler 2016), and HEX programs (Eiter, Redl, and Schüller 2016), to lazy grounding (Lefèvre and Nicolas 2009; Palù et al. 2009). Lazy grounding systems perform grounding of the rules during the stable model search, where a rule is instantiated only when its body is satisfied; in this way the grounding is done only for rules used during the search. The state-of-the-art lazy-grounding system ALPHA (Weinzierl 2017) also includes many techniques implemented in traditional ASP systems, such as learning, conflict-based heuristics, restarts, and so on. Bomanson, Janhunen, and Weinzierl (2019) implemented on top of ALPHA a techniques for lazily rewriting aggregates as rules. This latter compared with our approach shows similar performance on the grounding-intensive benchmarks supported by both systems. ALPHA differs from our approach since it avoids the grounding the entire program, where we compile as a propagator a subprogram, and automatically integrate it into the solving component of a CDCL-based solver. Note that our approach builds on optimized systems, whereas ALPHA requires to re-implement all existing techniques. Moreover, we mention a recent work (Lierler and Robbins 2021) that is based on the same principles of (Cuteri et al. 2020), where the grounding of some rules is done using a lazy propagator. However, (Lierler and Robbins 2021) does not yet include aggregates.

Conclusion

The grounding bottleneck is a limiting factor for ASP systems based on the ground&solve architecture. Compilation-based approaches proposed by Cuteri et al. (2020) offer the possibility to mitigate this issue in presence of constraints while keeping the benefits of state-of-the-art approaches. In this paper we extend the compilation-based approaches to handle aggregates, a relevant construct of ASP. Experiments confirm that our approach outperforms state-of-the-art systems on several ground-intensive programs with aggregates. Finally, we mention that benchmarks and executables are publicly-available (Dodaro, Mazzotta, and Ricca 2021a,b).

Acknowledgments

This work was partially supported by the Italian Ministry of Industrial Development (MISE) under project MAP4ID "Multipurpose Analytics Platform 4 Industrial Data", N. F/190138/01-03/X44. During the submission phase the order's name of the authors was not alphabetical in the submission system and it was not possible to fix it before the final submission. The order does not reflect the contribution of the authors to the paper. All authors have contributed equally.

References

Alviano, M.; Dodaro, C.; Leone, N.; and Ricca, F. 2015. Advances in WASP. In *LPNMR*, volume 9345 of *Lecture Notes in Computer Science*, 40–54. Springer.

Alviano, M.; Dodaro, C.; and Maratea, M. 2018. Shared aggregate sets in answer set programming. *Theory Pract. Log. Program.*, 18(3-4): 301–318.

Aziz, R. A.; Chu, G.; and Stuckey, P. J. 2013. Stable model semantics for founded bounds. *Theory Pract. Log. Program.*, 13(4-5): 517–532.

Balduccini, M.; and Lierler, Y. 2013. Integration Schemas for Constraint Answer Set Programming: a Case Study. *Theory Pract. Log. Program.*, 13(4-5-Online-Supplement).

Balduccini, M.; and Lierler, Y. 2017. Constraint answer set solver EZCSP and why integration schemas matter. *Theory Pract. Log. Program.*, 17(4): 462–515.

Bomanson, J.; Janhunen, T.; and Weinzierl, A. 2019. Enhancing Lazy Grounding with Lazy Normalization in Answer-Set Programming. In *AAAI*, 2694–2702. AAAI Press.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM*, 54(12): 92–103.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2016. Answer Set Programming: An Introduction to the Special Issue. *AI Mag.*, 37(3): 5–6.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 Input Language Format. *Theory Pract. Log. Program.*, 20(2): 294–309.

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

Cat, B. D.; Denecker, M.; Bruynooghe, M.; and Stuckey, P. J. 2015. Lazy Model Expansion: Interleaving Grounding with Search. *J. Artif. Intell. Res.*, 52: 235–286.

Clark, K. L. 1977. Negation as Failure. In *Logic and Data Bases*, Advances in Data Base Theory, 293–322. New York: Plemum Press.

Cuteri, B.; Dodaro, C.; Ricca, F.; and Schüller, P. 2019. Partial Compilation of ASP Programs. *Theory Pract. Log. Program.*, 19(5-6): 857–873.

Cuteri, B.; Dodaro, C.; Ricca, F.; and Schüller, P. 2020. Overcoming the Grounding Bottleneck Due to Constraints in ASP Solving: Constraints Become Propagators. In *IJCAI*, 1688–1694. ijcai.org.

Dodaro, C.; Mazzotta, G.; and Ricca, F. 2021a. Benchmarks: https://doi.org/10.5281/zenodo.5752555.

Dodaro, C.; Mazzotta, G.; and Ricca, F. 2021b. Executables: https://www.mat.unical.it/~dodaro/research/compilation/.

Dodaro, C.; and Ricca, F. 2020. The External Interface for Extending WASP. *Theory Pract. Log. Program.*, 20(2): 225–248.

Eiter, T.; Redl, C.; and Schüller, P. 2016. Problem Solving Using the HEX Family. In *Computational Models of Rationality*, 150–174. College Publications.

Erdem, E.; and Öztok, U. 2015. Generating explanations for biomedical queries. *Theory Pract. Log. Program.*, 15(1): 35–78.

Faber, W.; Pfeifer, G.; and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1): 278–298.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory Solving Made Easy with Clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASICS*, 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Gebser, M.; Leone, N.; Maratea, M.; Perri, S.; Ricca, F.; and Schaub, T. 2018. Evaluation Techniques and Systems for Answer Set Programming: a Survey. In *IJCAI*, 5450–5456. ijcai.org.

Gebser, M.; Maratea, M.; and Ricca, F. 2017. The Sixth Answer Set Programming Competition. *J. Artif. Intell. Res.*, 60: 41–95.

Gelfond, M.; and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Gener. Comput.*, 9(3/4): 365–386.

Kaufmann, B.; Leone, N.; Perri, S.; and Schaub, T. 2016. Grounding and Solving in Answer Set Programming. *AI Mag.*, 37(3): 25–32.

Lefèvre, C.; and Nicolas, P. 2009. The First Version of a New ASP Solver : ASPeRiX. In *LPNMR*, volume 5753 of *Lecture Notes in Computer Science*, 522–527. Springer.

Lierler, Y.; and Robbins, J. 2021. DualGrounder: Lazy Instantiation via Clingo Multi-shot Framework. In *JELIA*, volume 12678 of *Lecture Notes in Computer Science*, 435–441. Springer.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

Ostrowski, M.; and Schaub, T. 2012. ASP modulo CSP: The clingcon system. *Theory Pract. Log. Program.*, 12(4-5): 485–503.

Palù, A. D.; Dovier, A.; Pontelli, E.; and Rossi, G. 2009. GASP: Answer Set Programming with Lazy Grounding. *Fundam. Informaticae*, 96(3): 297–322.

Susman, B.; and Lierler, Y. 2016. SMT-Based Constraint Answer Set Solver EZSMT (System Description). In *ICLP (Technical Communications)*, volume 52 of *OASICS*, 1:1–1:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Weinzierl, A. 2017. Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, 191–204. Springer.