ECAI 2023
K. Gal et al. (Eds.)
© 2023 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA230316

Compilation of Tight ASP Programs

Carmine Dodaro , Giuseppe Mazzotta and Francesco Ricca

University of Calabria
ORCiD ID: Carmine Dodaro https://orcid.org/0000-0002-5617-5286, Giuseppe Mazzotta https://orcid.org/0000-0003-0125-0477, Francesco Ricca https://orcid.org/0000-0001-8218-3178

Abstract. Answer Set Programming (ASP) is a well-known AI formalism. Traditional ASP systems, that follow the "ground&solve" approach, are intrinsically limited by the so-called *grounding bottleneck*. Basically, the grounding step (i.e., variable-elimination) can be computationally expensive, and even unfeasible in several cases of practical interest. Recent work demonstrated that the grounding bottleneck can be partially overcome by compiling in external propagators subprograms acting as constraints. In this paper a novel compilation technique is presented that can be applied to tight normal programs; thus, the class of ASP programs that can be compiled is extended *beyond constraints*. The approach is implemented in the new system PROASP. PROASP skips entirely the grounding phase and performs solving by injecting custom propagators in GLUCOSE. An experiment, conducted on grounding-intensive ASP benchmarks, shows that PROASP is capable of solving instances that are out of reach for state-of-the-art ASP systems.

1 Introduction

Answer Set Programming (ASP) [9, 30] is a well-known AI formalism. Indeed, ASP has found extensive applications in AI [22] throughout the years that belong to various sub-fields, such as game theory [4], natural language processing and understanding [17, 39, 42], robotics [24], planning [43], and scheduling [19], among others [22]. The applicability of ASP is due to the combination of two features: a comparatively expressive language that can model hard problems [18] and efficient implementations [28]. Notably, the enhancement of ASP systems is a compelling research area as system performance plays a crucial role in the advancement of applications [28].

State-of-the-art ASP implementations follow the ground&solve approach [33], where the input program is first "grounded" to compute a variable-free equivalent propositional program, that is then "solved" by employing a CDCL-like algorithm [37] that computes the solutions (i.e., the answer sets of the input program). ground&solve ASP systems, such as CLINGO [26] and DLV [1], are intrinsically subject to the *grounding bottleneck*. Basically, the grounding step can be computationally expensive and even unfeasible in several cases of practical interest [10, 40].

Numerous efforts have been made to address the grounding bottleneck issue [28]. These include hybrid formalisms [7, 26, 40, 44], and *lazy grounding* architectures [8, 34, 36, 41, 45]. Rather than addressing the problem of ASP systems, hybrid approaches circumvent it by both expanding the language with novel constructs and connecting ASP systems with external sources of computation. On the other hand, lazy grounding techniques perform grounding during search, with the aim to prevent computing unnecessary instantiations. Despite showing promising initial results and continuously improving efficiency, the performance of lazy grounding techniques is still not on par with state-of-the-art systems [46]. Recent work demonstrated that the grounding bottleneck can be overcome by compiling subprograms in external propagators, that simulate the presence of their ground counterpart during the solving phase [15]. However, these approaches can only compile subprograms acting as constraints [16, 38]. Whether it is possible to devise a compiler for rules "generating" answer sets was left as an open question.

In this paper, a novel compilation technique and a grounding-less ASP solving architecture are proposed that target tight normal programs [23]. The resulting approach extends the class of ASP programs that can be compiled in a propagator *beyond constraints*; thus, the first positive answer to the above question is provided.

Tight programs have been considered a relevant class of programs since 1994, when Francois Fages [25] proved that stable models (i.e., the answer sets) of programs without positive loops (i.e., tight programs) can be characterized as the models of a propositional SAT formula, called "completion" [13]. As a matter of fact, tight programs have been playing a central role in the development of modern ASP systems [2, 31, 35, 32]; and, the techniques for evaluating tight programs are the core of all modern ASP solvers [28].

The compilation of tight programs can be obtained as follows. In a nutshell, the non-ground (i.e., with variables) input program is first pre-processed by applying a rewriting. The aim of this pre-processing is twofold, on the one hand, it adds rules that mimic the program completion [13]; and, on the other hand, it simplifies the implementation, since it produces rules of three kinds. (Details in Section 4). Then, the compiler processes the rewritten program to generate specific code for two tasks: (i) generation of (the useful part of) the Herbrand base, that is used to determine a set of propositional atoms that is sufficient to search for the answer sets of the program in input; (ii) synthesis of the solver component, where, basically, the code of a CDCL solver is enriched with custom procedures simulating the presence of ground rules. The resulting code is assembled by following an architecture where, first, the facts, modeling an input instance, are used to generate the Herbrand base (roughly, the propositional variables are computed); and, then, the CDCL solver search for answer sets by calling the propagators instead of performing the traditional inference on ground rules.

The novel approach is implemented in a new ASP system called PROASP. In PROASP ASP solving is implemented by injecting propagators in the GLUCOSE SAT solver [5]. Notably, the PROASP compiler produces a solver specific for the non-ground program in input that needs no grounder.

The performance of PROASP is evaluated by conducting a thorough analysis and comparison with state-of-the-art alternatives on ASP benchmarks that are known to be grounding-intensive. The primary objective is to evaluate the effectiveness of PROASP by highlighting its strengths and weaknesses. Furthermore, the study aims to measure the capability of PROASP to address the challenges posed by non-groundable ASP programs. In summary, the experiment demonstrates that PROASP is capable of solving efficiently instances that are out-or-reach for both traditional ASP systems and compilation approaches that are limited to constraints.

2 Answer Set Programming

ASP Syntax. *Terms* are either variables (i.e. strings starting with an uppercase letter) or constants (i.e. integer numbers or strings starting with a lowercase letter). An *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. Given an atom $a = p(t_1, \ldots, t_n)$, $\texttt{terms}(a)$ denotes the list of terms $t_1, \ldots, t_n$, and n is the *arity* of a. A *literal* is either an atom, a, or its negation $not\ a$, where not represents negation as failure. A literal is *ground* if it does not contain variables. A literal is *positive* (resp. *negative*) if it is of the form a (resp. $not\ a$). Given a literal l, the *complement* of l, denoted by $\bar{l}$, is $not\ a$ if $l = a$, or a if $l = not\ a$. Given a set of literals S, S^+, and S^- denote the sets of positive and negative literals in S, respectively. The complement of S, denoted by $\neg S$, is the set of literals $\{\bar{l} : l \in S\}$. A *rule* is an expression of the form $a \leftarrow l_1, \ldots, l_n$, where a is an atom called *head*, and $l_1, \ldots, l_n$ is a conjunction of literals called *body*. Given a rule r, H_r denotes either a or the set $\{a\}$, B_r denotes the set of literals $l_1, \ldots, l_n$. If H_r is empty, r is called *constraint*; if B_r is empty, r is called *fact*; otherwise, r is *normal*. A *program* is a finite set of rules.

Given a program P, the *Herbrand Universe* U_P of P, is the set of constants (occurring) in P; and, the *Herbrand Base* B_P is the set of ground atoms obtained from predicates of P and constants in U_P. Let r be a rule, its ground instantiation is obtained by substituting all variables in r with constants in U_P. The ground instantiation $ground(P)$ (or *grounding*) of the program P, is the union of the ground instantiations of rules in P. Given an ASP expression ϵ (i.e. term, atom, rule, etc.), $\texttt{vars}(\epsilon)$ denotes the list of variables occurring in ϵ, and $pred(\epsilon)$ denotes the set of predicates occurring in ϵ.

The *dependency graph* of program P is the graph $G_P = \langle V', E' \rangle$, where V' is the set of predicates appearing in P and E' contains an edge (p_1, p_2) if there exists a rule $r \in P$ such that p_2 occurs in H_r and p_1 occurs in B_r^+. The strongly connected components (SCC)s of G_P induce a partition of predicates of P, say $C_1, \ldots, C_n$, one for each SCC of G_P. The *ground dependency graph*, is the directed graph $\langle V, E \rangle$, where V is the set of atoms appearing in $ground(P)$ and E contains an edge (p_1, p_2) if there exists a rule $r \in ground(P)$ such that p_2 appears in H_r and p_1 appears in B_r^+. The program P is *tight* if its ground dependency graph is acyclic.

ASP Semantics. Given a program P, an interpretation I is a set of literals with atoms in B_P. 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. A conjunction of literals is true w.r.t I if the literals are true w.r.t. I, and false if at least one literal is false w.r.t. I. An interpretation I is *total* if for each atom $a \in B_P$, $a \in I$ or $not\ a \in I$, 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

Algorithm 1: CDCL for ASP solving

```
Input : An ASP Program $\Pi$
1  begin
2      $M$ := $\emptyset$
3      Loop
4          EagerPropagation($\Pi$, $M$);
5          if $M$ is inconsistent then
6              $\Pi$ := $\Pi \cup$ Learning($\Pi$, $M$);
7              $M$ := RestoreConsistency($\Pi$, $M$);
8              if $M$ is inconstent then return $\bot$ ;
9          if $M$ is total then
10             $C$ := LazyPropagation($\Pi$, $M$);
11             if $C \neq \emptyset$ then $\Pi$ := $\Pi \cup C$;
12             else return $M$;
13         else $M$ := PickBranchingLiteral($\Pi$) ;
```

P if for each rule $r \in ground(P)$, the head of r is true whenever the body of r is true. Given a program P and an interpretation I, the (Gelfond-Lifschitz) reduct [30] of P, denoted by P^I, is defined as the set of rules obtained from P by deleting those rules whose body is false w.r.t I and removing all negative literals that are true w.r.t. I from the body of remaining rules. Let I be a model for P, I is also an answer set (or *stable model*) for P if there is no $I' \subset I$ such that I' is a model for P^I. The program P is *coherent* if it admits at least one answer set, *incoherent* otherwise.

3 Compilation-based ASP solving

In this section, we recall the fundamental principles utilized by compilation-based approaches [15, 16, 38]. These approaches transform rules into a collection of specialized procedures called *propagators*. These propagators, once injected into a CDCL algorithm, are capable of mimicking rule inferences during the solving process. Compilation-based techniques in the literature [15, 16, 38] take a program P as input and split it into three subprograms: P', P_{eager}, and P_{lazy}. P' is processed in the traditional way, i.e., P' is first grounded in $\Pi = ground(P')$ and, then, fed in input to the CDCL algorithm shown in Algorithm 1. Meanwhile, P_{eager} and P_{lazy} are compiled into specialized propagators that are integrated into the solver as *eager* (line 4) or *lazy* propagators (line 10), respectively. In particular, the CDCL algorithm builds an answer set of Π by incrementally extending an empty interpretation M. The first step is eager propagation, corresponding to unit propagation in SAT solvers [37], which amounts to deriving the deterministic consequences of M that are implied by the program in input. Thus, the inferences due to both Π and P_{eager} are computed. Essentially, the standard approach is applied for ground rules in Π; whereas, the eager propagators added in the compilation-based approach simulate the inferences due to P_{eager} without grounding it in advance. After eager propagation, M can be either inconsistent or consistent. If it is inconsistent, the algorithm analyzes the inconsistency, adds a derived constraint to Π, and restores consistency by backtracking. If consistency cannot be restored, the algorithm returns $\bot$, indicating that Π has no answer sets. If M is consistent, the algorithm extends M with an unassigned literal selected using a heuristic and applies eager propagation again. If all atoms in Π are assigned w.r.t. M, then M is an answer set candidate. Indeed, lazy propagators are then applied to verify whether M is consistent w.r.t. P_{lazy}, and if so, the algorithm returns M. If not, lazy propagators return a set of violated rules that are added to Π, and the main loop is repeated again.

The effectiveness of compilation-based techniques relies heavily on how the program P is divided into subprograms, as noted in [14].

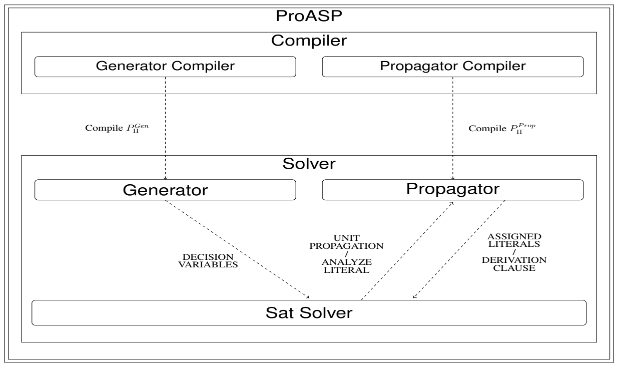


Figure 1: PROASP Architecture

Existing approaches [15, 16, 38] provide some syntactical requirements that have to be satisfied by the rules in both P_{eager}, and P_{lazy}. Basically, only programs behaving as constraints could be compiled, and also the atoms in the heads of rules in P^{eager} or P^{lazy} cannot appear in the head of any rule in P'. This might be restrictive.

4 The PROASP system

In this Section, we present a novel approach to the compilation of ASP programs that overcomes one of the main limitation of previous compilation-based approaches for ASP, namely the fact that they can compile only programs behaving as constraints. The new approach is applied to tight normal programs and has been implemented in the PROASP system.

Tight programs can be transformed to SAT formulas in a well-known way [23]; thus, they can be evaluated by resorting to an existing CDCL implementation for SAT [32, 35]. Actually, in almost all modern ASP solvers, the ground rules of an input program are converted in their representation in clauses that is processed together with their completion [13]. The completion is a set of clauses modeling that each true atom is *supported*, i.e., there exists a rule (at least one) having the body true in the model. For tight programs, this condition is sufficient to obtain correspondence between the answer sets of the program and the satisfying assignments of the so-built SAT formula. Following this pattern, we devise a novel grounding-less architecture for evaluating tight programs, illustrated in Figure 1.

PROASP has two main components, referred to as `Compiler` and `Solver`. The `Compiler` takes as input a non-ground ASP program P, and generates the code of an ASP solver, referred to as `Solver`, that can evaluate P. `Solver` is made of a CDCL SAT solver augmented with two additional modules: the `Generator` and the `Propagator`. As usual in ASP, the instance of a problem is assumed to be provided w.l.o.g. as a set of facts [9]. Given the instance in input, `Generator` generates (a subset of) the Herbrand base B_P, which is used to fill the CDCL solver data structures with propositional atoms (corresponding to the variables in SAT); whereas, the `Propagator` module contains the propagators that, very roughly, simulate the grounding of P (as it would have been properly transformed into a SAT formula). `Compiler` and `Solver` are detailed in the following subsections.

Comparing PROASP with existing compilation-based ASP solvers (see Section 3), we stress that the program in input does not have to be split into sub-programs, and no grounder is needed.

4.1 PROASP `Compiler`

Let Π denote the ASP program in input. We now introduce a rewriting of Π that ensures that (i) rules are treated uniformly, and (ii) the propagations ensuring atoms are supported are modeled by constraints. Without loss of generality, we assume there are no two different atoms sharing the same predicate name having different arity.

For a predicate p, let $\Pi_p = \{r_1, \ldots, r_n\}$ be the set of rules of Π s.t. the predicate of their head atom is p, then if $n > 1$, $unique(\Pi_p)$ is the program:

$$\begin{array}{ll} sup_r(\texttt{terms}(H_r)) \leftarrow B_r & \forall r \in \Pi_p \\ \leftarrow sup_r(\texttt{terms}(H_r)),\ not\ H_r & \forall r \in \Pi_p \\ \leftarrow p(\vec{x}),\ not\ sup_{r_1}(\vec{x}), \ldots, not\ sup_{r_n}(\vec{x}) & \end{array}$$

where $\vec{x}$ a list of m variables, and m is the arity of p; otherwise (i.e., $n = 1$) $unique(\Pi_p) = \Pi_p$. Then, P_Π is defined as the union of $unique(\Pi_p)$ for all p occurring in Π. Note that each predicate name occurs in at most one rule head in P_Π.

Example 1. *Let Π_{ex} be the following (sub)program:*

$$\begin{array}{ll} a(1, Z) & \leftarrow b(Y),\ c(Y, Z) \\ a(X, Z) & \leftarrow d(X, Y),\ c(Y, Z),\ not\ e(Z) \end{array}$$

In order to simplify the presentation, predicates $b,\ c,\ d,$ *and* e *do not appear in the head of any rule, so let us focus on predicate* a*. In this case* $\Pi_a = \{r_1, r_2\}$ *is rewritten as follows:*

$$\begin{array}{l} sup_{r_1}(1, Z) \leftarrow b(Y),\ c(Y, Z) \\ sup_{r_2}(X, Z) \leftarrow d(X, Y),\ c(Y, Z),\ not\ e(Z) \\ \leftarrow sup_{r_1}(1, Z),\ not\ a(1, Z) \\ \leftarrow sup_{r_2}(X, Z),\ not\ a(X, Z) \\ \leftarrow a(X_1, X_2),\ not\ sup_{r_1}(X_1, X_2),\ not\ sup_{r_2}(X_1, X_2) \end{array}$$

□

We define P_Π^{Prop} as the program obtained from P_Π by rewriting each normal rule $r \in P_\Pi$, as follows:

$$\begin{array}{lll} H_r & \leftarrow aux_r(\texttt{vars}(B_r^+)) & \\ & \leftarrow aux_r(\texttt{vars}(B_r^+)), \bar{l} & \forall l \in B_r \\ & \leftarrow l_1,\ \ldots,\ l_n,\ not\ aux_r(\texttt{vars}(B_r^+)) & \end{array}$$

where $B_r = l_1, \ldots,\ l_n$. Basically, P_Π^{Prop} models a form of completion [13] by means of additional constraints.

Example 2. *Let us consider the rule* $sup_{r_1}(1, Z) \leftarrow b(Y),\ c(Y, Z)$ *from Example 1. It is rewritten as follows:*

$$\begin{array}{ll} sup_{r_1}(1, Z) & \leftarrow aux_r(Y, Z) \\ & \leftarrow aux_r(Y, Z),\ not\ b(Y) \\ & \leftarrow aux_r(Y, Z),\ not\ c(Y, Z) \\ & \leftarrow b(Y),\ c(Y, Z),\ not\ aux_r(Y, Z) \end{array}$$

□

Program P_Π^{Prop} is used to build the Propagator module. We now define program P_Π^{Gen} that is used to build the Generator module.

Given a rule $r \in \Pi$, let $p = pred(H_r)$, then $supRule(r)$ is the rule $sup_r(\texttt{terms}(H_r)) \leftarrow B_r$ if $|\Pi_p| > 1$, otherwise $supRule(r)$ is r. Moreover, for each rule $r' \in P_\Pi$ that is not a constraint, $auxRule(r')$ is the rule $H_{r'} \leftarrow aux_{r'}(\texttt{vars}(B_{r'}^+))$. The program P_Π^{Gen} is obtained by rewriting each rule $r \in \Pi$ as follows:

$$\begin{array}{ll} B_{r''} \leftarrow B_{r'} & \\ H_{r''} \leftarrow B_{r''} & \\ H_r \leftarrow H_{r'} & if\ r \neq r' \end{array}$$

where $r' = supRule(r)$ and $r'' = auxRule(r')$

Algorithm 2: Sub-procedure generated for the rule $aux_{r_2}(X,Y,Z) \leftarrow d(X,Y),\ c(Y,Z),\ not\ e(Z)$

```
Input  : A set of atoms $\mathcal{B}$, a set of atoms $F \subseteq \mathcal{B}$
Output: A set of ground atoms matching $aux_{r_2}(X,Y,Z)$
1 begin
2     $atoms := \emptyset$;
3     forall $l_1 \in \{a \in \mathcal{B} : match(a, d(\_,\_))\}$ do
4         $x := l_1[0]$;    $y := l_1[1]$;
5         forall $l_2 \in \{a \in \mathcal{B} : match(a, c(y,\_)\}$ do
6             $z := l_2[1]$;    $l_3 := not\ e(z)$;
7             if $\overline{l_3} \notin F$ then $atoms := atoms \cup \{aux_{r_2}(x,y,z)\}$ ;
8     $\mathcal{B} := \mathcal{B} \cup atoms$;
```

Example 3. *Let us consider the program Π_{ex} from Example 1, then P_{ex}^{Prop} is the following:*

$$
\begin{array}{l}
\leftarrow sup_{r_1}(1,Z),\ not\ a(1,Z)\\
\leftarrow sup_{r_2}(X,Z),\ not\ a(X,Z)\\
\leftarrow a(X_1,X_2),\ not\ sup_{r_1}(X_1,X_2),\ not\ sup_{r_2}(X_1,X_2)\\
sup_{r_1}(1,Z) \leftarrow aux_{r_1}(Y,Z)
\end{array}
$$

$$
\begin{array}{l}
\leftarrow aux_{r_1}(Y,Z),\ not\ b(Y)\\
\leftarrow aux_{r_1}(Y,Z),\ not\ c(Y,Z)\\
\leftarrow b(Y),\ c(Y,Z),\ not\ aux_{r_1}(Y,Z)\\
sup_{r_2}(Y,Z) \leftarrow aux_{r_2}(X,Y,Z)\\
\leftarrow aux_{r_2}(X,Y,Z),\ not\ d(X,Y)\\
\leftarrow aux_{r_2}(X,Y,Z),\ not\ c(Y,Z)\\
\leftarrow aux_{r_2}(X,Y,Z),\ e(Z)\\
\leftarrow d(X,Y),\ c(Y,Z),\ not\ e(Z),\ not\ aux_{r_2}(X,Y,Z)
\end{array}
$$

and P_{ex}^{Gen} is the following:

$$
\begin{array}{l}
aux_{r_1}(Y,Z) \leftarrow b(Y),\ c(Y,Z)\\
aux_{r_2}(X,Y,Z) \leftarrow d(X,Y),\ c(Y,Z),\ not\ e(Z)\\
sup_{r_1}(1,Z) \leftarrow aux_{r_1}(Y,Z)\\
sup_{r_2}(Y,Z) \leftarrow aux_{r_2}(X,Y,Z)\\
a(X_1,X_2) \leftarrow sup_{r_1}(X_1,X_2)\\
a(X_1,X_2) \leftarrow sup_{r_2}(X_1,X_2)
\end{array}
$$

□

Once programs P_{Π}^{Prop} and P_{Π}^{Gen} are computed, they undergo a compilation process. On the one hand, P_{Π}^{Prop} is compiled by following the same algorithms described in [38], and this results in the `Propagator` module of our architecture. On the other hand, P_{Π}^{Gen} is also compiled but in a different way. The behavior is similar to a deductive database system that computes all variable substitutions starting from input facts. To this end, the `Compiler` computes the strongly connected components (SCCs) of P_{Π}^{Gen}, and process subprograms in the topological order induced by the SCCs. In particular, for each component C_i, the subprogram $\lambda_i = \{r \in P_{\Pi}^{Gen} : pred(H_r) \in C_i\}$ is compiled as a procedure $proc\_\lambda_i(\cdot)$ that generates all ground atoms of the form $p(\cdot)$ such that p occurs in C_i, and $p(\cdot)$ appears in at least one rule head.

To clarify the idea of the procedures generated by the compiler, we present in the following an example of compilation. In particular, the sub-procedure generated for the rule $aux_{r_2}(X,Y,Z) \leftarrow d(X,Y),\ c(Y,Z),\ not\ e(Z)$, referred to as r in the following, is reported in pseudo-code as Algorithm 2. Specifically, Algorithm 2 first creates an empty set of ground atoms that will be used to collect all ground atoms of the form $aux_{r_2}(\cdot,\cdot,\cdot)$. Then, Algorithm 2 has a different nested block for each literal l_i ($i = 1..|B_r|$) occurring in B_r, where the block is a **for loop** for each positive literal and an **if** statement for each negative literal. Indeed, Algorithm 2 contains three nested blocks, i.e. two **for loops** and one **if** statement, generated for $d(X,Y)$, $c(Y,Z)$, and $not\ e(Z)$, respectively. Then, when the scope of the **if** statement is reached, the set of literals l_1, l_2, and l_3 represents a possible instantiation of r. Thus, $aux_{r_2}(x,y,z)$ is added to the minimal subset of atoms needed for stable model computation. For instance, suppose that $\mathcal{B} = \{d(1,2), c(2,4), e(3)\}$, then we might have $l_1 = d(1,2)$, $l_2 = c(2,4)$, $l_3 = not\ e(4)$. Therefore, in this case, we can derive $aux_{r_2}(1,2,4)$.

Hence, the `Generator` module is assembled as a sequence of sub-procedures as the ones described before, according to the SCCs of the program P_{Π}^{Gen}. The set of all atoms computed by this module becomes the set of variables given as input to the SAT solver. After the production of the `Generator` module, the `Compiler` creates the `Propagator` module. Specifically, each rule $r \in P_{\Pi}^{Prop}$ is compiled into a propagator sub-procedure, called $Prop_r$ which evaluates possible propagations of r. In particular, two different propagation strategies have been implemented, respectively for constraints and normal rules. We refer the reader to [38] for the details about the generation of the sub-procedures. Here we show an example of the generated sub-procedures, reported as Algorithms 3 and 4. In particular, Algorithm 3 is a propagator for the constraint $c : \leftarrow aux_{r_2}(X,Y,Z),\ not\ d(X,Y)$, whereas Algorithm 4 is a propagator for the rule $r : sup_{r_2}(X,Z) \leftarrow aux_{r_2}(X,Y,Z)$. Algorithm 3 evaluates propagations of ground instantiations of c containing a literal l that has been added to the candidate stable model M. Thus, if the predicate of l is aux_{r_2} and l is a positive literal

Algorithm 3: Propag. for $\leftarrow aux_{r_2}(X,Y,Z),\ not\ d(X,Y)$

```
Input  : A literal $l$, an interpretation $M$
Output: A set of literals $M_l$
1 begin
2     $M_l := \emptyset$;
3     if $pred(l) = $ “$aux_{r_2}$” and $l \in M^+$ then
4         $x := l[0]$;    $y := l[1]$;    $z := l[2]$;    $l_2 := not\ d(x,y)$;
5         if $\overline{l_2} \notin M^+$ then $M_l := M_l \cup \{\overline{l_2}\}$ ;
6     else
7         if $pred(l) = $ “$d$” and $l \in M^-$ then
8             $x := l[0]$;    $y := l[1]$;
9             $T := \{a \in \mathcal{B} : match(a, aux_{r_2}(x,y,\cdot)\}$
10            forall $l_1 \in T$ do $M_l := M_l \cup \{\overline{l_1}\}$;
11    return $M_l$
```

Algorithm 4: Propag. for $sup_{r_2}(X,Z) \leftarrow aux_{r_2}(X,Y,Z)$

```
Input  : A literal $l$, an interpretation $M$
Output: A set of literals $M_l$
1 begin
2     $M_l := \emptyset$;
3     if $pred(l) = $ “$sup_{r_2}$” then
4         $x := l[0]$;    $z := l[1]$;
5         $T := \{a \in \mathcal{B} : match(a, aux_{r_2}(x,\cdot,z)\}$;
6         if $l \in M^+$ then
7             if $T \cap M = \emptyset$ and $|T| = 1$ then $M_l := M_l \cup T$ ;
8         else if $l \in M^-$ then $M_l := M_l \cup \neg T$ ;
9     else if $pred(l) = $ “$aux_{r_2}$” then
10        $x := [0]$;    $y := l[1]$;    $z := l[2]$;
11        if $l \in M^+$ then $M_l := M_l \cup \{sup_{r_2}(x,z)\}$ ;
12        else if $l \in M^-$ then
13            $T := \{a \in \mathcal{B} : match(a, aux_{r_2}(x,\cdot,z)\}$;
14            if $\neg T \subseteq M^-$ then $M_l := M_l \cup \{not\ sup_{r_2}(x,z)\}$ ;
15    return $M_l$
```

Table 1: Comparison of the different solvers on grounding-intensive benchmarks.

Benchmark	#	ProASP			WaspProp			Wasp			Clingo			Alpha		
		SO	TO	MO	SO	TO	MO	SO	TO	MO	SO	TO	MO	SO	TO	MO
(NPRC)	110	**110**	0	0	**110**	0	0	**110**	0	0	**110**	0	0	**110**	0	0
(P)	50	**23**	27	0	12	38	0	0	50	0	0	48	2	0	45	5
(QG)	100	**20**	0	80	15	0	85	12	3	85	5	0	95	5	40	55
(SM)	314	**230**	84	0	225	89	0	197	117	0	213	4	97	28	286	0
(WAT)	62	36	14	12	**50**	0	12	**50**	0	12	**50**	0	12	0	62	0

then $aux_{r_2}(X,Y,Z)$ is substituted by l, and $d(X,Y)$ is propagated. Otherwise, if the predicate of l is d and l is a negative literal then $not\ d(X,Y)$ is replaced by l, and all literals of the form $not\ aux_{r_2}(X,Y,\cdot)$ are propagated. Analogously, Algorithm 4 evaluates propagations of ground instantiations of r containing l either in H_r or B_r. Thus, if the predicate of l is sup_{r_2} then $sup_{r_2}(X,Z)$ is matched to l by applying a proper variable substitution. If l is a positive literal, then it means that there exists an instantiation of r, r', such that $H_{r'}$ is true w.r.t. M. Thus, the propagator ensures that if only one literal is left such that $l' = aux_{r_2}(X,\cdot,Z)$ this is propagated to true. Instead, if l is a negative literal, then there exists an instantiation of r, r', such that $H_{r'}$ is false w.r.t. M. Thus, the propagator propagates all literals of the form $aux_{r_2}(X,\cdot,Z)$ as false (i.e., $not\ aux_{r_2}(X,\cdot,Z)$ is added to M). On the other hand, if the predicate of l is aux_{r_2} then $aux_{r_2}(X,Y,Z)$ is matched to l by applying a proper variable substitution. Thus, if l is a positive literal then there exists an instantiation of r, r', where $B_{r'}$ is true w.r.t. M. Thus, $H_{r'}$ is propagated as true. Otherwise, if l is a negative literal and $sup_{r_2}(X,Z)$ is true w.r.t. M, the propagator ensures that if there is only one literal $l' = aux_{r_2}(X,\cdot,Z)$, then it is propagated to true.

4.2 ProASP `Solver`

The `Solver` for an input program Π is obtained by compiling (e.g., using g++), the SAT solver, the Generator, and the Propagator together in the same executable. The `Solver` reads as input an instance (set of facts) F, and, as the first step, it calls the Generator. The resulting set of ground atoms, say $\mathcal{B}$, is used to initialize the SAT solver data structures. A variable is added for each atom in $\mathcal{B}$, and a unit clause is added for each fact in F. Then, to limit iterations during propagation, variables are mapped to the propagators they can affect, similarly to *watched literals* in standard SAT solvers. For example, the watched literals for the propagator reported in Algorithm 3 are the literals of the form $aux_{r_2}(\cdot,\cdot,\cdot)$ and $not\ d(\cdot,\cdot)$. Then, the SAT solver is started, and the CDCL takes place as usual, alternating propagation and decision (cfr. Algorithm: 1). For example, consider the constraint $\leftarrow aux_{r_2}(X,Y,Z),\ not\ d(X,Y)$ and its propagator reported in Algorithm 3, and suppose that $aux_{r_2}(1,2,3)$ is added to M, then $d(1,2)$ is derived (and later added to M). In case M becomes inconsistent, the Learning procedure is invoked. During this process the propagator might be asked to reconstruct the clause that implied M, to be used as usual for learning a conflict clause [37]. In the previous example, suppose that $not\ d(1,2)$ and $d(1,2)$ are both in M, the set of literals causing their propagation, i.e., $aux_{r_2}(1,2,3)$ for $d(1,2)$ is returned by the propagator. (For a detailed description of propagators we refer the reader to [38].)

5 Experiments

In this section, we present the experiments evaluating the performance of the ProASP system on grounding-intensive benchmarks.

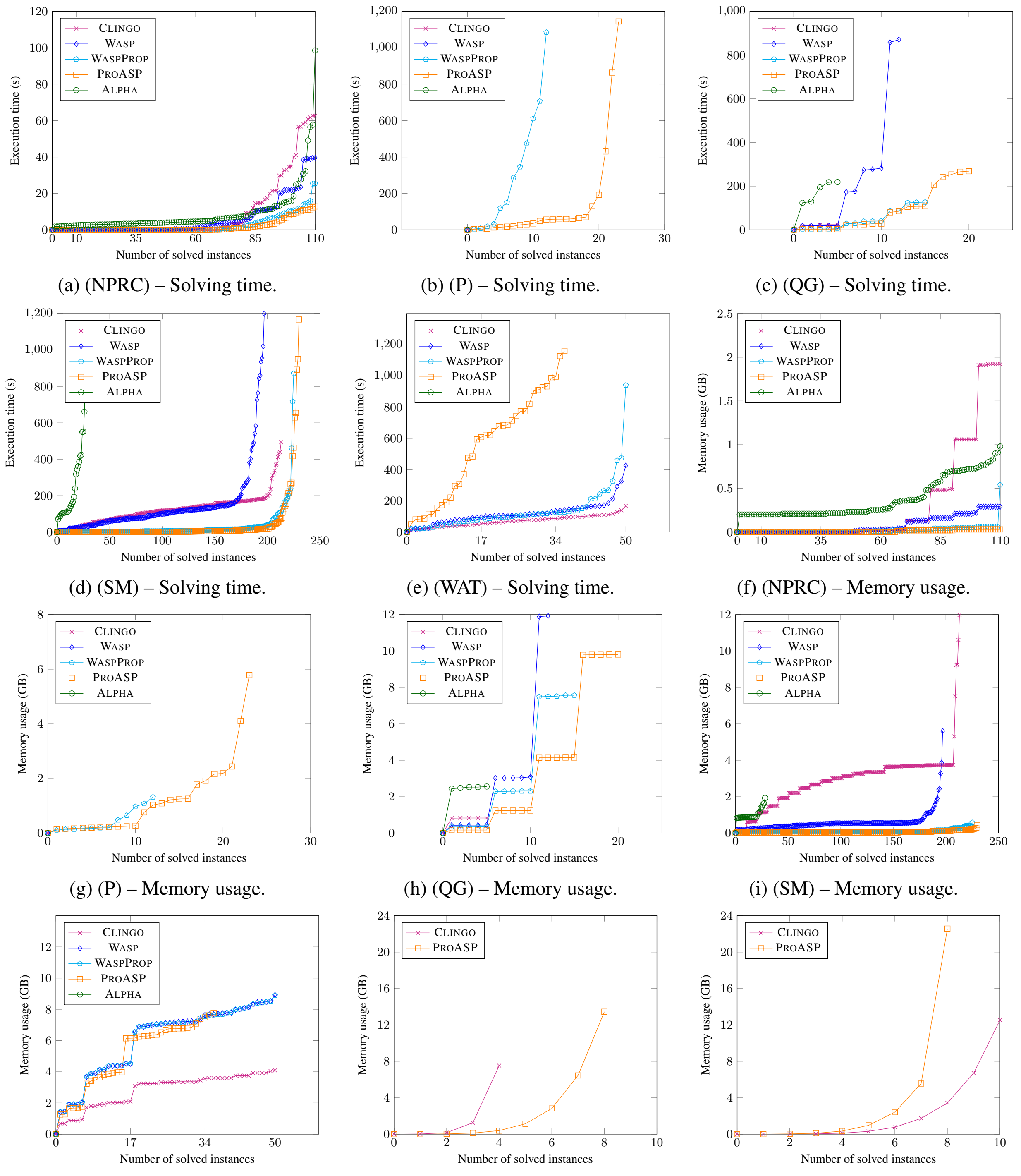
Specifically, ProASP has been compared with *(i)* WaspProp v. cb67c17 [38] where propagators are nested into the solver Wasp [3] and Gringo [27] is used as grounder. Here, due to the syntactical requirements, the compilation can only be applied on constraints; *(ii)* plain version of Wasp v. d87f3f0 using Gringo as grounder; *(iii)* Clingo [26] v. 5.6.2; *(iv)* Alpha [45] v. 0.7.0.

As for the benchmarks, we considered different grounding-intensive benchmarks from the literature, namely: Packing problem (P), Quasi Group (QG), Stable Marriage (SM), Non-Partition Removal Coloring (NPRC), and Weight Assignment Tree (WAT). The problem instances for benchmarks (P), (SM), (NPRC), and (WAT) were taken from previous studies, such as [11, 15, 29]. Moreover, for benchmark (SM), we extended the number of instances by generating novel ones with higher numbers of individuals and varying the percentage of expressed preferences as described in [15]. The (QG) problem consists of placing the numbers from 1 to n into an $n \times n$ matrix, M, in such a way that each row and each column does not contain the same number twice. We generated different instances with different values of n (i.e., from 50 to 1000), and for each value of n, we generated five instances by randomly initializing the matrix M with a random sample of the set $\{1,\ldots,n\}$.

All experiments were executed on an Intel(R) Xeon(R) CPU E5-4610 v2 @ 2.30GHz running Debian Linux (3.16.0-4-amd64), with memory and CPU time (i.e. user+system) limited of 12GB and 1200 seconds, respectively, and each system was limited to run in a single core. We report that, in all the considered benchmarks, the compilation time was negligible for both ProASP and WaspProp. Benchmarks and executables are available at [20].

Results. The results of the experimental evaluation are presented in Table 1 and Figures (2b)–(2j). Table 1 shows the total number of instances for each benchmark (#), and the number of instances solved (SO), the number of instances where the solver exceeded the given time (TO) or memory (MO) limits, for each benchmark and solver. Figure 2 includes cactus plots that depict the time and memory consumption for each benchmark and solver. In a cactus plot, instances are sorted by memory (or time) usage, and a point (i, j) indicates that a solver is capable of solving the i-th instance with a memory (or time) limit of j gigabytes or seconds, respectively.

In general, the results show that ProASP delivered the best performance by solving the highest number of instances and reducing memory consumption overall. The effectiveness of compilation-based approaches is evident in the performance of benchmarks (P), (QG), (SM), and (NPRC), where both ProASP and WaspProp outperformed Clingo and Wasp in both solving time (Figures 2a–2e) and memory usage (Figures 2f–2j). Additionally, the ability to compile the entire program resulted in a significant improvement for ProASP. Specifically, ProASP solved 21 more instances than WaspProp while using less memory in the aforementioned benchmarks. In contrast, the (WAT) domain revealed some limitations of the compilation techniques. Specifically, Clingo proved to be the



(a) (NPRC) – Solving time. (b) (P) – Solving time. (c) (QG) – Solving time.
(d) (SM) – Solving time. (e) (WAT) – Solving time. (f) (NPRC) – Memory usage.
(g) (P) – Memory usage. (h) (QG) – Memory usage. (i) (SM) – Memory usage.
(j) (WAT) – Memory usage. (k) Synthetic (1) – Memory usage. (l) Synthetic (2) – Memory usage.

Figure 2: Comparison of the performance of the different solvers.

optimal choice for addressing this problem. Our experiments showed that the usage of propagators introduced overhead, which is evident when comparing the performances of WASP with WASPPROP. While the memory consumption is comparable, the execution time starts to increase more rapidly after the 40th instance (as shown in Figure 2e). This overhead is even more pronounced in PROASP since the entire program is simulated by propagators. This drawback is partly due to a less informed heuristic that guides the CDCL solver and also to a high number of auxiliary atoms introduced by our rewriting to simulate the program's completion. We also mention that PROASP outperformed the lazy-solver ALPHA by solving more than 276 instances overall and significantly reducing memory consumption on solved instances. However, due to its design strategy, ALPHA used less memory on unsolved instances.

Synthetic benchmarks. To provide a more detailed investigation into the performance of our approach, we conducted an additional experiment on two synthetic benchmarks. This allowed us to highlight both the strengths and weaknesses of PROASP w.r.t. CLINGO. The first benchmark, referred to as Synthetic (1), is designed to showcase scenarios where PROASP is expected to have an advantage. It involves encoding a join among six binary relations, denoted as $a_1, a_2, ..., a_6$, and is formulated as follows:

$$\begin{array}{ll}
a_1(X,Y) & \leftarrow d(X),\ d(Y),\ not\ na_1(X,Y) \\
na_1(X,Y) & \leftarrow d(X),\ d(Y),\ not\ a_1(X,Y) \\
\cdots & \\
a_6(X,Y) & \leftarrow d(X),\ d(Y),\ not\ na_6(X,Y) \\
na_6(X,Y) & \leftarrow d(X),\ d(Y),\ not\ a_6(X,Y) \\
b(X_1,\ldots,X_7) & \leftarrow a_1(X_1,X_2), a_1(X_2,X_3), \ldots, a_6(X_6,X_7) \\
c(X_1,\ldots,X_7) & \leftarrow a_1(X_1,X_2), a_1(X_2,X_3), \ldots, a_6(X_6,X_7) \\
 & \leftarrow b(\_,\_,\_,\_,X,Y,Z), c(Z,Y,X,\_,\_,\_,\_)
\end{array}$$

The second benchmark, referred to as Synthetic (2), aims to highlight a limitation of PROASP that was already observed in the (WAT) benchmark. Indeed, it encodes a join between two relations of arity four and includes a projection over three terms:

$$\begin{array}{ll}
a_1(X,Y,Z,W) & \leftarrow d(X),\ d(Y),\ d(Z),\ d(W), \\
 & \quad not\ na_1(X,Y,Z,W) \\
na_1(X,Y,Z,W) & \leftarrow d(X),\ d(Y),\ d(Z),\ d(W), \\
 & \quad not\ a_1(X,Y,Z,W) \\
a_2(X,Y,Z,W) & \leftarrow d(X),\ d(Y),\ d(Z),\ d(W), \\
 & \quad not\ na_2(X,Y,Z,W) \\
na_2(X,Y,Z,W) & \leftarrow d(X),\ d(Y),\ d(Z),\ d(W), \\
 & \quad not\ a_2(X,Y,Z,W) \\
b(X,Y,Z) & \leftarrow a_1(X,\_,\_,Y), a_2(Y,\_,\_,Z) \\
c(X,Y,Z) & \leftarrow a_1(X,\_,\_,Y), a_2(Y,\_,\_,Z) \\
 & \leftarrow b(X_1,Y,Z), c(Z,Y,X_2)
\end{array}$$

For both benchmarks we generated 10 instances of increasing size by varying the number of atoms over the predicate d, from 3 to 12. Moreover, we increased the memory limit to 24GB, since the grounding phase is highly memory demanding, and we report only the memory consumption since the solving time after grounding is negligible.

As expected, obtained results highlighted a significant improvement introduced by PROASP in benchmark Synthetic (1) (Figure 2k), where the rate at which memory usage increases in PROASP is significantly slower than that in CLINGO, since, in this encoding, there are no projections. Instead, concerning benchmark Synthetic (2) (Figure 2l), we observe that PROASP has a larger usage of memory, since storing the auxiliary atoms (i.e., tuples of terms), together with indices used for computing joins, resulted to be heavier than storing ground program (where aux atoms cost as one integer) for large domains of the predicate d.

6 Related work

The grounding bottleneck is a known limitation of state-of-the-art ASP systems [8, 16], like CLINGO [26] and DLV [1]. Several approaches to overcome the grounding bottleneck have been proposed, which can be divided in three main classes: hybrid approaches, lazy-grounding, and compilation-based.

Hybrid approaches are based on the extension of the base language with additional constructs for connecting ASP solvers with external solvers. These include Constraints Answer Set Programming (CASP) [6, 7, 12, 40, 44], ASP Modulo Theories [26], and HEX programs [21]. While effective, these systems do not address the grounding bottleneck in ASP systems. Instead, they circumvent the issue by shifting the complexity from the logic program to external sources of computation such as constraint solvers and SMT solvers.

Lazy grounding systems perform the grounding of rules during the search for an answer set [8, 34, 36, 41, 45, 46]. In these approaches, a rule is instantiated only when its body is satisfied, thus grounding is done only for rules that are used during the search. The state-of-the-art lazy-grounding system ALPHA [45, 46] combines lazy instantiation techniques with learning, conflict-based heuristics, restarts, phase saving, etc. A key difference between ALPHA and PROASP is that the first discovers the space of propositional atoms during search, whereas PROASP computes it at the beginning. This choice might give advantages to ALPHA when the size of (the useful part of) the Herbrand base is already prohibitively large; on the other hand, it gives advantages to PROASP both in terms of visibility of the search space, and complete compatibility with standard SAT-solving technology. PROASP includes GLUCOSE as it is (with its highly-optimized SAT-solving algorithms and data structures), whereas in ALPHA all these techniques were re-implemented to be blended with lazy grounding. In the same category as ALPHA is the DualGrounder [36], which performs lazy instantiation resorting to the multi-shot API of CLINGO. ALPHA and DualGrounder can outperform traditional ASP systems on grounding-intensive benchmarks and are comparable in performance (as shown in [36]). Moreover, both ALPHA and DualGrounder support a richer input language than PROASP.

PROASP belongs to the compilation-based approaches together with the extensions of the WASP [3] solver with compiled external propagators [15, 16, 38], called WASPPROP. PROASP and WASPPROP share the idea of replacing rules with propagators. On the one hand, WASPPROP can handle aggregates, that are not currently supported by PROASP; on the other hand, PROASP is not limited to subprograms acting as constraints and needs no grounder.

7 Conclusion

ASP systems based on the ground&solve architecture are limited by the grounding bottleneck. Compilation-based approaches have shown promise in addressing this problem, but they were limited to subprograms that behave as constraints.

This paper presents a novel compilation technique that can be used to evaluate tight ASP programs. Thus, the class of ASP programs that can be evaluated by compilation-based techniques is extended *beyond constraints*. The approach is implemented in the new system PROASP, that follows a grounding-less compilation-based architecture. The PROASP compiler generates a custom ASP solver for a given non-ground program, that extends the GLUCOSE SAT solver with propagators. An experiment conducted on grounding-intensive ASP benchmarks shows that PROASP is capable of solving grounding-intensive instances that are out of reach for state-of-the-art ASP systems. Since the evaluation of tight ASP programs is at the core of modern ASP solvers, this work lays the groundwork for a new generation of grounding-less ASP solvers based on compilation.

As future work, there are plans to eventually support the entire ASP-Core 2 standard in the long term. However, it should be noted that the compilation of each advanced ASP construct presents its own unique set of non-trivial challenges. Additional directions for future work follow from the analysis reported in Section 5. Indeed, both the compilation of support propagation and the lazy generation of derived symbols are techniques that might improve PROASP performance on known corner cases.

Acknowledgements

This work was supported by Italian Ministry of Research (MUR) under PRIN project PINPOINT, CUP H23C22000280006, PNRR projects FAIR "Future AI Research" - Spoke 9 - WP9.1 - CUP H23C22000860006, and Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006.

References

[1] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari, 'The ASP system DLV2', in *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, pp. 215–221. Springer, (2017).

[2] Mario Alviano and Carmine Dodaro, 'Completion of disjunctive logic programs', in *IJCAI*, pp. 886–892. IJCAI/AAAI Press, (2016).

[3] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca, 'Advances in WASP', in *LPNMR*, volume 9345 of *Lecture Notes in Computer Science*, pp. 40–54. Springer, (2015).

[4] Giovanni Amendola, Gianluigi Greco, Nicola Leone, and Pierfrancesco Veltri, 'Modeling and reasoning about NTU games via answer set programming', in *IJCAI*, pp. 38–45. IJCAI/AAAI Press, (2016).

[5] Gilles Audemard and Laurent Simon, 'On the glucose SAT solver', *Int. J. Artif. Intell. Tools*, **27**(1), 1840001:1–1840001:25, (2018).

[6] Rehan Abdul Aziz, Geoffrey Chu, and Peter J. Stuckey, 'Stable model semantics for founded bounds', *Theory Pract. Log. Program.*, **13**(4-5), 517–532, (2013).

[7] Marcello Balduccini and Yuliya Lierler, 'Constraint answer set solver EZCSP and why integration schemas matter', *Theory Pract. Log. Program.*, **17**(4), 462–515, (2017).

[8] Jori Bomanson, Tomi Janhunen, and Antonius Weinzierl, 'Enhancing lazy grounding with lazy normalization in answer-set programming', in *AAAI*, pp. 2694–2702. AAAI Press, (2019).

[9] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski, 'Answer set programming at a glance', *Commun. ACM*, **54**(12), 92–103, (2011).

[10] Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca, 'Design and results of the fifth answer set programming competition', *Artif. Intell.*, **231**, 151–181, (2016).

[11] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca, 'The third open answer set programming competition', *Theory Pract. Log. Program.*, **14**(1), 117–135, (2014).

[12] Broes De Cat, Marc Denecker, Maurice Bruynooghe, and Peter J. Stuckey, 'Lazy model expansion: Interleaving grounding with search', *J. Artif. Intell. Res.*, **52**, 235–286, (2015).

[13] Keith L. Clark, 'Negation as failure', in *Logic and Data Bases*, Advances in Data Base Theory, pp. 293–322, New York, (1977). Plemum Press.

[14] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller, 'Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis', *Theory Pract. Log. Program.*, **17**(5-6), 780–799, (2017).

[15] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller, 'Partial compilation of ASP programs', *Theory Pract. Log. Program.*, **19**(5-6), 857–873, (2019).

[16] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller, 'Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators', in *IJCAI*, pp. 1688–1694. ijcai.org, (2020).

[17] Bernardo Cuteri, Kristian Reale, and Francesco Ricca, 'A logic-based question answering system for cultural heritage', in *JELIA*, volume 11468 of *LNCS*, pp. 526–541. Springer, (2019).

[18] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov, 'Complexity and expressive power of logic programming', *ACM Comput. Surv.*, **33**(3), 374–425, (2001).

[19] Carmine Dodaro and Marco Maratea, 'Nurse scheduling via answer set programming', in *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, pp. 301–307. Springer, (2017).

[20] Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. Benchmarks. https://doi.org/10.5281/zenodo.8179111, 2023.

[21] Thomas Eiter, Christoph Redl, and Peter Schüller, 'Problem solving using the HEX family', in *Computational Models of Rationality*, pp. 150–174. College Publications, (2016).

[22] Esra Erdem, Michael Gelfond, and Nicola Leone, 'Applications of answer set programming', *AI Mag.*, **37**(3), 53–68, (2016).

[23] Esra Erdem and Vladimir Lifschitz, 'Tight logic programs', *Theory Pract. Log. Program.*, **3**(4-5), 499–518, (2003).

[24] Esra Erdem and Volkan Patoglu, 'Applications of ASP in robotics', *Künstliche Intell.*, **32**(2-3), 143–149, (2018).

[25] François Fages, 'Consistency of clark's completion and existence of stable models', *Methods Log. Comput. Sci.*, **1**(1), 51–60, (1994).

[26] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko, 'Theory solving made easy with clingo 5', in *ICLP (Technical Communications)*, volume 52 of *OASICS*, pp. 2:1–2:15. Schloss Dagstuhl, (2016).

[27] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub, 'Advances in *gringo* series 3', in *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pp. 345–351. Springer, (2011).

[28] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub, 'Evaluation techniques and systems for answer set programming: a survey', in *IJCAI*, pp. 5450–5456. ijcai.org, (2018).

[29] Martin Gebser, Marco Maratea, and Francesco Ricca, 'The sixth answer set programming competition', *J. Artif. Intell. Res.*, **60**, 41–95, (2017).

[30] Michael Gelfond and Vladimir Lifschitz, 'Classical negation in logic programs and disjunctive databases', *New Gener. Comput.*, **9**(3/4), 365–386, (1991).

[31] Tomi Janhunen, 'Representing normal programs with clauses', in *ECAI*, pp. 358–362. IOS Press, (2004).

[32] Tomi Janhunen, 'Implementing stable-unstable semantics with ASPTOOLS and clingo', in *PADL*, volume 13165 of *Lecture Notes in Computer Science*, pp. 135–153. Springer, (2022).

[33] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub, 'Grounding and solving in answer set programming', *AI Mag.*, **37**(3), 25–32, (2016).

[34] Claire Lefèvre and Pascal Nicolas, 'The first version of a new ASP solver : Asperix', in *LPNMR*, volume 5753 of *Lecture Notes in Computer Science*, pp. 522–527. Springer, (2009).

[35] Yuliya Lierler and Marco Maratea, 'Cmodels-2: Sat-based answer set solver enhanced to non-tight programs', in *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, pp. 346–350. Springer, (2004).

[36] Yuliya Lierler and Justin Robbins, 'Dualgrounder: Lazy instantiation via clingo multi-shot framework', in *JELIA*, volume 12678 of *Lecture Notes in Computer Science*, pp. 435–441. Springer, (2021).

[37] João Marques-Silva, Inês Lynce, and Sharad Malik, 'Conflict-driven clause learning SAT solvers', in *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 133–182, IOS Press, (2021).

[38] Giuseppe Mazzotta, Francesco Ricca, and Carmine Dodaro, 'Compilation of aggregates in ASP systems', in *AAAI*, pp. 5834–5841. AAAI Press, (2022).

[39] Arindam Mitra, Peter Clark, Oyvind Tafjord, and Chitta Baral, 'Declarative question answering over knowledge bases containing natural language text with answer set programming', in *AAAI*, pp. 3003–3010. AAAI Press, (2019).

[40] Max Ostrowski and Torsten Schaub, 'ASP modulo CSP: the clingcon system', *Theory Pract. Log. Program.*, **12**(4-5), 485–503, (2012).

[41] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi, 'GASP: answer set programming with lazy grounding', *Fundam. Informaticae*, **96**(3), 297–322, (2009).

[42] Peter Schüller, 'Modeling variations of first-order horn abduction in answer set programming', *Fundam. Informaticae*, **149**(1-2), 159–207, (2016).

[43] Tran Cao Son, Enrico Pontelli, Marcello Balduccini, and Torsten Schaub, 'Answer set planning: A survey', *Theory Pract. Log. Program.*, **23**(1), 226–298, (2023).

[44] Benjamin Susman and Yuliya Lierler, 'Smt-based constraint answer set solver EZSMT (system description)', in *ICLP (Technical Communications)*, volume 52 of *OASIcs*, pp. 1:1–1:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, (2016).

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

[46] Antonius Weinzierl, Richard Taupe, and Gerhard Friedrich, 'Advancing lazy-grounding ASP solving techniques - restarts, phase saving, heuristics, and more', *Theory Pract. Log. Program.*, **20**(5), 609–624, (2020).