Enhancing Compilation-based ASP Solving with Postponed Atom Discovery

Andrea Cuteri, Giuseppe Mazzotta and Francesco Ricca

Department of Mathematics and Computer Science, University of Calabria, Rende, Italy

Abstract

Answer Set Programming (ASP) systems following the Ground&Solve approach are affected by the grounding bottleneck problem, where the grounding phase can cause a combinatorial explosion in program size, making solving unfeasible. Compilation-based ASP solving addresses this by simulating rule inferences through ad hoc propagators, avoiding the grounding phase. However, existing approaches require all propositional atoms to be computed upfront, resulting in overhead as the number of generated atoms grows. In this paper, we investigate how compilation-based technique can be enhanced by deferring the evaluation of rules generating large number of atoms, and so enabling atoms to be discovered lazily during solving. Preliminary results demonstrate significant performance improvements over traditional compilation-based solvers.

Keywords

Answer Set Programming, Grounding Bottleneck, Compilation-based ASP Solving, Compiled propagators

1. Introduction

Answer Set Programming (ASP) [1, 2] is a well-established declarative paradigm based on the stable model semantics [2, 3]. Nowadays, ASP has found extensive applications in a variety of fields, including Planning [4], Scheduling [5, 6, 7], Robotics [8], Natural Language Processing [9, 10, 11, 12], and Databases [13, 14, 15], among many others [16, 17]. The success of ASP stems from the two main strengths of such formalism. First, it provides a highly expressive language that enables the concise modeling of hard combinatorial problems [18]. Second, powerful solvers [19] make it a practical solution for tackling real-world context [17, 20] as well as industrial applications [21, 22, 23, 24].

Standard ASP systems, such as CLINGO [25] and DLV [26], follow the Ground&Solve approach [27]. Basically, the evaluation of a program is performed in two phases: *grounding*, which instantiates rules by replacing variables with constants, and *solving*, where answer sets are computed using CDCL-based algorithms [28]. While this workflow is effective in many scenarios, it suffers from a major limitation known as the grounding bottleneck [19]. In several practical cases [29, 30], the grounding phase causes a combinatorial explosion in the size of the program, making the solving unfeasible. Thus, the optimization of ASP systems continues to be a pivotal research area, as system performance plays a critical role in enabling the development of efficient applications [19].

To overcome this limitation, various techniques have been proposed, including hybrid approaches [31, 25, 30, 32], lazy grounding [33, 34, 35, 36, 37], complexity-aware rewritings [38], and compilation-based solving [39, 40, 41, 42].

Compilation-based techniques tackle the grounding bottleneck by translating non-ground rules into specialized procedures, known as propagators, which simulate rules' inferences during solving without materializing the corresponding ground rules. Empirical evidence showed that these techniques effectively tackled the grounding bottleneck problem by solving instances that were out-of-reach for state-of-the-art systems [41, 42, 43]. However, compilation-based systems require propositional atoms, needed to compute an answer set, to be generated ahead of solving. As it has been pointed out by Dodaro

Joint Proceedings of the Workshops and Doctoral Consortium of the 41st International Conference on Logic Programming, September 9–13, 2025, Rende, Italy

andrea.cuteri@unical.it (A. Cuteri); giuseppe.mazzotta@unical.it (G. Mazzotta); francesco.ricca@unical.it (F. Ricca)
0009-0000-7629-7347 (A. Cuteri); 0000-0003-0125-0477 (G. Mazzotta); 0000-0001-8218-3178 (F. Ricca)

et al. (2023) when the number of atoms grows large, this upfront computation leads to considerable overhead.

In this paper, we describe by examples a possible strategy to overcome this limitation and apply it to a few use cases. The idea is to discover some of the atoms incrementally during solving. These atoms are created dynamically, only when needed for truth assignment, instead of being precomputed, thus eliminating the overhead of the upfront atom computation. In particular, we propose an implementation of such technique on top of the compilation-based ASP system PROASP [42, 43] and preliminary experiments aimed at assessing the impact of the proposed technique. Obtained results show that lazy atom discovery significantly reduces both memory usage and runtime, offering a scalable and efficient alternative to traditional compilation-based solving.

2. Preliminaries

In this section we provide an overview of the ASP language, focusing on the concepts relevant for understanding the contribution of this paper. A more comprehensive description can be found in the existing literature. [1, 44, 45].

ASP Syntax. A term can either be a variable or a constant. A variable is a string starting with upper case letter, while a constant is a number or a string starting with lowercase letter. An atom is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate of arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms. A literal is either an atom a or its negation $\sim a$, where $\sim$ represents negation as failure. Given a literal $l = a$ (resp. $\sim a$), $\bar{l}$ denotes the complement of l that is $\sim a$ (resp. a); while $terms(l)$ denotes the set of terms appearing in literal l. A literal l is *ground* if it does not contain any variable. A *positive literal* is of the form a, where a is an atom, while $\sim a$ is a *negative literal.* For a set of literals S, S^+ (resp. S^-) denotes the set of positive (resp. negative) literals appearing in S. A rule is an expression of the form $h \leftarrow l_1, \ldots, l_n$ where h is an atom referred to as *head* that can be also omitted, and $l_1, \ldots, l_n$, with $n > 0$, is a conjunction of literals referred to as $body$. Given a rule r, $B(r)$ is the set of literals appearing in the body of r while $H(r)$ is the set of atoms appearing in the head of r. A rule r is a constraint if $H(r) = \emptyset$; or it is a fact if $B(r) = \emptyset$. Moreover, r is *safe* if each variable appearing in r appears also in $B(r)^+$. A *program* Π is a set of safe rules. Given an ASP expression ϵ (a program, a rule etc.), $pred(\epsilon)$ and $vars(\epsilon)$ denotes respectively the set of predicates and variables appearing in ϵ. Given a rule r, $head(r) \subseteq pred(r)$ is the set of predicates appearing in $H(r)$. This notation extends also to programs, and so $head(\Pi)$ is the set of predicates appearing in the head of some rules in Π. Given a program Π, the *Herbrand Universe* U_Π is the set of constants occurring in Π, while the *Herbrand base* B_Π is the set of ground atoms that can be obtained from predicates of Π and constants in U_Π. The *dependency graph* of program Π is a labeled graph $G_\Pi = \langle V, E \rangle$, where V is the set of predicates appearing in Π and E contains a positive (resp. negative) edge from p to q if there exists a rule $r \in \Pi$ such that p appears in $B(r)^+$ (resp. p appears in $B(r)^-$) and $q \in head(r)$. Program Π is said to be *stratified* if there exist no loop in G_Π involving negative edges; *tight* if G_Π does not contain positive loops [46]. We focus on tight ASP programs, as this class is sufficiently expressive for our purposes while remaining amenable to modeling many complex problems [29], and corresponds to the class supported by PROASP [42, 43].

Stable Model Semantics. Given a program Π, an *interpretation* I is a set of literals over atoms in B_Π. A literal l is true w.r.t. I (i.e., $I \models l$) if $l \in I$; false (i.e. $I \not\models l$) if $\bar{l} \in I$; undefined otherwise. A conjunction of literals $Conj$ is true w.r.t. I (i.e., $I \models Conj$) if $I \models l$ for each l in $Conj$; false (i.e. $I \not\models Conj$) if there exists l in $Conj$ such that $I \not\models l$; undefined otherwise. An interpretation I is *total* if for each atom $a \in B_\Pi$, $a \in I$ or $\sim a \in I$. An interpretation I is *consistent* if it does not exist $l \in I$ such that $\bar{l} \in I$. An atom $a \in B_\Pi$ is *supported* in Π w.r.t. I if there exists $r \in ground(\Pi)$ such that $H(r) = a$ and $I \models B(r)$. Given a rule $r \in ground(\Pi)$, then r is satisfied w.r.t. I if $I \models H(r)$ or $I \not\models B(r)$. A total and consistent interpretation I is a *model* of Π if all rules in $ground(\Pi)$ are satisfied. Let I be a model of Π, then I is a *supported model* if each positive literal in I is supported in Π w.r.t. I. Moreover, I is a *stable model* (or an *answer set*) of Π if I is a $\subset$-minimal model of its Gelfond-Lifschitz reduct [2]. For a program Π, $AS(\Pi)$ denotes the set of answer set of Π. For the class of programs considered in

this paper (i.e., tight programs) supported models coincide with answer sets.

3. The ProASP system

Standard ASP systems [26, 25] compute stable models of a program Π in two stages. First, a grounder module computes $ground(\Pi)$, and then a solver module computes answer sets of Π by implementing an extension of the CDCL (Conflict Driven Clause Learning) with *propagators* specific for ASP [27]. Such an approach is affected by a limitation known as *grounding bottleneck*. In many cases of practical interest $ground(\Pi)$ contains a large number of rules and atoms that cannot be stored in memory, thus preventing the solving stage from even beginning. Compilation-based ASP solvers tackle such an issue by compiling a logic program into an ad-hoc solver, made of rule-specific propagators, that simulate inferences deriving from $ground(\Pi)$ without explicitly materializing ground rules. In what follows we recall how the compilation-based ASP solver PROASP works and we discuss possible limitations of this approach.

3.1. Compilation of ASP prorgams in PROASP

PROASP is a compilation-based ASP solver which supports the class of tight ASP programs. The main idea behind the PROASP approach is the following: *"Given an ASP program Π, we build an ad-hoc solver for evaluating Π. Then, for each set of facts F the ad-hoc solver searches for an answer set of $\Pi \cup F$."*

Consider the following program Π:

$$\begin{array}{lrcl} r_1: & a(X) & \leftarrow & d(X),\ not\ na(X) \\ r_2: & na(X) & \leftarrow & d(X),\ not\ a(X) \\ r_3: & & \leftarrow & a(X),\ a(Y),\ X < Y. \end{array}$$

If we consider rules r_1 and r_2, and a possible constant x, then the following propagations may derive from them:

- $a(x)$ is true if and only if $d(x)$ is true and $na(x)$ is false.
- $na(x)$ is true if and only if $d(x)$ is true and $a(x)$ is false.

These propagations guarantee that rules r_1 and r_2 are satisfied and atoms over predicates a and na are supported w.r.t. Π. On the other hand, if we consider the constraint r_3 the following propagation may derive from it:

- if there exists x such that $a(x)$ is true then for each value $y > x$, $a(y)$ must be false;
- if there exists y such that $a(y)$ is true then for each value $x < y$, $a(x)$ must be false.

Such propagations guarantee that the constraint r_3 is satisfied.

Based on these inferences we can design a rule-specific propagator for each $r \in \Pi$. For space reasons we report only a possible propagator for r_1 (Algorithm 1) and r_3 (Algorithm 2), since the propagator for r_2 is equal to (Algorithm 1) modulo predicate renaming.

Given the set of facts $F = \{d(1),\ d(2)\}$, if we want to compute an answer set of $\Pi \cup F$ we need to generate the set of relevant atoms that is needed to compute such an answer set (i.e. a subset of the Herbrand Base), which is $\{a(1),\ a(2),\ na(1),\ na(2)\}$. By giving as input generated atoms to an implementation of CDCL (i.e., a SAT solver) equipped with rule-specific propagators we can compute an answer set of $\Pi \cup F$ as follows.

Classical CDCL implementations follow the *choose-propagate-learn* pattern starting from an empty interpretation I. At each step, a literal l is heuristically chosen and added to I. Then l is propagated for deriving further literals from rules in Π. In our setting propagation is demanded to rule-specific propagators (Algorithms 1 and 2) which infer new literals to be added to I. During propagation, if the interpretation I becomes inconsistent, then the *learn* phase is started for analyzing conflicting literals (i.e., $l \in I$ and $\bar{l} \in I$) to restore consistency of I. If consistency can be restored, then the process

Algorithm 1 Propagator for $a(X) \leftarrow d(X), not\ na(X)$.

```
Input : An interpretation I, and a literal l ∈ I
begin
    if pred(l) = "a":
        x := l[0]
        if l ∈ I+:
            propagate {d(x), ~na(x)}
        else:
            if d(x) ∈ I+:
                propagate {na(x)}
            else if ~na(x) ∈ I−:
                propagate {~d(x)}
    if pred(l) = "d":
        x := l[0]
        if l ∈ I+:
            if ~na(x) ∈ I−:
                propagate {a(x)}
        else:
            propagate {~a(x)}
    if pred(l) = "na":
        x := l[0]
        if l ∈ I−:
            if d(x) ∈ I+:
                propagate {a(x)}
        else:
            propagate {~a(x)}
```

Algorithm 2 Propagator for $\leftarrow a(X), a(Y), X < Y$.

```
Input : An interpretation I, and a literal l ∈ I
begin
    if pred(l) = "a" and l ∈ I+:
        x := l[0]
        for all y ∈ U_Π s.t. y > x:
            propagate ~a(y)
        y := l[0]
        for all x ∈ U_Π s.t. x < y:
            propagate ~a(x)
```

back-jumps to the choices that led to the conflict and continues the search. On the other hand, if consistency cannot be restored, then the algorithm stops since program Π has no models. Finally, as soon as I is total and consistent then I is an answer set of $\Pi \cup F$.

The compilation-based ASP solver PROASP generalizes the example discussed so far, allowing the compilation of tight ASP programs into ad-hoc solvers [43].

Given a (non-ground) program Π, at compilation stage PROASP extracts two subprograms, namely a *propagator* and *generator* program from Π. The propagator program is obtained by normalizing rules of Π in specific rule forms which allow to have a uniform compilation approach while preserving inferences of the original program. Conversely, the generator program is made of rules defining the domain of each predicate appearing in the propagator program and so it will be used to generate the subset of relevant atoms needed for computing answer sets of Π. Propagator and generator programs are compiled, respectively, into two modules, namely *Propagator* and *Generator* module. The resulting solver, namely Π-solver, is obtained by integrating Propagator and Generator modules together with the SAT Solver Glucose.

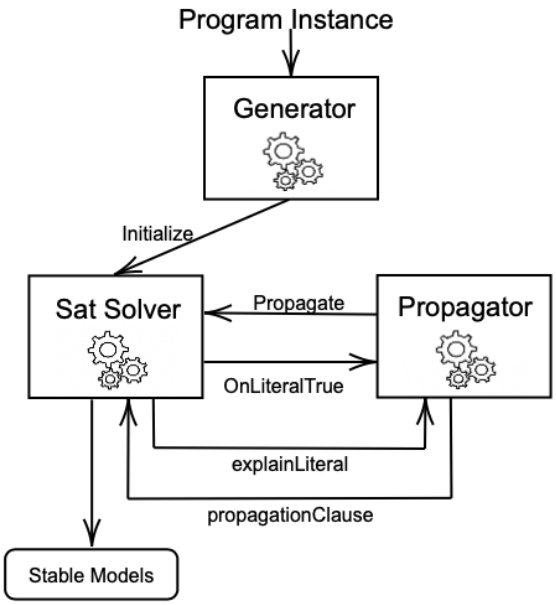


Figure 1: Compiled Solver

As reported in Figure 1, at the solving stage, the Π-solver takes as input a program instance F (i.e., a set of facts) and starts by generating ground atoms appearing in the propagator program, by means of the *Generator* module. Generated atoms are fed into the *Glucose* SAT solver which starts the CDCL from an empty interpretation I. During the CDCL, as soon as a literal is added to I, the SAT solver notifies the *Propagator* which computes new inferences (if any). By proceeding in this way, the SAT solver stops if I becomes total, and so I is an answer set, or consistency cannot be restored and so $\Pi \cup F$ does not have any answer set.

Empirical evaluations [41, 42, 43] show that compiled solvers allow to effectively tackle the grounding bottleneck problem by allowing to solve instances that were out-of-reach for standard systems while remaining competitive on problems that do not present grounding issues. However, as it has been pointed out by Dodaro et al. (2023), compiled solvers suffer whenever the number of (auxiliary) atoms obtained in the generation phase increases. In these cases, compiled solvers require a lot of computational resources, even more than standard ones.

4. Lazy atom generation in PROASP

To reduce the number of atoms that are generated upfront by PROASP, we propose a novel compilation-based technique that postpones the evaluation of problematic rules (i.e., rules generating a large number of atoms) to the solving phase. To present our approach we first introduce the notion of *lazy program split* which identifies the syntactic restrictions under which rules can be compiled with lazy atom generation.

Definition 1 (Lazy Program Split). *Let* Π *be a program and* $L \subseteq \Pi$. *Then* L *is a lazy program split for* Π *if* L *is stratified and* $head(L) \cap head(\Pi \setminus L) = \emptyset$.

Intuitively, a lazy program split is a stratified subprogram $L \subseteq \Pi$ such that predicates in the head of some rule in L do not appear in the head of any rule of $\Pi \setminus L$. Stratification is required to ensure that only deterministic inferences can be derived from L. The second condition instead allows to separate the evaluation of L from the standard PROASP pipeline.

To evaluate a program Π, state-of-the-art ASP solvers transform rules in $ground(\Pi)$ by applying the well-known Clark's completion [47]. By means of this transformation, solvers are able to compute

supported models by leveraging unit propagation of program clauses. However, this transformation may generate a large number of clauses and requires ground rules to be materialized, which is detrimental when the program is grounding intensive. To avoid such transformation for rules in a lazy program split L, it is possible to leverage the Fitting operator [48]. Such an operator guarantees that rules in $ground(L)$ are satisfied and that each positive literal in an answer set is supported.

Definition 2 (Fitting 1985). *Given a program Π and a (partial) interpretation I, the Fitting operator Φ_Π is defined as:*

- $\Phi_\Pi(I) = T_\Pi(I) \cup \overline{\gamma_\Pi(I)}$
- $T_\Pi(I) = \{H(r) \mid r \in ground(\Pi) \wedge I \models B(r)\}$
- $\gamma_\Pi(I) = \{a \in B_\Pi \mid \forall r \in ground(\Pi)\ s.t.\ H(r) = a, I \not\models B(r)\}$.

Intuitively, given an interpretation I, the operator $T_\Pi(I)$ derives the head of each rule in $ground(\Pi)$ having a true body w.r.t. I. In this way rules in Π are satisfied and atoms in $T_\Pi(I)$ are supported in Π w.r.t. I. On the other hand, $\gamma_\Pi(I)$ derives all those atoms that cannot be supported in Π w.r.t. I. Thus, such atoms cannot be true in an answer set of Π that extends I.

Based on the aforementioned properties of the Fitting operator, a lazy program split L of a program Π can be compiled into ad-hoc propagators, namely *Consequence* and *Support* propagators, which mimic, respectively, the application of T_L and γ_L on rules in L.

More in detail, the *Consequence* operator is implemented via a bottom-up evaluation that builds instantiations of rules in L that have a true body w.r.t. a (partial) interpretation I and infers the head of such rules. Conversely, the *Support* operator is implemented via a top-down evaluation that, given an atom a searches for an instantiation of a rule in L that (i) has a in the head; and (ii) whose body is not false w.r.t. I.

4.1. Lazy atom discovery in ProASP

Here we discuss in a more detailed way, how we implemented our techniques on top of the PROASP system. To enable lazy atom discovery during solving we extended both the compilation and solving pipelines that are reported in Figures 2 and 3.

Extended Compilation. Let Π be a program and L be a lazy program split of Π. Our objective is to compile L into ad-hoc propagators that simulate the application of the Fitting operator on rules in L. Whereas, the program $\Pi \setminus L$ follows the aforementioned PROASP compilation pipeline.

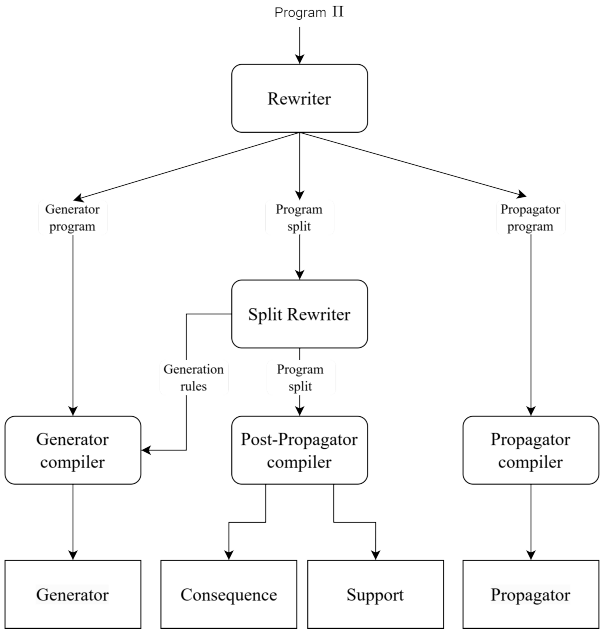
According to the definition of lazy program split, we know that predicates in $head(L)$ do not appear in the head of any rule of $\Pi \setminus L$. However, it is important to point out that predicates in $head(L)$ may appear in the body of some rules in $\Pi \setminus L$. In this case, PROASP propagators require knowledge of the existence of such atoms, and thus, it is required that PROASP's *Generator* module generates those atoms over predicates in $head(L)$ appearing in the body of rules of $\Pi \setminus L$, referred to as *interface atoms*. To this end, we introduce a rewriting technique which produces the rules needed for generating interface atoms.

Definition 3 (Interface Program). *Given a program Π and a lazy program split L of Π, then $interface(\Pi, L)$ contains, for each $r \in \Pi \setminus L$, the following rules:*

$$\begin{aligned} lit &\leftarrow B_\epsilon(r) \quad \forall lit \in B_\lambda(r)^+ \\ \overline{lit} &\leftarrow B_\epsilon(r) \quad \forall lit \in B_\lambda(r)^- \end{aligned}$$

where $B_\lambda(r)$ denotes the conjunction of literals $l \in B(r)$ such that $pred(l) \in head(L)$ and $B_\epsilon(r)$ denotes the conjunction of literals $l \in B(r)$ such that $pred(l) \in head(\Pi \setminus L)$.

Figure 2: Extended PROASP compiler



Example 4.1. *Let* Π *be the following program:*

$$\begin{aligned} a(X) &\leftarrow d(X), {\sim}na(X) \\ na(X) &\leftarrow d(X), {\sim}a(X) \\ b(X) &\leftarrow d(X), {\sim}nb(X) \\ nb(X) &\leftarrow d(X), {\sim}b(X) \\ c(X,Y) &\leftarrow a(X), b(Y) \\ &\leftarrow c(1,Y), a(Y) \end{aligned}$$

Let $L = \{c(X,Y) \leftarrow a(X), b(Y)\}$ *then* L *is a lazy program split for* Π*, since it is stratified and* c *does not appear in the head of any rule of* $\Pi \setminus L$*. In this case the interface program* $interface(\Pi, L)$ *contains only the rule* $c(1,Y) \leftarrow a(Y)$*, which is obtained from the constraint* $\leftarrow c(1,Y), a(Y)$*.*

However, there could be cases in which rules of the interface program are unsafe. If in program Π from Example 4.1 we add the rule $\leftarrow c(X,Y), a(Y)$ to Π, then $interface(\Pi, L)$ contains also the rule $c(X,Y) \leftarrow a(Y)$ which is unsafe since variable X appears only in the head of the rule. To avoid these cases, we restrict our approach to programs that satisfy a stronger notion of safety, namely *lazy safety*.

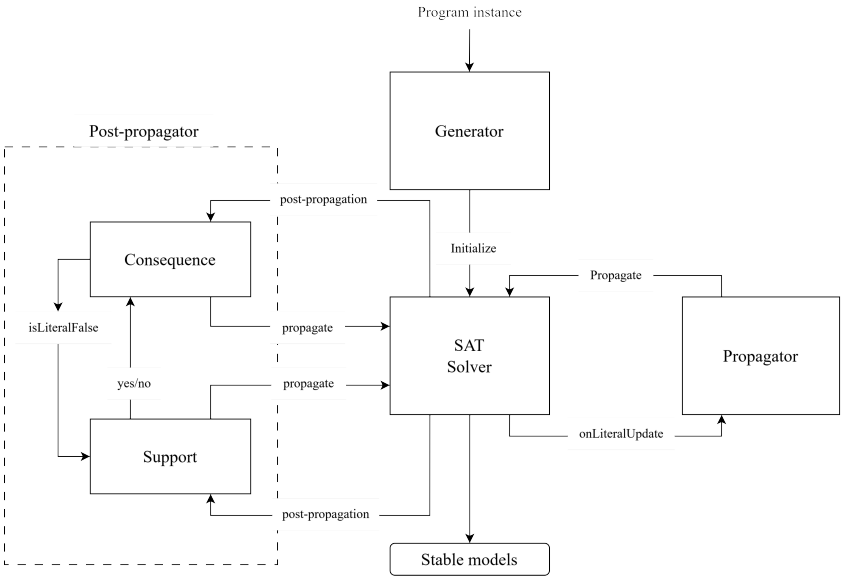
Definition 4 (Lazy Safety). *Given a program* Π *and a split* L*,* Π *is lazy safe if for each rule* $r \in \Pi$ $vars(B_\lambda(r)) \subseteq vars(B_\epsilon(r)^+)$*, where* $B_\lambda(r)$ *denotes the conjunction of literals* $l \in B(r)$ *such that* $pred(l) \in head(L)$ *and* $B_\epsilon(r)$ *denotes the conjunction of literals* l *such that* $pred(l) \in head(\Pi \setminus L)$*.*

Intuitively, lazy safety requires that atoms over predicate in $head(L)$ are bounded by variables appearing in positive literals not in $head(\Pi \setminus L)$. If a program is lazy safe then the interface program, $interface(\Pi, L)$ only contains safe rules. Note that, this restriction does not introduce any loss of generality since it is possible to use an auxiliary predicate that gives bound to unsafe variables by considering all possible constants in the Herbrand universe. However, this is not ideal since the *Generator* will produce more atoms than the ones produced by standard grounders such as GRINGO [49] and I-DLV [50] that perform an intelligent bottom-up grounding.

We are now ready to discuss the extended PROASP's compilation pipeline reported in Figure 2.

Given a program Π, and a lazy program split L, the *Rewriter* module rewrites all the rules in $\Pi \setminus L$ by following the standard PROASP methodology [43]. Thus, it produces as output the generator and the propagator program, while rules in L remain in their original form.

Figure 3: Extended PROASP solver



At this point, the generator program obtained by PROASP does not contain any rule defining interface atoms, and this is an issue since such atoms, as it has been previously pointed out, may appear in the body of some rule in $\Pi \setminus L$. In this case the PROASP solver may erroneously consider interface atoms as false whereas they could be discovered lazily during solving.

To this end, the *Split Rewriter* computes $interface(\Pi, L)$ which will be used to generate ad-hoc generator procedures, nested into the Generator module, that are able to produce the interface atoms.

At this point, each $r \in L$ is ready to be compiled into ad-hoc post-propagators which simulate the application of both T_L and γ_L operators on the rule r. In the following section we report an example of compiled propagators which will clarify how compiled post-propagators work. As a result of the compilation of L, we obtain two modules, namely *Consequence* and *Support* which will be integrated in the compiled solver for postponing the evaluation of the lazy program split L.

Extended Solving. The architecture of the resulting PROASP solver is depicted in Figure 3. At the solving stage, the compiled solver takes as input a program instance F, that is a set of facts, and the generator module produces the subset of relevant atoms that appear in the propagator program. Among atoms generated by the generator module, we need to distinguish between those over predicates in $head(\Pi \setminus L)$ and those over predicates in $head(L)$ (i.e., the interface atoms). Atoms whose predicate is in $head(\Pi \setminus L)$ are entirely handled by the SAT solver and the propagator module. Conversely, the truth value of atoms whose predicate is in $head(L)$ must be determined through the *Consequence* and *Support* modules. Among possible atoms over predicates in $head(L)$, only the interface atoms have been generated while the remaining one are unknown to the solver. Since interface atoms are indeed solver's decision variables then the solver can decide their truth value; while unknown ones can only be discovered by post-propagators during solving. To guarantee that solver choices are consistent with rules in L, at each post-propagator call, the SAT solver uses the *Consequence* module for deriving both interface and unknown atoms as true. In particular, interface atoms will be assigned into the SAT solver, while unknown atoms are lazily discovered and added to the current interpretation. Unknown atoms that are never derived by the *Consequence* module can be inferred as false. Note that, in this setting, false atoms can only be derived after that all solver's decision variables have been assigned to a truth value. Even though this approach guarantees correctness, it is not applicable in practice, since lazy program splits may contain negation and so we may propagate atoms too late during solving. To anticipate such propagations, when some unknown atoms are encountered during bottom-up evaluation of the *Consequence* module, the *Support* module is leveraged to infer false unknown atoms. On the other hand,

Algorithm 3 Compiled Fixpoint for $b(X,Y) \leftarrow a(X), a(Y)$.

```
Input : An interpretation I
begin
    D = ∅
    T_1 = {a(x) ∈ I}
    For t_1 ∈ T_1
        x := t_1[0]
        T_2 = {a(y) ∈ I}
        For t_2 ∈ T_2
            y := t_2[0]
            h := b(x, y)
            if b(x, y) ∉ I^+
                D := D ∪ {h}
    return D
```

to infer false interface atoms the SAT solver calls *Support* module at each post-propagator call. As soon as atoms derived either from *Consequence* or *Support* module were already assigned by the solver to a different truth value (i.e., we obtained an inconsistent interpretation), a conflict is generated. Such a conflict is handled by the SAT solver which resolves it, if possible, by analyzing the literals that derived conflictual literals (i.e., reasons of the propagations). If the conflict can be resolved then the solving restarts up to a total and consistent interpretation is constructed (i.e., an answer set is found) or the incoherence of the program is proved.

4.2. An example of compilation

In this section we provide an example of compilation of lazy program split into post-propagators. Recall that post-propagators are lower priority propagators that are used in ASP systems to perform additional checks required by stable model semantics [51]. Consider the following program Π:

$$\begin{aligned} a(X) &\leftarrow d(X),\ {\sim}na(X). \\ na(X) &\leftarrow d(X),\ {\sim}a(X). \\ c(X) &\leftarrow d(X),\ {\sim}nc(X). \\ nc(X) &\leftarrow d(X),\ {\sim}c(X). \\ b(X,Y) &\leftarrow a(X),\ a(Y). \\ &\leftarrow b(X,1),\ c(X). \end{aligned}$$

In this case, the rule $b(X,Y) \leftarrow a(X),\ a(Y)$ is itself the lazy program split. Thus, we consider $L = \{b(X,Y) \leftarrow a(X),\ a(Y)\}$ as the subprogram whose evaluation should be postponed and $\Pi \setminus L$ as the subprogram, made of remaining rules of Π, that follows the standard PROASP approach. We now describe how the compiled solver is obtained. As first step of the compilation stage, the compiler takes as input the program Π where rules in L are labeled as lazy program split. The rewriting process transforms program $\Pi \setminus L$ into the corresponding *Generator* and *Propagator* programs. Note that the generator program is augmented with generation rules in $interface(\Pi, L)$. In this case, the predicate b appears only in the constraint $\leftarrow b(X,1),\ c(X)$ and so the only rule in $interface(\Pi, L)$ is $b(X,1) \leftarrow c(X)$. At this point, we are ready to compile generator and propagator programs, as well as the lazy program split L. The former are compiled into *Generator* and *Propagator* modules while the latter is compiled into *Consequence* and *Support* modules. More precisely, since L only contains the rule $b(X,Y) \leftarrow a(X),\ a(Y)$ then the *Consequence* module only contains the compiled procedure reported in Algorithm 3. Analogously, the *Support* module only contains the compiled procedure reported in Algorithm 4. At this point, the compiled solver is obtained by integrating together compiled modules and the *Glucose* SAT solver.

We are now ready to describe how a program instance F is solved by the compiled solver. Let $F = \{d(1), d(2), d(3)\}$, the *Generator* module takes F as input and generates the subset of relevant

Algorithm 4 Compiled *Support* for $b(X,Y) \leftarrow a(X), a(Y)$.

Input: An atom t, an interpretation I, a set of literal R

```
1  begin
2      if pred(t) = "b"
3          x := t[0]
4          y := t[1]
5          if ~a(x) ∉ I
6              if ~a(y) ∉ I
7                  return ⊥
8              else R := R ∪ {a(y)}
9          else R := R ∪ {a(x)}
10     return ⊤
```

atoms needed to compute an answer set of $\Pi \cup F$, that are:

$$At = \begin{cases} a(1), a(2), a(3) \\ na(1), na(2), na(3) \\ c(1), c(2), c(3) \\ nc(1), nc(2), nc(3) \\ b(1,1), b(2,1), b(3,1) \end{cases}$$

Note that only a part of the domain of predicate b is generated. More precisely, only the relevant part that appears in the constraint $\leftarrow b(X,1), c(X)$.

Generated atoms in At become decision variables of the SAT solver, which starts the CDCL to build an answer set of $\Pi \cup F$ starting from an empty interpretation I.

As first step, the SAT solver adds F to I and then makes the first heuristic choice. Let us suppose that the SAT solver chooses $a(1)$ and adds it to I. At this point the *Propagator* module computes inferences coming from $\Pi \setminus L$. More precisely, $na(1)$ is derived as false, from rule $na(X) \leftarrow d(X),\ {\sim}a(X)$, and so ${\sim}na(1)$ is added to I. Since there are no further derivations from rules in $\Pi \setminus L$ then the SAT solver calls the post-propagator and in particular *Consequence* and *Support* modules. The *Consequence* module takes as input I and evaluates the rule $r : b(X,Y) \leftarrow a(X), a(Y)$. Since $a(1)$ is true, then the *Consequence* module is able to find an instantiation of r having a true body w.r.t. I, that is $b(1,1) \leftarrow a(1), a(1)$. Thus, $b(1,1)$ is inferred as true. On the other hand, the *Support* module searches for a supporting rule for atoms $b(2,1)$ and $b(3,1)$. By proceeding in a top-down fashion (i.e., substituting first $b(2,1)$ and then $b(3,1)$ to the head of r), the *Support* module builds the instantiations of r of the form $b(2,1) \leftarrow a(2), a(1)$, and $b(3,1) \leftarrow a(3), a(1)$ that have an undefined body w.r.t. I. Thus, the *Support* module does not derive any of $b(2,1)$ and $b(3,1)$ since they can still be supported w.r.t. I. Therefore, the post-propagator infers only $b(1,1)$ that matches $\Phi_L(I) = \{b(1,1)\}$. At this point, the SAT solver propagates the inferences of the post-propagator that cause further inferences from the constraint $\leftarrow b(X,1), c(X)$ and the rule $nc(X) \leftarrow d(X), {\sim}c(X)$. In particular, $c(1)$ is inferred as false and $nc(1)$ is inferred as true. As a result, ${\sim}c(1)$ and $nc(1)$ are added to I. Since no further inferences are computed by the *Propagator* module and the post-propagator one, then the SAT solver makes a new choice. Let $a(2)$ be that choice, then $a(2)$ is added to I. As in the previous case, the *Propagator* module infers ${\sim}na(2)$ from rules in $\Pi \setminus L$ and then the SAT solver invokes the post-propagator. Similarly to the previous call, the *Consequence* module derives $b(2,1), b(1,2), b(2,2)$; whereas the *Support* module checks whether $b(3,1)$ can be supported. Since $a(3)$ is undefined w.r.t. I and $a(1)$ is true, then $b(3,1)$ can still be supported. In this case, we can observe that $b(1,2)$ and $b(2,2)$ are lazily discovered atoms since they were not generated upfront but they must be true in I in order to let it become an answer set. On the other hand, $b(2,1)$ is known to the SAT solver and so it is propagated causing further inferences from rules in $\Pi \setminus L$. In particular, from constraint $\leftarrow b(X,1), c(X)$ and rule $nc(X) \leftarrow d(X), {\sim}c(X)$, ${\sim}c(2)$ and $nc(2)$ are added to I. At this point, the SAT solver makes another choice, for example ${\sim}a(3)$. As a consequence, the propagator module derives $na(3)$ as true and so $na(3)$ is added to I. Afterwards, the SAT solver calls the post-propagator. Here,

the *Consequence* module does not infer further atoms; while the *Support* module is not able to find a supporting rule for $b(3,1)$ and so $\sim b(3,1)$ is inferred. The inference of $\sim b(3,1)$ does not cause any derivations and so the SAT solver makes the last choice. Assume that $c(3)$ is chosen as true, then the Propagator module infers $\sim nc(3)$ and the SAT solver stops since all decision variables are assigned. Finally, $b(1,3), b(2,3), b(3,2), b(3,3)$ that are never discovered by the *Consequence* module are inferred as false and the compiled solver returns the following answer set of $\Pi \cup F$:

$$I = \left\{ \begin{array}{l} d(1),\ d(2),\ d(3), \\ a(1),\ a(2),\ \sim a(3), \\ \sim na(1),\ \sim na(2),\ na(3), \\ \sim c(1),\ \sim c(2),\ c(3), \\ nc(1),\ nc(2),\ \sim nc(3), \\ b(1,1),\ b(2,1),\ \sim b(3,1), \\ b(1,2),\ b(2,2),\ \sim b(3,2), \\ \sim b(1,3),\ \sim b(2,3),\ \sim b(3,3) \end{array} \right.$$

5. Preliminary Experiments

For measuring the impact of lazy atom discovery on PROASP performance, we conducted an empirical evaluation over synthetic benchmarks that were considered by Dodaro et al. (2023) for highlighting strengths and weaknesses of the compiled solver.

The first benchmark we considered is Synth1 on which compilation-based solvers showed significant overhead. More precisely, the compiled program in this benchmark is the following:

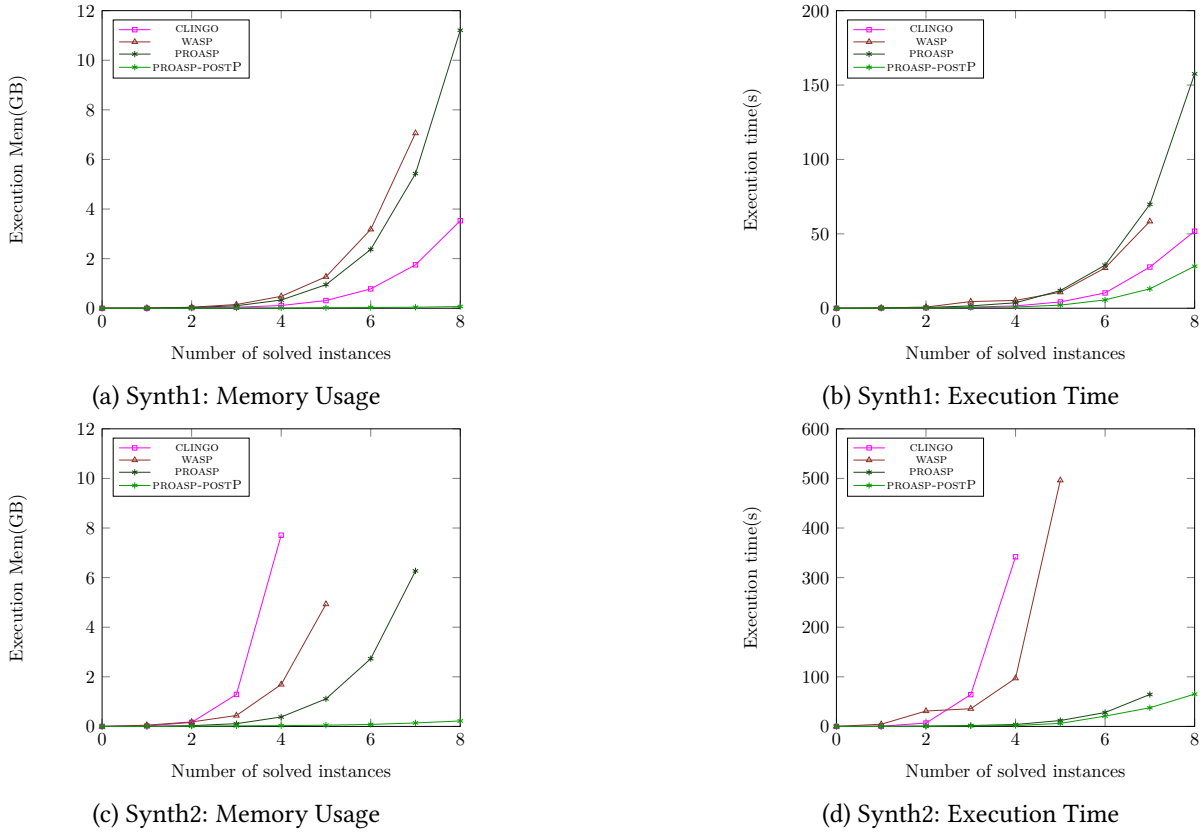
$$\begin{array}{lll} r_1: & a_1(X,Y,Z,W) & :- d(X),\ d(Y),\ d(Z),\ d(W),\ not\ na_1(X,Y,Z,W) \\ r_2: & na_1(X,Y,Z,W) & :- d(X),\ d(Y),\ d(Z),\ d(W),\ not\ a_1(X,Y,Z,W) \\ r_3: & a_2(X,Y,Z,W) & :- d(X),\ d(Y),\ d(Z), d(W),\ not\ na_2(X,Y,Z,W) \\ r_4: & na_2(X,Y,Z,W) & :- d(X),\ d(Y),\ d(Z),\ d(W),\ not\ a_2(X,Y,Z,W) \\ r_5: & b(X,Y,Z) & :- a_1(X,\_,\_,Y), a_2(Y,\_,\_,Z) \\ r_6: & c(X,Y,Z) & :- a_1(X,\_,\_,Y), a_2(Y,\_,\_,Z) \\ r_7: & & :- b(X_1,Y,Z), c(Z,Y,X_2) \end{array}$$

Synth1 is made of two four-arity predicates guesses, two rules that compute a join over the guessed predicates and one constraint for trashing a portion of the guessed models. For this benchmark we consider a lazy program split that contains rules r_5, r_6 and r_7. By compiling such a split with our technique into post-propagators, we are able to lazily generate atoms for predicates b and c. The corresponding version of PROASP without lazy atom discovery is forced to generate all b and c atoms plus the auxiliary atoms produced by the PROASP rewriting described in [42].

Conversely, the second benchmark we considered is *Synth2* on which compilation resulted to be very effective. The program considered in Synth2 benchmark is the following:

$$\begin{array}{lll} r_1: & a_1(X,Y) & :- d(X),\ d(Y),\ not\ na_1(X,Y) \\ r_2: & na_1(X,Y) & :- d(X),\ d(Y),\ not\ a_1(X,Y) \\ \ldots & & \\ r_{11}: & a_6(X,Y) & :- d(X),\ d(Y),\ not\ na_6(X,Y) \\ r_{12}: & na_6(X,Y) & :- d(X),\ d(Y),\ not\ a_6(X,Y) \\ r_{13}: & b(X_1,\ldots,X_7) & :- a_1(X_1,X_2), a_2(X_2,X_3), \ldots, a_6(X_6,X_7) \\ r_{14}: & c(X_1,\ldots,X_7) & :- a_1(X_1,X_2), a_2(X_2,X_3), \ldots, a_6(X_6,X_7) \\ r_{15}: & & :- b(\_,\_,\_,W,X,Y,Z), c(Z,Y,X,W,\_,\_,\_) \end{array}$$

As it can be observed, it has a structure that is similar to *Synth1*, except that this time there is a guess over six two-arity predicates. Also for this benchmark we consider a split which consists of the two rules that define a join over the result of the guessed predicated (namely r_{13} and r_{14}) and the constraint



(a) Synth1: Memory Usage

(b) Synth1: Execution Time

(c) Synth2: Memory Usage

(d) Synth2: Execution Time

Figure 4: Synthetic benchmarks

that trashes part of the guessed models. Just like the previous benchmark, the version of PROASP with post-propagators able to lazily generate atoms of predicates b and c. All the experiments were run 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 to 12GB and 1200 seconds, respectively. Each system was limited to run on a single core.

Obtained results showed that our technique is capable of improving PROASP performance. In particular, the introduction of lazy atom discovery in the *synth2* benchmark allowed to improve already good results obtained by PROASP by allowing lazy generation of atoms for predicates b and c. Concerning *Synth1* instead, we can see that the version of PROASP with lazy atom discovery was capable not only to improve PROASP performance, but to make the system achieve better results w.r.t. Ground & Solve systems that were winners for the benchmark. This is due to two reasons: the first is that atoms for predicates b and c were lazily generated, and the second is that the compilation of rules defining b and c into post-propagator allowed to avoid the PROASP rewriting that introduces auxiliary atoms.

6. Related work

Tackling the grounding bottleneck in ASP evaluation is a crucial problem which inspired a significant body of literature. Existing proposals tackled the grounding problem in different ways and each one present its strengths and limitations. Hybrid formalisms, such as Constraints Answer Set Programming (CASP) [52, 31, 53, 30, 32], ASP Modulo Theories [25], and HEX programs [54], integrate ASP systems with external sources of computation in order to shift the problem out of the ASP systems realm. Conversely, Lazy grounding systems [33, 34, 35, 36, 37], materialize ground rules only when their body is satisfied during solving. Albeit promising, these systems follow a novel solving strategy that still does not match the performance of state-of-the-art systems [55]. The techniques presented in this paper, instead, keeps the well-matured CDCL approach at solving stage but shifts the evaluation of program

splits without materializing ground rules.

Complexity-driven program rewritings [38, 56] tackle the grounding bottleneck by translating programs into different formalisms, such as propositional epistemic logic programs, or into disjunctive programs. Such translation can be exponential in the worst case but the obtained programs feature smaller ground instantiations.

Compilation-based approaches [39, 40, 41, 42], are at the core of the technique presented in this paper. These approaches mitigate the grounding bottleneck by translating non-ground rules into specialized procedures that simulate inferences avoiding (entirely) the grounding phase [41, 42, 43]. As it has already been discussed in previous sections, these techniques present some overhead when the number of generated atoms increases, but by enabling lazy atom discovery, it is possible to reduce such overhead.

7. Conclusions

ASP systems based on the Ground & Solve approach are affected by the Grounding Bottleneck. In the last years, proposed compilation-based ASP solvers were able to effectively address this issue. However, existing compilation-based systems still require the upfront materialization of ground atoms, which can be detrimental in certain cases, as noted by [42]. In this work, we describe how to extend the compilation-based technique to avoid the generation of all atoms in advance. We implemented this idea on top of PROASP. Achieved results showed substantial performance gains. As future work, we plan to extend our technique and experiments, and also to study how to support recursive programs that at the moment cannot be handled by PROASP.

Acknowledgments

This work was supported by the Italian Ministry of Industrial Development (MISE) under project EI-TWIN n. F/310168/05/X56 CUP B29J24000680005; and by the Italian Ministry of Research (MUR) under projects: PNRR FAIR - Spoke 9 - WP 9.1 CUP H23C22000860006, and Tech4You CUP H23C22000370006.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

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

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

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

[4] T. C. Son, E. Pontelli, M. Balduccini, T. Schaub, Answer set planning: A survey, Theory Pract. Log. Program. 23 (2023) 226–298.

[5] C. Dodaro, G. Galatà, M. Maratea, I. Porro, Operating room scheduling via answer set programming, in: AI*IA, volume 11298 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 445–459.

[6] M. Cardellini, P. D. Nardi, C. Dodaro, G. Galatà, A. Giardini, M. Maratea, I. Porro, A two-phase ASP encoding for solving rehabilitation scheduling, in: RuleML+RR, volume 12851 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 111–125.

[7] G. Grasso, S. Iiritano, N. Leone, V. Lio, F. Ricca, F. Scalise, An asp-based system for team-building in the gioia-tauro seaport, in: PADL, volume 5937 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 40–42.

[8] E. Erdem, V. Patoglu, Applications of ASP in robotics, Künstliche Intell. 32 (2018) 143–149.
[9] B. Cuteri, K. Reale, F. Ricca, A logic-based question answering system for cultural heritage, volume 11468 of *LNCS*, Springer, 2019, pp. 526–541.
[10] A. Mitra, P. Clark, O. Tafjord, C. Baral, Declarative question answering over knowledge bases containing natural language text with answer set programming, AAAI Press, 2019, pp. 3003–3010.
[11] P. Schüller, Modeling variations of first-order horn abduction in answer set programming, Fundam. Informaticae 149 (2016) 159–207.
[12] Z. Yang, A. Ishay, J. Lee, Coupling large language models with logic programming for robust and general reasoning from text, Association for Computational Linguistics, 2023, pp. 5186–5219.
[13] T. Eiter, M. Fink, G. Greco, D. Lembo, Repair localization for query answering from inconsistent databases, ACM Trans. Database Syst. 33 (2008) 10:1–10:51.
[14] M. Arenas, L. E. Bertossi, J. Chomicki, Consistent query answers in inconsistent databases, ACM Press, 1999, pp. 68–79.
[15] M. Manna, F. Ricca, G. Terracina, Taming primary key violations to query large inconsistent data via ASP, Theory Pract. Log. Program. 15 (2015) 696–710.
[16] G. Amendola, C. Dodaro, N. Leone, F. Ricca, On the application of answer set programming to the conference paper assignment problem, in: G. Adorni, S. Cagnoni, M. Gori, M. Maratea (Eds.), AI*IA 2016: Advances in Artificial Intelligence - XVth International Conference of the Italian Association for Artificial Intelligence, Genova, Italy, November 29 - December 1, 2016, Proceedings, volume 10037 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 164–178. URL: https://doi.org/10.1007/978-3-319-49130-1_13. doi:10.1007/978-3-319-49130-1_13.
[17] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (2016) 53–68.
[18] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surv. 33 (2001) 374–425.
[19] M. Gebser, N. Leone, M. Maratea, S. Perri, F. Ricca, T. Schaub, Evaluation techniques and systems for answer set programming: a survey, in: IJCAI, ijcai.org, 2018, pp. 5450–5456.
[20] A. A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E. C. Teppan, Industrial applications of answer set programming, Künstliche Intell. 32 (2018) 165–176.
[21] G. Francescutto, K. Schekotihin, M. M. S. El-Kholany, Solving a multi-resource partial-ordering flexible variant of the job-shop scheduling problem with hybrid ASP, volume 12678 of *LNCS*, Springer, 2021, pp. 313–328.
[22] V. Barbara, M. Guarascio, N. Leone, G. Manco, A. Quarta, F. Ricca, E. Ritacco, Neuro-symbolic AI for compliance checking of electrical control panels, Theory Pract. Log. Program. 23 (2023) 748–764.
[23] L. Müller, P. Wanko, C. Haubelt, T. Schaub, Investigating methods for aspmt-based design space exploration in evolutionary product design, Int. J. Parallel Program. 52 (2024) 59–92.
[24] D. Rajaratnam, T. Schaub, P. Wanko, K. Chen, S. Liu, T. C. Son, Solving an industrial-scale warehouse delivery problem with answer set programming modulo difference constraints, Algorithms 16 (2023) 216.
[25] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, volume 52 of *OASICS*, Schloss Dagstuhl, 2016, pp. 2:1–2:15.
[26] M. Alviano, F. Calimeri, C. Dodaro, D. Fuscà, N. Leone, S. Perri, F. Ricca, P. Veltri, J. Zangari, The ASP system DLV2, volume 10377 of *LNCS*, Springer, 2017, pp. 215–221.
[27] B. Kaufmann, N. Leone, S. Perri, T. Schaub, Grounding and solving in answer set programming, AI Mag. 37 (2016) 25–32.
[28] J. Marques-Silva, I. Lynce, S. Malik, Conflict-driven clause learning SAT solvers, volume 336 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2021, pp. 133–182.
[29] F. Calimeri, M. Gebser, M. Maratea, F. Ricca, Design and results of the fifth answer set programming competition, Artif. Intell. 231 (2016) 151–181.
[30] M. Ostrowski, T. Schaub, ASP modulo CSP: the clingcon system, Theory Pract. Log. Program. 12 (2012) 485–503.

[31] M. Balduccini, Y. Lierler, Constraint answer set solver EZCSP and why integration schemas matter, Theory Pract. Log. Program. 17 (2017) 462–515.
[32] B. Susman, Y. Lierler, Smt-based constraint answer set solver EZSMT (system description), volume 52 of *OASIcs*, Schloss Dagstuhl, 2016, pp. 1:1–1:15.
[33] J. Bomanson, T. Janhunen, A. Weinzierl, Enhancing lazy grounding with lazy normalization in answer-set programming, AAAI Press, 2019, pp. 2694–2702.
[34] C. Lefèvre, P. Nicolas, The first version of a new ASP solver : Asperix, volume 5753 of *LNCS*, Springer, 2009, pp. 522–527.
[35] Y. Lierler, J. Robbins, Dualgrounder: Lazy instantiation via clingo multi-shot framework, volume 12678 of *LNCS*, Springer, 2021, pp. 435–441.
[36] A. D. Palù, A. Dovier, E. Pontelli, G. Rossi, GASP: answer set programming with lazy grounding, Fundam. Informaticae 96 (2009) 297–322.
[37] A. Weinzierl, Blending lazy-grounding and CDNL search for answer-set solving, volume 10377 of *LNCS*, Springer, 2017, pp. 191–204.
[38] V. Besin, M. Hecher, S. Woltran, Body-decoupled grounding via solving: A novel approach on the ASP bottleneck, in: IJCAI, ijcai.org, 2022, pp. 2546–2552.
[39] B. Cuteri, C. Dodaro, F. Ricca, P. Schüller, Partial compilation of ASP programs, Theory Pract. Log. Program. 19 (2019) 857–873. URL: https://doi.org/10.1017/S1471068419000231. doi:10.1017/S1471068419000231.
[40] B. Cuteri, C. Dodaro, F. Ricca, P. Schüller, Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators, in: IJCAI, ijcai.org, 2020, pp. 1688–1694.
[41] G. Mazzotta, F. Ricca, C. Dodaro, Compilation of aggregates in ASP systems, AAAI Press, 2022, pp. 5834–5841.
[42] C. Dodaro, G. Mazzotta, F. Ricca, Compilation of tight ASP programs, volume 372 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2023, pp. 557–564. URL: https://doi.org/10.3233/FAIA230316. doi:10.3233/FAIA230316.
[43] C. Dodaro, G. Mazzotta, F. Ricca, Blending grounding and compilation for efficient ASP solving, in: Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR, 2024.
[44] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2012. URL: https://doi.org/10.2200/S00457ED1V01Y201211AIM019. doi:10.2200/S00457ED1V01Y201211AIM019.
[45] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, Asp-core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309.
[46] E. Erdem, V. Lifschitz, Tight logic programs, Theory Pract. Log. Program. 3 (2003) 499–518.
[47] K. L. Clark, Negation as failure, in: Logic and Data Bases, Advances in Data Base Theory, Plemum Press, New York, 1977, pp. 293–322.
[48] M. Fitting, A deterministic prolog fixpoint semantics, J. Log. Program. 2 (1985) 111–118. URL: https://doi.org/10.1016/0743-1066(85)90014-7. doi:10.1016/0743-1066(85)90014-7.
[49] M. Gebser, R. Kaminski, A. König, T. Schaub, Advances in *gringo* series 3, in: LPNMR, volume 6645 of *LNCS*, Springer, 2011, pp. 345–351.
[50] F. Calimeri, D. Fuscà, S. Perri, J. Zangari, I-DLV: the new intelligent grounder of DLV, Intelligenza Artificiale 11 (2017) 5–20. URL: https://doi.org/10.3233/IA-170104. doi:10.3233/IA-170104.
[51] C. Dodaro, Design and implementation of modern CDCL ASP solvers, Intelligenza Artificiale 18 (2024) 239–259.
[52] R. A. Aziz, G. Chu, P. J. Stuckey, Stable model semantics for founded bounds, Theory Pract. Log. Program. 13 (2013) 517–532.
[53] B. D. Cat, M. Denecker, M. Bruynooghe, P. J. Stuckey, Lazy model expansion: Interleaving grounding with search, J. Artif. Intell. Res. 52 (2015) 235–286.
[54] T. Eiter, C. Redl, P. Schüller, Problem solving using the HEX family, in: Computational Models of Rationality, College Publications, 2016, pp. 150–174.

[55] A. Weinzierl, R. Taupe, G. Friedrich, Advancing lazy-grounding ASP solving techniques - restarts, phase saving, heuristics, and more, Theory Pract. Log. Program. 20 (2020) 609–624.
[56] A. Beiser, M. Hecher, K. Unalan, S. Woltran, Bypassing the ASP bottleneck: Hybrid grounding by splitting and rewriting, in: IJCAI, ijcai.org, 2024, pp. 3250–3258.