Enhancing compilation-based ASP solving

Andrea Cuteri

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

Abstract

Answer Set Programming(ASP) is a logic-based formalism for knowledge representation and reasoning. State-of-the-art ASP systems that adopt the Ground & Solve approach are limited by the grounding bottleneck. This issue arises whenever the grounding produces a large propositional program that cannot be efficiently handled during solving. To overcome this limitation, compilation-based solutions have been introduced which led to the development of alternative ASP systems. However, compilation-based ASP systems generate in advance the propositional atoms required for solving the problem. As a consequence, as soon as a large number of propositional atoms is required, also compilation-based approaches exhibit overhead. To address this issue, we develop a novel compilation technique that enables lazy atom discovery during solving. Preliminary results confirmed the effectiveness of our approach.

Keywords

Answer Set Programming, Grounding Bottleneck, Compiled Propagators,

1. Introduction

Answer Set Programming (ASP) [1] is a logic-based formalism rooted in the stable model semantics [2]. Over the years, ASP has found many interesting applications ranging from planning [3] to scheduling [4, 5, 6] and many others [7]. Standard ASP systems such as *CLINGO* [8] and *DLV* [9] are based on the Ground&Solve approach [10]. The grounding step consists of eliminating variables from non-ground programs in order to obtain propositional equivalent programs. The solving step typically exploits a CDCL-like algorithm [11] for evaluating ground programs. Depending on the considered problem, computing the equivalent variable-free program might be infeasible. This problem is known as the grounding bottleneck [12]. Tackling this problem led to the development of alternative ASP Solvers such as compilation-based [13, 14, 15, 16], hybrid [17, 18], complexity-driven [19, 20], and lazy grounding [21, 22] systems. In particular, compilation-based solvers revealed to be among the most promising types of systems. Compilation-based solvers translate non-ground (sub)programs into ad-hoc procedures capable of mimicking the ground counterpart of an ASP program during solving without explicitly materializing it. However, available compilation-based approaches require to generate in advance the propositional atoms needed for solving the problem (i.e., computing answer sets) and whenever this set of atoms becomes large, these systems exhibit overhead [15]. The first part of my research has been focused on overcoming this limitation of compilation-based ASP solvers. In particular, we extended the PROASP solver with a compilation technique aimed at enabling lazy atom discovery during solving instead of generating them in advance. An empirical evaluation shows that lazy atom discovery significantly improves the performance of the PROASP solver, enabling the compilation of ASP programs that were previously considered unsuitable, into lightweight and efficient compiled solvers.

2. Existing Literature

Over the last years, compilation-based solvers have been developed as a technique for overcoming the grounding bottleneck problem. The idea at the heart of compilation-based solvers is to encode non-ground rules into ad-hoc propagators which are able to simulate rule inferences during solving without

ICLP DC 2025: 21st Doctoral Consortium on Logic Programming, September 2025, Rende, Italy.
andrea.cuteri@unical.it (A. Cuteri)
0009-0000-7629-7347 (A. Cuteri)

materializing any ground instantiation. Several solutions of this type were proposed over the years, starting from systems capable of compiling only constraints [13, 14] to solutions capable of compiling entire programs [15], and up to solutions capable of mixing grounding and compilation [16]. All these techniques tackled the grounding bottleneck effectively, allowing to solve out-of-reach instances for state-of-the-art systems and remaining competitive on solving problems not affected by grounding issues. Despite the fact that such techniques completely avoid rules grounding materialization, they need to generate all propositional atoms required to compute the answer sets of ASP programs. As soon as the number of generated atoms increases, compiled solvers exhibit significant overhead [15].

Among alternative approaches, developed for overcoming the grounding bottleneck, it is worth mentioning lazy-grounding systems [21, 22, 23, 24, 25], hybrid formalism [17, 18, 26, 27], and complexity-driven rewritings [19, 20]. Lazy grounding entirely skips the grounding of the program and generates, during solving, ground rules only when their body is true w.r.t. the current interpretation. Even though these systems effectively addressed the grounding bottleneck they showed poor performance, hardly achieving comparable results w.r.t. state-of-the-art ASP solvers [28]. Hybrid formalisms extend the base ASP language by integrating external sources of computation. Complexity-driven program rewritings encode the original program into a different formalism, such as propositional epistemic logic programs, or into a disjunctive program generating smaller ground programs. Even though these encodings are in general easier to evaluate and have shown promising results in some practical scenarios [19], they can be exponential in the worst case.

3. Preliminaries

In this section some basic notions about ASP syntax and semantics are given which will be used for introducing compilation-bases ASP solving.

ASP Syntax. A *term* can be either constant or a variable. *Constants* are strings starting with lower case letters or positive integers while *variables* are terms starting with upper case letters. An *atom* a is an expression of the form $p(t_1, ..., t_n)$ where p is a predicate of arity n and $t_1, ..., t_n$ are terms. A predicate of arity zero is said to be propositional. A *literal* l is an atom a, or its negation $not\ a$ where not represents negation as failure. A literal l is *positive* if it is of the form a, *negative* otherwise. The negative counterpart of literal l is denoted as $\bar{l}$. If all the terms of l are constants l is said to be a *ground* literal. A propositional literal is always ground. Given a literal l, $pred(l)$ denotes the predicate name of l. A rule is an expression of the form: $a \leftarrow\ l_1, ..., l_n.$ where a is a positive literal referred to as head of the rule, while $l_1,\ ...,\ l_k$ is a conjunction of literals referred to as body of the rule. $H(r)$ represents the set of atoms that appear in the head of the rule(it has a size of at most one). $B(r)$ represents the set of literals that appear in the body of the rule. A rule r is *ground* if both $H(r)$ and $B(r)$ are ground. $B(r)$ is ground if all the literals of $B(r)$ are ground literals. Similarly, $H(r)$ is ground if its atom is ground. An empty set of literals is trivially ground. A rule r is a *fact* if $B(r) = \emptyset$, it is a *constraint* if $H(r) = \emptyset$. A program Π is a set of rules. Given an ASP expression ϵ (a program, a rule etc.), $pred(\epsilon)$ and $vars(\epsilon)$ denote 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 appearing in Π. The *Herbrand Base*, B_Π, of program Π is the set of all possible ground atoms that can be built using predicates in Π and constants in U_Π. The ground instantiation $ground(\Pi)$ of a program Π is the union of all the ground instantiations of rules of Π. The dependency graph of a program Π, G_Π, is a directed labelled graph where the nodes are predicates appearing in Π, and the set of the edges contains a positive (resp. negative) edge (u, v) if there exists a rule $r \in \Pi$ where $u(t_1, \ldots, t_n)$ is a positive (resp. negative) literal in the body of r and $v(t_1, \ldots, t_m)$ the atom in the head of r. The Strongly connected components (SCCs) of G_Π create a partition of predicates of Π. A program Π is said to be stratified if there is no recursion through negation. In other words, Π is stratified if inside G_Π there is no cycle that involves a negative edge.

The *ground dependency graph* of Π is a dependency graph defined as above, constructed from rules in $ground(\Pi)$. A program Π is *tight* if its dependency graph G_Π does not contain cycles, it is *locally tight* if the ground dependency graph does not contain cycles.

ASP Semantics. Given a program Π, an interpretation I is a set of literals whose atoms belong to B_Π. A literal l is *true* w.r.t. I if $l \in I$, it is *false* if $\bar{l} \in I$, it is *undefined* otherwise. An interpretation I is *inconsistent* when both a literal l and its complement $\bar{l}$ belong to I; otherwise it is *consistent*. An interpretation is said to be *total* if for each atom $a \in B_\Pi$ either a or $\overline{a}$ belongs to I, it is partial otherwise. An interpretation that is both *total* and *consistent* is a *model* of Π if inside $ground(\Pi)$ the head of each rule r is true whenever the body of r is true. Given a program Π and an interpretation I, the Gelfond-Lifschitz reduct [2] of Π w.r.t. I, denoted by Π^I is a ground program that has all the rules of Π whose body was true w.r.t. I. Moreover, each rule r of Π^I does not contain the negative literals that were true w.r.t. I in the corresponding rule r of Π. I is an *answer set*(or a *stable model*) for Π if there is no interpretation I_1 s.t. $I_1^+ \subset I^+$ and I_1^+ is a model for Π^I. A program that admits at least one stable model is *coherent*, it is incoherent otherwise.

4. Research Plan

Approaches based on compilation showed promising results by effectively addressing the grounding bottleneck and achieving better performances w.r.t. state-of-the-art systems over grounding-intensive benchmarks and yet obtaining fairly good results on benchmarks that are not grounding-intensive. Even though effective, compilation-based systems are not a silver bullet for solving all types of ASP programs. First of all, compilation-based techniques proposed so far target only a restricted class of ASP programs (i.e., tight programs). Secondly, proposed techniques present a significant overhead when the number of propositional atoms required to compute an answer set increases. Therefore, my research plan is to study whether it is possible to overcome current limitations of compilation-based ASP systems and extend compilation to the entire ASP-Core-2 standard [29]. The first research direction on which I am currently working is the development of a novel compilation-based technique which avoids upfront atom materialization and postpones atom discovery to the solving phase. In this direction, I will study novel compilation-based techniques for compiling recursive and disjunctive programs. These two classes of programs require considering more involved and even more complex propagation scenarios than those considered so far. Together with these broader classes of ASP programs, it would be interesting to investigate the possibility of leveraging compilation for the evaluation of programs with weak constraints.

5. Preliminary Results

In this first phase of the research we focused in improving compilation-based ASP solvers by defining a technique, namely *lazy atom discovery*, for reducing the amount of propositional atoms that these solvers need to generate before starting the solving phase. Such an approach has been formalized and it has been accepted for publication at JELIA 2025.

5.1. Lazy atom discovery

The PROASP system. PROASP [15] is a compilation-based solver for the class of tight ASP programs. A compiled PROASP solver is made of two modules, namely *Generator* and *Propagator* that are integrated with the *Glucose* SAT solver. The execution of a compiled PROASP solver for a program Π over an arbitrary instance F (i.e., a set of facts) consists of two steps. As a first step the compiled *Generator* module is responsible for generating atoms that are required for solving the problem. In the second step, instead, the*Glucose* SAT solver takes as input generated atoms and starts the CDCL algorithm for computing an answer set of $\Pi \cup F$. During solving, the absence of ground rules forces the SAT solver

to delegate the entire CDCL propagation phase to the compiled *Propagator* module, which derives inferences from the compiled rules (i.e., those in Π) based on the problem instance F.

PROASP with lazy atom discovery. To enable lazy atom discovery, we first identify a subprogram L of an input program Π, namely *lazy program split*, which complies to some syntactic restrictions needed to postpone its evaluation during solving. In particular, L must be a stratified subprogram such that $head(L) \cap head(\Pi \setminus L) = \emptyset$. That is, predicates appearing in the head of rules of L cannot appear in the head of any rule in $\Pi \setminus L$. Thus, inferences coming from L are deterministic and the subprogram L does not affect the compilation pipeline of PROASP for the remaining rules, i.e., $\Pi \setminus L$. Once the lazy program split $L \subseteq \Pi$ has been identified, the program $\Pi \setminus L$ follows the typical compilation process described in [15]. Conversely, L is compiled with a novel technique into a post-propagator module (i.e., lower priority propagators). This module, that from now on we will call *postP*, is responsible for simulating rules in $ground(L)$ and for lazily discovering atoms in $ground(L)$. To obtain the *postP* module, we design a compilation strategy that does two main tasks. The first is to extract additional generation rules that are compiled into the PROASP *Generator module* for generating atoms from predicates in $head(L)$ that appear in $ground(\Pi \setminus L)$. Note that atoms generated through these additional generation rules become decision variables of the solver and therefore the solver can assign a truth value to them. The second is to compile rules from L into post-propagators capable of mimicking rules in $ground(L)$. One post-propagator for each SCC of program L is compiled. Each SCC post-propagator is made of two components. A *fixpoint* procedure, for deriving true atoms from L, and a *propagateToFalse* procedure for deriving false atoms from L by checking support of atoms a over predicates in $head(L)$. The architecture of PROASP with post-propagators is illustrated in Figure 1.

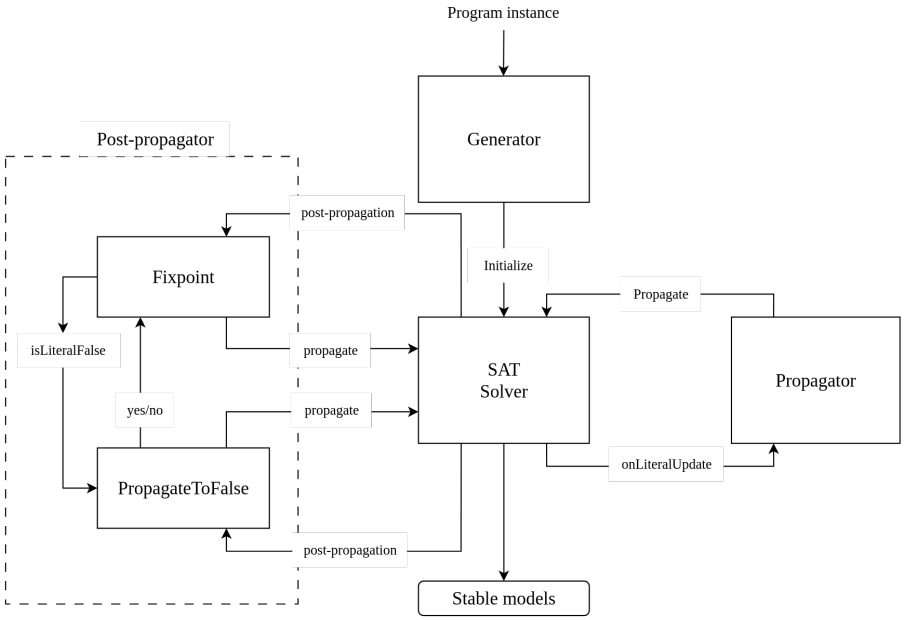


Figure 1: Extended ProASP Architecture

Example of compilation For space reasons only the result of the compilation over an example program is shown. Consider the following subprogram L:

$$r(X,Y) \leftarrow b(X,Y),\ not\ c(X).$$

$$rnf(X,Y) \leftarrow r(X,Y),\ not\ f(X,Y).$$

Algorithm 1: Fixpoint for component $\{rnf\}$

```
Input  : lits : a set of propagated literals
Output : C: a set of conflictual literals
1  begin
2      T = list(lits)
3      C = ∅
4      while T.size() > 0 do
5          l_0 = T.pop()
6          if pred(l_0) = "r" then
7              x := l_0[0]; y := l_0[1]
8              l_1 = match("f", {x, y})
9              if l_1 = null ∨ false(l_1)) then
10                 h = new tuple("rnf", {x, y}, ⊤)
11                 l_2 = match("rnf", {x, y})
12                 if not storage.has(h) ∨ undef(l_2) then
13                     storage.addTrue(h)
14                     T.push(h)
15                 else
16                     if false(h) then
17                         C = C ∪ h
18     return C
```

L is a stratified program that complies to the above restrictions and therefore it can be compiled into a *postP* module. Consider now program $\Pi \setminus L$ defined as follows:

$$\begin{array}{rcl} b(X,Y) & \leftarrow & d(X),\ t(X,Y)\ not\ nb(X,Y). \\ nb(X,Y) & \leftarrow & d(X),\ t(X,Y)\ not\ b(X,Y). \\ & \leftarrow & rnf(1,Z), e(Z). \end{array}$$

As already discussed, generation rules need to be extracted from $\Pi \setminus L$. In this case, only the constraint in $\Pi \setminus L$ contains a literal in $head(L)$, and thus only the following generation rule is extracted.

$$rnf(1,Z) \ \leftarrow \ e(Z).$$

The above generation rule produces all the atoms of the predicate *rnf* that appear in the constraint. As it can be observed, here atoms over predicate r are no longer materialized while the predicate rnf is partially generated (i.e., only atoms of the form $rnf(1,\_)$).

In the given example, the *postP* module is made of two SCC-propagators. We call these SCC propagators *PropR* and *PropRnf*, just like the predicates defined by the components.

During solving, the post-propagator module is invoked after PROASP propagators reach a fixed point (i.e., the SAT solver made a choice and all deterministic inferences coming from such a choice have been inferred and propagated). First, the *fixpoint* procedure is invoked for deriving true consequences in program L according to the current interpretation. Then, the *propagateToFalse* procedure is invoked for deriving atoms in L that can no longer be supported. Propagations coming from the SAT solver and the Propagator module are cached by *postP*. Such propagations are shuffled (using a watched-like structure) to SCC-propagators which will use them as starters in the *fixpoint* procedure. A starter is a literal of a rule which is fixed for iterating all ground instantiations of the rule containing such a literal. Each SCC-propagator is interested in propagations of tuples belonging to predicates that appear inside the SCC, both in the body and in the head of its rules. In the example, *PropR* would be interested in predicates $\{r, b, c\}$, while *PropRnf* would be interested in predicates $\{rnf, r, f\}$. To give an idea of how *fixpoint* is structured, we provide an example of the *fixpoint* procedure for component $\{rnf\}$ in

the pseudo-code in Algorithm 1. The example is almost trivial since only one starter is needed and the while loop will never repeat in this case. However, the goal of the example is to show the typical structure of a *fixpoint* procedure for an SCC without going into the technical details.

Once *fixpoint* completes for a given SCC, a set C of conflictual literals is returned. Whenever such a set is non-empty, the SAT solver enters the learning phase to resolve the conflict and restore consistency, if possible.

If no conflict is generated by the Fixpoint procedures, *propagateToFalse* is called. *propagateToFalse* can perform false propagation for atoms a defined in L by checking all the rules that have a as head. If the check succeeds, all rules having a as head are false and therefore *postP* propagates a to false. The evaluation of rules from L by *propagateToFalse* is made top-down by entering the rules with a as starter. During *propagateToFalse*, a non-generated atom a_1 belonging to the predicate set of some $p \in head(L)$ might be encountered. This scenario is plausible since the predicate set of p is incomplete. Note that there is a restriction on rules of L which guarantees that following a specific rule instantiation order, tuples of L that are encountered are bound (the order is: head, then predicates not defined in L and then the remaining ones). What we have to do at this point is to generate dummy tuples like a_1 to proceed with *propagateToFalse*.

To provide a meaningful example of the *propagateToFalse* procedure, we should consider the same program from the previous example in which the rule:

$$rnf(X,Y) \leftarrow r(X,Y),\ not\ f(X,Y).$$

becomes:

$$rnf(X,Y) \leftarrow a(Z),\ r(X,Z),\ not\ f(X,Y).$$

propagateToFalse for component $\{rnf\}$ is defined as described in the pseudo-code in Algorithm 2. *propagateToFalse* is also called before starting the solving phase over all atoms a s.t. $pred(a) \in head(L)$ that were generated by the *Generator* module. This preliminary call tries to propagate atoms from L to false as soon as possible. Atoms propagated to false at this step have no possibility to become true again.

Once the CDCL algorithm starts, the *propagateToFalse* procedure is invoked just after *fixpoint* completes without conflicts. The atoms that need to be checked via *propagateToFalse* are tuples (always from program L) that are true without support. These can be either true choices of the solver that did not yet find a support or tuples that have lost their support after a backjump made by the solver due to a conflict. To understand when a tuple loses its support, we keep the rule that generated it. When a supporting tuple changes truth value, all its supported tuples are marked as to-check. This is not memory-expansive as long as there are multiple rules that might generate the same tuple. Things become much worse when there is a one-to-one mapping between tuples and ground rules of Π.
propagateToFalse is both necessary for guaranteeing the correctness of *ProASP* with *postP* and a solution for anticipating false propagations in such a way that branches in the search space can be closed at a higher level.

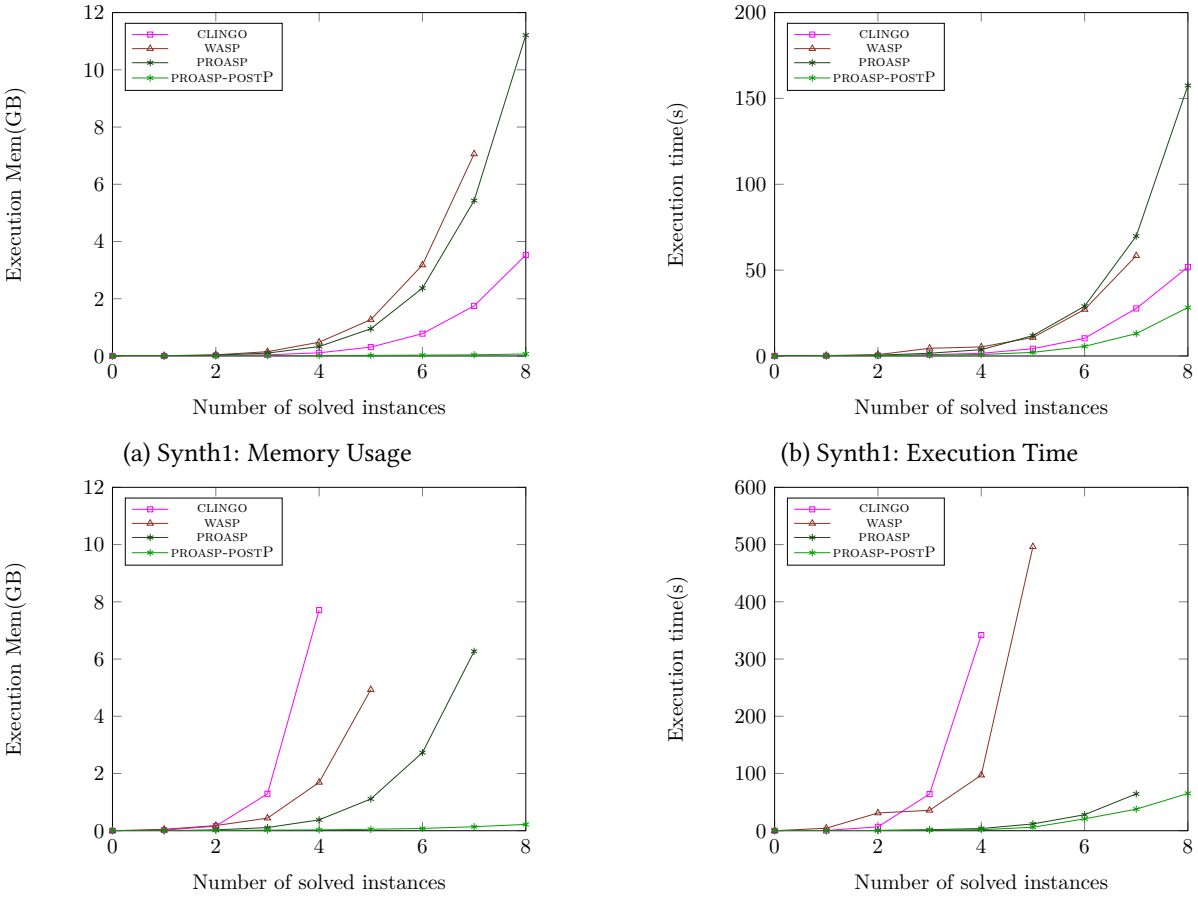
Experiments To assess the impact of lazy atom discovery on PROASP performance, we conducted an empirical evaluation over the synthetic benchmarks that were considered in [15] for highlighting strengths and weaknesses of the solver. Experiments were run on an Intel(R) Xeon(R) CPU E5-4610 v2 @ 2.30GHz 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 performances. The introduction of lazy atom discovery in the *Synth2* benchmark allowed to improve already good results for PROASP. Concerning *Synth1*, instead we can see that the version of PROASP with lazy atom discovery was capable of achieving better results than Ground & Solve systems that on this benchmark were the winners of the comparison.

Algorithm 2: Propagate to false for component {rnf}

```
Input  : a - The atom to propagate, reason - propagation reason
Output : canPropagate : propagation can be done with reason or not
1  begin
2      l_0 = a
3      if predicate(l_0) = "rnf" then
4          x := l_0[0]
5          y := l_0[1]
6          undefA = matchUndef("a", {})
7          if undefA.size() ≠ 0 then
8              reason.clear()
9              storage.saveUndefSupport(l_0, undefA[0])
10             return ⊥
11         else
12             falseA := matchFalse("a", {})
13             forall l_1 in falseA do
14                 reason.push(l_1)
15             trueA := matchTrue("a", {})
16             forall l_1 in trueA do
17                 z := l_1[0]
18                 l_2 = match("r", {x, z})
19                 if undef(l_2) then
20                     storage.saveUndefSupport(l_0, l_2)
21                     return ⊥
22                 if l_2 = null then
23                     l_2 = new tuple("r", {x, z}, unk)
24                 canProp = propR.propagateToFalse(l_2, reason)
25                 if not canProp then
26                     return ⊥
27                 else
28                     l_3 = match("f", {x, y})
29                     if true(l_3) then
30                         reason.push(l_3)
31                     if undef(l_3 ∨ false(l_3) then
32                         reason.clear()
33                         storage.saveUndefSupport(l_0, l_3)
34                         return ⊥
35             return⊤
```

6. Conclusions

Traditional ASP systems based on the Ground & Solve approach are affected by the grounding bottleneck which limits the applicability of ASP to many real world scenarios. Recently, compilation-based ASP solvers have proven effective in addressing this issue. Nevertheless, existing compilation-based systems still require the upfront materialization of ground atoms, which can be detrimental in certain cases, as noted by [15]. In this work, we introduced a compilation-based technique that avoids the generation of all atoms in advance. We implemented these techniques on top of PROASP. Achieved results showed substantial performance gains. As future work, we plan to extend our technique to support recursive programs that, at the moment, cannot be handled by PROASP.



(a) Synth1: Memory Usage

(b) Synth1: Execution Time

(c) Synth2: Memory Usage

(d) Synth2: Execution Time

Figure 2: Synthetic benchmarks

7. 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] T. C. Son, E. Pontelli, M. Balduccini, T. Schaub, Answer set planning: A survey, TPLP 23 (2023) 226–298.

[4] C. Dodaro, M. Maratea, Nurse scheduling via answer set programming, volume 10377 of *LNCS*, Springer, 2017, pp. 301–307.

[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] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (2016) 53–68.

[8] M. Gebser, R. Kaminski, A. König, T. Schaub, Advances in *gringo* series 3, in: LPNMR, volume 6645 of *LNCS*, Springer, 2011, pp. 345–351.
[9] M. Alviano, C. Dodaro, N. Leone, F. Ricca, Advances in WASP, volume 9345 of *LNCS*, Springer, 2015, pp. 40–54.
[10] B. Kaufmann, N. Leone, S. Perri, T. Schaub, Grounding and solving in answer set programming, AI Mag. 37 (2016) 25–32.
[11] 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.
[12] 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.
[13] 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.
[14] G. Mazzotta, F. Ricca, C. Dodaro, Compilation of aggregates in ASP systems, AAAI Press, 2022, pp. 5834–5841.
[15] 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.
[16] 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.
[17] M. Balduccini, Y. Lierler, Constraint answer set solver EZCSP and why integration schemas matter, TPLP 17 (2017) 462–515.
[18] 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.
[19] 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.
[20] 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.
[21] J. Bomanson, T. Janhunen, A. Weinzierl, Enhancing lazy grounding with lazy normalization in answer-set programming, AAAI Press, 2019, pp. 2694–2702.
[22] C. Lefèvre, P. Nicolas, The first version of a new ASP solver : Asperix, volume 5753 of *LNCS*, Springer, 2009, pp. 522–527.
[23] Y. Lierler, J. Robbins, Dualgrounder: Lazy instantiation via clingo multi-shot framework, volume 12678 of *LNCS*, Springer, 2021, pp. 435–441.
[24] A. D. Palù, A. Dovier, E. Pontelli, G. Rossi, GASP: answer set programming with lazy grounding, Fundam. Informaticae 96 (2009) 297–322.
[25] A. Weinzierl, Blending lazy-grounding and CDNL search for answer-set solving, volume 10377 of *LNCS*, Springer, 2017, pp. 191–204.
[26] M. Ostrowski, T. Schaub, ASP modulo CSP: the clingcon system, TPLP 12 (2012) 485–503.
[27] 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.
[28] A. Weinzierl, R. Taupe, G. Friedrich, Advancing lazy-grounding ASP solving techniques - restarts, phase saving, heuristics, and more, TPLP 20 (2020) 609–624.
[29] 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.