Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning
Main Track

Blending Grounding and Compilation for Efficient ASP Solving

Carmine Dodaro , Giuseppe Mazzotta , Francesco Ricca
University of Calabria
carmine.dodaro@unical.it giuseppe.mazzotta@unical.it francesco.ricca@unical.it

Abstract

Answer Set Programming (ASP) is a widely recognized formalism for Knowledge Representation and Reasoning. Traditional ASP systems, that employ the ground and solve architecture, are subject to the grounding bottleneck (i.e., variable-elimination can exhaust all computational resources). Compilation-based approaches have recently demonstrated how grounding can be effectively bypassed by compiling rules into propagators that simulate them. However, compiling an entire ASP program is not always advantageous. In this paper, we present both a program rewriting technique and an algorithm for the compilation of grounding that allow for unrestricted blending of grounding and compilation. We implement these techniques in a hybrid ASP system that compares favourably with state-of-the-art ASP solvers on established benchmarks.

1 Introduction

Answer Set Programming (ASP) (Brewka, Eiter, and Truszczynski 2011; Gelfond and Lifschitz 1991) is a well-known formalism for Knowledge Representation and Reasoning. ASP has been applied in several areas (Erdem, Gelfond, and Leone 2016), some notable examples are Planning (Son et al. 2023), Scheduling (Dodaro and Maratea 2017; Cardellini et al. 2023), Robotics (Erdem and Patoglu 2018), Natural Language Processing (Cuteri, Reale, and Ricca 2019; Mitra et al. 2019; Schüller 2016; Yang, Ishay, and Lee 2023), and Databases (Eiter et al. 2008; Arenas, Bertossi, and Chomicki 1999; Manna, Ricca, and Terracina 2015). Notably, ASP is the core technology in many real-world applications of industrial interest (Francescutto, Schekotihin, and El-Kholany 2021; Barbara et al. 2023; Müller et al. 2024; Rajaratnam et al. 2023).

The applicability of ASP is mostly due to two features: a comparatively expressive language that can model hard problems (Dantsin et al. 2001), and the availability of efficient implementations (Gebser et al. 2018). On the one hand, the language of ASP has established roots in the stable model semantics (Gelfond and Lifschitz 1991; Lifschitz 2010) and a standardized syntax (Calimeri et al. 2020). On the other hand, the enhancement of ASP systems is still a compelling research area, as system performance is often fundamental in the development of effective applications (Gebser et al. 2018).

Traditional ASP systems are based on the Ground&Solve approach (Kaufmann et al. 2016). In few words, the input program is first "grounded" to compute a variable-free equivalent propositional program; then, the grounded program is "solved" by employing a CDCL-like algorithm (Marques-Silva, Lynce, and Malik 2021) that computes its answer sets. Ground&Solve systems such as CLINGO (Gebser et al. 2016) and DLV (Alviano et al. 2017) are serving honorably the cause. However, they intrinsically suffer from the so-called *grounding bottleneck*, i.e., getting rid of variables already consumes all the computational resources (i.e., time and/or space) in several cases of practical interest (Calimeri et al. 2016; Ostrowski and Schaub 2012).

The grounding bottleneck (Gebser et al. 2018) has been approached from several perspectives. These include hybrid formalisms (Balduccini and Lierler 2017; Gebser et al. 2016; Ostrowski and Schaub 2012; Susman and Lierler 2016), *lazy grounding* architectures (Bomanson, Janhunen, and Weinzierl 2019; Lefèvre and Nicolas 2009; Lierler and Robbins 2021; Palù et al. 2009; Weinzierl 2017), complexity-driven program rewritings (Besin, Hecher, and Woltran 2023; Besin, Hecher, and Woltran 2022), propagators and program compilation (Cuteri et al. 2019; Cuteri et al. 2020; Mazzotta, Ricca, and Dodaro 2022; Dodaro, Mazzotta, and Ricca 2023). Hybrid approaches basically circumvent the problem of ASP systems by expanding the language with novel constructs and connecting ASP systems with external sources of computation. Lazy grounding techniques perform grounding during the search, aiming to prevent computing unnecessary instantiations; however, they have yet to match the performance of state-of-the-art systems (Weinzierl, Taupe, and Friedrich 2020) in common cases. Program rewriting methods (Besin, Hecher, and Woltran 2022; Besin, Hecher, and Woltran 2023) tackle the problem by transforming the original normal program into a different form, such as targeting a different formalism like propositional epistemic logic programs or generating a smaller ground disjunctive program. The programs resulting from the transformation are easier to evaluate, although the translation process can be exponential in the worst case (yet highly promising in practical scenarios) (Besin, Hecher, and Woltran 2022). Compilation-based approaches demonstrated that the grounding can be skipped in some relevant

cases by compiling rules in propagators that simulate rules. The first iterations of compilation-based approaches were effective in compiling subprograms acting as constraints (Cuteri et al. 2020; Mazzotta, Ricca, and Dodaro 2022). More recently, the PROASP system (Dodaro, Mazzotta, and Ricca 2023) demonstrated that it is possible to devise a compiler also for rules "generating" answer sets (i.e., involving non-stratified negation). In PROASP, a tight (Erdem and Lifschitz 2003) non-ground (aggregate-free) input program is first pre-processed by applying a rewriting encompassing program completion (Clark 1977) and normalization (i.e., it produces rules of two kinds). Then, the compiler generates code for both Herbrand base generation and rule propagators. That code is injected in the CDCL solver GLUCOSE (Audemard and Simon 2009) to initialize variables and simulate the presence of ground rules, respectively. In this way, the PROASP compiler produces a solver specific for the non-ground program in input that needs no grounder. Clearly, PROASP implements an approach that is basically at the antipodes of Ground&Solve, since it requires that all rules of a program are compiled in propagators.

Empirical evidence shows that compiling all rules of an ASP program does not pay off in all cases w.r.t. traditional approaches (Mazzotta, Ricca, and Dodaro 2022). This is not surprising. On the one hand, it is well-established that there is no free lunch in ASP solving (Gebser, Maratea, and Ricca 2020), i.e., no single algorithm is the best in all cases; on the other hand, the following example makes it more intuitive that the same principle also applies to the choice between grounding and compilation. First, consider the case of a *small groundable* instance of a program modelling a complex problem P. It is likely that a highly optimized solver like CLINGO is the best choice to solve it. However, when faced with a large, *non-groundable* instance of the same problem P, opting for PROASP might be a wiser choice. Now, what about intermediate cases? Under typical operational scenarios, it is plausible that certain rules of the program can be efficiently grounded and, thus, are amenable to be processed using the traditional architecture; whereas, for the remaining part of the program, which is made of rules that are subject to the grounding bottleneck, a compilation-based system could offer a viable solution for their evaluation. This latter case (which is more common in practice than one might expect) cannot be approached in the best possible way with current state-of-the-art ASP systems, that either ground or compile everything.

In this paper, we focus on this latter scenario. We build on the ground of PROASP and enhance its compilation-based architecture to seamlessly blend grounding and compilation. This novel approach allows for an unrestricted blending of both methods, and aims at achieving superior performance by combining the advantages of both approaches. Specifically, to achieve our objectives, we have extended PROASP in multiple directions providing the following contributions:

1. **A program rewriting technique** that rearranges the program so that grounded rules (no matter if they are involved in the same negative cycles or not) can coexist with those that will be handled by propagators, and extends the PROASP one also to handle aggregates.
2. **A compilation algorithm for grounding** that processes non-ground rules to generate specific imperative code that performs variable elimination.
3. **A hybrid ASP solver** that extends PROASP to support in the same system both grounding and compilation of ASP programs with aggregates.
4. **An empirical analysis** that assesses the behaviour of the new approach and compares it with state-of-the-art alternatives on well-known ASP benchmarks.

In our approach, users can distinguish between the rules of the ASP program that need to be grounded and those that need to be compiled. The system first applies a pre-processing technique that extends the normalization of PROASP to allow grounded rules to coexist with propagators, no matter if they are involved in a recursive definition. Then, the rules to be compiled follow the usual path of PROASP, i.e., they are compiled in propagators; whereas the remaining rules are compiled in a code that performs grounding, i.e., code that generates propositional clauses that are stored in the GLUCOSE data structures. The resulting ASP system can blend compilation and grounding. Moreover, we remark that a compiled grounder has never been proposed in the literature. Indeed, the compilation was focused on the generation of propagators (i.e., code performing inference, not grounding), and the compilation of aggregates, introduced by Mazzotta, Ricca, and Dodaro (2022), was not supported by the previous version of PROASP.

The results of the experiment presented in the paper confirm that grounding and compilation can be blended effectively. Indeed, our approach can outperform state-of-the-art ASP solvers on benchmarks that are either ground-intensive or solving-intensive (or both).

2 Preliminaries

In this section we provide some basic knowledge of the language of Answer Set Programming (ASP), and we recall the main working principles of Ground&Solve and Compilation-based ASP systems, that are needed to dive into the details of the main contributions of this paper.

2.1 The Language of ASP

In ASP *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). A *standard 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 *standard literal* is either a standard atom a or its negation $not\ a$, where not represents negation as failure. Given a standard literal $l = a$ (resp. $l = not\, a$), $\bar{l}$ denotes the complement of l, that is $not\, a$ (resp. a), and $\mathcal{T}(l)$ denotes the list of terms appearing in l. A standard literal is *ground* if it does not contain any variable. A *symbolic set* is a pair for the form $\langle V : Conj \rangle$, where V is a list of variables and $Conj$ is a conjunction of standard literals. A *ground set* is a set of pairs of the form $\langle t : conj \rangle$, where t is a list of constants and $conj$ is a conjunction of ground standard literals. An *aggregate function* is an expression of the form $f(S)$, where

$f \in \{\#sum, \#count\}$ is an aggregate function symbol, and S is a symbolic or ground set. An *aggregate atom* is an expression of the form $f(S) \prec t$, where $f(S)$ is an aggregate function, $\prec \in \{\leq, <, \geq, >\}$ is a comparison operator, and t is a term called *guard*. An *atom* is either a standard or an aggregate atom. A *literal* is either an atom a or its negation $not\, a$. A literal of the form a is said to be *positive*, otherwise it is *negative*. A *rule* is an expression of the form $h \leftarrow l_1, \ldots, l_n$ where h is a standard atom referred to as *head* that can also be omitted, and $l_1, \ldots, l_n$, with $n \geq 0$, is a conjunction of literals referred to as *body*. For a rule r, H_r denotes the set of standard atoms in the head of r, B_r denotes the set of standard literals in the body of r, and B_r^a denotes the set of aggregate literals in the body of r. Without loss of generality in this paper we assume that for each rule r, $|B_r^a| \leq 1$. A rule r is a *constraint* if $H_r = \emptyset$ or it is a *fact* if $B_r = B_r^a = \emptyset$. Given a rule r, the *global variables* of r are all the variables appearing in B_r or in the guard of an aggregate in B_r^a; r is *safe* if each global variable and each variable in H_r appears in some positive literal in B_r. A *program* is a set of safe rules. Given an ASP expression ϵ (i.e. literals, rule, etc.), $\mathcal{V}(\epsilon)$ and $\mathcal{P}(\epsilon)$ denote, respectively, the set of variables and predicates in ϵ. Given a program Π, $\mathcal{H}(\Pi)$ denotes the set of atoms appearing in the head of some rules in Π; the dependency graph of Π, denoted by G_Π, is a directed labeled graph whose nodes are the predicates in Π and there exists an edge $(u, v, +)$ (resp. $(u, v, -)$) if there exists a rule $r \in \Pi$ such that u appears in some positive (resp. negative) literal in the body of r and v appears in the head of r. A program Π is said to be *tight* (Fages 1994) if G_Π has no loops involving only positive edges. Such definition has been successively relaxed by Fandinno and Lifschitz (2022), introducing the class of *locally tight* programs, that are programs free of positive recursion after grounding. In the following we restrict our attention to locally tight programs that is the class ASP program supported by our approach. Let $p \in \mathcal{P}(\Pi)$, Π_p denotes the set of rules $r \in \Pi$ such that p appears in H_r. Given a program Π, the *Herbrand Universe* of Π, denoted by $\mathcal{U}_\Pi$, is the set of constants in Π; the *Herbrand Base* of Π, denoted by $\mathcal{B}_\Pi$, is the set of ground standard atoms that can be built from predicates in Π and constants in $\mathcal{U}_\Pi$. A substitution σ w.r.t. Π, is a mapping from a set of variables V to constants in $\mathcal{U}_\Pi$. Given an ASP expression ϵ occurring in a program Π, a substitution σ from $\mathcal{V}(\epsilon)$ to constants in $\mathcal{U}_\Pi$ is a *well-formed substitution*, and $\sigma(\epsilon)$ denotes the expression obtained from ϵ by replacing variables with values they are mapped to. Given a symbolic set S, $inst(S)$ denotes the ground set $\{\sigma(S) \mid \sigma \text{ is a well-formed substitution for } S\}$. Given a rule $r \in \Pi$, a *global substitution* for r is a substitution from the global variables of r to constants in $\mathcal{U}_\Pi$; and $ground(r)$ denotes the set of ground instantiation of the form $\sigma(H_r) \leftarrow \sigma(B_r), A$, where σ is a global substitution for r, and $A = f(inst(\sigma(S))) \prec \sigma(T)$ if $B_r^a = \{f(S) \prec T\}$, otherwise A is omitted. For a program Π, $ground(\Pi)$ is the set of ground instantiations of rules in Π.

Regarding the semantics of ASP, we recall that, given a program Π, an *interpretation* I (i.e. a set of standard literals over atoms in $\mathcal{B}_\Pi$) is an answer set of Π iff (i) I is total and consistent (i.e., for each $a \in \mathcal{B}_\Pi$ either $a \in I$ or $not\, a \in I$); (ii) I is a model, namely for each rule $r \in ground(\Pi)$ either the head of r is true w.r.t. I or the body of r is false w.r.t. I; and (iii) I is a minimal model of its FLP-reduct (Faber, Pfeifer, and Leone 2011).

A program Π is said to be coherent if it admits at least one answer set, otherwise it is incoherent. For further details about the ASP semantics, we refer the reader to (Gelfond and Lifschitz 1991; Faber, Pfeifer, and Leone 2011; Calimeri et al. 2020).

2.2 Evaluation of ASP Programs

The Ground&Solve Approach. Traditional ASP solvers employ the Ground&Solve approach, which is based on two components, grounder and solver. The former takes as input a program P and produces $ground(\Pi)$, which is later on processed by the solver to produce answer sets. Specifically, answer sets are produced using a CDCL-based (Silva and Sakallah 1999) algorithm, extended with *propagators* specific for ASP (Kaufmann et al. 2016). The algorithm is based on three main components, a *choice* heuristic, a *propagation* function, and a *learning* strategy. The idea is to build an answer set step-by-step starting from an empty interpretation I. At each step, a literal is heuristically selected and added to I (*choice*). Then, *propagators* are used to extend I with the deterministic consequences of this choice and their *reason*, i.e., literals in I leading to the propagation. If the propagation leads to an inconsistency in I (i.e. $p, \overline{p} \in I$ for some $p \in \mathcal{B}$), the algorithm *learns* a new constraint using the reason of each propagated literal, undoes the choices leading to the inconsistency, and restores the consistency of I. This process is repeated until I is an answer set or the consistency of I cannot be restored, showing that no answer sets can be found.

Compilation-based Approach. Compilation-based ASP-solving was revealed to be very promising in tackling the grounding bottleneck problem that affects the Ground&Solve approach (Ostrowski and Schaub 2012; Calimeri et al. 2016). The idea behind such approaches is to compile an ASP program into a set of custom propagators that are able to simulate rules' inferences during the solving without grounding them at all. Recently, a compilation-based ASP solver, known as PROASP, has been introduced (Dodaro, Mazzotta, and Ricca 2023). Briefly, PROASP produces an ad-hoc solver for a non-ground program that could be used to solve multiple instances of such a problem expressed as a set of facts. More precisely, given a non-ground program P, the compilation stage generates two modules, the *Generator* and the *Propagator*. The former generates the set of relevant atoms needed to compute the answer sets, while the latter simulates the inferences of rules in P during the execution of the CDCL algorithm. To this end, the input program Π is used to generate two programs Π^{gen}, and Π^{prop}. Π^{gen} is compiled into a set of procedures that will be used to perform a bottom-up evaluation of rules in Π^{gen} starting from a set of input facts. These procedures are assembled by following the topological sort of the dependency graph of Π^{gen}, and originate the Generator

module of the output solver. On the other hand, the program Π^{prop} is first rewritten by applying an algorithm that resorts the completion (Clark 1977) of the program, adjusted for the non-ground setting. Then, the resulting program is compiled into custom propagators that will be further integrated into a dedicated module of the output solver, that is the Propagator. The third component of the PROASP solver is an extended version of the SAT solver GLUCOSE (Audemard and Simon 2009), which is initialized with the atoms produced by the Generator and integrates the generated Propagator module, which is used during propagation and learning stages.

3 Blending Grounding and Compilation

In this section, we describe the compilation of a hybrid solver starting from a tight program Π. Initially, we partition Π into two distinct programs, denoted as G and C. Subsequently, the hybrid solver incorporates a grounder module responsible for generating rule instantiations of rules in G, and a set of specialized propagators tailored for rules in C. Initially, rules in G are rewritten as described below, and then grounded. The resulting ground rules are then processed by the CDCL algorithm, where Clark's completion (Clark 1977) is first applied, and then they are directly integrated into the solver.

One of the problems of creating a hybrid solver is to interleave the main CDCL algorithm and the set of dedicated propagators. Specifically, the CDCL algorithm operates on Clark's completion of its input program. However, in our hybrid approach, the CDCL algorithm operates only on a portion of the input program, i.e. the ground instantiation of the rules in G, and the other rules are compiled as a set of dedicated propagators. As a consequence, Clark's completion may be incorrect in this case. As an example, consider that Π includes the rules $r_1 : p(X) \leftarrow a(X)$ and $r_2 : p(X) \leftarrow b(X)$, where $p(X)$ appears in the head. If both r_1 and r_2 are in G, Clark's completion of each atom over the predicate p, say $p(1)$, would include a set of clauses to state that $p(1)$ is true iff one between $a(1)$ and $b(1)$ is true. However, if only r_1 is in G, the completion would include a set of clauses stating that $p(1)$ is true iff $a(1)$ is true, which is incorrect. To address this issue, we perform an additional rewriting technique, described in the following.

Definition 1. *Let Π be a program partitioned into two subprograms G and C, then $blend(\Pi) = \mathcal{G} \cup \mathcal{C}$ where $\mathcal{G}$ is obtained from G by adding a rule of the form $p(V_1, \ldots, V_n) \leftarrow p'(V_1, \ldots, V_n)$, for each predicate $p \in \mathcal{P}(\mathcal{H}(G)) \cap \mathcal{P}(\mathcal{H}(C))$ of arity n, where $V_1, \ldots, V_n$ is a list of distinct variables, and p' does not occur in Π. Instead, $\mathcal{C}$ is obtained from C by rewriting each rule of the form $p(t_1, \ldots, t_n) \leftarrow l_1, \ldots, l_m$ as $p'(t_1, \ldots, t_n) \leftarrow l_1, \ldots, l_m$, where $p \in \mathcal{P}(\mathcal{H}(G)) \cap \mathcal{P}(\mathcal{H}(C))$, and p' is not in Π.*

Intuitively, for an atom p, p' represents an alias of p for the rules in C, then the Clark's completion of p includes also the atom p' so that the truth of p' is handled using propagators created from C. The following example should clarify the rewriting technique.

Example 1. *Let Π be a program partitioned into two subprograms G and C, where $G = \{a(X,Y) \leftarrow b(X,Y), not\, c(Y)\}$ and $C = \{r_1, r_2\}$, with $r_1 = a(1,Y) \leftarrow d(Y)$ and $r_2 = a(X,1) \leftarrow e(X)$. Then $blend(\Pi) = \mathcal{G} \cup \mathcal{C}$ where $\mathcal{G}$ is the program:*

$$a(X,Y) \leftarrow b(X,Y), not\, c(Y)$$
$$a(X_1,X_2) \leftarrow a'(X_1,X_2)$$

and $\mathcal{C}$ is the program:

$$a'(1,Y) \leftarrow d(Y) \qquad a'(X,1) \leftarrow e(X).$$

Starting from $blend(\Pi)$, the goal is to construct a pair of programs $\langle \Pi^{prop}, \Pi^{gen} \rangle$ to create a solver following the PROASP approach, extended with the compilation of some rules into a compiled grounder. Specifically, all rules in $\mathcal{G}$ are stored in Π^{gen}. The rewriting techniques described below are applied to $\mathcal{C}$, resulting in some rules added to Π^{prop} and others to Π^{gen}. Rules added to Π^{gen} that are also in $\mathcal{G}$ are compiled into a grounder (as detailed in Section 4), while other rules are compiled into a generator, which has the role of generating the atoms to be added in GLUCOSE by simulating the grounding procedure on the rules of Π^{gen} without producing the ground instantiation.

To show how $\langle \Pi^{prop}, \Pi^{gen} \rangle$ are created, we introduce different transformations used to process an input program Π.

The first transformation is applied to rules containing aggregates and is described in the following.

Definition 2. *Let Π be a program, and a rule $r \in \Pi$ of the form:*

$$H_r \leftarrow B_r, f\{V : Conj\} \prec T \tag{1}$$

$agg(\Pi, r)$ denotes the pair $\langle \Pi_r^{prop}, \Pi_r^{gen} \rangle$. Specifically, Π_r^{prop} denotes the program:

$$\begin{aligned} as_r(S,V) &\leftarrow dm_r(S), Conj \\ agg_r(S) &\leftarrow dm_r(S), f\{V : as_r(S,V)\} \prec T \\ H_r &\leftarrow B_r, agg_r(S) \end{aligned}$$

where S is the set of global variables of r that appear also in $f\{V : Conj\} \prec T$, as_r, and agg_r are fresh predicates and dm_r denotes a fresh predicate modelling the domain of variables in S for the rule r.
Instead, Π_r^{gen} denotes the program $\{agg_r(S) \leftarrow dm_r(S)\}$, where S is the set of global variables of r that appear also in $f\{V : Conj\} \prec T$, and dm_r denotes a fresh predicate modelling the domain of variables in S for the rule r.

Intuitively, Definition 2 resorts the transformation introduced by Mazzotta, Ricca, and Dodaro (2022) and is used to split the propagations of a rule r of the form (1) into more affordable steps. More precisely, $agg(\Pi, r)$ introduces: (1) a rule for modelling the truth of all the literals in $Conj$ by means of atoms over the fresh predicate as_r; (2) a rule that models the truth of the aggregate atom by means of atoms over the fresh predicate agg_r; (3) a rule to reconstruct the original rule r; and (4) a generator rule that defines the domains of the fresh predicate agg_r according to the domain of the global variables of r. The domain of the global variables is modeled by external atoms (Gebser et al. 2016) of predicate dm_r. More precisely, such atoms are determined during the generation process by projecting the possible values of variables in S from the instantiations of B_r.

Example 2. *Let $r \in \Pi$ be a rule of the form:*

$$a(X) \leftarrow b(X,Y), \#count\{Z : c(Y,Z), not\ d(Z)\} \geq 2$$

Then $agg(\Pi, r) = \langle \Pi_r^{prop}, \Pi_r^{gen} \rangle$ where $\Pi_r^{gen} = \{agg_r(Y) \leftarrow dm_r(Y)\}$ and Π_r^{prop} is the program:

$$\begin{aligned} as_r(Y,Z) &\leftarrow dm_r(Y), c(Y,Z), not\ d(Z) \\ agg_r(Y) &\leftarrow dm_r(Y), \#count\{Z : as_r(Y,Z)\} \geq 2 \\ a(X) &\leftarrow b(X,Y), agg_r(Y) \end{aligned}$$

The second transformation is instead applied for optimizing the completion of rules forming an even loop through negation and is detailed in the following.

Let Π be a program, and SCC be the strongly connected components of G_Π, $neg(\Pi)$ denotes the set of components $\{p, q\} \in SCC$ such that $\Pi_p \cup \Pi_q$ is of the form:

$$\begin{aligned} p(t_1, \ldots, t_n) &\leftarrow l_1, \ldots, l_m, not\ q(t_1, \ldots, t_n) \\ q(t_1, \ldots, t_n) &\leftarrow l_1, \ldots, l_m, not\ p(t_1, \ldots, t_n) \end{aligned}$$

Definition 3. *Let Π be a program, and $Comp = \{p, q\} \in neg(\Pi)$, then $neg\_completion(\Pi, Comp)$ denotes the pair of programs $\langle \Pi_{Comp}^{prop}, \Pi_{Comp}^{gen} \rangle$, where Π_{Comp}^{prop} denotes the program:*

$$\begin{aligned} aux_{p,q}(\vec{t}) &\leftarrow l_1, \ldots, l_m \\ &\leftarrow p(\vec{t}), not\ aux_{p,q}(\vec{t}) \\ &\leftarrow q(\vec{t}), not\ aux_{p,q}(\vec{t}) \\ &\leftarrow aux_{p,q}(\vec{t}), p(\vec{t}), q(\vec{t}) \\ &\leftarrow aux_{p,q}(\vec{t}), not\ p(\vec{t}), not\ q(\vec{t}) \end{aligned}$$

and Π_{Comp}^{gen} denotes the program:

$$p(\vec{t}) \leftarrow aux_{p,q}(\vec{t}) \qquad q(\vec{t}) \leftarrow aux_{p,q}(\vec{t})$$

where $\vec{t} = t_1, \ldots, t_n$ are the terms appearing in the head of the rules in $\Pi_p \cup \Pi_q$, and $aux_{p,q}$ is a fresh predicate not appearing in Π.

Intuitively, if Π is a program of the form $\{a \leftarrow not\ b, b \leftarrow not\ a\}$, then a is true if and only if b is false (since a and b do not appear in any other rule of Π). This intuition is generalized by the transformation $neg\_completion$ that, given two rules r_1, r_2, introduces a rule for modelling the truth of the shared body between r_1 and r_2 by means of a fresh predicate, say aux_{r_1,r_2}, and a set of constraints that ensures (i) whenever H_{r_1} or H_{r_2} are true then the corresponding auxiliary atom must be true; and (ii) for each true auxiliary atom, it is not possible that H_{r_1} and H_{r_2} are both true or both false. Together with such rules, $neg\_completion$ introduces two generator rules that are used to generate the domain of predicates in H_{r_1} and H_{r_2} by using the fresh predicate aux_{r_1,r_2}.

Example 3. *Let Π be the following program:*

$$\begin{aligned} a(X) &\leftarrow b(X,Y), not\ c(Y), not\ na(X) \\ na(X) &\leftarrow b(X,Y), not\ c(Y), not\ a(X) \end{aligned}$$

Then, SCC comprises $\{b\}$, $\{c\}$, $\{a, na\}$, and $neg(\Pi)$=$\{\{a, na\}\}$. Let $Comp = \{a, na\}$, then $neg\_completion(\Pi, Comp) = \langle \Pi_{Comp}^{prop}, \Pi_{Comp}^{gen} \rangle$ where Π_{Comp}^{prop} is the program:

$$\begin{aligned} aux_{a,na}(X) &\leftarrow b(X,Y), not\ c(Y) \\ &\leftarrow aux_{a,na}(X), a(X), na(X) \\ &\leftarrow aux_{a,na}(X), not\ a(X), not\ na(X) \end{aligned}$$

and Π_{Comp}^{gen} is the program:

$$a(X) \leftarrow aux_{a,na}(X) \qquad na(X) \leftarrow aux_{a,na}(X).$$

Finally, the Clark's completion is applied to the program by performing the transformations described in the following.

Definition 4 ((Dodaro, Mazzotta, and Ricca 2023))**.** *Let Π be a program, $p \in \mathcal{P}(\Pi)$ be a predicate of arity m such that $\Pi_p = \{r_1, \ldots, r_n\}$, with $n \geq 2$, and $\vec{V} = V_1, \ldots, V_m$ be a list of variables. Then $unique(\Pi, p)$ denotes the pair of programs $\langle \Pi_p^{prop}, \Pi_p^{gen} \rangle$ where $\Pi_p^{gen} = \{H_r \leftarrow sup_r(\mathcal{T}(H_r)) \mid r \in \Pi_p\}$ and Π_p^{prop} denotes the program:*

$$\begin{aligned} &sup_r(\mathcal{T}(H_r)) \leftarrow B_r, B_{r^a} && \forall r \in \Pi_p \\ &\leftarrow sup_r(\mathcal{T}(H_r)), not\ H_r && \forall r \in \Pi_p \\ &\leftarrow p(\vec{V}), not\ sup_{r_1}(\vec{V}), \ldots, not\ sup_{r_n}(\vec{V}) \end{aligned}$$

Basically, for a program Π and a predicate $p \in \Pi$, $unique(\Pi, p)$ rewrites all those rules $r \in \Pi$ having p in the head, by substituting p with a fresh predicate sup_r that captures the existence of a supporting rule for an atom over predicate p, and adds a set of constraints simulating the support propagations for the atoms over predicate p. Moreover, the generator program contains a rule for each predicate sup_r that is used to generate the possible atoms over predicate p.

Example 4. *Let Π be the following program:*

$$\begin{aligned} r_1: \quad a(X) &\leftarrow b(X,Y), not\ c(Y) \\ r_2: \quad a(Z) &\leftarrow d(Z), not\ c(Z) \end{aligned}$$

$unique(\Pi, a) = \langle \Pi_a^{prop}, \Pi_a^{gen} \rangle$ where Π_a^{prop} denotes the program:

$$\begin{aligned} sup_{r_1}(X) &\leftarrow b(X,Y), not\ c(Y) \\ sup_{r_2}(Z) &\leftarrow d(Z), not\ c(Z) \\ &\leftarrow sup_{r_1}(X), not\ a(X) \\ &\leftarrow sup_{r_2}(Z), not\ a(Z) \\ &\leftarrow a(X_1), not\ sup_{r_1}(X_1), not\ sup_{r_2}(X_1) \end{aligned}$$

and Π_a^{gen} denotes the program:

$$a(X) \leftarrow sup_{r_1}(X) \qquad a(Z) \leftarrow sup_{r_2}(Z).$$

Definition 5 ((Mazzotta, Ricca, and Dodaro 2022))**.** *Given a program Π, and a rule $r \in \Pi$ such that r is of the form:*

$$H_r \leftarrow B_r \tag{2}$$

where $H_r \neq \emptyset$, $B_r = l_1, \ldots, l_n$ is a conjunction of standard literals, and $\vec{V} = \mathcal{V}(B_r)$. Then $completion(\Pi, r)$ denotes the pair $\langle \Pi_r^{prop}, \Pi_r^{gen} \rangle$, where Π_r^{gen} denotes the program $\{aux_r(\vec{V}) \leftarrow l_1, \ldots, l_n\}$, and Π_r^{prop} denotes the program:

$$\begin{aligned} &H_r \leftarrow aux_r(\vec{V}) \\ &\leftarrow aux_r(\vec{V}), \bar{l} && \forall l \in B_r \\ &\leftarrow l_1, \ldots, l_n, not\ aux_r(\vec{V}). \end{aligned}$$

Intuitively, *completion* transformation rewrites a rule r of the form (2) for simulating the well-known Clark's completion (Clark 1977) for the non-ground setting. This is done, by introducing an auxiliary predicate aux_r, and a set of constraints for each literal in B_r. In this case, the generator program contains a rule for generating all those atoms over aux_r starting from literals in B_r.

Example 5. *Let $r \in \Pi$ be the rule $a(X) \leftarrow b(X), not\ c(X)$. Then $completion(\Pi, r) = \langle \Pi_r^{prop}, \Pi_r^{gen} \rangle$, where $\Pi_r^{gen} = \{aux_r(X) \leftarrow b(X), not\ c(X)\}$, and Π_r^{prop} is the program:*

$$\begin{aligned} a(X) \leftarrow &\ aux_r(X) \\ \leftarrow &\ aux_r(X), not\ b(X) \\ \leftarrow &\ aux_r(X), c(X) \\ \leftarrow &\ b(X), not\ c(X), not\ aux_r(X) \end{aligned}$$

After applying the transformations described before, Π can be compiled into a hybrid solver. Specifically, the goal is to generate a pair of programs $\langle \Pi^{gen}, \Pi^{prop} \rangle$ such that Π^{gen} contains the rules that should be grounded together with the rules that are used for generating the ground atoms that are needed to compute the answer set of Π, while Π^{prop} contains the rules that should be compiled into a collection of custom propagators. Thus, given a program Π partitioned in two subprograms G and C, as first step we compute $blend(\Pi) = \langle \mathcal{G}, \mathcal{C} \rangle$. Then, $\mathcal{G}$ is added to Π^{gen}, while rules in $\mathcal{C}$ are subject to further transformations. First of all, for each rule $r \in \mathcal{C}$ of the form (1) we compute $agg(\mathcal{C}, r) = \langle \mathcal{C}_r^{prop}, \mathcal{C}_r^{gen} \rangle$. As in the previous step, the rules in $\mathcal{C}_r^{gen}$ are added to Π^{gen}, while r is replaced by aggregate-free rules in $\mathcal{C}_r^{prop}$. The remaining rules in $\mathcal{C}_r^{prop}$ (i.e., rules containing aggregates) are added to the final propagator program Π^{prop}. At this point, the program $\mathcal{C}$ is transformed by looking at the SCCs of $\mathcal{C}$. More precisely, for each component $c \in neg(\Pi)$, we compute $neg\_completion(\mathcal{C}, c) = \langle \mathcal{C}_c^{prop}, \mathcal{C}_c^{gen} \rangle$. While $\mathcal{C}_c^{gen}$ is added to the final generator program Π^{gen}, $\mathcal{C}_c^{prop}$ is substituted to the rules in $\mathcal{C}$ whose head predicate appears in c. The next step of the pipeline produces a program whose predicate appears in at most one rule head. This is achieved by computing $unique(\mathcal{C}, p) = \langle \mathcal{C}_p^{prop}, \mathcal{C}_p^{gen} \rangle$ for each predicate p such that $|\mathcal{C}_p| \geq 2$. In particular, for such predicate p, the rules in $\mathcal{C}_p^{gen}$ are added to Π^{gen} while the rules in $\mathcal{C}_p$ are substituted by $\mathcal{C}_p^{prop}$. Finally, we apply the completion introduced by Definition 5 to each rule $r \in \mathcal{C}$ of the form (2). In particular, for such rules we compute $completion(\mathcal{C}, r) = \langle \mathcal{C}_r^{prop}, \mathcal{C}_r^{gen} \rangle$ that will be added, respectively, to Π^{prop} and Π^{gen}. The remaining rules in $\mathcal{C}$ are basically constraints that are directly added to Π^{prop}.

4 Compiled Grounder

In this section we describe the algorithm that allows to compile the variable-elimination step (grounding). Given an ASP program Π, the compiler for the grounding outputs the code of a procedure that will compute the instantiation of Π from the facts given in input to the solver. In the following, we adopt the code convention used by Mazzotta, Ricca, and Dodaro (2022). More precisely, the code enclosed between $\ll\gg$ is printed by the compiler as it is, and the code enclosed in $[\![]\!]_t$, is first substituted with its runtime value before being printed. For example, let $V = X, Y$ and $Conj = a(X), b(Y)$, Algorithm 1 at line 12 prints $g\_set = g\_set \cup \sigma(X, Y : a(X), b(Y))$.

As in standard grounders (Kaufmann et al. 2016), the input program Π is analysed to identify body-head dependencies. In particular, the SCCs of the dependency graph G_Π of Π, and the corresponding topological sort of G_Π are computed. Then, since a rule r is associated with a SCC C such that $\mathcal{P}(H_r) \subseteq C$, each rule of Π is processed according to that order, as in a bottom-up program evaluation (Kaufmann et al. 2016). Basically, the compiler prints the code of each rule so that the resulting algorithm performs the same evaluation of a standard grounder, which respects body-head dependencies. (The code iterating over components is rather straightforward and is omitted for space reasons).

Algorithm 1 CompileRuleGrounder

```
   Input  : A rule r, i.e., H_r ← B_r, B_r^a
   Output: Prints grounder procedure for the rule r
1  begin
2      ≪aggr = ∅≫
3      ≪rules = ∅≫
4      ≪constraints = ∅≫
5      ≪σ = ϵ≫
6      CompileStandardLiterals(B_r,"l")
7      if ∃ f{V : Conj} ≺ T ∈ B_r^a then
8          ≪  v := σ( [[vars(B_r) ∩ vars(Conj)]]_t )≫
9          ≪  if ∄⟨v, _⟩ ∈ aggr then≫
10         ≪      g_set = ∅≫
11             CompileStandardLiterals(Conj,"t")
12         ≪      g_set = g_set ∪ {σ( [[V : Conj]]_t )}≫
13             CloseLocalScopes(Conj,"t")
14         ≪      aggr = aggr ∪ {⟨v, g_set⟩}≫
15         ≪      U = U ∪ {aux_agg(v)}≫
16         ≪  fi≫
17     ≪  body = σ( [[B_r]]_t )≫
18     if |B_r^a| = 1 then
19         ≪  body = body ∪ {aux_agg(v)}≫
20     if ∃ h ∈ H_r then
21         ≪  rules = rules ∪ {⟨σ( [[h]]_t ), body⟩}≫
22     else
23         ≪constraints = constraints ∪ {body}≫
24     CloseLocalScopes(B_r,"l")
```

Algorithm 1 reports the compiler code designed to process a non-ground rule r, of the form $H_r \leftarrow B_r, B_r^a$, and prints a procedure capable of computing ground instantiations of r. The resulting variable-elimination algorithm works on a set of atoms F over predicates in Π, representing the input facts, and a set of atoms U, representing undefined atoms computed by evaluating preceding components.

The objective of the generated code is to compute the set of pairs $\langle h, B \rangle$ where h is a ground instantiation of H_r and B is a ground instantiation of B_r, B_r^a having h in the head. In detail, Algorithm 1 starts by printing the code that initializes the structures to accumulate the instantiations of r (lines 2–4). Then, the initialization of a variable substitution σ as an empty substitution is printed (line 5). The next step is to compile the literals in B_r by printing a sequence of nested code blocks, one for each literal in B_r, that iterates over possible instantiations of B_r. This is done by means of Algorithm CompileStandardLiterals that prints the nested loop join algorithm (Ceri, Gottlob, and Tanca 1990) used for compiling the body of constraints by (Cuteri et al. 2020). (More detail in the supplementary material.) Subsequently, Algorithm 1 checks whether the body of r contains aggregates (line 7). If this is the case, the code for grounding the

aggregate is generated within the innermost block processing literals in B_r. Since the grounding of aggregate atoms relies on the global variables in r, the procedure creates a ground set for each ground instantiation of these variables, called shared variables, that appear in the aggregate (i.e. for an aggregate $f\{V : Conj\} \prec T, \mathcal{V}(B_r) \cap \mathcal{V}(Conj)$). Thus, (line 9) an if-statement is printed that checks whether there exists a ground set for the current instantiation of the shared variables v, and the code for evaluating the aggregate set for v is printed inside. The conjunction of the aggregate ($Conj$) is processed, as done for rule bodies, by means of COMPILESTANDARDLITERALS; next, the code for accumulating in the ground set g_set the pair $\sigma(V : Conj)$ is printed. Function CLOSELOCALSCOPES prints terminating delimiters for each block of code generated up to now (for $Conj$). Once the nested blocks (for $Conj$) are closed, the code stores the newly generated ground set for the shared variables v. Additionally, a fresh auxiliary atom ($aux_{agg}(v)$) is generated (line 15) to represent the truth value of the aggregate atom. This atom is then incorporated into the body (line 19), to be managed in the solver via a dedicated propagator as usual (Kaufmann et al. 2016). At this point (from line 17), the compiler prints the code that generates an instantiation of r from the current σ. Indeed, it prints the code that applies σ to the body of r. In case r is not a constraint ($H_r \neq \emptyset$), the code that applies σ to the head is printed, otherwise the code instantiating a ground constraint of the form $\leftarrow body$ is printed. Finally, the nested blocks generated for handling B_r are closed (last line). (An example of code generated by Algorithm 1 is reported in the supplementary material.)

5 Implementation and Experiments

In this section, we begin by outlining specific implementation details of our approach. Following this, we present the results of an empirical evaluation assessing the performance of the new approach. This evaluation was conducted using publicly available benchmarks with the goal of providing an answer to the following questions: (i) Is compiled grounding effective? (ii) Does the blending of grounding and compilation provide advantages? (iii) Is there a way to select what to compile and what to ground? (iv) What is the overall impact of blending grounding and compilation?

Implementation. We implement our approach on the top of the PROASP[1] system. To achieve this, the input handling was extended to apply the rewriting described in Section 3, which enables blending grounding and compilation; also, subprograms suitable for compilation or grounding are identified. Then, an extended *Generator Compiler* produces either Herbrand base computation code or variable-elimination code according to the methodology presented in Section 4. We also extended the *Propagator Compiler* to support the compilation of rules containing aggregates, as proposed by Mazzotta, Ricca, and Dodaro (2022).

Hardware and Software Setup. The experiments were run on a system with 2.30GHz Intel(R) Xeon(R) Gold 5118 CPU and 512GB of RAM with Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-137-generic x86_64). Execution time and memory were limited to 1800 seconds and 8192 MB, respectively. Each system was limited to run in a single core. All the material needed to reproduce our experiment can be downloaded from https://t.ly/_yJIK.

Benchmarks. For our analysis, we considered several hard benchmarks used in the literature to compare the performance of ASP systems, including benchmarks from ASP competitions (Calimeri et al. 2016), and benchmarks for compilation-based ASP solvers (Mazzotta, Ricca, and Dodaro 2022; Dodaro, Mazzotta, and Ricca 2023), for a total of 14 benchmarks and 2366 instances. The full list of benchmarks is reported in the first column of Table 1, where the column *#* indicates the number of instances, column *Gr.Int.* indicates whether the domain is grounding-intensive or not (we flag a problem as grounding-intensive if the grounding causes a system to exceed the memory limit in at least one instance), columns *SO* and *Sum t.* indicate, for each compared method, the number of solved instances and the sum of solving times in seconds, respectively.

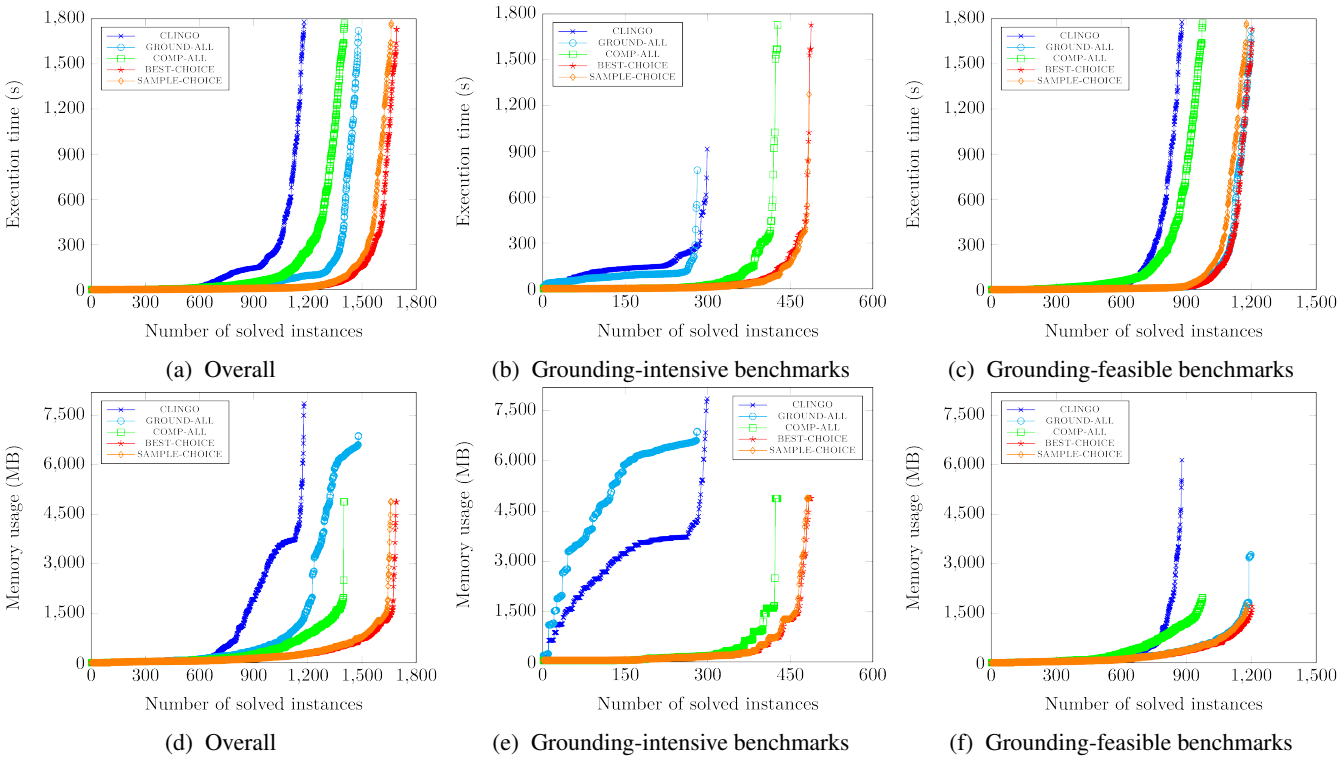
Compared methods. We compare the following methods: The ASP system CLINGO (Gebser et al. 2016) (v.5.4.0) as a representative of a state-of-the-art Ground&Solve implementation, and eight variants of the PROASP system. We label GROUND-ALL the PROASP version configured to follow the Ground&Solve architecture (i.e., all rules are grounded), where grounding is obtained via the novel compiled-grounding technique of Section 4. We label COMPILE-ALL the PROASP system following a pure compilation-based approach, i.e., no rule is grounded. COMPILE-ALL corresponds to the system presented in (Dodaro, Mazzotta, and Ricca 2023) enhanced to support aggregates on the lines of (Mazzotta, Ricca, and Dodaro 2022). Then, we consider six additional variants of PROASP representing different ways of blending grounding and compilation by splitting programs by rule type. We opted for this pragmatic choice for the following reasons. First observe that, given a program, the number of different possible hybrid solvers that could be obtained is exponential in the number of its rules, e.g., for a small program of 10 rules there are 2^{10}=1024 possible configurations. Additionally, since different benchmarks employ different encodings, it would be hard to draw general observations from such a program-specific analysis. Thus, pragmatically we split each each program according to a coarse-grained syntactic criterion that solicits separately different types of propagators. This way, we identified the following three types of rules: (CS) constraints, that is rules with an empty head not containing aggregates; (AG) rules containing aggregates; and (RL) all the other normal rules. Also, we consider their combinations, i.e., RL+CS, RL+AG and CS+AG. (Note that COMP-ALL would be COMP-RL+AG+CS). Thus, a label of the form COMP-TYPE is used to label the variant configured to compile rules of the type TYPE. In practice, we built 112 PROASP binaries (i.e., 8 versions for 14 domains).

Results. The results of the experiments are reported in Table 1, where there is a line for each benchmark problem, and the best-performing method in a domain is outlined in bold.

[1] https://github.com/MazzottaG/ProASP.git

Benchmark	#	Gr. Int.	CLINGO		GROUND-ALL		COMP-CS		COMP-RL		COMP-AG		COMP-RL+CS		COMP-RL+AG		COMP-CS+AG		COMP-ALL	
			SO	Sum t.	SO	Sum t.	SO	Sum t.	SO	Sum t.	SO	Sum t.	SO	Sum t.	SO	Sum t.	SO	Sum t.	SO	Sum t.
Packing Problem	50	✓	-	-	-	-	**48**	**1290.07**	-	-	-	-	19	2271.87	-	-	**48**	**1290.07**	19	2271.87
Quasi Group	100	✓	5	105.68	5	72.05	**15**	1079.8	5	63.82	5	72.05	**15**	**432.28**	5	63.82	**15**	1079.8	**15**	**432.28**
Non Part. Rem. Col.	110	✗	**110**	825.95	**110**	353.77	**110**	**53.05**	**110**	536.81	**110**	353.77	**110**	176.12	**110**	536.81	**110**	**53.05**	**110**	176.12
Stable Marriage	314	✓	209	22799.96	207	19145.57	236	6270.74	205	18546.01	207	19145.57	**240**	16086.1	205	18546.01	236	6270.74	**240**	16086.1
Graph Colouring	60	✗	**31**	10480.28	28	10839.99	24	9981.4	27	10981.51	28	10839.99	24	10871.3	27	10981.51	24	9981.4	24	10871.3
Hanoi Tower	60	✗	59	9115.25	**60**	**2802.27**	**60**	5793.16	53	18949.99	**60**	**2802.27**	52	18753.86	53	18949.99	**60**	5793.16	52	18753.86
Knight Tour	300	✗	54	13559.99	**300**	**1688.95**	295	4303.56	**300**	11762.4	**300**	**1688.95**	**300**	9171.51	**300**	11762.4	295	4303.56	**300**	9171.51
Maximal Clique	50	✗	**50**	52.77	**50**	26.42	**50**	**23.91**	**50**	40.56	**50**	26.42	**50**	39.83	**50**	40.56	**50**	**23.91**	**50**	39.83
Weighted Sequence	65	✗	**61**	13980.69	56	17947.97	58	20581.17	44	24502.89	56	17947.97	47	29629.55	44	24502.89	58	20581.17	47	29629.55
Bottle Filling	100	✗	**100**	803.8	**100**	784.49	**100**	847.97	**100**	**683.96**	**100**	3848.93	**100**	770.38	**100**	3802.24	**100**	3727.3	**100**	3678.29
Comp. Assignment	302	✗	125	56264.24	**226**	**6882.1**	**226**	7361.74	219	23488.53	194	37028.71	216	19029.91	85	34775.51	193	36545.65	84	32975.39
Crossing Minim.	255	✗	**228**	16889.52	189	43645.63	179	38633.15	166	60856.17	192	40463.01	165	59755.4	170	58014.91	183	38920.36	166	60502.43
Incr. Scheduling	500	✓	85	21141.85	69	5765.6	151	1513.48	70	6679.62	68	5676.84	153	3252.5	68	3486.77	**185**	12709.91	153	11080.44
Visit All	100	✗	62	4980.16	**80**	27215.43	63	13954.48	57	18061.3	60	23243.34	58	21753.64	41	3141.12	43	4859.79	43	7068.99
Overall	2366	-	1179	171000.14	1480	137170.24	**1615**	111687.68	1406	195153.57	1430	163137.82	1549	191994.25	1258	188604.54	1600	146139.87	1403	202737.96

Table 1: Comparison of the different degree of blending with state-of-the-art solver CLINGO.



(a) Overall (b) Grounding-intensive benchmarks (c) Grounding-feasible benchmarks

(d) Overall (e) Grounding-intensive benchmarks (f) Grounding-feasible benchmarks

Figure 1: Execution time and memory usage comparison.

First, we answer question (i), and we compare side by side CLINGO with GROUND-ALL. We observe that the latter solves more instances overall (1480 vs 1179). However, a domain-wise analysis shows that the two methods have their own strengths. CLINGO is preferable in Graph Colouring, Weighted Sequence, and Crossing Minimization, whereas GROUND-ALL is faster in Hanoi Tower, Knight Tour, Component Assignment, and Visit All. Diving into the details, CLINGO is very fast in benchmarks where the program simplifications it applies after grounding provide an advantage; this is more visible in ground-intensive benchmarks. GROUND-ALL seems faster where a direct and fast instantiation leaves more time to solving.

We now broaden our analysis to study the behaviour of all the compared methods. As expected, there is no solution that is definitely the best in all the domains. In Crossing Minimization the Ground&Solve is the best approach; indeed, CLINGO is the fastest, and GROUND-ALL is the best PROASP version. On the other hand, in Quasi Group, the fastest is a pure compilation-based approach. Nonetheless, the best performance is often obtained by a hybrid strategy. Indeed, the three best-performing methods overall are COMP-CS, COMP-CS+AG, and COMP-RL+CS, which solve 1615, 1600 and 1549 instances, respectively. We additionally report that we also run ALPHA system (Weinzierl 2017). Overall, ALPHA is not competitive with either CLINGO or PROASP, solving less than 250 instances out of 2366. On the same benchmarks, CLINGO solves 1179 instances and the best version of PROASP solves 1615 instances. All in all, blending grounding and compilation is often beneficial (this answers question (ii)).

Understanding the conditions under which compilation

and grounding are beneficial is crucial, especially to address question (iii) and develop a strategy for selecting suitable blends. Observe that the nature of compilation-based approaches prevents from applying existing algorithm selection strategies (Maratea, Pulina, and Ricca 2014). Indeed, what to compile has to be decided at the beginning, and there are different binaries for different domains. Running all possible versions over all the instances is often unpractical. However, given a reasonable number of candidate versions, one can pragmatically sample the space of instances, perform a short run, and then make an informed decision. Our sampling approach is based on the instance size, measured by the number of facts (which impacts the grounding size). First, we efficiently compute the number of facts for each instance in a given benchmark. Then, we categorize instances into four intervals based on quartiles. Subsequently, we randomly select k instances from each interval to create a sample of the benchmark. For this experiment, we set $k = 2$ (considering only a few instances), and for each domain, we select the best PROASP version from the sample. We label the resulting method SAMPLE-CHOICE, and we compare it with the best possible choice (labelled BEST-CHOICE) in each benchmark, and the extreme evaluation methods, namely GROUND-ALL, COMP-ALL, and CLINGO. The overall behaviour is shown in the cactus plots of Figure 1a and Figure 1d, respectively plotting time and memory consumption. In a cactus plot, instances are sorted by memory or time usage. A point (i, j) denotes that a solver solves the i-th instance within a limit of j MB or seconds, respectively.

Note that SAMPLE-CHOICE closely aligns with BEST-CHOICE, validating the effectiveness of the sampling strategy described earlier. Few instances were adequate to make a sound decision (this answers question (iii)). Furthermore, Figure 1a highlights the significant effectiveness of blending grounding and compilation. In particular, BEST-CHOICE solves 513, 212, and 289 more instances (in less time) than CLINGO, GROUND-ALL, and COMP-ALL, respectively. Figure 1d confirms that blending grounding and compilation also leads to improved memory usage.

Finally, we study the behaviour of grounding and compilation (to answer question (iv)). The cactus plots of Figure 1b and Figure 1c summarize the performance of compared methods depending on whether the problem is ground-intensive or not, respectively. In the grounding-intensive setting compilation is advantageous, indeed in Figure 1b the COMP-ALL line is very close to BEST-CHOICE. Also note that GROUND-ALL and CLINGO perform similarly. On the other hand, in Figure 1c, i.e., where grounding is not an issue, GROUND-ALL is almost superimposed to BEST-CHOICE. Similar considerations hold from the perspective of memory usage, cfr., Figure 1e and Figure 1f.

6 Related Work

In recent years, several techniques have been proposed in order to mitigate the grounding bottleneck. Constraints Answer Set Programming (CASP) (Aziz, Chu, and Stuckey 2013; Balduccini and Lierler 2017; Cat et al. 2015; Ostrowski and Schaub 2012; Susman and Lierler 2016), ASP Modulo Theories (Gebser et al. 2016), and HEX programs (Eiter, Redl, and Schüller 2016) opt for extending the language with additional constructs for connecting ASP with external solvers. Although these systems are effective, they shift the complexity from ASP to external sources. Program rewriting methods (Besin, Hecher, and Woltran 2022; Besin, Hecher, and Woltran 2023) address the issue by converting the input program into alternative forms, like epistemic logic programs or disjunctive programs. While the resulting programs are easier to evaluate, the translation process may be exponential in the worst case, yet highly effective in some scenarios (Besin, Hecher, and Woltran 2022). Anyhow the techniques of (Besin, Hecher, and Woltran 2022) that produce an ASP program are orthogonal and could be combined with compilation. An alternative approach (Stéphan 2021) translates an ASP program into a Constraint Handling Rules program. Lazy-grounding systems (Bomanson, Janhunen, and Weinzierl 2019; Lefèvre and Nicolas 2009; Lierler and Robbins 2021; Palù et al. 2009; Weinzierl 2017; Weinzierl, Taupe, and Friedrich 2020), instead, tackle the bottleneck by grounding rules during the search, i.e., when their body is satisfied during the solving. Two prominent examples are ALPHA (Weinzierl 2017; Weinzierl, Taupe, and Friedrich 2020) and DualGrounder (Lierler and Robbins 2021). The former combines lazy instantiation with ASP solving techniques (such as clause-learning, conflict-based heuristics, etc.); while DualGrounder performs lazy instantiation resorting to the multi-shot API of CLINGO. Both outperform traditional ASP systems on grounding-intensive benchmarks but are not competitive on solving-intensive ones (Lierler and Robbins 2021). This is due to the fact that they discover the space of propositional atoms during the solving, which is advantageous only if the grounding is unfeasible. Compilation-based approaches (Cuteri et al. 2019; Cuteri et al. 2020; Mazzotta, Ricca, and Dodaro 2022; Dodaro, Mazzotta, and Ricca 2023) avoid grounding by compiling non-ground rules into ad-hoc propagators that are able to simulate the inferences of such rules. Compilation-based have broader applicability than lazy grounders, because they generate the Herbrand base in advance. The first approaches (Cuteri et al. 2019; Cuteri et al. 2020; Mazzotta, Ricca, and Dodaro 2022) targeted only constraints and aggregates; whereas the PROASP system (Dodaro, Mazzotta, and Ricca 2023) removed the syntactic restrictions of previous approaches, and support the compilation of tight programs (Erdem and Lifschitz 2003). Tight programs are a relevant class of programs that has been playing a central role in the development of modern SAT-based ASP systems (Lierler and Maratea 2004; Janhunen 2022).

This work extends PROASP by incorporating aggregate compilation (following (Mazzotta, Ricca, and Dodaro 2022)) and introduces the ability to blend grounding and compilation. While PROASP also supports compiling the grounding procedure, similarly to ASP grounders such as GRINGO (Gebser et al. 2011) and IDLV (Calimeri et al. 2017), the key difference lies in how grounding is handled. PROASP compiles grounding into code specific to the program, whereas GRINGO and IDLV utilize general-purpose algorithms and support a broader input language.

7 Conclusion

This paper describes how to blend grounding and compilation in a hybrid ASP system. The new approach involves: (i) a program rewriting technique enabling the blending of grounded rules with propagators and the compilation of aggregates, and (ii) a compiler for the grounding phase, generating program-specific code for variable elimination. Moreover, we implemented the new techniques in an extended version of the PROASP solver. The results of an experiment, conducted on a rich collection of well-known benchmarks, show that blending grounding and compilation can give significant advantages over the base techniques and state-of-the-art implementations.

As future work, our plan is to extend PROASP to support the entire ASP-Core 2 standard, with a particular focus on non-tight programs, and also to study a more fine-grained (possibly automated) procedure for selecting the rules to ground/compile. This is a non trivial task that cannot be obtained by adapting existing algorithm selection techniques. This is due to the unreleased nature of compilation, which is a task that has to be performed before being able to execute any instance of the problem.

Acknowledgements

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, Tech4You CUP H23C22000370006, and PRIN PINPOINT CUP H23C22000280006.

References

Alviano, M.; Calimeri, F.; Dodaro, C.; Fuscà, D.; Leone, N.; Perri, S.; Ricca, F.; Veltri, P.; and Zangari, J. 2017. The ASP system DLV2. In *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, 215–221. Springer.

Arenas, M.; Bertossi, L. E.; and Chomicki, J. 1999. Consistent query answers in inconsistent databases. In *PODS*, 68–79. ACM Press.

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, 399–404.

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

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

Barbara, V.; Guarascio, M.; Leone, N.; Manco, G.; Quarta, A.; Ricca, F.; and Ritacco, E. 2023. Neuro-symbolic AI for compliance checking of electrical control panels. *Theory Pract. Log. Program.* 23(4):748–764.

Besin, V.; Hecher, M.; and Woltran, S. 2022. Body-decoupled grounding via solving: A novel approach on the ASP bottleneck. In *IJCAI*, 2546–2552. ijcai.org.

Besin, V.; Hecher, M.; and Woltran, S. 2023. On the structural complexity of grounding - tackling the ASP grounding bottleneck via epistemic programs and treewidth. In *ECAI*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, 247–254. IOS Press.

Bomanson, J.; Janhunen, T.; and Weinzierl, A. 2019. Enhancing lazy grounding with lazy normalization in answer-set programming. In *AAAI*, 2694–2702. AAAI Press.

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

Calimeri, F.; Gebser, M.; Maratea, M.; and Ricca, F. 2016. Design and results of the fifth answer set programming competition. *Artif. Intell.* 231:151–181.

Calimeri, F.; Fuscà, D.; Perri, S.; and Zangari, J. 2017. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale* 11(1):5–20.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. Asp-core-2 input language format. *Theory Pract. Log. Program.* 20(2):294–309.

Cardellini, M.; Dodaro, C.; Galatà, G.; Giardini, A.; Maratea, M.; Nisopoli, N.; and Porro, I. 2023. Rescheduling rehabilitation sessions with answer set programming. *J. Log. Comput.* 33(4):837–863.

Cat, B. D.; Denecker, M.; Bruynooghe, M.; and Stuckey, P. J. 2015. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res.* 52:235–286.

Ceri, S.; Gottlob, G.; and Tanca, L. 1990. *Logic Programming and Databases*. Surveys in computer science. Springer.

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

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

Cuteri, B.; Dodaro, C.; Ricca, F.; and Schüller, P. 2020. Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators. In *IJCAI*, 1688–1694. ijcai.org.

Cuteri, B.; Reale, K.; and Ricca, F. 2019. A logic-based question answering system for cultural heritage. In *JELIA*, volume 11468 of *LNCS*, 526–541. Springer.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33(3):374–425.

Dodaro, C., and Maratea, M. 2017. Nurse scheduling via answer set programming. In *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, 301–307. Springer.

Dodaro, C.; Mazzotta, G.; and Ricca, F. 2023. Compilation of tight ASP programs. In *ECAI*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, 557–564. IOS Press.

Eiter, T.; Fink, M.; Greco, G.; and Lembo, D. 2008.

Repair localization for query answering from inconsistent databases. *ACM Trans. Database Syst.* 33(2):10:1–10:51.

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

Erdem, E., and Lifschitz, V. 2003. Tight logic programs. *Theory Pract. Log. Program.* 3(4-5):499–518.

Erdem, E., and Patoglu, V. 2018. Applications of ASP in robotics. *Künstliche Intell.* 32(2-3):143–149.

Erdem, E.; Gelfond, M.; and Leone, N. 2016. Applications of answer set programming. *AI Mag.* 37(3):53–68.

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

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

Fandinno, J., and Lifschitz, V. 2022. Verification of locally tight programs. *CoRR* abs/2204.10789.

Francescutto, G.; Schekotihin, K.; and El-Kholany, M. M. S. 2021. Solving a multi-resource partial-ordering flexible variant of the job-shop scheduling problem with hybrid ASP. In *JELIA*, volume 12678 of *Lecture Notes in Computer Science*, 313–328. Springer.

Gebser, M.; Kaminski, R.; König, A.; and Schaub, T. 2011. Advances in *gringo* series 3. In *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, 345–351. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory solving made easy with clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASICS*, 2:1–2:15. Schloss Dagstuhl.

Gebser, M.; Leone, N.; Maratea, M.; Perri, S.; Ricca, F.; and Schaub, T. 2018. Evaluation techniques and systems for answer set programming: a survey. In *IJCAI*, 5450–5456. ijcai.org.

Gebser, M.; Maratea, M.; and Ricca, F. 2020. The seventh answer set programming competition: Design and results. *Theory Pract. Log. Program.* 20(2):176–204.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Gener. Comput.* 9(3/4):365–386.

Janhunen, T. 2022. Implementing stable-unstable semantics with ASPTOOLS and clingo. In *PADL*, volume 13165 of *Lecture Notes in Computer Science*, 135–153. Springer.

Kaufmann, B.; Leone, N.; Perri, S.; and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Mag.* 37(3):25–32.

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

Lierler, Y., and Maratea, M. 2004. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, 346–350. Springer.

Lierler, Y., and Robbins, J. 2021. Dualgrounder: Lazy instantiation via clingo multi-shot framework. In *JELIA*, volume 12678 of *Lecture Notes in Computer Science*, 435–441. Springer.

Lifschitz, V. 2010. Thirteen definitions of a stable model. In *Fields of Logic and Computation*, volume 6300 of *Lecture Notes in Computer Science*, 488–503. Springer.

Manna, M.; Ricca, F.; and Terracina, G. 2015. Taming primary key violations to query large inconsistent data via ASP. *Theory Pract. Log. Program.* 15(4-5):696–710.

Maratea, M.; Pulina, L.; and Ricca, F. 2014. A multi-engine approach to answer-set programming. *TPLP* 14(6):841–868.

Marques-Silva, J.; Lynce, I.; and Malik, S. 2021. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 133–182.

Mazzotta, G.; Ricca, F.; and Dodaro, C. 2022. Compilation of aggregates in ASP systems. In *AAAI*, 5834–5841. AAAI Press.

Mitra, A.; Clark, P.; Tafjord, O.; and Baral, C. 2019. Declarative question answering over knowledge bases containing natural language text with answer set programming. In *AAAI*, 3003–3010. AAAI Press.

Müller, L.; Wanko, P.; Haubelt, C.; and Schaub, T. 2024. Investigating methods for aspmt-based design space exploration in evolutionary product design. *Int. J. Parallel Program.* 52(1):59–92.

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

Palù, A. D.; Dovier, A.; Pontelli, E.; and Rossi, G. 2009. GASP: answer set programming with lazy grounding. *Fundam. Informaticae* 96(3):297–322.

Rajaratnam, D.; Schaub, T.; Wanko, P.; Chen, K.; Liu, S.; and Son, T. C. 2023. Solving an industrial-scale warehouse delivery problem with answer set programming modulo difference constraints. *Algorithms* 16(4):216.

Schüller, P. 2016. Modeling variations of first-order horn abduction in answer set programming. *Fundam. Informaticae* 149(1-2):159–207.

Silva, J. P. M., and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers* 48(5):506–521.

Son, T. C.; Pontelli, E.; Balduccini, M.; and Schaub, T. 2023. Answer set planning: A survey. *Theory Pract. Log. Program.* 23(1):226–298.

Stéphan, I. 2021. First-order ASP programs as CHR programs. In *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, 881–888. ACM.

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

Weinzierl, A.; Taupe, R.; and Friedrich, G. 2020. Advancing lazy-grounding ASP solving techniques - restarts, phase saving, heuristics, and more. *Theory Pract. Log. Program.* 20(5):609–624.

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

Yang, Z.; Ishay, A.; and Lee, J. 2023. Coupling large language models with logic programming for robust and general reasoning from text. In *ACL (Findings)*, 5186–5219. Association for Computational Linguistics.