

Bypassing the ASP Bottleneck: Hybrid Grounding by Splitting and Rewriting

Alexander Beiser¹, Markus Hecher², Kaan Unalan¹ and Stefan Woltran¹

¹TU Wien, Vienna, Austria

²Massachusetts Institute of Technology, Cambridge, MA, United States

{alexander.beiser, kaan.unalan, stefan.woltran}@tuwien.ac.at, hecher@mit.edu,

Abstract

Answer Set Programming (ASP) is a key paradigm for problems in artificial intelligence and industrial contexts. In ASP, problems are modeled via a set of rules. Over the time this paradigm grew into a rich language, enabling complex rule types like aggregate expressions. Most practical ASP systems follow a ground-and-solve pattern, where rule schemes are grounded and resulting rules are solved. There, the so-called grounding bottleneck may prevent from solving, due to sheer grounding sizes. Recently, body-decoupled grounding (BDG) demonstrated how to reduce grounding sizes by delegating effort to solving. However, BDG provides limited interoperability with traditional grounders and only covers simple rule types. In this work, we establish hybrid grounding — based on a novel splitting theorem that allows us to freely combine BDG with traditional grounders. To mitigate huge groundings in practice, we define rewriting procedures for efficiently deferring grounding effort of aggregates to solving. Our experimental results indicate that this approach is competitive, especially for instances where traditional grounding fails.

1 Introduction

Answer set programming (ASP) [Brewka *et al.*, 2011; Gebser *et al.*, 2019] is a prominent modeling and solving framework, where problems are modeled by means of rules that form a (*logic*) program. The solutions of such programs are called *answer sets*, which are sets of atoms that fulfill every rule. A huge contribution to the success story and practical relevance of ASP is the availability of efficient systems [Gebser *et al.*, 2019; Alviano *et al.*, 2019]. These systems are based on a *ground-and-solve pattern*, where a *grounder* replaces first-order like variables by domain constants, thereby creating a ground program that is solved by means of an ASP solver.

Over the time, the ground-and-solve pattern enabled ASP to grow into a rich and expressive language supporting complex expressions like *aggregates* [Alviano and Faber, 2018]. This makes ASP a convenient tool for formulating problems

in domains like knowledge representation, artificial intelligence, and even in industrial contexts, see e.g., [Falkner *et al.*, 2018]. Aggregates are particularly useful to compactly express complex scenarios and are implemented in modern grounders, such as *gringo* [Kaminski and Schaub, 2022] and *idlv* [Calimeri *et al.*, 2019].

Undoubtedly, there is a huge downside to the ground-and-solve pattern, as it may lead to the well-known ASP *grounding bottleneck* [Gebser *et al.*, 2018; Cuteri *et al.*, 2020; Tsamoura *et al.*, 2020] — an issue, where the grounder creates an exponentially large program beyond solving capabilities. While this is a major source of inefficient encodings, this issue is far from surprising, as already the evaluation of normal programs is NEXPTIME-complete [Dantsin *et al.*, 2001]. However, as shown by Eiter *et al.*, [2007], for constant predicate arities the complexity of non-ground answer set existence increases by only one level in the polynomial hierarchy, compared to ground programs (i.e., hardness of deciding answer set existence grows for normal programs from NP [Marek and Truszczyński, 1991] to Σ_2^P and for disjunctive programs from Σ_2^P [Eiter and Gottlob, 1995] to Σ_3^P).

These insights have been recently utilized to mitigate the grounding bottleneck, resulting in *body-decoupled grounding (BDG)* [Besin *et al.*, 2022] and other variants [Dodaro *et al.*, 2023] of shifting effort from the grounder to the solver. BDG provides an efficient alternative to cope with large rule bodies, as body elements are grounded independently, i.e., it is exponential only in the maximum predicate arity and not in the number of variables of the whole body. However, this technique comes with limitations, as for small rule bodies it is outperformed by traditional grounding. In practice, some combination of both concepts is therefore well desired. To date (i) BDG supports only *very limited* interoperability with traditional grounding (if programs Π_1 and Π_2 shall be grounded via BDG and traditional grounding, respectively, a predicate is prohibited to appear in rule heads of both programs), and crucially, (ii) BDG does *not* support aggregates yet, which are one of the main sources of huge groundings.

Contributions¹. We address both shortcomings (i) and (ii):

1. First, we propose *hybrid grounding*, which aims at com-

¹Supplementary material including source code, benchmark instances and experimental results, are available online: <https://github.com/alex14123/newground/releases/tag/v2.0.0>.

binning the best of both traditional grounding and BDG. We thereby develop a *novel splitting theorem* to enable arbitrary partitions of a non-ground normal program Π into a tight program² Π_t with dense rule bodies³, and a potentially cyclic part Π_c , forming the remainder. Program Π_t is then grounded similar to BDG, where we avoid large grounding sizes caused by dense rule bodies and delay efforts to the ASP solver. In turn, Π_c benefits from efficient traditional grounding of small rule bodies.

2. We define novel rewriting methods on translating hard-to-ground aggregate expressions, such that we ease the grounding of aggregates with a dense body. We then utilize hybrid grounding, where the translated rules are subject to BDG, i.e., put into Π_t . Thereby, we provide an alternative approach beyond traditional variable instantiation. To deal with aggregation in different contexts, we provide variants of our translation.
3. Finally, we present an implementation of our translation-based approach on the ASP grounding bottleneck in Python, where we provide support for commonly used ASP aggregates. In our experimental evaluation, we compare two rewritings, demonstrating that hybrid grounding on these translations is capable of grounding dense aggregates beyond the state-of-the-art.

Related Work. The literature distinguishes alternative grounding approaches, ranging from classical techniques [Alviano *et al.*, 2019; Kaminski and Schaub, 2022], numerical techniques [Hippen and Lierler, 2021], lazy rule injection [Cuteri *et al.*, 2017], and lazy grounding [Weinzierl *et al.*, 2020], over ASP modulo theory, e.g., [Banbara *et al.*, 2017], and methods based on structural measures, e.g., [Bichler *et al.*, 2020; Mitchell, 2019; Calimeri *et al.*, 2019]. There are existing works on translating aggregates, e.g., [Bartholomew *et al.*, 2011; Zaniolo *et al.*, 2017; Pelov *et al.*, 2003; Bomanon *et al.*, 2014]. Earlier works on splitting, e.g., [Lifschitz and Turner, 1994] do not focus on grounding strategies.

2 Preliminaries

Ground ASP. Let ℓ, m, n be non-negative integers such that $\ell \leq m \leq n$; a_1, \dots, a_n be distinct propositional atoms. A (*disjunctive*) *program* P is a set of (*disjunctive*) *rules* $a_1 \vee \dots \vee a_\ell \leftarrow a_{\ell+1}, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n$. For a rule r , we let $H_r := \{a_1, \dots, a_\ell\}$, $B_r^+ := \{a_{\ell+1}, \dots, a_m\}$, and $B_r^- := \{a_{m+1}, \dots, a_n\}$. We denote the sets of *atoms* occurring in a rule r or in a program P by $\text{at}(r) := H_r \cup B_r^+ \cup B_r^-$ and $\text{at}(P) := \bigcup_{r \in P} \text{at}(r)$. A rule r is *normal* if $|H_r| \leq 1$ and a program P is *normal* if all its rules are normal. The *dependency graph* \mathcal{D}_P is the directed graph defined on vertices $\bigcup_{r \in P} H_r \cup B_r^+$, s.t. for every $r \in P$ two atoms $a \in B_r^+$ and $b \in H_r$ are joined by edge (a, b) . A *head-cycle* of \mathcal{D}_P is an $\{a, b\}$ -cycle⁴ for two distinct atoms $a, b \in H_r$ for some

rule $r \in P$. P is *head-cycle-free (HCF)* or *tight* if \mathcal{D}_P contains no head-cycle or cycle, respectively.

An *interpretation* I is a set of atoms. I satisfies a rule r if $(H_r \cup B_r^-) \cap I \neq \emptyset$ or $B_r^+ \setminus I \neq \emptyset$. I is a *model* of P if it satisfies all rules of P . The *Gelfond-Lifschitz (GL) reduct* of P under I is the program P^I obtained from P by first removing all rules r with $B_r^- \cap I \neq \emptyset$ and then removing all $\neg z$ where $z \in B_r^-$ from the remaining rules r . I is an *answer set* of a program P if I is a *minimal model* (w.r.t. \subseteq) of P^I . Deciding whether a program has an answer set is called *consistency*, which is Σ_2^P -complete [Eiter and Gottlob, 1995]. For normal and HCF programs, its complexity drops to NP-complete [Marek and Truszczyński, 1991; Ben-Eliyahu and Dechter, 1994]. For an interpretation I , a *level mapping* $\psi : I \rightarrow \{0, \dots, |I| - 1\}$ assigns every atom in I a unique value [Lin and Zhao, 2003; Janhunen, 2006]. Given an interpretation I of an HCF program Π and a level mapping ψ of I . An atom $a \in I$ is *founded* if for an $r \in \Pi$ with $a \in H_r$, we have $B_r^+ \subseteq I$, $\psi(b) < \psi(a)$ for every $b \in B_r^+$, and $I \cap B_r^- = I \cap (H_r \setminus \{a\}) = \emptyset$. I is an *answer set* of Π if (i) I is a model of Π , and (ii) all atoms in I are founded.

Non-ground ASP. We use vectors $\mathbf{X} = \langle x_1, \dots, x_m \rangle$, $\mathbf{Y} = \langle y_1, \dots, y_n \rangle$ in the usual way. We *combine vectors* by $\langle \mathbf{X}, \mathbf{Y} \rangle := \langle x_1, \dots, x_m, y_1, \dots, y_n \rangle$ and test whether x_1 is in \mathbf{X} by $x_1 \in \mathbf{X}$. Elements of vectors are in any fixed total order. For a set S , we *construct* its unique vector by $\langle S \rangle$.

A *term* is a constant, or a variable. Let p_1, \dots, p_n be predicates; each one takes *arity* $|p_i|$ many variables for $1 \leq i \leq n$. A *comparison* is a predicate of the form $t < u$, where t , and u are terms, and $< \in \{<, \leq, >, \geq, =, \neq\}$, in their usual meaning. A *literal* is a predicate, or its negation. A (*non-ground*) *program* Π is a set of (*non-ground*) *rules* of the form

$$p_1(\mathbf{X}_1) \vee \dots \vee p_\ell(\mathbf{X}_\ell) \leftarrow p_{\ell+1}(\mathbf{X}_{\ell+1}), \dots, p_m(\mathbf{X}_m), \quad (1)$$

$$\neg p_{m+1}(\mathbf{X}_{m+1}), \dots, \neg p_n(\mathbf{X}_n).$$

where for every *variable vector* \mathbf{X}_i we have $|\mathbf{X}_i| = |p_i|$, and whenever $x \in \langle \mathbf{X}_1, \dots, \mathbf{X}_\ell, \mathbf{X}_{m+1}, \dots, \mathbf{X}_n \rangle$, then $x \in \langle \mathbf{X}_{\ell+1}, \dots, \mathbf{X}_m \rangle$ (*safeness*). For a non-ground rule r , we define $H_r := \{p_1(\mathbf{X}_1), \dots, p_\ell(\mathbf{X}_\ell)\}$, $B_r^+ := \{p_{\ell+1}(\mathbf{X}_{\ell+1}), \dots, p_m(\mathbf{X}_m)\}$, as well as $B_r^- := \{p_{m+1}(\mathbf{X}_{m+1}), \dots, p_n(\mathbf{X}_n)\}$. We furthermore define $B_r := B_r^+ \cup B_r^-$, $\text{var}(r) := \{x \in \mathbf{X} \mid p(\mathbf{X}) \in H_r \cup B_r\}$, and use $\text{hpred}(\Pi) := \{p(\mathbf{X}) \in H_r \mid r \in \Pi\}$ and $\text{pred}(\Pi) := \{p(\mathbf{X}) \in (H_r \cup B_r) \mid r \in \Pi\}$. For brevity, we assume *variables are unique per rule*, i.e., for every two rules $r, r' \in \Pi$, $\text{var}(r) \cap \text{var}(r') = \emptyset$. Attributes *disjunctive*, *normal*, and *tight* carry over to rules (programs). In order to *ground* Π , we require a given set \mathcal{F} of *facts*, i.e., atoms of the form $p(\mathbf{D})$ with p being a predicate of Π and \mathbf{D} being a vector over *domain values* of size $|\mathbf{D}| = |p|$. We say that \mathbf{D} is part of the *domain* of Π , defined by $\text{dom}(\Pi) := \{d \in \mathbf{D} \mid p(\mathbf{D}) \in \mathcal{F}\}$. We refer to the *domain vectors* over $\text{dom}(\Pi)$ for a variable vector \mathbf{X} of size $|\mathbf{X}|$ by $\text{dom}(\mathbf{X})$. Let \mathbf{D} be a domain vector over variable vector \mathbf{X} and vector \mathbf{Y} containing only variables of \mathbf{X} . We refer to the *domain vector* \mathbf{D} *restricted to* \mathbf{Y} by $\mathbf{D}_\mathbf{Y}$. The *grounding* $\mathcal{G}(\Pi)$ comprises \mathcal{F} and rules obtained by replacing each rule r of Form (1) for

²We can relax this condition, which causes an additional sub-quadratic overhead in the form of level mappings [Janhunen, 2006].

³A rule body is *dense* if it contains a large number of predicates

⁴Let $G = (V, E)$ be a digraph and $W \subseteq V$. Then, a (directed) cycle in G is a W -cycle if it contains all vertices from W .

every $\mathbf{D} \in \text{dom}(\langle \text{var}(r) \rangle)$ by $p_1(\mathbf{D}_{\mathbf{X}_1}) \vee \dots \vee p_\ell(\mathbf{D}_{\mathbf{X}_\ell}) \leftarrow p_{\ell+1}(\mathbf{D}_{\mathbf{X}_{\ell+1}}), \dots, p_m(\mathbf{D}_{\mathbf{X}_m}), \neg p_{m+1}(\mathbf{D}_{\mathbf{X}_{m+1}}), \dots, \neg p_n(\mathbf{D}_{\mathbf{X}_n})$. The dependency graph \mathcal{D}_Π of the non-ground program Π is defined over the ground program $\mathcal{G}(\Pi)$; $\text{scc}(\Pi)$ refers to the set of *strongly-connected components (vertices)* of \mathcal{D}_Π .

Example 1. Consider the non-ground program $\Pi_1 := \{r_1\}$ with $r_1 = a(X, Y) \leftarrow b(X), c(Y, Z), d(Y)$ and $\mathcal{F}_1 := \{b(1); c(1, 2); c(2, 1); d(1)\}$. Observe that $\text{dom}(X) = \{1\}$ and $\text{dom}(Y) = \text{dom}(Z) = \{1, 2\}$. Grounding $\mathcal{G}(\Pi_1)$ comprises: $\{b(1); c(1, 2); c(2, 1); d(1); a(1, 1) \leftarrow b(1), c(1, 1), d(1); a(1, 1) \leftarrow b(1), c(1, 2), d(1); a(1, 2) \leftarrow b(1), c(2, 1), d(2); a(1, 2) \leftarrow b(1), c(2, 2), d(2)\}$. The answer set of Π_1 is $\{b(1); c(1, 2); c(2, 1); a(1, 1); d(1)\}$.

Aggregates. We denote aggregates as specified in the ASP-Core-2 standard [Calimeri *et al.*, 2020]. Let $\mathbf{T} = \langle t_1, \dots, t_o \rangle$ be a term vector and $\mathbf{L} = \langle l_1(\mathbf{Y}_1), \dots, l_k(\mathbf{Y}_k) \rangle$ be a literal vector. An (aggregate) element e is of the form $\mathbf{T} : \mathbf{L}$, where \mathbf{T} is the head of e of length o and \mathbf{L} is its body. We require variables in \mathbf{T} to occur in a non-negated predicate of \mathbf{L} . We refer to $E = \{e_1, \dots, e_\alpha\}$ as the *element multiset* ($|E| = \alpha$).

An aggregate is of the form $\text{aggr}\{E\} \prec u$, $\text{aggr} \in \{\text{count}, \text{sum}, \text{min}, \text{max}\}$, u is a term, and $\prec \in \{<, \leq, >, \geq, =, \neq\}$. An aggregate may occur in B_r of a non-ground rule r ; a variable X of some body of an element $e_j \in E$ that also occurs in B_r , is a *global variable*, i.e., the aggregate has a *variable dependency* on X .

To evaluate an aggregate $\text{aggr}\{E\} \prec u$, it is grounded, thereby replacing variables by domain values. Then, those ground element heads in E , whose element body hold, are collected, resulting in a set R of domain vectors. We apply aggr on R , where $\text{count}(R)$, $\text{sum}(R)$, $\text{min}(R)$, and $\text{max}(R)$ are evaluated under the usual semantics. Finally, we determine $\text{aggr}(R) \prec u$.

Example 2. Consider the non-ground program $\Pi_2 := \{r_2\}$ with $r_2 = \leftarrow e(X, Y), \text{count}\{A, B : e(A, B), f(A, X); A : g(A), e(A, Y); A : h(A)\} \geq 3$. and $\mathcal{F}_2 := \{e(1, 1); f(1, 1); g(1); g(2); h(1); h(3)\}$. Note that the element tuple sets are $R_1 = \{(1, 1)\}$, $R_2 = \{(1)\}$, and $R_3 = \{(1), (3)\}$. Further, the combined one is $R = R_1 \cup R_2 \cup R_3$. Grounding $\mathcal{G}(\Pi_2)$ results in the following program, which has no answer set: $\{e(1, 1); f(1, 1); g(1); g(2); h(1); h(3); \leftarrow e(1, 1), \text{count}\{(1, 1), (1), (3)\} \geq 3\}$

In general, we rewrite aggregate expressions to non-ground programs, for enabling efficient hybrid grounding. Still, we define for aggregate programs the dependency graph \mathcal{D}_Π , and $\text{scc}(\Pi)$ analogously. Further, we require a rule r containing an aggregate that the corresponding $p_i(\mathbf{D}_{\mathbf{X}_i}) \in H_r$, and $p_j(\mathbf{D}_{\mathbf{X}_j}) \in B_r^+$ are not contained in a cycle in \mathcal{D}_Π . However, there are proposals for recursive aggregate semantics, e.g., [Ferraris, 2011; Faber *et al.*, 2011; Gelfond and Zhang, 2019]. However, these are *not contained* in the agreed ASP-Core-2 standard, as supported by current ASP systems.

3 A Splitting Theorem for Hybrid Grounding

First, we provide new insights into a hybrid grounding strategy that allows us to significantly strengthen body-decoupled grounding (BDG) [Besin *et al.*, 2022]. More concretely, we

split a program into parts, which enables to individually combine body-decoupled grounding on dense tight rules, with classical grounding for the remainder of the program. This yields a stronger result, which we call *hybrid grounding*, where we provide a way to freely *split a non-ground program* Π into two parts: A dense tight program Π_t and the remainder $\Pi_c = \Pi \setminus \Pi_t$. This flexibility will then pave the way towards the efficient grounding of dense aggregates, using different translations to a (tight) program.

Example 3. We demonstrate the potential of hybrid grounding by a small example. Consider the non-ground program $\Pi_3 := \{r_{3a}; r_{3b}\}$ with $r_{3a} = a(X) \leftarrow b(X)$, $r_{3b} = a(X) \leftarrow c(X), e(A, B), e(A, C), e(B, C)$, and facts $\mathcal{F}_3 := \{b(1); c(2)\} \cup G$, where G is a graph given by edge facts (e) . Rule r_{3b} uses 4 (densely) interacting variables, i.e., intuitively r_{3b} is *denser* than r_{3a} . In practice, one would therefore ground r_{3a} with state-of-the-art grounders and r_{3b} with BDG, avoiding groundings that are cubic in the domain size.

BDG, in contrast to hybrid grounding, explicitly prohibits shared predicates (here: a) and the result would be indeed wrong. We derive $a(1)$ from r_{3a} using fact $b(1)$. Crucially, r_{3b} **can never** derive $a(1)$, as there is no fact $c(1)$. Since $a(1)$ was derived, but can not be justified by BDG (rule r_{3b}), the constructed program does not have an answer set.

In order to ensure correctness of hybrid grounding for any partitioning into two programs Π_c, Π_t , despite the fact that (shared) atoms can be derived by both programs (individually), we use auxiliary predicates p' for predicates p appearing in the head of the rules in Π_t . This will allow us to distinguish the (potentially interleaving) cases, where an atom is founded by Π_c and those where the atom is derived due to Π_t .

Hybrid Grounding Procedure. First, we discuss a simplified variant of hybrid grounding, where we only permit cyclic dependencies within Π_c , but not between Π_t and Π_c . Crucially, we ensure that atoms are correctly derived, despite shared predicates between Π_c and Π_t , thereby bijectively preserving all answer sets.

Figure 1 depicts hybrid grounding procedure \mathcal{H} . First, by Rules (2) we guess whether an atom can be derived by means of Π_t . Further, these rules ensure that atoms over auxiliary predicates h' (i.e., those derived by means of Π_t) are copied to atoms over h . This is crucial, as these rules ensure the link between Π_t and Π_c , but also contribute to rule satisfiability of Π_t , as discussed below. Then, Rules (3) ground Π_c by classical means (i.e., rule instantiation). Note that in this step, practical grounders might simplify or remove rules of Π_c . However, this can be avoided by grounding Π_c together with (non-ground versions of) Rules (2).

Then, Rules (4)–(8) ensure rule satisfiability of Π_t . This is achieved by guessing every potential instantiation in Rules (4), applying the saturation technique in Rules (8), and deriving rule satisfiability via avoiding rule body combinations due to Rules (5)–(7). Observe that the latter rules only use a *single (body) predicate* of Π_t . After that, rule satisfiability can be derived for all rules of Π_t by Rules (4), which is mandatory due to (8).

Finally, Rules (9)–(12) of Figure 1 ensure that, in addition to satisfiability of rules in Π_t , every atom that is guessed via

Rules (2) is indeed founded. To this end, it is sufficient to find a suitable rule instantiation via Rules (9) for such an atom, and derive unfoundedness for a rule, again, by decoupling body atoms due to Rules (10) and (11), such that *not* every rule yields unfoundedness for that atom, see Rules (12). Note that the latter rules are relaxed such that these rules are only generated for atoms that are derived due to Π_t , i.e., if the atom over the auxiliary predicate p' holds.

Despite Rules (2), we bijectively preserve all answer sets.

Lemma 1. *Given a partition of any non-ground HCF program Π into a program Π_c and a tight program $\Pi_t = \Pi \setminus \Pi_c$, such that for every $S \in \text{scc}(\Pi)$, we have $|E(S_{\Pi_t})| = 0$, where $E(S_{\Pi_t})$ is the edge set of the subgraph S_{Π_t} of \mathcal{D}_{Π_t} induced by $S \cap \text{at}(\Pi_t)$. Then, the answer sets of $\mathcal{H}(\Pi_c, \Pi_t)$ restricted to $\text{at}(\mathcal{G}(\Pi))$ bijectively match those of $\mathcal{G}(\Pi)$.*

Proof (Sketch). Correctness of BDG under the assumption $\text{hpred}(\Pi_c) \cap \text{hpred}(\Pi_t) = \emptyset$ has been shown in [Besin *et al.*, 2022]. In this case, predicates $\text{hpred}(\Pi_c)$, $\text{hpred}(\Pi_t)$ can only appear in rule bodies of Π_t , Π_c , respectively. Therefore, foundedness of atoms over predicates $\text{hpred}(\Pi_t)$ is checked by Rules (9)–(12) and foundedness of atoms over $\text{hpred}(\Pi_c)$ is derived directly by Rules (3). Satisfiability of Π_t is ensured by Rules (4)–(8); satisfiability of Π_c is treated by Rules (3).

Suppose $\text{hpred}(\Pi_c) \cap \text{hpred}(\Pi_t) \neq \emptyset$. As now atoms over $\text{hpred}(\Pi_c)$, $\text{hpred}(\Pi_t)$ may be derived in Π_c , Π_t but may also appear in rule heads of Π_t , Π_c , respectively, this might lead to incorrect results. We rename all head predicates $h(\mathbf{X}) \in \text{hpred}(\Pi_t)$ to $h'(\mathbf{X})$. This (again) ensures foundedness of atoms over $\text{hpred}(\Pi_t)$ and satisfiability of Π_t , only by Rules (9)–(12) and (4)–(8), respectively, while Π_t continues to use atoms derived by Π_c in its rule bodies. However, Π_t has to inform Π_c about atoms derived in Π_t , ensured by Rules (2). This still applies if Π_c is cyclic, as long as there is no shared cycle between Π_c and Π_t .

It remains to argue, why despite Rules (2), every answer set M of $\mathcal{H}(\Pi_c, \Pi_t)$ has a unique answer set $M \cap \text{at}(\Pi)$ of Π . Consider an arbitrary $h(\mathbf{D}) \in \text{at}(\mathcal{G}(\Pi))$. If $h(\mathbf{D}) \notin M$, $h'(\mathbf{D}) \notin M$ due to Rules (2), making $M \cup \{h(\mathbf{D})\}$ unique.

Suppose $h(\mathbf{D}) \in M$. Case $h'(\mathbf{D}) \in M$: Then, $h(\mathbf{D})$ has to be founded by Π_t . Hence, there is an $r \in \Pi_t$ with $\text{sat}_r \in M$ due to Rules (7). So, there is no answer set $M' \neq M$ identical to M over $\text{at}(\mathcal{G}(\Pi))$, s.t. $h'(\mathbf{D}) \notin M'$, since $\text{sat}_r \notin M'$.

Case $h'(\mathbf{D}) \notin M$: There is no answer set $M' \neq M$ identical to M over $\text{at}(\mathcal{G}(\Pi))$, s.t. $h'(\mathbf{D}) \in M'$, as M' violates (12). \square

Results. Next, we strengthen Lemma 1. Indeed, compared to BDG as presented in [Besin *et al.*, 2022], our hybrid strategy enables to freely split a non-ground HCF program Π into a tight program Π_t and the remainder $\Pi \setminus \Pi_t$. This yields the theorem below, which is the key result of hybrid grounding, permitting the (potentially cyclic) sharing of predicates between split programs Π_c , Π_t .

With level mappings, we can extend \mathcal{H} to work despite cycles over Π_c and Π_t . This yields reduction \mathcal{H}_{lv} (Appendix Figure 7), where we use auxiliary predicates to encode level mappings over each two atoms $p_1(\mathbf{D}_1)$, $p_2(\mathbf{D}_2)$ for any domain vectors \mathbf{D}_1 , \mathbf{D}_2 of a shared SCC, so that one of the atoms has precedence over the other one. We formalize that

this precedence is transitive, to avoid cyclic precedence via level mappings. Crucially, we then ensure that a head atom $h(\mathbf{D})$ of a rule r is unfounded, if a body atom over a predicate in $\text{pred}(r) \setminus \text{hpred}(r)$ does not precede $h(\mathbf{D})$. This prevents abusing cyclic foundedness between both Π_c and Π_t .

Theorem 1 (Grounding-Splitting Theorem). *Given a partition of any non-ground HCF program Π into a HCF program Π_c and a tight program $\Pi_t = \Pi \setminus \Pi_c$. The answer sets of $\mathcal{H}_{lv}(\Pi_c, \Pi_t)$ restricted to $\text{at}(\mathcal{G}(\Pi))$ match those of $\mathcal{G}(\Pi)$.*

Note that \mathcal{H}_{lv} can be further extended, such that Π_t is any normal (HCF) program. This requires level mappings as above, thereby also excluding cyclic foundedness within Π_t .

We still obtain polynomial runtimes for small domains, without considering traditional grounding, i.e., Rules (3).

Proposition 1 (Runtime). *Let Π be any tight non-ground program, where predicate arities are bounded by a . Then, $\mathcal{H}_{lv}(\emptyset, \Pi)$ runs in time $\mathcal{O}((|\Pi| + |\text{at}(\Pi)|) \cdot |\text{dom}(\Pi)|^{2 \cdot a})$.*

4 Aggregate Rewriting for Hybrid Grounding

State-of-the-art grounders easily run into the grounding-bottleneck in case of *dense aggregates*, whose body contains a large number of variables. With the help of hybrid grounding, it is possible to efficiently ground tight ASP programs, but utilizing this approach for aggregates requires further considerations. To this end, we present ideas behind our aggregate rewriting techniques. While these are presented along the lines of the count aggregate, they work for all aggregates, as detailed in the appendix.

4.1 Warm-up: How to Rewrite Aggregates

According to Proposition 1, the grounding time is mainly influenced by the maximum arity. Hence, the rewritten program should have a low maximum arity. Consider the simple monotonic count aggregate: $\text{count}\{X : p(X, Y)\} \geq 3$. Intuitively this aggregate holds whenever we derive at least three different element heads, i.e., three different instantiations for variable X . We implement this by (i) copying the element's body $(p(X, Y))$ three times and by (ii) adding an *alldiff* predicate to ensure that the tuple-heads (X) differ:

$$p(X_1, Y_1), p(X_2, Y_2), p(X_3, Y_3), \text{alldiff}(X_1, X_2, X_3)$$

To ensure low maximum arity, we encode *alldiff* by pairwise difference checks, i.e., $X_i \neq X_j$, for all pairs i, j .

As a first step towards generalizing the example, we provide a *generalization* for rules containing a single monotonic count aggregate (count_{\prec}) in the positive body of a rule r ($\text{count}_{\prec} \in B_r^+$). Further, we require count_{\prec} to have an integer bound $\geq u$ and a single element $e = T_e : B_e$, comprising head $T_e = \langle t_1, \dots, t_o \rangle$ and body $B_e = l_1(\mathbf{Y}_1), \dots, l_k(\mathbf{Y}_k)$. By extending our idea from above, we ensure that whenever u different element heads T_e exist, *the aggregate holds*. Therefore, we (i) add u many $B_{e,i}$'s, with variable indexing as described above, and (ii) add *alldiff* constraints for all (indexed, $1 \leq i \leq u$) $T_{e,i}$'s. The resulting rewritten rule is depicted in Equation (13), where $B_{rc}^+ := B_r^+ \setminus \{\text{count}_{\prec}\}$, yielding rewriting procedure $\mathcal{RM}^{\text{count}}$ (Rewriting Monotonic Count).

Glue Π_c to Π_t and Ground Π_c

$$h'(\mathbf{D}) \vee h'(\mathbf{D}) \leftarrow h(\mathbf{D}) \leftarrow h'(\mathbf{D}) \quad \text{for every } h(\mathbf{X}) \in \text{hpred}(\Pi_t), \mathbf{D} \in \text{dom}(\mathbf{X}) \quad (2)$$

$$r \quad \text{for every } r \in \mathcal{G}(\Pi_c) \quad (3)$$

Satisfiability of Π_t

$$\bigvee_{d \in \text{dom}(\mathbf{x})} \text{sat}_x(d) \leftarrow \text{sat} \leftarrow \text{sat}_{r_1}, \dots, \text{sat}_{r_n} \quad \text{for every } r \in \Pi_t, x \in \text{var}(r), \text{ where } \Pi_t = \{r_1, \dots, r_n\} \quad (4)$$

$$\text{sat}_r \leftarrow \text{sat}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \dots, \text{sat}_{x_\ell}(\mathbf{D}_{\langle x_\ell \rangle}), \neg p(\mathbf{D}) \quad \text{for every } r \in \Pi_t, p(\mathbf{X}) \in B_r^+, \mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \dots, x_\ell \rangle \quad (5)$$

$$\text{sat}_r \leftarrow \text{sat}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \dots, \text{sat}_{x_\ell}(\mathbf{D}_{\langle x_\ell \rangle}), p(\mathbf{D}) \quad \text{for every } r \in \Pi_t, p(\mathbf{X}) \in B_r^-, \mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \dots, x_\ell \rangle \quad (6)$$

$$\text{sat}_r \leftarrow \text{sat}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \dots, \text{sat}_{x_\ell}(\mathbf{D}_{\langle x_\ell \rangle}), h'(\mathbf{D}) \quad \text{for every } r \in \Pi_t, h(\mathbf{X}) \in H_r, \mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \dots, x_\ell \rangle \quad (7)$$

$$\text{sat}_x(d) \leftarrow \text{sat} \leftarrow \neg \text{sat} \quad \text{for every } r \in \Pi_t, x \in \text{var}(r), d \in \text{dom}(\mathbf{x}) \quad (8)$$

Prevent Unfoundedness of Atoms in Π_t

$$\bigvee_{d \in \text{dom}(y)} \text{uf}_y(\langle \mathbf{D}, d \rangle) \leftarrow h'(\mathbf{D}) \quad \text{for every } r \in \Pi_t, h(\mathbf{X}) \in H_r, \mathbf{D} \in \text{dom}(\mathbf{X}), y \in \text{var}(r), y \notin \mathbf{X} \quad (9)$$

$$\text{uf}_r(\mathbf{D}_\mathbf{X}) \leftarrow \text{uf}_{y_1}(\mathbf{D}_{\langle \mathbf{X}, y_1 \rangle}), \dots, \text{uf}_{y_\ell}(\mathbf{D}_{\langle \mathbf{X}, y_\ell \rangle}), \neg p(\mathbf{D}_\mathbf{Y}) \quad \text{for every } r \in \Pi_t, h(\mathbf{X}) \in H_r, p(\mathbf{Y}) \in B_r^+, \mathbf{D} \in \text{dom}(\langle \mathbf{X}, \mathbf{Y} \rangle), \mathbf{Y} = \langle y_1, \dots, y_\ell \rangle \quad (10)$$

$$\text{uf}_r(\mathbf{D}_\mathbf{X}) \leftarrow \text{uf}_{y_1}(\mathbf{D}_{\langle \mathbf{X}, y_1 \rangle}), \dots, \text{uf}_{y_\ell}(\mathbf{D}_{\langle \mathbf{X}, y_\ell \rangle}), p(\mathbf{D}_\mathbf{Y}) \quad \text{for every } r \in \Pi_t, h(\mathbf{X}) \in H_r, p(\mathbf{Y}) \in B_r^- \cup (H_r \setminus \{h(\mathbf{X})\}), \mathbf{D} \in \text{dom}(\langle \mathbf{X}, \mathbf{Y} \rangle), \mathbf{Y} = \langle y_1, \dots, y_\ell \rangle \quad (11)$$

$$\leftarrow \text{uf}_{r_1}(\mathbf{D}), \dots, \text{uf}_{r_m}(\mathbf{D}), h'(\mathbf{D}) \quad \text{for every } h(\mathbf{X}) \in \text{hpred}(\Pi_t), \mathbf{D} \in \text{dom}(\mathbf{X}), \{r_1, \dots, r_m\} = \{r \in \Pi_t \mid h(\mathbf{X}) \in H_r\} \quad (12)$$

Figure 1: Hybrid grounding procedure $\mathcal{H}(\Pi_c, \Pi_t)$, which creates a disjunctive *ground* program from a given non-ground HCF program $\Pi_c \cup \Pi_t$ such that Π_t is tight. We thereby interleave classical grounding on Π_c with body-decoupled grounding on Π_t .

Rewriting Procedure $\mathcal{RM}^{\text{count}}$

$$H_r \leftarrow B_{rc}^+, B_r^-, B_{e,1}, \dots, B_{e,u}, \text{alldiff}(\mathbf{T}_{e,1}, \dots, \mathbf{T}_{e,u}) \quad (13)$$

Results. Let $h = |\mathbf{T}|$ be the length of the element's head, and ϕ_r be the maximum arity of r (before rewriting). Then, Lemma 2 upper bounds the rewritten rule's arity.

Lemma 2 (Maximum arity of $\mathcal{RM}^{\text{count}}$). *Let r be a rule containing a single count aggregate, with a singleton element set, with u being an integer constant and \prec being \geq . The maximum arity of $\mathcal{RM}^{\text{count}}(r)$ is bounded by $\max\{2 \cdot |h|, \phi_r\}$.*

4.2 Generalizing the Idea

To this end, we need to address several issues: (i) multiple elements, (ii) variable dependencies, (iii) variable bounds, (iv) different comparison operators, and (v) different aggregate types. Many of these issues are interdependent and there is no single approach to address all issues at once. For brevity, we give an idea of how the generalization works for the *count* aggregate, by addressing issue (i) (and partly (iv)). For issues (ii), (iii), (iv) and (v), we refer to the appendix.

First we address issue (i), where we observe that a simple extension of $\mathcal{RM}^{\text{count}}$ to multiple elements fails for two reasons: On the one hand, for the count aggregate to hold in general, we need to consider every combination of u different element heads. On the other hand, element heads might have varying arity. We address both by introducing a *combined element*, which combines all elements into one conceptual element \mathcal{E} . Element \mathcal{E} has a *combined head* (\mathbf{T}'), and a *single body predicate* ($tp(\mathbf{T}')$), i.e., $\mathcal{E} = \mathbf{T}' : tp(\mathbf{T}')$. Note that the arity of the combined element head \mathbf{T}' has to be the

maximum arity over all elements e' ($|\mathbf{T}'| = \max_{e'} |\mathbf{T}_{e'}|$). Intuitively, a *ground atom* over predicate tp holds, whenever there is at least one aggregate element that finds it. Therefore, we introduce for each element $e = \mathbf{T}_e : B_e$ a new rule: $tp(\mathbf{T}_e) \leftarrow B_e$. However, this rule fails in case of different head element arities. Note, for each element e , its head arity $|\mathbf{T}_e|$ fulfills $|\mathbf{T}_e| \leq |\mathbf{T}'|$. So we adapt this idea in the following way: If for an element e we have $|\mathbf{T}'| - |\mathbf{T}_e| > 0$ we fill up the difference with fresh dummy constants c_j . The full rule is depicted in Equation (14a) below.

Example 4. *For example, given an aggregate count $\{X_1 : p(X_1); X_2, Y_2 : k(X_2, Y_2); X_3, Y_3, Z_3 : l(X_3, Y_3, Z_3)\} \geq u$, we get $|\mathbf{T}'| = 3$, and for elements 1, 2 and 3: $|\mathbf{T}_1| = 1$, $|\mathbf{T}_2| = 2$ and $|\mathbf{T}_3| = 3$. By applying our updated Equation (14a) we obtain the following three rules (c_1 , and c_2 are fresh constants): $\{tp(X_1, c_1, c_2) \leftarrow p(X_1); tp(X_2, Y_2, c_2) \leftarrow k(X_2, Y_2); tp(X_3, Y_3, Z_3) \leftarrow l(X_3, Y_3, Z_3)\}$*

Having the encoding of \mathcal{E} , we are ready to put together the pieces. To this end, we partly take into account issue (iv), thereby introducing a *count* $_{\prec}$ predicate and an appropriate new rule (Equation (14b)) such that *count* $_{\prec}$ holds whenever the aggregate holds. The encoding of Equation (14b) states u variable disjoint $tp(\mathbf{T}')$ predicates and the *alldiff* predicate to ensure pairwise differing \mathbf{T}' . Finally, in the original rule we replace the aggregate by predicate *count* $_{\prec}$ (Equation (14c)):

Rewriting Procedure $\mathcal{RS}^{\text{count}}$

$$\text{For every elem. } e: tp(\mathbf{T}', c_1, \dots, c_{|\mathbf{T}'| - |\mathbf{T}_e|}) \leftarrow B_e. \quad (14a)$$

$$\text{count}_{\prec} \leftarrow tp(\mathbf{T}'_1), \dots, tp(\mathbf{T}'_u), \text{alldiff}(\mathbf{T}'_1, \dots, \mathbf{T}'_u). \quad (14b)$$

$$H_r \leftarrow \text{count}_{\prec}, B_r^+, B_r^-. \quad (14c)$$

Scen.	#Insts.	Total #Solved			
		gringo	idlv	NaGG-gringo	NaGG-idlv
S1	50	26	50	50	50
S2	3500	250	980	1057	1000
S3-T4	3500	491	489	2526	2420
S3-T6	3500	204	204	2078	1997
S3-T8	3500	156	156	1718	1666
S4-T4	3500	231	439	1274	1168
S4-T6	3500	143	197	907	890
S4-T8	3500	121	156	756	700

Table 1: Number of solved instances for Scenarios S1–S4.

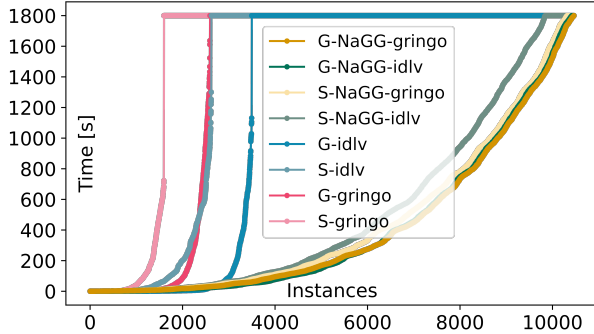


Figure 2: Cactus plot (S2, S3, S4), number of grounded (G), and combined (grounded and solved) (S) instances, for gringo, idlv, NaGG-gringo, and NaGG-idlv.

5 Implementation and Experimental Results

We implemented both hybrid grounding and aggregate rewritings, resulting in the system *Novel AggreGate Grounder* (NaGG). Our prototypical grounder is based on Python3 and the `clingo` 5.6 API. Efficient parsing of the input program is achieved by the `clingo Abstract Syntax Tree` (AST) library.

The main part of NaGG implements hybrid grounding according to the grounding-splitting theorem (Theorem 1). Thereby, NaGG provides the flavors NaGG-gringo (NaGG-idlv) indicating that NaGG takes care of the program part Π_t and gringo (idlv) grounds the remainder Π_c , respectively. By design, NaGG is most beneficial if Π_t comprises only rules of larger bodies (with many variable dependencies), whereas Π_c only contains rules of slim bodies. So, hybrid grounding avoids large groundings of Π_t , caused by exhaustive rule instantiation or auxiliary predicates of high arity.

Aggregates are handled by NaGG with a preprocessor, which implements multiple rewriting strategies (Appendix Table 2). Note that by construction our aggregate rewritings generate rules of large bodies (and many variable dependencies). The translation of aggregates therefore is one crucial use case that demonstrates the potential of this technique well.

5.1 Experimental Setup

We compared NaGG against state-of-the-art grounders gringo 5.6.2 and idlv 1.1.6. We compare four experimental setups, gringo, idlv, NaGG-gringo, and idlv-gringo. While we consider lazy grounding related, we only

compare NaGG to exact grounders. However, we expect that our approach is of interest for lazy grounding, as available systems do not support aggregates.

Benchmark Setting. We mainly measure *grounding sizes* (*times*) and *combined times* (grounding and solving). Grounding size thereby refers to the output size, which is either in *smodels* or *aspif* format for idlv and gringo, respectively. In our benchmarks, we *limit* available main memory (RAM) to 32GB (for each grounding or solving), and the overall runtime for both grounding and solving to 1800s. Plots only show grounding sizes up to 32GB. We used a benchmark system with an AMD Opteron 6272 with 225GB RAM on Debian10 with kernel 4.19.0-16-amd64. To rule out random results, we ran benchmark jobs multiple times.

Benchmark Scenarios. In order to evaluate NaGG, we design a series of benchmarks. The goal of these experiments is to demonstrate application areas and use cases for hybrid grounding strategies on dense aggregates. This seems to be particularly useful, as our approach can be readily incorporated into every grounder, making a *portfolio-based grounder*. We consider the following *benchmark scenarios*:

S1 Polygamy Stable Matching: We adapt the *stable marriage* problem (ASP Competition 2014), where we add an aggregate to permit polygamy for some individual(s).

S2 Relaxed NPRC: We relax non-partition-removal colorings [Weinzierl *et al.*, 2020], by adding an aggregate to permit some deleted node causing a disconnected graph.

S3 Traffic Connector: Decide whether there are connected subgraphs reaching at least a certain number of the nodes, s.t. we use at most one central node of degree $\geq k$ (traffic connector). k ranges from 4 (S3-T4), 6 (S3-T6), to 8 (S3-T8).

S4 Count Traffic Connectors: This is based on S3, where we count the number of traffic connectors via aggregates, thereby storing the aggregate result in a predicate. As above, k ranges from 4 (S4-T4), 6 (S4-T6) to 8 (S4-T8).

Benchmark Instances. We use both random and application instances. For S1, we take instances from the ASP competition (2014). We generate graphs with varying *instance density* (edge probability) and *instance size* for S2, S3 and S4, to discuss the impact of instance scalability on NaGG. For S3 and S4 we model different *rule densities* k , i.e., number of different body variables, allowing us to study rule scalability.

Hypotheses. We study *Hypotheses H1–H3*:

- H1: NaGG improves *overall solving performance* on crafted and application instances.
- H2: For NaGG, the *overall solving performance* suffers less from increased *rule density*, *instance density*, and *instance size*.
- H3: NaGG reduces *grounding sizes* and *grounding times* for dense scenarios and instances.

5.2 Experimental Results

We confirm H1 via Table 1 and Figure 2. On application instances gringo is only capable of grounding and solving 26 instances of S1, which is increased to **50** by NaGG-gringo. On crafted instances, e.g., S3-T6, gringo and idlv ground

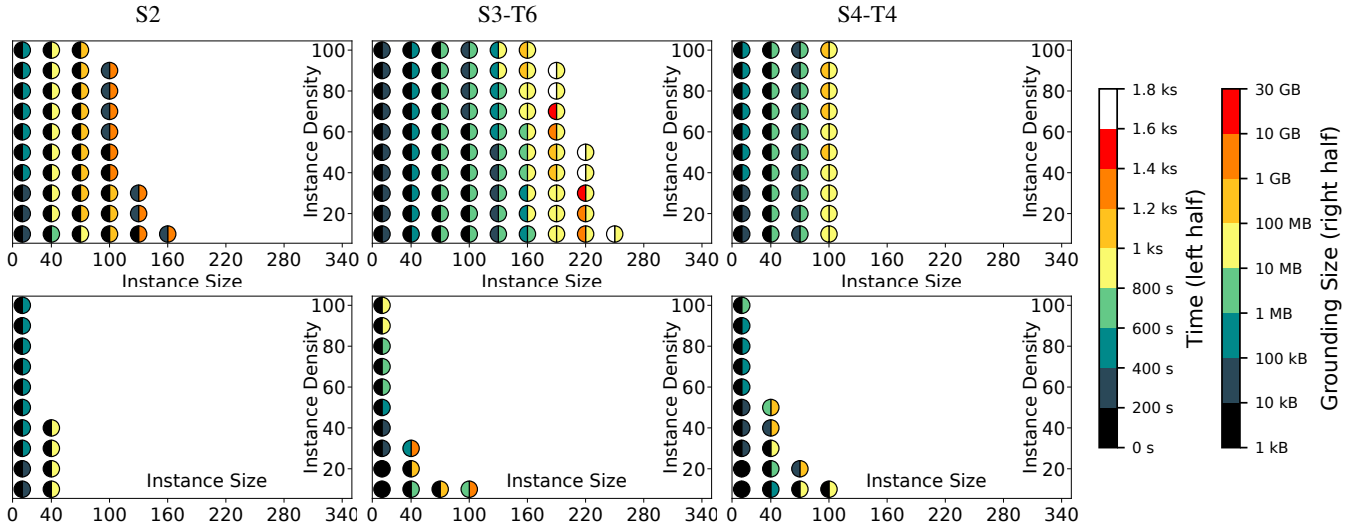


Figure 3: Combined solving profile of NaGG-gringo/NaGG-idlv (top row) and gringo/idlv (bottom row). S2 (grounding-time; first column; top: NaGG-gringo, bottom: gringo), S3-T6 (grounding-time; second column; top: NaGG-gringo, bottom: gringo), S4-T4 (combined-time; third column; top: NaGG-idlv, bottom: idlv). Circles mark instances solved (grounded) within 1800s and 32GB RAM.

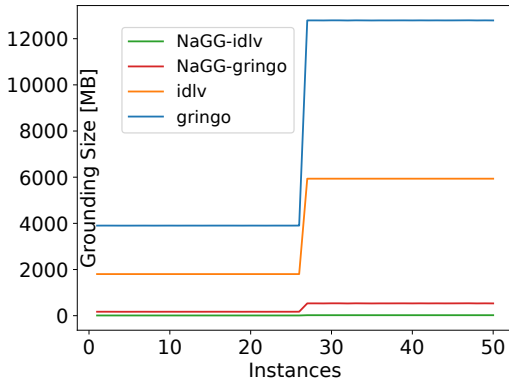


Figure 4: Grounding size for S1 per instance.

and solve 204 instances, which increases to **2078** and **1997** for NaGG-gringo and NaGG-idlv, respectively.

We continue with H2 and refer to Table 1 and Figure 3. Note that the rule densities for S3-T4 to S3-T8 are strictly monotonically increasing. For S3-T4, gringo and idlv ground and solve fewer than 500 instances, which increases to more than 2400 for NaGG, so the relative factor between standard grounders and NaGG is about **5**. For S3-T8 gringo and idlv ground and solve fewer than 160 instances, which increases to more than 1600 for NaGG, so a relative factor of about **10**. Therefore, the relative factor approximately **doubles**. Additionally, observe that in Figure 3 our approach (first row) remains relatively stable with increased instance density, compared to other grounders (second row). Further, we are able to ground and solve larger instances, as seen in Figure 3.

Lastly, we confirm H3 (using Figures 2–4). In Figure 3 (second column), it can be seen that, e.g., for instance size 100, density 10, the grounding size is reduced by a factor of

about **1000**. For these instances, also grounding time is reduced from between 600 and 800 to a value between **0** and **100**. Further, in S1 (Figure 4), we also reduce the grounding size by about a factor of 10. Lastly, Figure 2 confirms a reduction in grounding time, which gives a general trend. There, our approach grounds about 10000 instances, whereas standard grounders manage about 3000 instances.

6 Discussion and Conclusion

Aggregates are a crucial part of many ASP programs. In this work, we improved developments in the area of alternative grounding techniques. Our approach enables hybrid grounding, where we approach the grounding of dense aggregates by shifting efforts to solving, while still applying traditional grounding techniques for the remainder of the program. In order to do so, we establish a flexible and novel splitting theorem on interleaving these techniques with existing traditional groundings. Further, we present different rewriting techniques to translate non-ground aggregates to non-ground rules such that these rules can be efficiently treated by hybrid grounding. Our results show that state-of-the-art grounders can still be improved, despite years of development.

In upcoming works, we plan to tightly integrating our approach into standard grounders like gringo and idlv. Further, we expect hybrid grounding to be useful in the context of lazy grounding and big data. Especially for hard-to-ground instances and application areas, hybrid grounding might be an alternative to traditional approaches; recent work in planning [Corrêa *et al.*, 2023] underlines this perspective. Finally, we want to investigate rewriting techniques (e.g., [Brass and Dix, 1999]) for hybrid grounding. What about structural measures beyond treewidth, see, e.g., [Hecher, 2022; Hecher and Kiesel, 2023; Besin *et al.*, 2023]?

Acknowledgements

Authors are ordered alphabetically. The work has been carried out while Hecher visited the Simons Institute at UC Berkeley. Research is supported by the Austrian Science Fund (FWF), grant J4656; the Society for Research Funding in Lower Austria (GFF) grant ExzF-0004; and Vienna Science and Technology Fund (WWTF) grant ICT19-065.

References

- [Alviano and Faber, 2018] Mario Alviano and Wolfgang Faber. Aggregates in answer set programming. *KI - Künstliche Intelligenz*, 32(2-3):119–124, 2018.
- [Alviano et al., 2019] Mario Alviano, Giovanni Amendola, Carmine Dodaro, Nicola Leone, Marco Maratea, and Francesco Ricca. Evaluation of Disjunctive Programs in WASP. In *LPNMR’19*, volume 11481 of *LNCS*, pages 241–255. Springer, 2019.
- [Banbara et al., 2017] Mutsunori Banbara, Benjamin Kaufmann, Max Ostrowski, and Torsten Schaub. Clingcon: The next generation. *Theory Pract. Log. Program.*, 17(4):408–461, 2017.
- [Bartholomew et al., 2011] Michael Bartholomew, Joohyung Lee, and Yunsong Meng. First-order semantics of aggregates in answer set programming via modified circumscription. In *Papers from the 2011 AAAI Spring Symposium*, 2011.
- [Ben-Eliyahu and Dechter, 1994] Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12(1):53–87, 1994.
- [Besin et al., 2022] Viktor Besin, Markus Hecher, and Stefan Woltran. Body-Decoupled Grounding via Solving: A Novel Approach on the ASP Bottleneck. In *IJCAI’22*, pages 2546–2552. ijcai.org, 2022.
- [Besin et al., 2023] Viktor Besin, Markus Hecher, and Stefan Woltran. On the Structural Complexity of Grounding - Tackling the ASP Grounding Bottleneck via Epistemic Programs and Treewidth. In *26th European Conference on Artificial Intelligence (ECAI 2023)*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 247–254. IOS Press, 2023.
- [Bichler et al., 2020] Manuel Bichler, Michael Morak, and Stefan Woltran. Ipopt: A Rule Optimization Tool for Answer Set Programming. *Fundamenta Informaticae*, 177(3-4):275–296, 2020.
- [Bomanson et al., 2014] Jori Bomanson, Martin Gebser, and Tomi Janhunen. Improving the normalization of weight rules in answer set programs. In Eduardo Fermé and João Leite, editors, *JELIA’14*, pages 166–180. Springer-Verlag, 2014.
- [Brass and Dix, 1999] Stefan Brass and Jürgen Dix. Semantics of (disjunctive) logic programs based on partial evaluation. *J. Log. Program.*, 40(1):1–46, 1999.
- [Brewka et al., 2011] G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Comm. of the ACM*, 54(12):92–103, 2011.
- [Calimeri et al., 2019] Francesco Calimeri, Simona Perri, and Jessica Zangari. Optimizing answer set computation via heuristic-based decomposition. *Theory Pract. Log. Program.*, 19(4):603–628, 2019.
- [Calimeri et al., 2020] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Schaub Torsten. ASP-Core-2 input language format. *Theory Pract. Log. Program.*, 20(2):294–309, 2020.
- [Corrêa et al., 2023] Augusto B. Corrêa, Markus Hecher, Malte Helmert, Davide Mario Longo, Florian Pommerening, and Stefan Woltran. Grounding Planning Tasks Using Tree Decompositions and Iterated Solving. In *ICAPS’23*, pages 100–108. AAAI Press, 2023.
- [Cuteri et al., 2017] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *Theory Pract. Log. Program.*, 17(5-6):780–799, 2017.
- [Cuteri et al., 2020] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators. In *IJCAI’20*, pages 1688–1694. ijcai.org, 2020.
- [Dantsin et al., 2001] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.
- [Dodaro et al., 2023] Carmine Dodaro, Giuseppe Mazzotta, and Francesco Ricca. Answer Set Programming as SAT modulo Acyclicity. In *ECAI’23*. IOS Press, 2023. In Press.
- [Eiter and Gottlob, 1995] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323, 1995.
- [Eiter et al., 2007] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):123–165, 2007.
- [Faber et al., 2011] Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298, 2011.
- [Falkner et al., 2018] Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan. Industrial applications of answer set programming. *Künstliche Intell.*, 32(2-3):165–176, 2018.

- [Ferraris, 2011] Paolo Ferraris. Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic*, 12(4):Article 25 (40 pages), 2011.
- [Gebser *et al.*, 2018] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. Evaluation techniques and systems for answer set programming: A survey. In Jérôme Lang, editor, *IJCAI'18*, pages 5450–5456. ijcai.org, 2018.
- [Gebser *et al.*, 2019] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019.
- [Gelfond and Zhang, 2019] Michael Gelfond and Yuanlin Zhang. Vicious circle principle, aggregates, and formation of sets in ASP based languages. *Artificial Intelligence*, 275:28–77, 2019.
- [Hecher and Kiesel, 2023] Markus Hecher and Rafael Kiesel. The impact of structure in answer set counting: Fighting cycles and its limits. In *KR'23*, 2023. In Press.
- [Hecher, 2022] Markus Hecher. Treewidth-aware reductions of normal ASP to SAT - Is normal ASP harder than SAT after all? *Artif. Intell.*, 304:103651, 2022.
- [Hippen and Lierler, 2021] Nicholas Hippen and Yuliya Lierler. Estimating grounding sizes of logic programs under answer set semantics. In *JELIA'21*, volume 12678 of *LNCS*, pages 346–361. Springer, 2021.
- [Janhunen, 2006] Tomi Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86, 2006.
- [Kaminski and Schaub, 2022] Roland Kaminski and Torsten Schaub. On the foundations of grounding in answer set programming. *Theory Pract. Log. Program.*, pages 1–60, 2022.
- [Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a Logic Program. In *ICLP'94*, pages 23–37. MIT Press, 1994.
- [Lin and Zhao, 2003] Fangzhen Lin and Jicheng Zhao. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *IJCAI'03*, pages 853–858. Morgan Kaufmann, 2003.
- [Marek and Truszczyński, 1991] Wiktor Marek and Mirosław Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38(3):588–619, 1991.
- [Mitchell, 2019] David Mitchell. Guarded constraint models define treewidth preserving reductions. In *CP'19*, volume 11802 of *LNCS*, pages 350–365. Springer, 2019.
- [Pelov *et al.*, 2003] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Translation of aggregate programs to normal logic programs. In Marina De Vos and Alessandro Provetti, editors, *International Workshop on Answer Set Programming (ASP'03)*, pages 29–42, 2003.
- [Tsamoura *et al.*, 2020] Efthymia Tsamoura, Víctor Gutiérrez-Basulto, and Angelika Kimmig. Beyond the grounding bottleneck: Datalog techniques for inference in probabilistic logic programs. In *AAAI'20*, pages 10284–10291. AAAI Press, 2020.
- [Weinzierl *et al.*, 2020] Antonius Weinzierl, Richard Taupe, and Gerhard Friedrich. Advancing lazy-grounding ASP solving techniques - restarts, phase saving, heuristics, and more. *Theory Pract. Log. Program.*, 20(5):609–624, 2020.
- [Zaniolo *et al.*, 2017] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Theory Pract. Log. Program.*, 17(5-6):1048–1065, 2017.