Bypassing the ASP Bottleneck: Hybrid Grounding by Splitting and Rewriting

Alexander Beiser\(^1\), Markus Hecher\(^2\), Kaan Unalan\(^1\) and Stefan Woltran\(^1\)

\(^1\)TU Wien, Vienna, Austria

\(^2\)Massachusetts 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 \( \Sigma^p_2 \) and for disjunctive programs from \( \Sigma^p_2 \) [Eiter and Gottlob, 1995] to \( \Sigma^p_3 \)).

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 \( \Pi_1 \) and \( \Pi_2 \) 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.

Contributions 1. 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/alexsl4123/newground/releases/tag/v2.0.0.
bining 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 \( \Pi \) into a tight program\(^2\) \( \Pi \) with dense rule bodies\(^3\), and a potentially cyclic part \( \Pi_\ell \), forming the remainder. Program \( \Pi \) 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, \( \Pi_\ell \), 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 \( \Pi_\ell \). 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; Kaminska 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., [Barbara 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; Boman-son et al., 2014]. Earlier works on splitting, e.g., [Lifschitz and Turner, 1994] do not focus on grounding strategies.

2 Preliminaries

Ground ASP. Let \( \ell, m, n \) be non-negative integers such that \( \ell \leq m \leq n; a_1, \ldots, a_n \) be distinct propositional atoms. A (disjunctive) program \( \Pi \) is a set of (disjunctive) rules \( a_1 \lor \ldots \lor a_t \leftarrow a_{t+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \). For a rule \( r \), we let \( H_r := \{ a_1, \ldots, a_t \} \), \( B_r^+ := \{ a_{t+1}, \ldots, a_m \} \), and \( B_r^- := \{ \neg a_{m+1}, \ldots, \neg a_n \} \). We denote the sets of atoms occurring in a rule \( r \) or in a program \( \Pi \) by \( \text{at}(r) := H_r \cup B_r^+ \cup B_r^- \) and \( \text{at}(\Pi) := \bigcup_{r \in \Pi} \text{at}(r) \). A rule \( r \) is normal if \( |H_r| \leq 1 \) and a program \( \Pi \) is normal if all its rules are normal. The dependency graph \( D_{\Pi} \) is the directed graph defined on vertices \( \bigcup_{r \in \Pi} H_r \cup B_r^+ \), s.t. for every \( r \in \Pi \) two atoms \( a \in B_r^+ \) and \( b \in H_r \) are joined by edge \((a, b)\). A head-cycle of \( D_{\Pi} \) is an \((a, b)\)-cycle\(^4\) for two distinct atoms \( a, b \in H_r \) for some rule \( r \in \Pi \). \( P \) is head-cycle-free (HCF) or tight if \( D_{\Pi} \) 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 \( \Pi \) if it satisfies all rules of \( \Pi \). The Gelfond-Lifschitz (GL) reduct of \( \Pi \) under \( I \) is the program \( \Pi^I \) obtained from \( \Pi \) 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 \( \Pi \) if \( I \) is a minimal model w.r.t. \( \subseteq \) of \( \Pi \). Deciding whether a program has an answer set is called consistency, which is \( \Sigma_2^P \)-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 \( \psi: I \rightarrow \{0, \ldots, |I| - 1\} \) assigns every atom in \( I \) a unique value [Lin and Zhao, 2003; Janhunen, 2006]. Given an interpretation \( I \) of an HCF program \( \Pi \) and a level mapping \( \psi \) of \( I \). An atom \( a \in I \) is founded if for an \( r \in \Pi \) with \( a \in H_r \), \( B_r^+ \subseteq I \), \( \psi(b) < \psi(a) \) for every \( b \in B_r^+ \), and \( I \cap B_r^- = I \cap (H_r \setminus \{a\}) = \emptyset \). \( I \) is an answer set of \( \Pi \) if (i) \( I \) is a model of \( \Pi \), and (ii) all atoms in \( I \) are founded.

Non-ground ASP. We use vectors \( X := \langle x_1, \ldots, x_m \rangle \) in the usual way. We combine vectors by \( (X, Y) := \langle X_1, \ldots, Y_1 \rangle \) and test whether \( x_1 \) is in \( X \) by \( x_1 \in X \). 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 \( p_1, \ldots, p_n \) 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 \( \langle <, \leq, >, \geq, \neq \rangle \), in their usual meaning.

A literal is a predicate, or its negation. A (non-ground) program \( \Pi \) is a set of (non-ground) rules of the form

\[
\psi_1(X_1) \land \ldots \land \psi_t(X_t) \leftarrow \psi_{t+1}(X_{t+1}), \ldots, \psi_m(X_m), \quad (1)
\]


\[
\neg \psi_{m+1}(X_{m+1}), \ldots, \neg \psi_n(X_n).
\]

For every variable vector \( X \), we have \( |X| = |p_i| \) and whenever \( x \in \langle X_1, \ldots, X_t, X_{m+1}, \ldots, X_n \rangle \), then \( x \in \langle X_{t+1}, \ldots, X_m \rangle \) (safeness). For a non-ground rule \( r \), we define \( H_r := \{ \psi_1(X_1), \ldots, \psi_t(X_t) \} \), \( B_r^+ := \{ \psi_{t+1}(X_{t+1}), \ldots, \psi_m(X_m) \} \), as well as \( B_r^- := \{ \psi_{m+1}(X_{m+1}), \ldots, \psi_n(X_n) \} \). We furthermore define \( B_r := B_r^+ \cup B_r^- \), \( \text{var}(r) := X \in \{X \mid p(X) \in H_r \cup B_r \} \), and use \( \text{hpred}(\{I\}) := \{ p(X) \in H_r \mid r \in I \} \) and \( \text{pred}(\{I\}) := \{ p(X) \in H_r \cup B_r \mid r \in I \} \). For brevity, we assume variables are unique per rule, i.e., for every two rules \( r, r' \in I \), \( \text{var}(r) \cap \text{var}(r') = \emptyset \). Attributes disjunctive, normal, and tight carry over to rules (programs). In order to ground \( \Pi \), we require a given set \( F \) of facts, i.e., atoms of the form \( p(D) \) with \( p \) being a predicate of \( I \) and \( D \) being a vector over domain values of size \( |D| = |p| \). We say that \( D \) is part of the domain of \( I \), defined by \( \text{dom}(\Pi) := \{ d \in D \mid p(D) \in F \} \). We refer to the domain vectors over \( \text{dom}(\Pi) \) for a variable vector \( X \) of size \( |X| \) by \( \text{dom}(X) \). Let \( D \) be a domain vector over variable vector \( X \) and vector \( Y \) containing only variables of \( X \). We refer to the domain vector \( D \) restricted to \( Y \) by \( D_Y \). The grounding \( G(\Pi) \) comprises \( F \) and rules obtained by replacing each rule \( r \) of Form (1) for

\[3251\]
every $D \in \text{dom}((\text{var}(r)))$ by $p_1(D_{X_1}) \lor \ldots \lor p_m(D_{X_m}) \leftarrow p_{r+1}(D_{X_{r+1}}), \ldots, p_m(D_{X_m})$. The dependency graph $D_{\Pi}$ of the non-ground program $\Pi$ is defined over the ground program $G(\Pi)$; $\text{sc}(\Pi)$ refers to the set of strongly-connected components (vertices) of $D_{\Pi}$.

**Example 1.** Consider the non-ground program $\Pi_1 := \{r_1\}$ with $r_1 = a(X,Y) \leftarrow b(Y), c(Y,Z), d(Y)$ and $F_1 := \{b(1); c(1,2); d(2); 1\}$. Observe that $\text{dom}(X) = \{1\}$ and $\text{dom}(Y) = \text{dom}(Z) = \{1,2\}$. Grounding $G(\Pi_1)$ comprises: $\{b(1); c(1,2); d(1); a(1,1) \leftarrow b(1), c(1,1), d(1); a(1,1) \leftarrow b(1), c(1,2), d(1); a(1,2) \leftarrow b(1), c(2,1), d(2); a(1,2) \leftarrow b(1), c(2,2), d(2)\}$. The answer set of $\Pi_1$ is $\{b(1); c(1,2); d(1); a(1,1); d(1)\}$.

**Aggregates.** We denote aggregates as specified in the ASP-Core-2 standard [Calimeri et al., 2020]. Let $T = (t_1, \ldots, t_o)$ be a term vector and $L = (l_1(Y_1), \ldots, l_k(Y_k))$ be a literal vector. An (aggregate) element $e$ is of the form $T \cdot L$, where $T$ is the head of $e$ of length $o$ and $L$ is its body. We require variables in $T$ to occur in a non-negated predicate of $L$. We refer to $E = \{e_1, \ldots, e_o\}$ as the element multiset ($|E| = o$).

A (aggregate) is of the form $agg(E) < u$, $agg \in \{\text{count}, \text{sum}, \text{min}, \text{max}\}$, $u$ is a term, and $\langle \leq, <, \geq, =, \neq \rangle$. 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 $agg(E) < u$, 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 $agg$ 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 $agg(R) < u$.

**Example 2.** Consider the non-ground program $\Pi_2 := \{r_2\}$ with $r_2 = \langle e(X,Y), \text{count}(A,B): e(A,B), f(A,X): A: g(A), e(A,Y), A: h(A) \rangle \geq 3$, and $F_2 := \{e(1,1); f(1,1); g(1); g(2); h(1); h(3)\}$. 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 $G(\Pi_2)$ results in the following program, which has no answer set: $\langle e(1,1); f(1,1); g(1); g(2); h(1); h(3)\rangle < e(1,1), \text{count}(\{(1,1),(1),(3)\}) \geq 3$.

In general, we rewrite aggregate expressions across non-ground programs, for enabling efficient hybrid grounding. Still, we define for aggregate programs the dependency graph $D_{\Pi}$, and $\text{sc}(\Pi)$ analogously. Further, we require a rule $r$ containing an aggregate that the corresponding $p_i(D_{X_i}) \in H_r$, and $p_j(D_{X_j}) \in F_r^+$ are not contained in a cycle in $D_{\Pi}$. 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 $\Pi$ into two parts: A dense tight program $\Pi_t$ and the remainder $\Pi_r = \Pi \setminus \Pi_t$. 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 $F_3 := \{b(1); c(2)\} \cup G$, 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 $\Pi_t$, $\Pi_r$, 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 $\Pi_t$. This will allow us to distinguish the (potentially interleaving) cases, where an atom is founded by $\Pi_t$, and those where the atom is derived due to $\Pi_r$.

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

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

Then, Rules (4)–(8) ensure rule satisfiability of $\Pi_t$. 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 $\Pi_t$. After that, rule satisfiability can be derived for all rules of $\Pi_t$ 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 $\Pi_t$, 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 $\Pi$, 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 $\Pi$ into a program $\Pi_1$ and a tight program $\Pi_2$, such that for every $S \in \text{sc}(\Pi)$, we have $|E(S_{\Pi_1})| = 0$, where $E(S_{\Pi_1})$ is the edge set of the subgraph $S_{\Pi_1}$ of $G_{\Pi_1}$ induced by $S \cap \text{at}(\Pi_1)$. Then, the answer sets of $\mathcal{H}(\Pi_1, \Pi_2)$ restricted to $\mathcal{G}(\Pi_1)$ bijectively match those of $\mathcal{G}(\Pi)$.

**Proof (Sketch).** Correctness of BDG under the assumption $\text{hpred}(\Pi_1) \cap \text{hpred}(\Pi_2) = \emptyset$ has been shown in [Besin et al., 2022]. In this case, predicates $\text{hpred}(\Pi_1)$, $\text{hpred}(\Pi_2)$ can only appear in rule bodies of $\Pi_1$, $\Pi_2$, respectively. Therefore, foundedness of atoms over predicates $\text{hpred}(\Pi_i)$ is checked by Rules (9)–(12) and foundedness of atoms over $\text{hpred}(\Pi_1)$ is derived directly by Rules (3). Satisfiability of $\Pi$ is ensured by Rules (4)–(8); satisfiability of $\Pi_1$ is treated by Rules (3).

Suppose $\text{hpred}(\Pi_1) \cap \text{hpred}(\Pi_2) \neq \emptyset$. As now atoms over $\text{hpred}(\Pi_1)$, $\text{hpred}(\Pi_2)$ may be derived in $\Pi_1$, $\Pi_2$ but may also appear in rule heads of $\Pi_1$, $\Pi_2$, respectively, this might lead to incorrect results. We rename all head predicates $h(X) \in \text{hpred}(\Pi_1)$ to $h'(X)$. This (again) ensures foundedness of atoms over $\text{hpred}(\Pi_1)$ and satisfiability of $\Pi_1$, only by Rules (9)–(12) and (4)–(8), respectively, while $\Pi_2$ continues to use atoms derived by $\Pi_1$ in its rule bodies. However, $\Pi_2$ has to inform $\Pi_1$ about atoms derived in $\Pi_1$, ensured by Rules (2). This still applies if $\Pi_1$ is cyclic, as long as there is no shared cycle between $\Pi_1$ and $\Pi_2$.

It remains to argue, why despite Rules (2), every answer set $M$ of $\mathcal{H}(\Pi_1, \Pi_2)$ has a unique answer set $M \cap \text{at}(\Pi_2)$ of $\Pi_2$. Consider an arbitrary body $h(D) \in \text{at}(\mathcal{G}(\Pi))$. If $h(D) \notin M$, $h'(D) \notin M$ due to Rules (2), making $M \cup \{h(D)\}$ unique.

Suppose $h(D) \in M$. Case $h'(D) \in M$: Then, $h(D)$ has to be founded by $\Pi_2$. Hence, there is an $r \in \Pi_2$ with $sat_r \in M$ due to Rules (7). So, there is no answer set $M' \neq M$ identical to $M$ over $\text{at}(\mathcal{G}(\Pi))$, s.t. $h'(D) \notin M'$, since $sat_r \notin M'$.

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

**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 $\Pi$ into a tight program $\Pi_1$, and the remainder $\Pi_2 \setminus \Pi_1$. This yields the theorem below, which is the key result of hybrid grounding, permitting the (potentially cyclic) sharing of predicates between split programs $\Pi_1$, $\Pi_2$.

With level mappings, we can extend $\mathcal{H}$ to work despite cycles over $\Pi_1$ and $\Pi_2$. This yields reduction $\mathcal{H}_{\text{lev}}$ (Appendix Figure 7), where we use auxiliary predicates to encode level mappings over each two atoms $p_1(D_1)$, $p_2(D_2)$ for any domain vectors $D_1$, $D_2$ 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 $h(D)$ of a rule $r$ is unfounded, if a body atom over a predicate in pred(r) \ hpred(r) does not precede $h(D)$. This prevents abusing cyclic foundedness between both $\Pi_1$ and $\Pi_2$.

**Theorem 1 (Grounding-Splitting Theorem).** Given a partition of any non-ground HCF program $\Pi$ into a HCF program $\Pi_1$ and a tight program $\Pi_2 = \Pi \setminus \Pi_1$. The answer sets of $\mathcal{H}_{\text{lev}}(\Pi_1, \Pi_2)$ restricted to at($\mathcal{G}(\Pi)$) match those of $\mathcal{G}(\Pi)$.

Note that $\mathcal{H}_{\text{lev}}$ can be further extended, such that $\Pi_1$ is any normal (HCF) program. This requires level mappings as above, thereby also excluding cyclic foundedness within $\Pi_1$.

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

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

## 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 $(\text{count}_c)$ in the positive body of a rule $r$ $(\text{count}_c \in B_r^+)$. Further, we require $\text{count}_c$ to have an integer bound $\geq u$ and a single element $c = T_e : B_e$, comprising head $T_e = \{t_1, \ldots, t_u\}$ and body $B_e = l_1(Y_1), \ldots, l_u(Y_u)$. 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 $B_e$‘s, with variable indexing as described above, and (ii) add alldiff constraints for all (indexed, $1 \leq i \leq u$) $T_e$‘s. The resulting rewritten rule is depicted in Equation (13), where $B_r^+ := B_r^+ \setminus \{\text{count}_c\}$, yielding rewriting procedure $\mathcal{R}_{\text{MC}}$ (Rewriting Monotonic Count).
Glue \( \Pi_t \) to \( \Pi_c \) and Ground \( \Pi_c \)

\[ h'(D) \lor h''(D) \leftarrow h(D) \leftarrow h'(D) \]
\[ r \]

**Satisfaction of \( \Pi_t \)**

\[ \forall d \in \text{dom}(x) \]
\[ \text{sat} \leftarrow \text{sat}_x(D(x_1)), \ldots, \text{sat}_x(D(x_t)), \neg p(D) \]
\[ \text{sat}_r \leftarrow \text{sat}_x(D(x_1)), \ldots, \text{sat}_x(D(x_t)), p(D) \]
\[ \text{sat}_r \leftarrow \text{sat}_x(D(x_1)), \ldots, \text{sat}_x(D(x_t)), h'(D) \]
\[ \text{sat}_r(d) \leftarrow \text{sat} \leftarrow \neg \text{sat} \]

**Prevent Unfoundedness of Atoms in \( \Pi_t \)**

\[ \forall u, f_u((D, d)) \leftarrow h'(D) \]
\[ \forall d \in \text{dom}(y) \]
\[ u_r(D_X) \leftarrow u_f(D(X,y_1)), \ldots, u_f(D(X,y_t)), \neg p(D_Y) \]
\[ u_f(D_X) \leftarrow u_f(D(X,y_1)), \ldots, u_f(D(X,y_t)), p(D_Y) \]
\[ u_r(D), \ldots, u_r(D), h'(D) \]

**Rewriting Procedure \( \mathcal{RM}_c^{\text{count}} \)**

\[ H_t \leftarrow B^+_e, B^−_e, B_{e, 1}, \ldots, B_{e,u}, \text{alldiff}(T_{e, 1}, \ldots, T_{e,u}) \]  \( (13) \)

**Results.** Let \( h = |T| \) be the length of the element’s head, and \( \phi_r \) be the maximum arity of \( r \) (before rewriting). Then, Lemma 2 upper bounds the rewritten rule’s arity.

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

### 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 \( \mathcal{RM}_c^{\text{count}} \) 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 \( E \). Element \( E \) has a **combined head** \( (T') \), and a single **body predicate** \( (tp(T')) \), i.e., \( E = T' : tp(T') \). Note that the arity of the combined element head \( T' \) has to be the maximum arity over all elements \( e' (|T'| = \max_e |T'_e|) \). Intuitively, a **ground atom** over predicate \( tp \) holds, whenever there is at least one aggregate element that founds it. Therefore, we introduce for each element \( e = T_e : B_e \) a new rule: \( tp(T_e) \leftarrow B_e \). However, this rule fails in case of different head element arities. Note, for each element \( e \), its head arity \( |T_e| \) fulfills \( |T_e| \leq |T'| \). So we adapt this idea in the following way: If for an element \( e \) we have \( |T' | - |T_e| > 0 \) 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 \( |T'| = 3 \), and for elements 1, 2 and 3: \( |T_1| = 1 \), \( |T_2| = 2 \) and \( |T_3| = 3 \). By applying our updated Equation (14a) we obtain the following three rules \( (c_1, \text{and } c_2 \text{ are fresh constants}) \):

\[ \{tp(X_1, c_1, c_2) \leftarrow p(X_1) ; tp(X_2, Y_2, c_2) \leftarrow k(X_2, Y_2) ; tp(X_3, Y_3, Z_3) \leftarrow l(X_3, Y_3, Z_3)\} \]

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

**Rewriting Procedure \( \mathcal{RS}_c^{\text{count}} \)**

\[ \text{For every elem. } e : tp(T'_1, c_1, \ldots, c_{|T'_1| - |T'_e|}) \leftarrow B_e \]  \( (14a) \)
\[ \text{count}_{\leq} \leftarrow tp(T'_1), \ldots, tp(T'_u), \text{alldiff}(T'_1, \ldots, T'_u) \]  \( (14b) \)
\[ H_t \leftarrow \text{count}_{\leq}, B^+_e, B^−_e \]  \( (14c) \)
While we consider lazy grounding related, we only demonstrate the potential of this technique well. 

NaGG implements multiple rewriting strategies (Appendix Table 2). Our prototypical grounder is based on Python3 and the Novel AggreGate Grounder (NaGG), resulting in the system

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 $\Pi_k$ and gringo(idlv) grounds the remainder $\Pi_r$, respectively. By design, NaGG is most beneficial if $\Pi_r$ comprises only rules of larger bodies (with many variable dependencies), whereas $\Pi_k$ only contains rules of slim bodies. So, hybrid grounding avoids large groundings of $\Pi_r$, 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 NaGG-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...
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


[Zaniolo et al., 2017] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Inte-