Towards Lightweight Completion Formulas for Lazy Grounding in Answer Set Programming

Bart Bogaerts[1] , **Simon Marynissen**[1,2] , **Antonius Weinzierl**[3]
[1]Vrije Universiteit Brussel [2]KU Leuven [3]TU Wien
bart.bogaerts@vub.be, simon.marynissen@kuleuven.be, antonius.weinzierl@kr.tuwien.ac.at

Abstract

Lazy grounding is a technique for avoiding the so-called grounding bottleneck in Answer Set Programming (ASP). The core principle of lazy grounding is to only add parts of the grounding when they are needed to guarantee correctness of the underlying ASP solver. One of the main drawbacks of this approach is that a lot of (valuable) propagation is missed. In this work, we take a first step towards solving this problem by developing a theoretical framework for investigating *completion formulas* in the context of lazy grounding.

1 Introduction

Answer set programming (ASP) (Marek and Truszczyński 1999) is a well-known knowledge representation paradigm in which logic programs under the stable semantics (Gelfond and Lifschitz 1988) are used to encode problems in the complexity class NP and beyond. From a practical perspective, ASP offers users a rich first-order language, ASP-Core2 (Calimeri et al. 2013), to express knowledge in, and many efficient ASP solvers (Gebser, Maratea, and Ricca 2017) can subsequently be used to solve problems related to knowledge expressed in ASP-Core2.

Traditional ASP systems work in two phases. First, the input program is *grounded* (variables are eliminated). Second, a solver is used to find the stable models of the resulting ground theory. For a long time, the ASP community has focused strongly on developing efficient solvers, while only a few grounders were developed. Most modern ASP solvers are in essence extensions of satisfiability (SAT) (Marques Silva, Lynce, and Malik 2009) solvers, building on conflict-driven clause learning (CDCL) (Marques-Silva and Sakallah 1999). In recent years, in many formalisms that build on top of SAT, we have seen a move towards only generating parts of the SAT encoding on-the-fly, on moments when it is deemed useful for the solver. This idea lies at the heart of the CDCL(T) algorithm for SAT modulo theories (Barrett et al. 2009) and is embraced under the name lazy clause generation (Stuckey 2010) in constraint programming (Rossi, van Beek, and Walsh 2006). Answer set programming is no exception: the so-called *unfounded set* propagator and aggregate propagator are implemented using the same principles; when needed, they generate clauses for the underlying SAT algorithm. Additionally, lazy clause generation forms the basis of recent constraint ASP solvers (Banbara et al. 2017).

Lazy grounding takes the idea of lazily generating the SAT encoding one step further by also lazily performing the *grounding* process. That is, ASP rules are only instantiated when some algorithm detects that they are useful for the solver in its current state. The most prominent class of lazy grounding systems for ASP is based on *computation sequences* (Liu et al. 2007) and includes systems such as Omiga (Dao-Tran et al. 2012), GASP (Dal Palù et al. 2009), ASPeRiX (Lefèvre and Nicolas 2009) and the recently introduced ALPHA (Weinzierl 2017). The latter is the youngest and most modern of the family and the only one that integrates lazy grounding with a CDCL solver, resulting in superior search performance over its predecessors. Our work extends the ALPHA algorithm.

Contrary to more traditional ASP systems, lazy grounding systems aim more at applications in which the full grounding is so large that simply creating it would pose issues (e.g., if it does not fit in your main memory). This phenomenon is known as the *grounding bottleneck* (Balduccini, Lierler, and Schüller 2013). Examples of such problems include queries over a large graph; planning problems, with a very large number of potential time steps, or problems where the full grounding contains a lot of unnecessary information and the actual search problem is not very hard.

The essential idea underlying lazy grounding is that all parts of the grounding that do not help the solver in its quest to find a satisfying assignment (a stable model) or prove unsatisfiability are better not given to the solver since they only consume precious time and memory. Unfortunately, it is not easy to detect which parts that are and a trade-off shows up (Taupe, Weinzierl, and Friedrich 2019): producing larger parts of the grounding will improve search performance (e.g., propagation can prune larger parts of the search space) but grounding too much will — on the type of instances lazy grounding is built for — result in an unmanageable explosion of the ground theory. Lazy grounding systems and ground-and-solve systems reside on two extremes of this trade-off: the former produce a minimal required part of the theory to ensure correctness while the latter produce the entire bottom-up grounding.

Our work moves lazy grounding a bit more to eager side of this trade-off. Specifically, we focus on *completion formulas* (Clark 1978) that essentially express that when an atom is true, there must be a rule that *supports it* (a rule with

true body and that atom in the head). While ground-and-solve systems add these formulas (in the form of clauses) to their ground theory, lazy grounders cannot do this easily; the reason is that the set of ground rules that could derive a certain atom is not known (more instantiations could be found later on). Consider, for example the atom $p(a)$ and a rule $p(X) \leftarrow q(X,Y)$. where the set of ground instantiations of this rule with $p(a)$ in the head depends on the set of atoms over the binary predicate q. Unless those instances over q are fully grounded, a lazy grounder cannot add the corresponding completion formula. In this paper, we develop lightweight algorithms to detect when that set of rules is complete and hence, when completion formulas are added. Our hypothesis is that doing this will improve search performance without blowing up the grounding and as such result in overall improved performance of lazy-grounding ASP systems, and specifically the ALPHA system.

The main contribution of our paper is the development of a novel method to discover completion formulas during lazy grounding. Our method starts from a static analysis of the input program in which we discover functional dependencies between variable occurrences. During the search, this static analysis is then used to figure out the right moment to add the completion formulas in a manner that is inspired by the two-watched literal scheme from SAT to avoid adding the completion constraints on moments they have no chance of propagating anyway. We do not have an implementation of this idea available yet, but instead focus on the theoretical principles.

The rest of this paper is structured as follows. In Section 2 we recall some preliminaries. Section 3 contains the different methods for discovering completion formulas. In Section 4, we discuss extensions of our work that could be used to find even more completion formulas. We conclude in Section 5.

2 Preliminaries

We now introduce some preliminaries related to answer set programming in general and the ALPHA algorithm specifically. This section is based on the preliminaries of (Bogaerts and Weinzierl 2018).

Answer set programming. Let $\mathcal{C}$ be a set of *constants*, $\mathbb{V}$ be a set of *variables*, and $\mathcal{Q}$ be a set of *predicates*, each with an associated arity, i.e., elements of $\mathcal{Q}$ are of the form p/k where p is the predicate name and k its arity. We assume the existence of built-in predicates, such as equality, with a fixed interpretation. A (non-ground) *term* is an element of $\mathcal{C} \cup \mathbb{V}$.[1] The set of all terms is denoted $\mathcal{T}$. Our definition of a term does not allow for nesting. This eases our exposition, but is not essential for our results. For instance, it allows us to view $+$ as a ternary predicate $+/3$, i.e. $+(X,Y,Z)$ means that $X+Y=Z$. A (non-ground) *atom* is an expression of the form $p(t_1,\ldots,t_k)$ where $p/k \in \mathcal{Q}$ and $t_i \in \mathcal{T}$ for each i.

[1] Following Weinzierl (2017), we omit function symbols to simplify the presentation. All our results still hold in the presence of function symbols, except for termination, for which additional (syntactic) restrictions must be imposed.

The set of all atoms is denoted by $\mathcal{A}$. If $a \in \mathcal{A}$, then $\mathrm{var}(a)$ denotes the set of variables occurring in a. We say that a is *ground* if $\mathrm{var}(a) = \emptyset$. The set of all ground atoms is denoted $\mathcal{A}_{\mathrm{gr}}$. A *literal* is an atom p or its negation $\neg p$. The former is called a *positive* literal, the latter a *negative* literal. Slightly abusing notation, if l is a literal, we use $\neg l$ to denote the literal that is the negation of l, i.e., we use $\neg(\neg p)$ to denote p. The set of all literals is denoted $\mathcal{L}$ and the set of ground literals $\mathcal{L}_{\mathrm{gr}}$. A *clause* is a disjunction of literals. A *(normal) rule* is an expression of the form

$$p \leftarrow L$$

where p is an atom and L a set of literals. If r is such a rule, its *head*, *positive body*, *negative body* and *body* are defined as $\mathrm{H}(r) = p$, $\mathrm{B}^+(r) = \mathcal{A} \cap L$, $\mathrm{B}^-(r) = \{q \in \mathcal{A} \mid \neg q \in L\}$ and $\mathrm{B}(r) = L$ respectively. We call r a *fact* if $\mathrm{B}(r) = \emptyset$ and *ground* if p and all literals in L are ground. We use $\mathrm{var}(r)$ to denote the set of variables occurring in r, i.e.,

$$\mathrm{var}(r) = \mathrm{var}(p) \cup \bigcup_{q \in L} \mathrm{var}(q).$$

A rule r is *safe* if all variables in r occur in its positive body, i.e., if $\mathrm{var}(r) \subseteq \mathrm{var}(\mathrm{B}^+(r))$. A *logic program* $\mathcal{P}$ is a finite set of safe rules. $\mathcal{P}$ is ground if each $r \in \mathcal{P}$ is. In our examples, logic programs are presented in a more general format, using, e.g., choice rules (see (Calimeri et al. 2020)). These can easily be translated into the format considered here.

If X is a set of variables, a *grounding substitution* of X is a mapping $\sigma : X \to \mathcal{C}$. The set of all substitutions of X is denoted $sub(X)$ If e is an expression, a *grounding substitution* for e is a grounding subtitution of its variables. We write $[c_1/X_1,\ldots,c_n/X_n]$ for the substitution that maps each X_i to c_i and each other variable to itself. The result of applying a substitution σ to an expression e is the expression obtained by replacing all variables X by $\sigma(X)$ and is denoted $\sigma(e)$. The most general unifier of two substitutions is defined as usual (Martelli and Montanari 1982). A substitution σ extends a substitution τ if σ is equal to τ in the domain of τ. The *grounding* of a rule is given by

$$\mathrm{gr}(r) = \{\sigma(r) \mid \sigma \text{ is a grounding substitution}\}$$

and the (full) grounding of a program $\mathcal{P}$ is defined as $\mathrm{gr}(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} \mathrm{gr}(r)$.

A *(Herbrand) interpretation* I is a finite set of ground atoms. The satisfaction relation between interpretations and literals is given by

$$I \models p \text{ if } p \in I, \text{ and}$$
$$I \models \neg p \text{ if } p \notin I.$$

An interpretation satisfies a set L of literals if it satisfies each literal in L. A *partial (Herbrand) interpretation* $\mathcal{I}$ is a *consistent* set of ground literals (consistent here means that it does not contain both an atom and its negation). The value of a literal l in a partial interpretation $\mathcal{I}$ is $l^{\mathcal{I}} = \mathbf{t}$ if $l \in \mathcal{I}$, $\mathbf{f}$ if $\neg l \in \mathcal{I}$ and $\mathbf{u}$ otherwise.

Given a (partial) interpretation $\mathcal{I}$ and a ground program $\mathcal{P}$, we inductively define when an atom is *justified* (Denecker, Brewka, and Strass 2015) as follows. An atom p is *justified*

in $\mathcal{I}$ by $\mathcal{P}$ if there is a rule $r \in \mathcal{P}$ with $\mathrm{H}(r) = p$ such that each $q^+ \in \mathrm{B}^+(r)$ is justified in $\mathcal{I}$ by $\mathcal{P}$ and each $q^- \in \mathrm{B}^-(r)$ is false in $\mathcal{I}$. A built-in atom is justified in $\mathcal{I}$ by $\mathcal{P}$ if it is true in $\mathcal{I}$.

An interpretation I is a *model* of a ground program $\mathcal{P}$ if for each rule $r \in \mathcal{P}$ with $I \models \mathrm{B}(r)$, also $I \models \mathrm{H}(r)$. An interpretation I is a *stable model* (or *answer set*) of a ground program $\mathcal{P}$ (Gelfond and Lifschitz 1988) if it is a model of $\mathcal{P}$ and each true atom in I is justified in I by $\mathcal{P}$. This non-standard characterization of stable models coincides with the original reduct-based characterization, as shown by Denecker, Brewka, and Strass (2015) but simplifies the rest of our presentation. If $\mathcal{P}$ is non-ground, we say that I is an answer set of $\mathcal{P}$ if it is an answer set of $\mathrm{gr}(\mathcal{P})$. The set of all answer sets of $\mathcal{P}$ is denoted $\mathcal{AS}(\mathcal{P})$.

The ALPHA algorithm. We now recall the formalization of ALPHA of Bogaerts and Weinzierl (2018). This differs from the original presentation of Weinzierl (2017) in that it does not use the truth value MUST-BE-TRUE, but instead makes the justifiedness of atoms explicit. The state of ALPHA is a tuple $\langle \mathcal{P}, \mathcal{P}_g, C, \alpha, S_J \rangle$, where

- $\mathcal{P}$ is a logic program,
- $\mathcal{P}_g \subseteq \mathrm{gr}(\mathcal{P})$ is the so-far grounded program; we use $\Sigma_g \subseteq \mathcal{A}_{\mathrm{gr}}$ to denote the set of ground atoms that occur in $\mathcal{P}_g$,
- C is a set of (learned) clauses,
- α is the trail; this is a sequence of tuples (l, c) with l a literal and c either the symbol δ, a rule in $\mathcal{P}_g$ or a clause in C. α is restricted to not containing two tuples (l, c) and $(\neg l, c')$; in a tuple $(l, c) \in \alpha$, c represents the reason for making l true: either decision (denoted δ) or propagation because of some rule or clause; α implicitly determines a partial interpretation denoted $\mathcal{I}_\alpha = \{l \mid (l, c) \in \alpha \text{ for some } c\}$.
- $S_J \subseteq \mathcal{A}$ is the set of atoms that are *justified* by $\mathcal{P}_g$ in I_α.

For clause learning and propagation, a rule $p \leftarrow L$ is treated as the clause $p \vee \bigvee_{l \in L} \neg l$. Hence, whenever we refer to "a clause" in the following, we mean any rule in $\mathcal{P}_g$ (viewed as a clause) or any clause in C. We refer to *rules* whenever the rule structure is needed (for determining justified atoms).

ALPHA interleaves CDCL and grounding. It performs (iteratively) the following steps (listed by priority).

conflict If a clause in $C \cup \mathcal{P}_g$ is violated, analyze the conflict, learn a new clause (add to C) and back-jump (undo changes to α and S_J that happened since a certain point) following the so-called 1UIP schema (Zhang et al. 2001).

(unit) propagate If all literals of a clause $c \in C \cup \mathcal{P}_g$ except for l are false in $\mathcal{I}_\alpha$, add (l, c) to α.

justify If there is a rule r such that $\mathrm{B}^+(r) \subseteq S_J$ and $\neg\mathrm{B}^-(r) \subseteq \mathcal{I}_\alpha$, add $\mathrm{H}(r)$ to S_J.

ground If, for some grounding substitution σ and $r \in \mathcal{P}$, $\mathrm{B}^+(\sigma(r)) \subseteq \mathcal{I}_\alpha$, add $\sigma(r)$ to $\mathcal{P}_g$. In practice, when adding this rule, ALPHA makes a new – intermediate – propositional variable $\beta(\sigma(r))$ to represent the body of the rule, similar to (Anger et al. 2006).

decide Pick (using some heuristics (Taupe, Weinzierl, and Schenner 2017)) one atom p, occurring in $\mathcal{P}_g$ that is unknown in $\mathcal{I}_\alpha$ and add (p, δ) or $(\neg p, \delta)$ to α.[2]

justification-conflict If all atoms in $\mathcal{P}_g$ are assigned while some atom is true but not justified, learn a new clause that avoids visiting this assignment again. Worst-case the learned clause contains the negation of all decisions, but Bogaerts and Weinzierl (2018) developed more optimized analysis methods. After learning this clause, ALPHA backjumps.

3 Deriving Completion Formulas

We now discuss our modifications to the ALPHA algorithm that allow us to add completion formulas. There are two main problems to be tackled here: the first, and most fundamental is **Question 1:** *how to generate completion clauses*, or stated differently, how to find all the rules that can derive a certain atom, *without* creating the full grounding, and the second is, **Question 2:** *when to add completion formulas to the solver*. The general idea for the generation is that we will develop approximation methods that overapproximate the set of instantiations of rules that can derive a given atom based on a **static analysis** of the program. The reason why we look for an overapproximation is since in general finding the exact set of such instantiations would require a semantical analysis. Our methods below are designed based on the principle that such an overapproximation should be as tight as possible. Specifically, our methods will be based on *functional dependencies* and *determined predicates*.

This section starts by proving definitions for bounds. After that we explain how bounds can be used in ALPHA. The last subsection describes the different type of bounds and how they can be detected and combined.

3.1 Bounds

The core concept of our detection mechanism is the notion of *bounds*. We have already stated that we want to find overapproximations of grounding substitutions. We now formalize this.

Definition 3.1. *Given a rule r in a program $\mathcal{P}$. A grounding substitution σ is* relevant *in r with respect to $\mathcal{P}$ if $\mathrm{B}^+(\sigma(r))$ is justified in some partial interpretation of $\mathcal{P}$.*

The following lemma follows immediately from the characterization of stable models in terms of justifications (Denecker, Brewka, and Strass 2015).

Lemma 3.2. *Let I be an answer set of $\mathcal{P}$. If $I \models p$, then there is a rule r in $\mathcal{P}$ and a relevant substitution σ in r such that $\sigma(\mathrm{H}(r)) = p$.*

Proof. Since the justification characterization of answer sets, we know that p is justified in I. Then by the definition of justified, the proof follows. □

Definition 3.3. *Given a rule r and two sets X and Y of variables in r. A function $f\colon sub(X) \to 2^{sub(Y)}$ is called*

[2] ALPHA actually only allows deciding on certain atoms (those of the form $\beta(r)$), hence our presentation is slightly more general.

a bound *in r if for all $\sigma \in sub(X)$ it holds that $f(\sigma)$ is a superset of the elements $\tau \in sub(Y)$ for which there is a relevant substitution in r that extends both σ and τ.*

To denote that f is a bound, we write $f\colon X \curvearrowright Y$. If $X = \emptyset$, then we say Y is bounded by f in r. If $f(\sigma)$ contains at most one element for each $\sigma \in sub(X)$, then f is called a functional bound.

3.2 How to use bounds

Bounds can be used to calculate overapproximations of completion formulas. To start, assume that a predicate p is defined only in a single rule r. Assume there is a bound $f\colon \mathrm{var}(\mathrm{H}(r)) \curvearrowright \mathrm{var}(r)$, and let $\sigma \in sub(\mathrm{var}(\mathrm{H}(r)))$. Then with σ, we can determine an overapproximation of the completion formula of $h = \sigma(\mathrm{H}(r))$ as follows:

$$\neg h \vee \bigvee_{\tau \in f(\sigma)} \beta(\tau(r)).$$

The case when p has multiple rules is similar and is formalized in the following proposition.

Proposition 3.4. *Let h be a ground atom. Let $r_1, \ldots, r_n$ be the rules in $\mathcal{P}$ whose head unifies with h. Let σ_i denote the most general unifier of h and $\mathrm{H}(r_i)$. If there is a bound $f_i\colon \mathrm{var}(\mathrm{H}(r_i)) \curvearrowright \mathrm{var}(r_i)$ for all i, then*

$$\neg h \vee \bigvee_{1 \leq i \leq n} \bigvee_{\tau \in f_i(\sigma_i)} \beta(\tau(r_i))$$

holds in all answer sets of $\mathcal{P}$.

Proof. For all answer sets I of $\mathcal{P}$ for which $I \models \neg h$, the clause trivially holds in I. So assume an answer set I for which $I \models h$. This means there is a rule r_i in $\mathcal{P}$ that derives h. Hence by Lemma 3.2, there is a relevant substitution ρ in r that extends σ_i. This means that $I \models \beta(\rho(r))$. By the definition of a bound, it holds that $\rho \in f(\sigma)$. Therefore I satisfies the clause, which we needed to show. □

Remark 3.5. *By Lemma 3.11 and Lemma 3.12, it is sufficient to have a bound $\mathrm{var}(\mathrm{H}(r)) \curvearrowright \mathrm{var}(\mathrm{B}(r))$ for each rule r.*

For both the multiple and the single rule case, the generated clause might be unwieldy, in particular if the bounds are bad overapproximations. Therefore, it is crucial that good bounds are detected, which is discussed in the next subsection.

Of course, a question that remains unanswered is when such bounds should be added to the solver. We see two ways to do this.

The first way is a very lightweight mechanism that happens during the **ground** reasoning step. The idea is that as soon as all rules that can derive a specific head h have been grounded, then we add the completion formula for h. Keeping track of this can be done very cheaply: the bounds provide us with an upper bound on the number of rules that can derive a given atom; it suffices to keep track of a simple counter for each atom to know when the criterion is satisfied. As soon as this is the case, all the atoms $\beta(\tau(r_i))$ mentioned in Proposition 3.4 are defined in the solver and it makes sense to add the completion constraint. This method is very lightweight: it does not trigger additional grounding, does not change the fundamental algorithm underlying ALPHA, and only adds very few additional constraints. It does enable better pruning of the search space.

The second way is more proactive, but also more invasive. It happens during the **justification-conflict** reasoning step. If an atom h is true, but not justified, instead of triggering the justification analysis to resolve why this situation happens, we add the completion formula for h, thereby also avoiding the justification-conflict. However, since certain atoms $\beta(\tau(r_i))$ from Proposition 3.4 are not yet known to the solver, also these corresponding rules need to be grounded. For this reason the second way is more intrusive into the grounding algorithm.

3.3 How to find bounds

In the previous subsection, we showed how bounds can be used to improve the lazy grounding algorithm. We now turn our attention to the question of how to find bounds. In particular, the various types of bounds we define in this section can all be found using a static analysis of the program. We illustrate our methods in increasing difficulty, illustrating each of them with examples of rules we encountered in practice, in encodings of the 5th ASP competition (Calimeri et al. 2016).

Case 1: Non-projective rules The first case is very simple: in case all variables occurring in a rule also occur in the head, then we know that for each atom, there is at most one variable substitution that turns the head of the rule into the specified atom. We call such a rule *non-projective* since no body variables are projected out.

Proposition 3.6. *If r is a non-projective rule, i.e., if $\mathrm{var}(\mathrm{H}(r)) = \mathrm{var}(r)$, then the following is a bound:*

$$id\colon sub(\mathrm{var}(\mathrm{H}(r))) \to sub(\mathrm{var}(r))\colon \sigma \mapsto \{\sigma\}.$$

Proof. Take $\sigma \in sub(\mathrm{var}(\mathrm{H}(r)))$. Let $\tau \in sub(\mathrm{var}(r))$ for which there is a relevant substitution ρ in r that extends both σ and τ. Then $\tau = \rho = \sigma$. Therefore $\tau \in id(\sigma)$, which proves that id is a bound. □

In case a predicate has a single non-projective rule, for each ground instance of the rule, the head is in fact equivalent to the body. This is a very specific and restricted case. We mention it here for two reasons. First of all, this is the only case for which ALPHA, without our extensions already adds completion constraints. Secondly, this (restricted) situation does show up in practical problems. For instance the following rule was taken from the new *Knight Tour with Holes* encoding used in the 5th ASP competition (Calimeri et al. 2016).

$$move(X, Y, XX, YY) \\ \leftarrow valid(X, Y, XX, YY), \neg other(X, Y, XX, YY).$$

Of course, if all the rules for a predicate are non-projective, then we can combine the trivial bounds on each rule to find a completion formula; however, this is is not yet detected in the existing ALPHA algorithm.

Case 2: Direct functional dependencies In certain cases, the body of a rule can contain variables the head does not, yet without increasing the number of instantiations that can derive the same head. This happens especially if some arithmetic operations are present. To illustrate this, consider the rule

$$\{gt(A,X,U)\} \leftarrow elem(A,X), comUnit(U), comUnit(U_1), U_1 = U+1, rule(A), U < X.$$

taken from the new Partner Units encoding used in the 5th ASP competition (Calimeri et al. 2016). This type of pattern occurs quite often, also for instance in Tower of Hanoi and in many temporal problems in which a time parameter is incremented by one or in problems over a grid in which coordinates are incremented by one. We can see that even though the variable U_1 occurs only in the body of the rule, for each instantation of the head there can be at most one grounding substitution of the rule that derives it. Hence, if all rules for gt have this structure, the completion can also be detected here. We now formalize this idea.

If p is a predicate with arity n, by p^j (with $1 \leq j \leq n$) we denote the j^{th} argument position of p. For any set J of argument positions, denote by $sub(J)$ the set of assignments of constants to the positions in J. A tuple of constants $c_1, \ldots, c_n$, is succinctly denoted by $\overline{c}$. If $p(\overline{c})$ is an atom and J a set of argument positions in p, we write $\overline{c}|_J$ to denote the element in $sub(J)$ that maps each $p^j \in J$ to c_j.

Definition 3.7. *A ground atom h is* relevant *in $\mathcal{P}$ if there is a rule r in $\mathcal{P}$ and a relevant grounding substitution σ in r such that $\sigma(\mathrm{H}(r)) = h$. A ground built-in atom is* relevant *in $\mathcal{P}$ if it is true.*

Definition 3.8. *Let J and K be sets of argument positions of a predicate p in $\mathcal{P}$. We say that $J \to K$ is a* functional dependency *if for all $\sigma \in sub(J)$, there exists at most one $\tau \in sub(K)$ and relevant atom $p(\overline{c})$ in $\mathcal{P}$ such that $\overline{c}|_J = \sigma$ and $\overline{c}|_K = \tau$.*

For instance, if p is equality, the following are some functional dependencies: $\{=^1\} \to \{=^2\}$, $\{=^2\} \to \{=^1\}$, $\{=^1, =^2\} \to \{=^1\}$. Of the ones mentioned here, the last one is the least interesting. Another example is the predicate $+/3$. It has among others the following functional dependencies: $\{+^1, +^2\} \to \{+^3\}$, $\{+^1, +^3\} \to \{+^2\}$, $\{+^3, +^2\} \to \{+^1\}$.

If a built-in predicate p with arity n occurs in the positive body of a rule r, then a functional dependency of p determines a bound in r.

Proposition 3.9. *Assume p is a built-in predicate and $p(\overline{t}) \in \mathrm{B}^+(r)$. A functional dependency $J \to K$ of p induces a functional bound (denoted $p(\overline{t})^{J \to K}$) in r:*

$$\mathrm{var}(\{t_i \mid p^i \in J\}) \curvearrowright \mathrm{var}(\{t_i \mid p^i \in K\}).$$

Proof. Let $X = \mathrm{var}(\{t_i \mid p^i \in J\})$ and $Y = \mathrm{var}(\{t_i \mid p^i \in K\})$. Let $\sigma \in sub(X)$. Since $J \to K$ is a functional dependency, there exists at most one $\tau_\sigma \in sub(Y)$ such that the atom $p(\overline{t})$ is satisfied under some extension of both σ and τ_σ. Define

$$f\colon sub(X) \to 2^{sub(Y)}$$

mapping a σ to $\{\tau_\sigma\}$ if τ_σ exists and $\emptyset$ otherwise. We prove that f is a bound; hence take any $\sigma \in sub(X)$. If there is no $\tau \in sub(Y)$ for which there is a relevant substitution in r that extends both σ and τ, then we are done. So suppose, there is such a τ. We prove that $\tau = \tau_\sigma$. Any relevant extension in r of both τ and σ justifies $p(\overline{X})$; hence satisfies $p(\overline{X})$. By definition of τ_σ we have that $\tau = \tau_\sigma$. Therefore, $\tau \in f(\sigma)$. This proves that f is a bound. That f is functional follows directly from its definition. □

As we will see later, bounds originating from functional dependencies of built-in predicates will act as a base case for further functional bounds.

Case 3: Determined predicates Given a program $\mathcal{P}$ we call a predicate *determined* if its defined only by facts. The interpretation of determined predicates can be computed efficiently prior to the solving process, and their value can be used to find bounds on the instantiations of other rules. An example can be found in graph coloring, in which a rule

$$colored(N) \leftarrow assign(N,C), color(C) \tag{1}$$

expresses that a node is colored if it is assigned a color. The predicate $color$ here is determined since it is given by facts. Thus, we know that for each node n, there are at most as many instances of the rule that derive $colored(n)$ as there are colors. Notably, the completion contraint that would be added by taking this into account, is exactly the redundant constraint that was added manually in the graph coloring experiments of Leutgeb and Weinzierl (2017) to help lazy grounding, i.e.

$$\neg colored(n) \vee assign(n, col_1) \vee \cdots \vee assign(n, col_k)$$

Our new methods obtain this constraint automatically, thereby easing the life of the modeler.

Proposition 3.10. *Let r be a rule with $d(\overline{t}) \in \mathrm{B}^+(r)$ and d a determined predicate. In that case there exists a bound $\emptyset \curvearrowright X$, where X is the set of variables in $\overline{t}$.*

Proof. Every fact $d(\overline{c})$ for a tuple of constants $\overline{c}$ corresponds to at most one element $\sigma_{\overline{c}}$ in $sub(X)$. Since d is given by facts, we can enumerate its interpretation I^d. Let

$$f\colon sub(\emptyset) \to 2^{sub(X)}\colon \sigma \mapsto \{\sigma_{\overline{c}} \mid \overline{c} \in I^d\}$$

We prove that f is a bound. Take $\sigma \in sub(\emptyset)$. Note that σ is necessarily the trivial substitution. Take $\tau \in sub(X)$ for which there is a relevant substitution in r that extends both σ and τ. We prove that $\tau \in f(\sigma)$, i.e. $\tau = \sigma_{\overline{c}}$ for some $\overline{c} \in I^d$. By the existence of that relevant substitution in r, we have that $d(\overline{t})$ is satisfied under τ; hence τ is equal to some $\sigma_{\overline{c}}$ for some $\overline{c} \in I^d$. This proves that f is a bound. □

Typical ASP encodings of graph coloring do not contain the rule (1) but instead use the rule

$$colored(N) \leftarrow assign(N,C).$$

Even in this case, it is possible to determine that C is bounded by a determined predicate by inspecting the defining rules of $assign$. This is formalized in the remainder of this section.

Case 4: Combining bounds Bounds can be obtained from other bounds in several ways. We already found three base cases of bounds, given in Propositions 3.6, 3.9, and 3.10:

1. If $Y \subseteq X \subseteq \mathrm{var}(r)$, then $id\colon X \curvearrowright Y$ is a bound, where id is the function mapping σ to $\{\sigma\}$.
2. The bound $p(\bar{t})^{J \to K}$ induced by a built-in atom $p(\bar{t}) \in \mathrm{B}^+(r)$ with functional dependency $J \to K$.
3. The bound induced by an atom $d(\bar{t}) \in \mathrm{B}^+(r)$ for a determined predicate d.

Additionally, bounds of different types can be altered or combined to get new bounds, as shown in the following lemmas.

Lemma 3.11. *Let $f\colon X \curvearrowright Y$ be a bound in r. Then for any $X \subseteq X'$ and $Y' \subseteq Y'$, the function*

$$f'\colon sub(X') \to 2^{sub(Y')}\colon \sigma \mapsto \{\tau|_{Y'} \mid \tau \in f(\sigma|_X)\}$$

is also a bound. ($\sigma|_X$ denotes σ restricted to the variables in X)

Proof. Take $\sigma \in sub(X')$. Let $\tau' \in sub(Y')$ for which there exists a relevant substitution ρ in r that extends both σ and τ'. We prove that $\tau' \in f(\sigma)$, i.e. there exist a $\tau \in f(\sigma|_X)$ such that $\tau' = \tau|_{Y'}$. Take $\tau = \rho|_Y$. By definition, $\tau|_{Y'} = \tau'$. We know that ρ extends both $\sigma|_X$ and τ. Therefore, since f is a bound, it holds that $\tau \in f(\sigma|_X)$. This proves that f' is a bound. □

Lemma 3.12. *Let $f\colon X \curvearrowright Y$ be a bound in r and let $U \subseteq \mathrm{var}(r)$. Let h denote the function*

$$h\colon sub(X \cup U) \to 2^{sub(Y \cup U)}$$

where

$$h(\sigma) = \{\tau \cdot \sigma|_{U \setminus Y} \mid \tau \in f(\sigma|_X)\}$$

and $\cdot$ is used to denote the combination of two disjoint projected substitutions. The function h is a bound from $X \cup U$ to $Y \cup U$.

Proof. Take $\sigma \in sub(X \cup U)$. Let $\tau \in sub(Y \cup U)$ for which there is a relevant substitution ρ in r that extends both σ and τ. We prove that $\tau \in h(\sigma)$. We know that ρ also extends both $\sigma|_X$ and $\tau|_Y$. Now, since f is a bound, $\tau|_Y \in f(\sigma|_X)$. Since ρ extends both σ and τ, it holds that $\sigma|_{U \setminus Y} = \tau|_{U \setminus Y}$ because $U \setminus Y$ is contained in the domains of both σ and τ. Therefore $\tau = \tau|_Y \cdot \tau|_{U \setminus Y} = \tau' \cdot \sigma|_{U \setminus Y}$ for some $\tau' \in f(\sigma|_X)$. This proves that $\tau \in h(\sigma)$; hence h is a bound. □

Lemma 3.13. *If $f\colon X \curvearrowright Y$ and $g\colon Y \curvearrowright Z$ are bounds in r, then the following function is a bound:*

$$h\colon sub(X) \to 2^{sub(Z)}\colon \sigma \mapsto \bigcup_{\tau \in f(\sigma)} g(\tau)$$

Proof. Take $\sigma \in sub(X)$. Let $\upsilon \in sub(Z)$ for which there is a relevant substitution ρ in r that extends both σ and υ. As usual we prove that $\upsilon \in h(\sigma)$. Take $\tau = \rho|_Y$. Then ρ is a relevant substitution that extends both τ and υ. Therefore, since g is a bound, $\upsilon \in g(\tau)$. Likewise, ρ is a relevant substitution that extends both σ and τ. Hence, $\tau \in f(\sigma)$ since f is a bound. Combining this proves that $\upsilon \in h(\sigma)$; hence proving that h is a bound. □

If only functional bounds are considered, then Lemma 3.12 and Lemma 3.13, together with our first base case forms the axiomatic system for functional dependencies developed by Armstrong (1974). To illustrate the combination of bounds, consider a rule

$$h(X) \leftarrow +(X, 1, Z), = (Z, U).$$

In this case, $X \curvearrowright U$ is a functional bound in r: by using the functional dependency of $+$ we see that $X \curvearrowright Z$ is a functional bound; by using the dependencies of $=$, we see that $Z \curvearrowright U$ is functional bound, hence we can combine them, by using Lemma 3.13, to get the desired dependency.

Even more is possible. If $f\colon X \curvearrowright Y$ and $g\colon X \curvearrowright Y$ are bounds, then the pointwise union and intersection are also bounds. While the union will not be of much benefit for finding good overapproximations of completion formulas, the intersection of two bounds can be useful since it allows for more precise approximations.

Case 5: Bounds on argument positions We have shown that if d is a determined predicate, then it induces a bound. However, sometimes bounds by determined predicates are not explicit. For instance, in the graph coloring example it would make perfect sense to drop $color(C)$ from the body of the rule since the fact that C is a color should follow already from its occurrence in $assign(N, C)$, resulting in the rule

$$colored(N) \leftarrow assign(N, C).$$

However, from the definition of $assign$, one can see that that C is bound by the determined predicate $color$ and hence the completion constraint could, in principle, still be derived. We now formally show how to do this.

Definition 3.14. *Let p be a predicate with arity n in a program $\mathcal{P}$ and J and K be sets of argument positions in p. If f is a function from $sub(J)$ to $2^{sub(K)}$ such that for every relevant atom $p(\bar{c})$ in $\mathcal{P}$ it holds that $\bar{c}|_K \in f(\bar{c}|_J)$, then f is said to be a* bound *in p, which we denote by $f\colon J \curvearrowright K$. If $J = \emptyset$, then we say K is bounded by f.*

Bounds in rules and predicates are not independent: bounds in rules determine bounds on argument positions and vice versa. This is formalized in the following two propositions.

Proposition 3.15. *Let p be a predicate symbol and J and K sets of argument positions in p. Assume that for each rule r of the form $p(\bar{t}) \leftarrow \varphi$ in $\mathcal{P}$, $f_r\colon \mathrm{var}(\bar{t}|_J) \curvearrowright \mathrm{var}(\bar{t}|_K)$ is a bound in r, then the union of these f_r induces a bound in p.*

Proof. Let A be any set of argument positions in p. Then A corresponds uniquely to a set $V_r \subseteq \mathrm{var}(\mathrm{H}(r))$ for each rule r of p, and V_r is the same for each rule r of p. Therefore, this set is denoted V. It is straightforward that $sub(A)$ is in a one-to-one relation with $sub(V)$. Misusing notation, we assume $sub(A) = sub(V)$. Then, we can define $f\colon sub(J) \to 2^{sub(K)}$ mapping σ to $\cup_r f_r(\sigma)$. We now

prove that f is a bound in p. Hence, take a relevant atom $p(\overline{c})$ in $\mathcal{P}$. It sufficies to prove that $\overline{c}|_K \in f(\overline{c}|_J)$. Since $p(\overline{c})$ is relevant, there is a rule r of p and a relevant grounding substitution ρ such that $\rho(\mathrm{H}(r)) = p(\overline{c})$. By the one-to-one correspondence between $sub(J)$ and $sub(V_J)$ and $sub(K)$ and $sub(V_K)$, we know that $\overline{c}|_J \in sub(V_J)$ and $\overline{c}|_K \in sub(V_K)$. Therefore, since f_r is a bound, we know that $\overline{c}|_K \in f_r(\overline{c}|_J)$. Hence, $\overline{c}|_K \in f(\overline{c}|_J)$, which proves that f is a bound in p. □

A simple example illustrating this proposition is as follows: Suppose we have the following rules for p:

$$p(X,Y) \leftarrow X = Y + 1.$$
$$p(X,Y) \leftarrow X = Y - 1.$$

Both rules have functional bounds from X to Y and vice versa. By taking the union of these two bounds, we get the bound $p^1 \curvearrowright p^2$ where X is mapped to $\{X-1, X+1\}$. This shows that functional bounds on rules do not necessarily give rise to functional bounds on argument positions.

If new bounds in predicates are detected, then these can be used to find new bounds in rules analogous to Proposition 3.9.

Proposition 3.16. *Let p be a predicate with a bound $f\colon J \curvearrowright K$ in p. If $p(\overline{t}) \in \mathrm{B}^+(r)$, then there is a bound*

$$\mathrm{var}\left(\overline{t}|_J\right) \curvearrowright \mathrm{var}\left(\overline{t}|_K\right)$$

in r. This bound is functional, if f is functional.

Proof. Let $X = \mathrm{var}\left(\overline{t}|_J\right)$ and $Y = \mathrm{var}\left(\overline{t}|_K\right)$. Any element $\tau' \in sub(K)$ corresponds to a unique element $\tau \in sub(Y)$. Similarly, any $\sigma \in sub(X)$ corresponds to a unique element $\sigma \in sub(J)$. Define

$$g\colon sub(X) \to 2^{sub(Y)}\colon \sigma \mapsto \{\tau \mid \tau' \in f(\sigma')\}$$

Take $\sigma \in sub(X)$. Let $\tau \in sub(Y)$ and let ρ be a relevant substitution in r that extends both σ and τ. We prove that $\tau \in f(\sigma)$. Since f is a bound in p, for each relevant atom $p(\overline{c})$ it holds that $\overline{c}|_K \in f(\overline{c}|_J)$. Since ρ is relevant, we know that $p(\overline{t})$ is justified; hence $p(\rho(\overline{t}))$ is a relevant atom. Therefore, $\rho(\overline{t})|_K \in f(\rho(\overline{t})|_J)$ because f is a bound. We can see that $\rho(\overline{t})|_J$ corresponds to σ and $\rho(\overline{t})|_K$ corresponds to τ, which completes the proof. □

The interaction between Proposition 3.15 and Proposition 3.16 is shown in the following example program:

$$u(1..3).\ w(3..5).$$
$$p(A,B) \leftarrow u(A), w(B).$$
$$q(B) \leftarrow p(C,B).$$
$$r(X,Y) \leftarrow q(Y), X = Y.$$
$$r(X,Y) \leftarrow p(X,Y).$$
$$o(a) \leftarrow r(X,a).$$

We know that both u and w are determined predicates. Therefore, in the rule of p, A is bounded by u and B bounded by w. This indicates that p^1 is bounded by u and p^2 is bounded by w. Similarly, q^1 is bounded by w. In the first rule of r, Y is bounded by w, and by transitivity X is bounded by w as well. In the second rule of r, X is bounded by u and Y bounded by w. Therefore, r^1 is bounded by the union of u and w, while r^2 is bounded by w. Finally, we obtain the following completion formula for o:

$$\neg o(a) \vee r(1,a) \vee r(2,a) \vee r(3,a) \vee r(4,a) \vee r(5,a)$$

In theory, to find bounds we repeat the two steps below until a fixpoint is reached:

1. find all bounds on variables in rules (using a fixpoint procedure, using the base cases and lemmas in Case 4 and Proposition 3.16)
2. find all bounds on argument positions of predicates (using a fixpoint procedure, using Proposition 3.15) (we can restrict ourselves to the predicates occurring in positive bodies, since that are the only predicates useful for generating completion formulas)

4 Future work

To tackle this problem in its most general form, one could develop methods similar to *grounding with bounds* (Wittocx, Mariën, and Denecker 2010) that were developed in the context of model expansion (Mitchell and Ternovska 2005) for an extensions of first-order logic (Denecker and Ternovska 2008) that closely relates to answer set programming (Denecker et al. 2019).

While the cases studied in the previous section allow for adding completion constraints in a wide variety of applications, we see the current work as a stepping stone towards a more extensive theory of approximations that enable adding completion constraints. In this section, we provide several directions in which the current work can be extended.

Dynamic overapproximations The approximations developed and described in the previous section can all be determined statically. However, during solving sometimes more consequences at decision level zero are derived. Taking these also into account (instead of just the determined predicates) can result in better approximations and hence more completion constraints.

More bounds in predicates For finding new opportunities to add completion formulas, it is necessary that (especially functional) bounds between argument positions are detected, eventhough they are not directly used in generating the completion formulas. This detection can be done by syntactic means, such as inspecting their defining rules, or by semantic means (De Cat and Bruynooghe 2013). We already supplied Proposition 3.15, however this is not sufficient to find all useful bounds.

For example, in each rule below we have functional bounds $\{2,3\} \curvearrowright \{4,5\}$ and $\{4,5\} \curvearrowright \{2,3\}$, but the complete predicate has the following fundamental functional bounds $\{1,2,3\} \curvearrowright \{4,5\}$ and $\{1,4,5\} \curvearrowright \{2,3\}$. This is because if you know the first argument position, then you know the rule that is used. If for example you have $neighbor(n,X,Y,XX,YY)$ in the positive body of a rule,

then you know the first rule is applicable: $X = XX$ and $Y = YY - 1$.

$$neighbor(D, X, Y, X, YY) \leftarrow D = n, Y = YY - 1.$$
$$neighbor(D, X, Y, X, YY) \leftarrow D = s, Y = YY + 1.$$
$$neighbor(D, X, Y, XX, Y) \leftarrow D = w, X = XX - 1.$$
$$neighbor(D, X, Y, XX, Y) \leftarrow D = e, X = XX + 1.$$

These dependencies are not detected by the double fixpoint procedure. Intuitively, what is going on here is that the first argument of $neighbor$ is inherently linked to which rule is applicable. Depending on that first argument, we can decide *which* functional dependency can be generalized to the predicate level (but it is not always the same).

5 Conclusion

In this paper, we highlighted the issue of missing completion formulas in lazy grounding and provided lightweight solutions for this issue based on static program analysis. In our theoretical analysis, we found that the completion formulas that can now be added are in some cases identical to redundant constraints added to improve search performance; hence, usage of our techniques eliminates this burden for the programmer.

Our next step in this research will be implementing the presented ideas and experimenting to find out what their impact is on the runtime of lazy grounders.

In Section 4, we identified several directions in which this work can continue that would allow for the detection of even more completion constraints. We intend to evaluate these as well in follow-up research.

References

Anger, C.; Gebser, M.; Janhunen, T.; and Schaub, T. 2006. What's a head without a body? In Brewka, G.; Coradeschi, S.; Perini, A.; and Traverso, P., eds., *ECAI*, 769–770. IOS Press.

Armstrong, W. W. 1974. Dependency structures of data base relationships. *IFIP Congress* 580–583.

Balduccini, M.; Lierler, Y.; and Schüller, P. 2013. Prolog and ASP inference under one roof. In Cabalar, P., and Son, T. C., eds., *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, volume 8148 of *LNCS*, 148–160. Springer.

Banbara, M.; Kaufmann, B.; Ostrowski, M.; and Schaub, T. 2017. Clingcon: The next generation. *TPLP* 17(4):408–461.

Barrett, C. W.; Sebastiani, R.; Seshia, S. A.; and Tinelli, C. 2009. Satisfiability modulo theories. In Biere et al. (2009). 825–885.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bogaerts, B., and Weinzierl, A. 2018. Exploiting justifications for lazy grounding of answer set programs. In Lang, J., ed., *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, 1737–1745. ijcai.org.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Ricca, F.; and Schaub, T. 2013. ASP-Core-2 input language format. Technical report, ASP Standardization Working Group.

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

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 input language format. *TPLP* 20(2):294–309.

Clark, K. L. 1978. Negation as failure. In *Logic and Data Bases*, 293–322. Plenum Press.

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

Dao-Tran, M.; Eiter, T.; Fink, M.; Weidinger, G.; and Weinzierl, A. 2012. Omiga: An open minded grounding on-the-fly answer set solver. In del Cerro, L. F.; Herzig, A.; and Mengin, J., eds., *JELIA*, volume 7519 of *LNCS*, 480–483. Springer.

De Cat, B., and Bruynooghe, M. 2013. Detection and exploitation of functional dependencies for model generation. *TPLP* 13(4–5):471–485.

Denecker, M., and Ternovska, E. 2008. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.* 9(2):14:1–14:52.

Denecker, M.; Lierler, Y.; Truszczynski, M.; and Vennekens, J. 2019. The informal semantics of answer set programming: A Tarskian perspective. *CoRR* abs/1901.09125.

Denecker, M.; Brewka, G.; and Strass, H. 2015. A formal theory of justifications. In Calimeri, F.; Ianni, G.; and Truszczyński, M., eds., *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *Lecture Notes in Computer Science*, 250–264. Springer.

Gebser, M.; Maratea, M.; and Ricca, F. 2017. The sixth answer set programming competition. *J. Artif. Intell. Res.* 60:41–95.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R. A., and Bowen, K. A., eds., *ICLP/SLP*, 1070–1080. MIT Press.

Lefèvre, C., and Nicolas, P. 2009. The first version of a new ASP solver: ASPeRiX. In Erdem, E.; Lin, F.; and Schaub, T., eds., *LPNMR*, volume 5753 of *LNCS*, 522–527. Springer.

Leutgeb, L., and Weinzierl, A. 2017. Techniques for efficient lazy-grounding ASP solving. In Seipel, D.; Hanus, M.; and Abreu, S., eds., *Declare 2017 – Conference on Declarative Programming, proceedings*, number 499 in Institut für Informatik technical report, 123–138.

Liu, L.; Pontelli, E.; Son, T. C.; and Truszczynski, M. 2007. Logic programs with abstract constraint atoms: The role of computations. In Dahl, V., and Niemelä, I., eds., *Logic Programming, 23rd International Conference, ICLP 2007,*

Porto, Portugal, September 8-13, 2007, Proceedings, volume 4670 of *Lecture Notes in Computer Science*, 286–301. Springer.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In Apt, K. R.; Marek, V.; Truszczyński, M.; and Warren, D. S., eds., *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag. 375–398.

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

Marques Silva, J. P.; Lynce, I.; and Malik, S. 2009. Conflict-driven clause learning SAT solvers. In Biere et al. (2009). 131–153.

Martelli, A., and Montanari, U. 1982. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.* 4(2):258–282.

Mitchell, D. G., and Ternovska, E. 2005. A framework for representing and solving NP search problems. In Veloso, M. M., and Kambhampati, S., eds., *AAAI*, 430–435. AAAI Press / The MIT Press.

Rossi, F.; van Beek, P.; and Walsh, T., eds. 2006. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier.

Stuckey, P. J. 2010. Lazy clause generation: Combining the power of SAT and CP (and mip?) solving. In Lodi, A.; Milano, M.; and Toth, P., eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, volume 6140 of *Lecture Notes in Computer Science*, 5–9. Springer.

Taupe, R.; Weinzierl, A.; and Friedrich, G. 2019. Degrees of laziness in grounding - effects of lazy-grounding strategies on ASP solving. In Balduccini, M.; Lierler, Y.; and Woltran, S., eds., *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, volume 11481 of *Lecture Notes in Computer Science*, 298–311. Springer.

Taupe, R.; Weinzierl, A.; and Schenner, G. 2017. Introducing Heuristics for Lazy-Grounding ASP Solving. In *1st International Workshop on Practical Aspects of Answer Set Programming*.

Weinzierl, A. 2017. Blending lazy-grounding and CDNL search for answer-set solving. In Balduccini, M., and Janhunen, T., eds., *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, volume 10377 of *Lecture Notes in Computer Science*, 191–204. Springer.

Wittocx, J.; Mariën, M.; and Denecker, M. 2010. Grounding FO and FO(ID) with bounds. *J. Artif. Intell. Res. (JAIR)* 38:223–269.

Zhang, L.; Madigan, C. F.; Moskewicz, M. W.; and Malik, S. 2001. Efficient conflict driven learning in Boolean satisfiability solver. In *ICCAD*, 279–285.