Enhancing Lazy Grounding with Lazy Normalization in Answer-Set Programming

Jori Bomanson,[1] **Tomi Janhunen,**[1] **Antonius Weinzierl**[1,2]
[1]Department of Computer Science, Aalto University, Finland
[2]Institute of Logic and Computation, TU Wien, Austria
jori.bomanson@aalto.fi, tomi.janhunen@aalto.fi, antonius.weinzierl@kr.tuwien.ac.at

Abstract

Answer-Set Programming (ASP) is an expressive rule-based knowledge-representation formalism. Lazy grounding is a solving technique that avoids the well-known grounding bottleneck of traditional ASP evaluation but is restricted to normal rules, severely limiting its expressive power. In this work, we introduce a framework to handle aggregates by normalizing them on demand during lazy grounding, hence relieving the restrictions of lazy grounding significantly. We term our approach as lazy normalization and demonstrate its feasibility for different types of aggregates. Asymptotic behavior is analyzed and correctness of the presented lazy normalizations is shown. Benchmark results indicate that lazy normalization can bring up-to exponential gains in space and time as well as enable ASP to be used in new application areas.

1 Introduction

Answer-Set Programming (ASP) is an expressive rule-based knowledge-representation formalism whose success is much due to efficient solver technology available for evaluation (Gebser, Kaufmann, and Schaub 2012; Alviano et al. 2013; Leone et al. 2002). State-of-the-art ASP systems follow the ground-and-solve approach, where a first-order input program is turned into a corresponding *ground* (variable-free) program for which answer sets are then computed. But, in the worst case, the resulting ground program may become exponentially larger than the original non-ground input program. Even a polynomial-size increase may already be prohibitive in practice. This drawback impairs the scalability of ASP for practical applications (cf. (Falkner et al. 2016)) and is known as the grounding bottleneck of ASP. To circumvent such blow-ups, *lazy-grounding* ASP solvers have been developed; cf. (Palù et al. 2009; Lefèvre et al. 2017; Dao-Tran et al. 2012; Weinzierl 2017). The main idea is to interleave grounding with solving and to generate only ground rules necessary in each position of the search space.

However, existing lazy-grounding ASP systems only accept *normal rules* as input and they do not support a broad range of syntactic primitives as defined by, e.g., the ASP *core language* (Calimeri et al. 2012). Most notably missing are *aggregates*, which occur in many ASP programs, because they are highly expressive and enable a programmer to state complex conditions in a very concise manner. The importance of aggregates is witnessed by a rich body of research, see e.g., (Greco 1999; Simons, Niemelä, and Soininen 2002; Liu and Truszczynski 2006; Ferraris 2011; Faber, Pfeifer, and Leone 2011; Gelfond and Zhang 2014; Alviano, Faber, and Gebser 2015). The integration of aggregates into lazy-grounding ASP systems, however, has not been attempted so far, although there is work on first-order rewriting of aggregates for ground-and-solve ASP systems (Polleres et al. 2013). Realizing aggregates is feasible as native extensions of solvers in the form of propagators or by unfolding (monotone) ground aggregates as normal rules. The latter is known as *normalization* (Janhunen and Niemelä 2011; Bomanson and Janhunen 2013). Both approaches are logical avenues of research in the lazy grounding setting. However, normalization is particularly appealing because it readily benefits from existing lazy solving techniques, such that conflict-driven learning and lazy instantiation naturally carry over to normalized aggregates. Furthermore, it easily allows to revise encodings of aggregates in a systematic fashion. For these reasons, we focus on normalization as the implementation strategy in this work. Moreover, we concentrate on *monotone* aggregates, and in particular count and sum aggregates with only lower bounds and non-negative weights. This provides a natural basis for implementing aggregates in practice, and paves a way for future support of *non-monotone* aggregates via rewritings into monotone ones (Alviano, Faber, and Gebser 2015).

Lazy grounding poses some unexpected challenges to normalization, because matters like enumerating all ground instances that are trivial in the ground-and-solve approach suddenly become challenging: for every variable X it is unclear what the ground instances are, how many of them will appear, and in what order. E.g., counting inherently requires that the counted ground atoms are totally ordered; so for an aggregate counting the cardinality of $p(X)$ it is not known whether the ground instances $p(a)$ and $p(c)$ will be grounded lazily at some point, nor is it then known if $p(c)$ is the second ground instance since some later grounded $p(b)$ might come between $p(a)$ and $p(c)$ in a natural order. Thus it is not clear whether normalization can be efficiently applied in a lazy-grounding setting. Another challenge is that the normalization of an aggregate must not require any portion of

the program to be fully grounded, since due to predicate dependency this can easily cascade and require large portions of the program to be fully grounded and hence degenerate into requiring a full grounding of the input program.

Our investigation revealed that ground instance enumeration is the key principle to enable non-ground normalization in a lazy-grounding setting and, based on this, the contributions of this paper are: **(1)** the introduction of an elegant framework for *lazy normalization* where the result of normalization can be instantiated lazily; **(2)** effective lazy normalizations for *counting* and *summation* aggregates with lower bounds; **(3)** a far optimized lazy normalization for count aggregates based on lazily generated sorting circuits, inheriting attractive properties from ground-and-solve normalizations; **(4)** proof-of-concept implementations of lazy normalizations within the Alpha solver; **(5)** several benchmarks showing, i.a., that lazy normalization opens up new application areas like simulation with many time steps, and enables up-to exponential savings in space and time; and **(6)** an unprecedented lazy-grounding ASP evaluation beyond normal rules and a significant step towards the full expressive power of ASP in lazy-grounding ASP.

Following preliminaries in Section 2, Section 3 introduces the lazy normalization framework. Actual lazy normalizations are presented in Section 4 whereas their evaluation takes place in Section 5. Related work and generalizations are addressed in Section 6, and Section 7 concludes.

2 Preliminaries

We refer to (Eiter, Ianni, and Krennwallner 2009) for a comprehensive introduction to ASP, and present here briefly a fragment of ASP containing aggregates. A rule is of the form

$$a_1 \vee \ldots \vee a_m \leftarrow b_1, \ldots, b_n$$

where $m, n \geq 0$ and $a_1, \ldots, a_m$ are atoms over a first-order language $\mathcal{L}$, and $b_1, \ldots, b_n$ are literals over $\mathcal{L}$, where a literal is either an atom a, a negated atom $not\, a$, or an aggregate. Aggregates are of the form

$$l \prec \#func\{T_1 : l_1; \ldots; T_n : l_n\} \prec u \qquad (1)$$

where l (resp. u) are integer lower (resp. upper) bounds; $\prec$ the comparison relation $\leq$ or $<$; $\#func$ the aggregate function $\#count$, $\#sum$ or $\#max$; $T_1, \ldots, T_n$ the (lists of) terms involving non-negative weights over which the aggregate is evaluated; and $l_1, \ldots, l_n$ the (conjunctions of) positive and negative atoms that specify the condition(s) under which the corresponding terms should be considered by the aggregate function. The upper bound of an aggregate may be omitted, leaving a lower-bounded aggregated.

The *head* of a rule r is the set $H(r) = \{a_1, \ldots, a_m\}$, the *body* is $B(r) = \{b_1, \ldots, b_n\}$, and the positive body is $B^+(r) = B(r) \cap \mathcal{L}$. A rule with $m = 0$ is a *constraint*, a rule with $n = 0$ is a *fact*. A rule r is a *ground instance* of another rule if it is the result of substituting all first-order variables in that other rule with ground terms, r is *ground* if it is a ground instance of itself, and r is called *normal* if $m \leq 1$, i.e., its head contains at most one atom. The ASP core language (Calimeri et al. 2012) standardizes further language constructs such as comparison relations and choice rules.

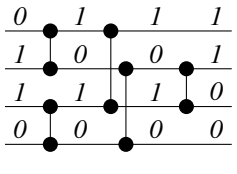
A program P is a set of rules. Its Herbrand base $HB(P)$ is the set of all ground atoms based on the predicates, constants and function symbols in P. An *interpretation* $I \subseteq HB(P)$ is an assignment of truth values to $HB(P)$. Given an interpretation I, the *satisfaction relation* $\models$ for ground objects is such that $I \models a$ if the atom a is assigned true, i.e., if $a \in I$; $I \models not\, a$ if $I \not\models a$; $I \models ag$ for an aggregate ag of the form (1) if the application of $\#func$ to the set of first elements of tuples T_i for which each literal in l_i is satisfied by I returns a value within the lower and upper bounds l and u in the sense of $\prec$ (cf. (Calimeri et al. 2012) for details); $I \models r$ if r is a ground rule and $I \models b_i$ for all $b_i \in B(r)$ implies that $I \models a_j$ for some $a_j \in H(r)$; $I \models P$ holds if $I \models r$ for all $r \in P$. For first-order programs P, $I \models P$ if $I \models r'$ for all ground instances r' of all rules $r \in P$. The *reduct* of a program P w.r.t. I is the set of ground instances of rules r in P whose body is satisfied in I. An interpretation I is a *model* of P if $I \models P$ and an *answer set* of P if it is a $\subseteq$-minimal model of the reduct of P w.r.t. I.

The semantics of aggregates as defined in (Calimeri et al. 2012) follows the intuition, for example an aggregate $ag{=}7 \leq \#sum\{3{:}p(a);\ 4{:}p(b);\ 5{:}p(c)\}$ is true whenever the sum of 3, 4, and 5 is at least 7, given that $p(a)$, $p(b)$, and $p(c)$ holds, respectively. This holds whenever two out of the three literals $p(a)$, $p(b)$, and $p(c)$ are true. Given an interpretation I with $I \models p(a)$, $I \not\models p(b)$, and $I \models p(c)$, then the sum amounts to $3 + 5 = 8 \geq 7$, so $I \models ag$.

In general, lazy grounding is an implementation technique that avoids upfront grounding and instead produces individual ground instances of rules only as they become relevant, implying that more ground rules appear gradually during solving. Moreover, a typical lazy grounder recursively guesses and deduces tentative truth values for atoms, which are detracted or revised in a backtracking process upon conflict. A ground instance r' of a normal rule r becomes relevant when these tentative assignments amount to a partial interpretation I that satisfies the positive body $B^+(r')$ of r' and subsequently the ground head $H(r')$ is obtained.

Count and sum aggregates in ground programs can be normalized into sets of normal rules (Bomanson 2017). In particular, count aggregates can be replaced with encodings of *odd-even merge sorting networks* (Batcher 1968). A sorting network is a data-oblivious sorting algorithm for a fixed number of inputs that conditionally swaps input element pairs at predetermined positions (Batcher 1968).

Example 1. *Below is a sorting network with four wires and a depth of three operating on Boolean inputs from left to right. Values on vertically connected wires are sorted using* comparators*, whose wires $I < J$ and depths D we give in facts $comp(I, J, D)$ on the right.*




```
comp(1,2,1).
comp(3,4,1).
comp(1,3,2).
comp(2,4,2).
comp(2,3,3).
```

For instance, the top left comparator $comp(1, 2, 1)$ sorts the pair 01 *to* 10*. Observe that the sorted output on the very*

right expresses the cardinality of the inputs in such a way that the k^{th} output tells whether k or more inputs are true.

These networks and their evaluation on Boolean inputs can be conveniently encoded in ASP due to their data-oblivious nature. With such encodings, a ground aggregate with n input terms and a bound of k can be turned into $\mathcal{O}(n(\log k)^2)$ normal rules involving auxiliary atoms in the same order of magnitude (Bomanson 2017).

3 Framework for Lazy-Grounding First-Order Normalization

The expressive power of aggregates is reflected by the high numbers of normal rules required to state them otherwise. In the process of normalization, an aggregate is replaced by a sub-program solely consisting of normal rules. In the traditional ground-and-solve approach, this takes place after grounding which means that all atoms occuring in a particular aggregate are already ground and known a priori. This facilitates the normalization step substantially, since the dimensions and parameters of the aggregate have been fixed.

Example 2. *Consider an aggregate $ag = 2 \leq \#count\{X : p(X)\}$ and the set of constants $\mathcal{C} = \{a, b, c\}$ acting as the universe for instantiation. The resulting grounding of ag is $2 \leq \#count\{a : p(a);\ b : p(b);\ c : p(c)\}$. One potential normalization of this aggregate consists of*

$$holds(ag) \leftarrow p(a), p(b). \quad holds(ag) \leftarrow p(a), p(c).$$
$$holds(ag) \leftarrow p(b), p(c).$$

These rules simply encode subsets of conditions that suffice to reach the given lower bound 2 of the aggregate.

It is not immediately clear how to normalize aggregates when rules are instantiated lazily. If all ground instances of some predicate $p(X)$ are determined, it may require the recursive grounding of rules used to derive atoms over p, potentially degenerating into a full grounding of the entire program. Therefore, lazy-grounding normalization *should not* presume the knowledge of all ground instances of predicates involved. This is why normalizations deployed in the traditional ground-and-solve approach are not directly feasible.

In Example 2, the normalization effectively enumerates all ground instances of $p(X)$. In the lazy-grounding setting, they are not readily available and forming such an enumeration becomes a challenge of its own. In principle, there is a natural order for ground terms (cf. (Calimeri et al. 2012)), but the exploitation of this order in lazy normalization is jeopardized by two factors. First, particular ground terms might never occur as arguments of predicates due to the rules of the program. Second, the underlying lazy-grounding solver could produce ground instances of predicates against such a predetermined order of terms. In the following, we present a surprisingly simple yet practical solution to the enumeration problem in question: *built-in* atoms that keep track of the order in which ground terms are lazily formed.

Definition 1. *An* enumeration built-in *is an atom*

$$enum(ag, t, i)$$

where the term ag identifies an aggregate with a grounding order, t is a ground term, and i is the index of t in the order.

In practice, when a given ground term t is formed by the solver for the first time, the index $i \geq 1$ for $enum(ag, t, i)$ can be computed as $i' + 1$ for the largest index i' in the ground atoms $enum(ag, t', i')$ generated so far. As a consequence, the index of t is unique for each identifier ag and the indices introduced for ag are contiguous. On the other hand, the term t may be assigned different indices for different aggregates, i.e., $enum(ag_1, t, i)$ and $enum(ag_2, t, j)$ with $ag_1 \neq ag_2$ and $i \neq j$ may be formed. As regards implementation, the enumeration built-in can be realized by using a counter and a map data structure for each aggregate.

Since first-order variables are naturally available in the lazy-grounding setting, it is possible to encode particular normalizations in terms of non-ground rules, to be subsequently instantiated on demand only. This paves the way for fixed-size first-order normalizations that do not depend on the actual instantiations of aggregates. To preserve the answer sets of the original, non-normal ASP program $\mathcal{P}$, the sub-program realizing the aggregate evaluation must be kept transparent with respect to reported answer sets. This holds in particular for the enumeration built-in, whose extension depends on the order of grounding. To formalize this, all predicates occurring in $\mathcal{P}$ are considered *visible* while any auxiliary predicates introduced in the normalization of $\mathcal{P}$ are kept *invisible* as detailed by the following definition.

Definition 2. *Given a finite set of rules R, a* lazy normalization program *of R is a pair (S, G) of sets of normal rules such that (i) all rules in S are over invisible predicates and (ii) all rules in G are over both visible and invisible predicates, including the built-in enumeration predicate.*

Intuitively, a lazy normalization program (S, G) consists of a sub-program S for the actual normalization of aggregates and a *gluing* program G that connects S into its context. Typically, such a normalization takes place in a context of some program P that is already normal. However, Definition 2 is also applicable to individual rules r by setting $R = \{r\}$ and allowing non-normal contexts. The roles played by the different programs are demonstrated below.

Example 3. *Consider the rule $r =$*

$$ok \leftarrow 10 \leq \#max\{V : pick(I), cand(I, V)\}.$$

with an aggregate in the context of the normal program

$$P = \left\{ \begin{array}{l} cand(a, 7).\ cand(b, 12).\ cand(c, 13). \\ pick(I) \leftarrow not\ drop(I),\ cand(A, V). \\ drop(I) \leftarrow not\ pick(I),\ cand(A, V). \end{array} \right\}$$

Intuitively, the program $P \cup \{r\}$ determines if the highest value of picked items is at least 10 units. One potential lazy normalization program (S, G) of r is given by

$$S = \{out(ag) \leftarrow B \leq V,\ in(ag, I, V),\ bound(ag, B).\}$$

$$G = \left\{ \begin{array}{ll} in(ag, I, V) \leftarrow pick(I),\ cand(I, V). & \\ bound(ag, 10). & ok \leftarrow out(ag). \end{array} \right\}$$

Here ag identifies the aggregate in r and S checks the lower bound of the aggregate, which boils down to the simple question of whether there exists some aggregate input whose value is higher than the bound. The gluing program G feeds the picked items as input to S, sets the bound to 10, and derives the original head atom if the aggregate is satisfied.

Note that the built-in predicate $enum$ is not needed here but will be vital for the normalizations presented later. As regards the semantics of lazy normalization programs, answer sets are obtained as projections over visible atoms:

Definition 3. *Given a lazy normalization program (S, G) of a rule set R with aggregates, a set A of ground atoms is an* answer set *of (S, G) in the context of a program P, if there is an ordinary answer set $A^+ \supseteq A$ of $P \cup S \cup G$ such that $A = \{a \in A^+ \mid$ the atom a is based on a predicate in $P \cup R\}$.*

Example 4 (Ex. 3 continued)**.** *One answer set of r in the context of P is $A = \{cand(a, 7), cand(b, 12), cand(c, 13), pick(a), pick(b), drop(c), ok\}$. The corresponding answer set A^+ of $P \cup S \cup G$ extends A by $bound(ag, 10)$, $in(ag, a, 7)$, $in(ag, b, 12)$, and $out(ag)$. Likewise, one can show that the answer sets of $P \cup \{r\}$ and $P \cup S \cup G$ coincide in the sense of Definition 3.*

Next, we address the correctness of lazy normalization in a simplified setting where a first-order program has only one rule r involving an aggregate. Let (S_r, G_r) be the corresponding normalization program in the context P forming the rest of the program, i.e., G_r glues the first-order normalization S_r with P. Since P is an arbitrary context for $S_r \cup G_r$, the correctness of $S_r \cup G_r$ as a normalization of r must aim at the *strong equivalence* of r and $S_r \cup G_r$ to cater for arbitrary interactions between r and P as well as recursion through the aggregate in r. However, since auxiliary predicates are allowed in $S_r \cup G_r$, we must resort to a notion of strong equivalence (Woltran 2004) that excludes any interactions through such auxiliary predicates.

Definition 4. *Let r be a first-order rule with an aggregate. A first-order normalization (S_r, G_r) of r is called* faithful*, if r and $S_r \cup G_r$ induce the same answer sets in every context P not referring to the auxiliary predicates of (S_r, G_r).*

Let us illustrate the faithfulness of normalization in practice.

Example 5. *Consider a rule $r = ok \leftarrow 1 \leq \#count\{1, X : p(X)\}$ with a count aggregate ag and its first-order normalization determined by S_r and G_r below:*

$$S_r = \{\ out(ag) \leftarrow in(ag, X).\ \}$$
$$G_r = \{\ in(ag, X) \leftarrow p(X).\quad ok \leftarrow out(ag).\ \}$$

Intuitively, the atom ok is included in an answer set A iff $p(t) \in A$ for some ground term t. Such an inclusion is equally feasible given either r or (S_r, G_r). Also, an empty answer set is obtained in the recursive context of $p(a) \leftarrow ok$.

As regards negative literals $not\, a$ occurring in an aggregate, it is possible to remove such occurrences by substituting a new atom na defined by a normal rule $na \leftarrow not\, a$. Thus, since normalizations to be presented in Section 4 are based on positive programs, we may assume without loss of generality that aggregates and normalizations addressed in correctness proofs are negation-free. The proposition below formalizes our proof strategy based on Herbrand models.

Proposition 1. *A* negation-free *first-order normalization (S_r, G_r) of a* negation-free *aggregate rule r is faithful, if the classical Herbrand models of r coincide with those of $S_r \cup R_r$, neglecting the $\subseteq$-minimal interpretations of any auxiliary predicates used in the normalization (S_r, G_r).*

4 Lazy-Normalization of Aggregates

In what follows, we apply our normalization framework and obtain the first normalizations of count aggregates in lazy-grounding answer-set solving. We present two novel evaluation programs S with a gluing program G to form two alternative lazy normalization programs (S, G).

Definition 5. *Given a rule r of the form*

$$h \leftarrow K \leq \#count\{T_1 : l_1; \ldots; T_n : l_n\}, dom(K), B.$$

where $T_1, \ldots, T_n$ are (lists of) terms and $l_1, \ldots, l_n$ are (lists of) literals, B is a list of ordinary literals, and the aggregate is identified by ag, the gluing program *G_r for r is:*

$$\{\ in(ag, element\_tuple(T_i)) \leftarrow l_i, B. \qquad (\forall i : 1 \leq i \leq n)$$
$$idx(R, I) \leftarrow in(R, X), enum(R, X, I).$$
$$bound(ag, K) \leftarrow dom(K).$$
$$h \leftarrow out(ag, K), dom(K), B.\ \}$$

The terms T_i of satisfied input literals l_i are captured by the in predicate and then mapped to indices obtained via $enum$ for storage in idx. Based on these indices and lower bounds marked with $bound$, the evaluation program S is expected to define the output atom $out(R, K)$, which substitutes the original aggregate in the final gluing rule for h.

Counting Grid. Our first evaluation program is essentially a first-order version of the *counting grid* encoding of ground cardinality rules (Janhunen and Niemelä 2011) or equivalently the *sequential counter* encoding of cardinality constraints (Hölldobler, Manthey, and Steinke 2012).

Definition 6. *The* counting grid *evaluation program S is:*

$$\{\ span(R, 1..I{-}1) \leftarrow idx(R, I).$$
$$sum(R, 0, 0) \leftarrow idx(R, \_).$$
$$sum(R, I, S) \leftarrow sum(R, I{-}1, S), span(R, I).$$
$$sum(R, I, S{+}1) \leftarrow sum(R, I{-}1, S), idx(R, I),$$
$$\qquad bound(R, K), S < K.$$
$$out(R, K) \leftarrow bound(R, K), K \leq S, sum(R, \_, S).\ \}$$

This encoding defines $sum(R, I, S)$ to be equivalent to a count aggregate with a lower bound S and inputs $idx(R, 1), \ldots, idx(R, I)$. The definition proceeds by induction on I: for each atom $idx(R, I)$, any bounds already satisfied at $I - 1$ are carried over and also incremented by one if the atom is satisfied. An output atom $out(R, K)$ is derived if a bound K of the original aggregate is reached at any index I. The use of any index I instead of the last index only enables lazy-grounding to work properly without having to unnecessarily ground all input atoms of the aggregate.

Proposition 2. *The ground size of a counting grid program is $\mathcal{O}(kn)$ for n inputs grounded so far and the bound k.*

Sorting Network Application. Next we present an alternative encoding that yields a more compact grounding. It consists of two parts, one defining a sorting network (SN), and another applying (any) SN to evaluate count aggregates. The idea of an SN is that its k^{th} sorted output value is true exactly if at least k of the inputs are true. We first start with the part that applies a given SN.

Definition 7. *The* sorting network application (SNA) program A *is*

$$
\begin{aligned}
\{\ & span(R, 1..I) \leftarrow idx(R, I).\\
& v(R, I, D) \leftarrow idx(R, I), D = 0.\\
& v(R, I, D) \leftarrow v(R, I, D{-}1), comp(I, \_, D), dh(R, D).\\
& v(R, I, D) \leftarrow v(R, J, D{-}1), comp(I, J, D), dh(R, D).\\
& v(R, J, D) \leftarrow v(R, I, D{-}1), comp(I, J, D), dh(R, D),\\
& \qquad v(R, J, D{-}1).\\
& v(R, I, D) \leftarrow v(R, I, D{-}1), pass(I, D), dh(R, D).\\
& out(R, K) \leftarrow bound(R, K), v(R, K, D), done(N, D),\\
& \qquad K{\leq}N.\\
& out(R, K) \leftarrow bound(R, K), K \leq 0.\ \}
\end{aligned}
$$

For a given aggregate, the $span$ predicate marks a range of indices that covers all its inputs, and v encodes *wire values* computed from those inputs based on an SN. In figures of networks such as the one in Example 1, wire values are the 0s and 1s overlaying the network. Here, satisfied atoms $v(R, I, D)$ mark indices I and depths D that correspond to the vertical and horizontal coordinates of the 1s, respectively. The comparators $comp(I, J, D)$ are shown as vertical wires in the figures and are implemented by taking as inputs the values of the wires I and J from the previous depth, from the left of the comparators, and computing new values for I and J as the disjunction and conjunction of those inputs, respectively. This causes true values to gravitate towards low indices. Wire values at *passthroughs* $pass(I, D)$, corresponding to unconnected wires and depths, are simply copied from the previous depth. The predicates $comp$ and $pass$ are defined by the encoding of the SN, which also indicates via a predicate $dh(R, D)$ the range of depths D that are relevant for sorting the inputs of an aggregate R, as well as via predicate $done(N, D)$ depths D at which exactly N of the first inputs are known to be sorted. The $done(N, D)$ predicate guarantees that aggregate bounds up to N are implied by corresponding wire values at depth D, and this is used in defining the output out of the network. Inferring the output based on multiple stages of the computation, instead of the final sorted output, is crucial for lazy-grounding, as the output may be inferred early without fully grounding.

Proposition 3. *The ground size of an SNA program is* $\mathcal{O}(nd)$ *ground rules for* n *inputs and a relevant depth* d*.*

In practice, d is a function of n determined by the SN.

Example 6. *The SN from Ex. 1 is encoded by* $comp(1, 2, 1)$*,* $comp(3, 4, 1)$ $comp(1, 3, 2)$*,* $comp(2, 4, 2)$*,* $comp(2, 3, 3)$*,* $pass(1, 3)$*,* $pass(4, 3)$*,* $done(1, 0)$*,* $done(2, 1)$*, and* $done(4, 3)$*. Let us then consider a count aggregate with four grounded input terms, such that* $idx(ag, 1..4)$ *have been generated using* $enum$ *and, moreover, that* $bound(ag, 2)$*,* $idx(ag, 2..3)$*, and* $dh(ag, 1..3)$ *hold. The encoding yields the locations of* 1*s in Ex. 1 as* $v(ag, 2, 0)$*,* $v(ag, 3, 0)$*,* $v(ag, 1, 1)$*,* $v(ag, 3, 1)$*,* $v(ag, 1, 2)$*,* $v(ag, 3, 2)$*,* $v(ag, 1, 3)$*, and* $v(ag, 2, 3)$*. Finally, the output* $out(ag, 2)$ *depends on* $v(ag, 2, d)$ *for* $d = 1, 3$ *since* $done$ *declares that enough values are sorted at these depths; and the case with* $d = 3$ *gives the result* $out(ag, 2)$*.*

Odd-Even Sorting Network. Encoding an SN is a matter of determining which pairs of wires are mutually connected by a comparator at a given depth. We employ Batcher's odd-even sorting networks (Batcher 1968), since these are well known for encodings of cardinality constraints in SAT (Eén and Sörensson 2006) as well as for normalizations of ground cardinality rules (Bomanson and Janhunen 2013). In the ground case they are typically realized via two intertwined divide-and-conquer algorithms, which generate networks that sort and merge, respectively. To make a first-order encoding possible, we substitute the divide-and-conquer algorithms by a closed form expression (Bekbolatov 2015) that unfolds them. The encoding allows that a single ground instance of it can support any number of SNA programs instantiated for count aggregates, even for differing sizes.

Definition 8. *The* odd-even sorting network (OESN) program F *is*

$$
\begin{aligned}
\{\ & part(P) \leftarrow span(\_, I), P = P_1 + 1, log2(I - 1, P_1).\\
& lvl(1, 1, 1) \leftarrow part(1).\\
& lvl(L, P + 1, D + L) \leftarrow L = 1..P + 1, lvl(P, P, D),\\
& \qquad part(P + 1).\\
& comp(I, J, D) \leftarrow lvl(1, P, D), span(\_, I), I < J,\\
& \qquad J = (I - 1) \oplus 2^{(P-1)} + 1.\\
& comp(I, J, D) \leftarrow lvl(L, P, D), span(\_, I), J = I + S,\\
& \qquad 1 < L, N \neq 0, N \neq B - 1, 1 = N \bmod 2, \Phi.\\
& pass(I, D) \leftarrow lvl(L,P,D), span(\_, I), 1 < L, N = 0, \Phi.\\
& pass(I, D) \leftarrow lvl(L,P,D), span(\_, I), 1{<}L, N{=}B{-}1, \Phi.\\
& dh(R, 1..D) \leftarrow span(R, N{+}1), done(N,\_), done(2N,D).\\
& done(N, D) \leftarrow log2(N, P), lvl(P, P, D). \qquad done(1, 0).\\
& log2(Ip_2, I) \leftarrow Ip_2 = 2^I, I = 0..30.\ \}
\end{aligned}
$$

where $\oplus$ *stands for bit-wise XOR and* Φ *for the conditions:* $B = 2^L, S = 2^{(P-L)}, N = (I-1)/S - ((I-1)/S/B) \cdot B.$

The $part$ predicate indicates how large parts of an SN are required to accommodate the largest seen aggregate. Part numbers from 1 to p suffice for 2^p inputs. The atom $lvl(L, P, D)$ associates depths D with these parts P and *levels* L within the parts. These concepts are depicted in Figure 1 and they stem from the two levels of hierarchy in the typical divide-and-conquer generation algorithms. The rules for $comp$ apply the closed form expression, while the rules for $pass$ complement the conditions appropriately to detect wires untouched by comparators. The rule for dh states that for an aggregate with n inputs, as large depths are necessary as are required to sort $2^{\lceil \log n \rceil}$ inputs.

Proposition 4. *The OESN program when up to* n *input literals are needed has* $\mathcal{O}(n(\log n)^2)$ *ground rules. The combined grounded size of the SNA and OESN programs for a single aggregate is* $\mathcal{O}(n(\log n)^2)$*.*

Example 7 (Ex. 6 continued)**.** *The SNA program* A *also yields* $span(ag, 1..3)$*, based on which the definition program* F *produces the used sorting network in form of the atoms* $part(1)$*,* $part(2)$*,* $lvl(1, 1, 1)$*,* $lvl(1, 2, 2)$*,* $lvl(2, 2, 3)$*,* $comp(1, 2, 1)$*,* $comp(3, 4, 1)$*,* $comp(1, 3, 2)$*,* $comp(2, 4, 2)$*,*

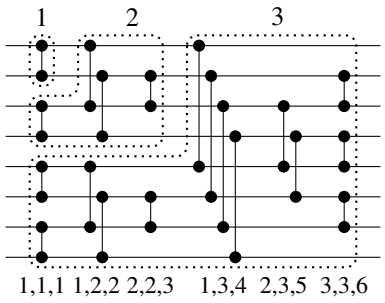


Figure 1: An odd-even sorting network for eight inputs with part numbers captured by $part$ at the top and tuples captured by lvl for each layer at the bottom. Part 1 sorts two inputs, parts 1-2 sort four, and parts 1-3 sort all eight inputs. Each part yields a $depth(N, D)$ atom where N and D are the greatest wire and depth indices of comparators within them, respectively.

$comp(2,3,3)$, $pass(1,3)$, $dh(ag,1)$, $dh(ag,2)$, $dh(ag,3)$, $done(1,0)$, $done(2,1)$, *and* $done(4,3)$.

Definition 9. *Let r be a rule of the form $h \leftarrow K \leq \#count\{T_1 : l_1; \ldots; T_n : l_n\}, dom(K)$ with a count aggregate. The* count aggregate lazy normalization *of r is the lazy normalization program $(A_r \cup F_r, G_r)$ where A_r, F_r, and G_r are the SNA, OESN, and gluing programs for r, resp.*

The following correctness result can be established, essentially using the strategy provided by Proposition 1.

Theorem 1. *Given a rule r with a count aggregate, the lazy normalization $(A_r \cup F_r, G_r)$ is faithful.*

Lazy-Normalization of Sum Aggregates. Sum aggregates are similar to count aggregates in the sense that they consider each ground instance and increment some aggregated value, with the difference that counting increases the value by 1 while summation increases by a given value.

The gluing program G_r is the same as in Definition 5, except that the in predicate is now used in the form of $in(ag, element\_tuple(T_i), V_i)$ where T_i is a tuple of terms as before and V_i is the value of T_i, which is the first term in T_i (Calimeri et al. 2012). The evaluation program is almost the same as in Definition 6 and mainly differs by the rule

$$sum(R, I, S + V) \leftarrow sum(R, I - 1, S), idx(R, I, V).$$

where $S+V$ substitutes $S+1$ in the counting grid encoding. The respective normalization program is defined below.

Definition 10. *Let r be a rule $h \leftarrow K \leq \#sum\{T_1 : l_1; \ldots; T_n : l_n\}, dom(K)$ with a sum aggregate. The* sum aggregate lazy normalization *of r is the lazy normalization program (S_r, G_r) where S_r is the summation grid encoding and G_r is the gluing program for r.*

Theorem 2. *Given a rule r with a sum aggregate, the lazy normalization (S_r, G_r) is faithful.*

5 Evaluation

Lazy grounding is a recent approach aimed at solving large-scale instances in the long run and it is partially incompatible with existing ground-and-solve techniques, which often require a complete grounding. Thus lazy-grounding cannot directly be put on top of ground-and-solve systems. Consequently, lazy-grounding ASP systems lack a decade of solver optimizations and a variety of solving techniques (e.g., equivalence preprocessing, learned clause deletion, and rapid restarts). The ASP competition benchmarks, however, are tuned to exercise those techniques while problems where grounding is an issue are explicitly excluded, see (Gebser, Maratea, and Ricca 2017). We therefore evaluate our approach, implemented in the lazy-grounding ASP system Alpha (Weinzierl 2017), on a custom, yet meaningful, set of benchmarks. Since no other lazy-grounding ASP system supports aggregates, we can only compare with ground-and-solve systems and chose Clingo (Gebser et al. 2016) and Dlv2 (Alviano et al. 2017) as representatives. The evaluated version of Clingo is 5.2.2 and the version of Dlv2 is 2.0. If not otherwise indicated, each benchmark had 300 seconds and 8GB of memory on a single core of a Linux cluster with Intel Xeon E5-2680 v3 CPUs. For each instance we report CPU time averaged over 10 runs of computing the first 10 answer sets. Count aggregates are evaluated using the compact normalization.

Simulation. ASP is very suitable for encoding rule-based simulations, but the number of potential outcomes is prohibitive for ground-and-solve systems once many time steps are to be simulated. As this benchmark shows, lazy grounding with lazy normalization does not suffer from that issue. This benchmark simulates a robot free to move on the edges of a random graph as long as its energy resources are not depleted, where depletion is checked using a count aggregate. To avoid trivial solutions, visits to some pairs of nodes are mutually exclusive and the robot may not end in its starting position. Figure 2a shows the result of this benchmark for time steps (and graph sizes) from 100 up to 2000, each one being run on 10 random instances. Clingo is faster only for 100 time steps, while at 200 time steps Alpha is already faster. Clingo runs out of memory starting at 600 time steps while Alpha can still deal with instances of size 2000. Dlv2 suffers from the same issues as Clingo and performs even worse: it is always slower than Alpha and runs out of memory at 500 time steps.

Large Input Aggregates. We evaluate lazy normalization on two problem classes with aggregates that are non-trivial to ground: a plain sum aggregate and an indegree counting of a graph. The aggregates in these programs range over a domain whose elements are guessed to avoid trivial grounding. In the plain *Summation* benchmark, lazy-normalization easily handles increasing domain sizes, while Clingo runs into timeouts (cf. Figure 2b). We report no numbers for Dlv2 here, because it was unable to compute correct answer sets for this benchmark program. The *Dynamic Indegree Counting* benchmark counts the indegrees of ver-

Size	Alpha	Clingo	Dlv2
100	2.6(0)	0.8(0)	4.7(0)
200	3.2(0)	6.2(0)	100.1(3)
500	9.1(0)	110.5(0)	memout
600	7.7(0)	memout	memout
800	27.8(0)	memout	memout
1000	47.4(1)	memout	memout
2000	86.8(1)	memout	memout

(a) Simulation.

Size	Alpha	Clingo
100	1.9(0)	1.2(0)
150	2.2(0)	5.2(0)
200	2.5(0)	15.2(0)
250	2.9(0)	37.5(0)
300	3.0(0)	96.8(0)
350	3.2(0)	247.0(3)
1000	6.5(0)	300.0(10)

(b) Summation.

Size	Alpha	Clingo	Dlv2
1k (100/10)	4.4(0)	1.5(0)	2.5(0)
3k (100/30)	13.0(0)	13.7(0)	8.1(0)
4k (200/10)	11.3(0)	25.1(0)	4.0(0)
5k (100/50)	18.7(0)	38.9(0)	3.4(0)
6k (250/10)	13.9(0)	62.7(0)	6.1(0)
12k (200/30)	36.5(0)	271.8(1)	9.6(0)
25k (500/10)	47.9(0)	300.0(10)	39.3(0)

(c) Dynamic Indegree Counting.

Figure 2: Benchmark results with runtimes in seconds and timeouts in parentheses. Size is: (a) number of time steps and graph size, (b) size of the domain that numbers are selected from, and (c) approx. number of edges (number of vertices / percentage of edge presence).

tices of a (random) graph, after one edge is removed non-deterministically. Clingo is fast for small graphs, but lazy normalization out-performs it on larger ones (cf. Figure 2c). Interestingly, Dlv2 performs better than Alpha and Clingo.

Exponential Space Saving. Lazy normalization can save up-to exponential space, as shown by the next ASP program:

```
dom(0..1).   { a;b;c }.   :- a, b.
exp(X) :- a,b, dom(X).
holds :- 2 <= #count { X_1, X_2, ..., X_n :
           exp(X_1), exp(X_2), ..., exp(X_n) }.
```

There is a domain dom with 2 elements and a non-deterministic guess over a, b and c such that a and b cannot both hold (cf. (Calimeri et al. 2012) for the employed language constructs). Atoms $exp(0)$ and $exp(1)$ are derived if both a and b are true (which they are for no answer set). Finally, an aggregate counts the number of n-tuples over exp. The results of this benchmark are shown in Table 1, where size is the number n of variables in the aggregate and 40 GB of memory are provided. Memory consumption of Clingo is growing exponentially until it hits 40 GB with size 26, where it runs out of memory. Dlv2 performs similarly but has a significantly higher memory consumption and exhibits some peculiar behavior starting at size 20, where it runs into timeouts despite being able to compute all actual answer sets of the program after a few seconds. Starting at size 26, however, Dlv2 also runs out of memory before being able to compute any answer set. In contrast to that, Alpha never runs out of memory, requires about 80 MB for small instances (below size 30), and a constant amount of about 620 MB for larger ones. Closer investigation of the latter cases revealed a inefficiency in the grounding component of Alpha, occuring only for rules with hundreds of body literals. The 620 MB memory requirement is a subsequent artifact of Java's garbage collection (GC) algorithm. Comparing Alpha with Clingo and Dlv2, it is apparent that lazy normalization can bring exponential space savings.

Comparing Normalizations. Since both lazy normalizations of count aggregates were realized in Alpha, we can also show that the compact lazy normalization based on the odd-even sorting network performs much better than the counting grid-based one.

	Alpha		Clingo		Dlv2	
Size	Time	Memory	Time	Memory	Time	Memory
10	0.8	70 MB	0.0	3 MB	0.0	5 MB
18	0.8	83 MB	0.8	124 MB	249.3	1.4 GB
20	0.9	82 MB	3.3	504 MB	900.0	6.3 GB
22	0.8	82 MB	14.7	2.06 GB	900.0	12.0 GB
24	0.9	82 MB	63.7	8.47 GB	900.0	37.6 GB
26	0.9	83 MB	370.0	34.9 GB	–	memout
28	0.9	87 MB	–	memout	–	memout
500	12.3	620 MB	–	memout	–	memout
1000	80.0	616 MB	–	memout	–	memout

Table 1: Exponential Space Saving benchmark: time and space consumption of Alpha, Clingo, and Dlv2. Size is the number of variables in the aggregate. Maximum memory is 40 GB and timeout is at 900s.

6 Discussion

Related Work. Lazy normalization, in principle, can be added to any lazy-grounding ASP system. So far, these are: GASP (Palù et al. 2009), ASPeRiX (Lefèvre et al. 2017), Omiga (Dao-Tran et al. 2012), and Alpha (Weinzierl 2017). These solvers all have in common that they process only normal rules, i.e., they allow for only ordinary literals in rule bodies and no disjunction in heads, and in particular, they do not support aggregates. Recent developments around Alpha are on exploiting justifications (Bogaerts and Weinzierl 2018) as well as its use in evaluating external atoms (Eiter, Kaminski, and Weinzierl 2017). Furthermore, Alpha is the only lazy-grounding ASP system with efficient techniques for search (CDNL), hence it was taken as basis for this work.

Related to lazy grounding is recent work in (Cuteri et al. 2017) exploring lazy instantiation for constraints alone, i.e., rules without heads, through use of propagators. Although aggregates are not addressed therein, the approach may efficiently solve some of our benchmark problems, provided that the benchmarks were refactored to utilize a mix of constraints, aggregates, and additional guess-and-check atoms in place of the current rules with aggregates.

The first aggregate-based extensions of ASP were *choice rules*, *cardinality rules*, and *weight rules* (Simons, Niemelä, and Soininen 2002) where the latter two involve counting and summation aggregates in the terminology of this paper. Ground-and-solve systems support such rules by means of either native functions for propagation and calculation of

reasons behind conflicts (Gebser et al. 2009) or normalization on the ground level (Bomanson 2017). A first-order normalization of these rules is already present in (Polleres et al. 2013) along with the motivation of debugging answer-set programs involving aggregates. Interestingly, the normalization schemes of Polleres et al. (2013) rely on a fixed total order on input terms in contrast with the dynamic order created by the enumeration built-in devised in this work. On the downside, their encodings suffer from higher space complexity, which is witnessed by the fact that an early version of our normalization tool LP2NORMAL (Janhunen and Niemelä 2011) produces more compact normalizations than what is obtained by grounding the first-order translations devised by Polleres et al. (2013).

Aggregates have their roots in database operations (see, e.g., (Greco 1999)). The approach therein is based on DATALOG programs where dedicated predicates are used to express summation, for instance. Consequently, linear constraints can be embedded into rules. In particular, recursion through such special predicates is forbidden in terms of stratification. The evaluation of summation aggregates resembles our approach in the sense that (i) it is based on first-order rewriting and (ii) terms contributing to the sum are ordered and the aggregate is recursively evaluated in a chosen order. However, this order is produced upfront via nondeterministic choice, which is suitable for the implementation considered in (Greco 1999) but impractical in ASP solving. In contrast, we commit the next available position i in an aggregate ag to a ground term t on-the-fly by using the built-in predicate $enum(ag, t, i)$. In this way, the evaluation of the aggregate need not be delayed until the completion of the resulting order of terms.

Aggregates in ASP have also been studied on a higher level of abstraction and the central properties of aggregates were identified when, e.g., *monotone*, *anti-monotone*, and *convex* aggregates were studied (Liu and Truszczynski 2006). Since ASP deploys recursive definitions in terms of rules, the introduction of aggregates posed a challenge from the semantic perspective and a number of proposals were made; cf. (Ferraris 2011; Faber, Pfeifer, and Leone 2011; Gelfond and Zhang 2014)). Results on rewriting aggregates, such as *non-monotone* aggregates into monotone ones (Alviano, Faber, and Gebser 2015), support the general idea adopted in this paper that monotone aggregates provide a natural basis for implementing aggregates in practical ASP systems. We note that proper disjunctive rules seem indispensable for such rewritings in general, and therefore such techniques do not fall strictly under the concept of normalization as understood in this paper.

Convex Aggregates. Monotone aggregates serve as natural primitives in practical implementations and more complex aggregates can be realized on top of them. To illustrate this, let us consider counting and summation aggregates with *upper bounds* in addition to lower bounds, such as in:

$$ok \leftarrow 10 \leq \#sum\{V, I : pick(I), cand(I, V)\} \leq 20.$$

The body of the rule is an example of a convex aggregate condition that can be understood as a conjunction of monotone and an anti-monotone aggregate conditions. This can be made explicit by rewriting the rule using the ideas from (Simons, Niemelä, and Soininen 2002):

$$enough \leftarrow 10 \leq \#sum\{V, I : pick(I), cand(I, V)\}.$$
$$toomuch \leftarrow 21 \leq \#sum\{V, I : pick(I), cand(I, V)\}.$$
$$ok \leftarrow enough, not\ toomuch.$$

Considering performance, both bounds can now be evaluated lazily, but the upper bound poses some challenge: if the solver assigns ok true and $toomuch$ false before all relevant ground instances of $pick(\cdot)$ are generated, then backtracking may be necessary if new instances of $pick(\cdot)$ are grounded later on. Besides that, convex aggregates can be evaluated within our framework, but an implementation is future work.

7 Conclusion

In this paper, we presented a lean framework for *lazy normalization*, which enables the first-order normalization of aggregates subject to lazy grounding. To illustrate the feasibility of the framework, we devised several novel first-order encodings of counting and summation aggregates. Their correctness and asymptotic behavior was formally analyzed. To the best of our knowledge, this is the first systematic approach to the evaluation of aggregates in a lazy-grounding setting.

The lazy grounding ASP system Alpha was extended to support lazy normalization of aggregates and benchmarks demonstrate that lazy normalization can outperform the state-of-the-art ground-and-solve system Clingo on aggregate evaluation if grounding is an issue. Lazy grounding may also bring up-to exponential savings in space and time, as shown experimentally. Furthermore, the results on running rule-based simulations for hundreds of time steps indicate a potential new application area for ASP solving based on lazy grounding and lazy normalization. Together this suggests that lazy normalization is a useful new technique for ASP solving and knowledge representation in general.

In future work, we plan to investigate further normalization schemes for monotone/convex aggregates, potentially with upper bounds and other aggregate types like average. Lazy normalization may also prove useful with other formalisms such as FO(ID) proposed by de Cat et al. (2015).

Acknowledgments

This work has been supported by the Academy of Finland, project 251170 and the research project *DynaCon (FFG-PNr.: 861263)*, which is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" between 2017 and 2020 (see https://iktderzukunft.at/en/ for more information).

References

Alviano, M.; Dodaro, C.; Faber, W.; Leone, N.; and Ricca, F. 2013. WASP: A native ASP solver based on constraint learning. In Cabalar and Son (2013), 54–66.

Alviano, M.; Calimeri, F.; Dodaro, C.; Fuscà, D.; Leone, N.; Perri, S.; Ricca, F.; Veltri, P.; and Zangari, J. 2017. The

ASP system DLV2. In *LPNMR*, volume 10377 of *LNCS*, 215–221.

Alviano, M.; Faber, W.; and Gebser, M. 2015. Rewriting recursive aggregates in answer set programming: back to monotonicity. *Theory and Practice of Logic Programming* 15(4-5):559–573.

Batcher, K. E. 1968. Sorting networks and their applications. In *AFIPS*, volume 32 of *AFIPS Conference Proceedings*, 307–314.

Bekbolatov, R. 2015. Batcher's odd-even merge based sorting network node partner calculation. Available at http://sparkydots.blogspot.fi/2015/05/batchers-odd-even-merging-network.html.

Bogaerts, B., and Weinzierl, A. 2018. Exploiting justifications for lazy grounding of answer set programs. In *IJCAI*, 1737–1745.

Bomanson, J., and Janhunen, T. 2013. Normalizing cardinality rules using merging and sorting constructions. In *LPNMR*, volume 8148 of *LNCS*, 187–199.

Bomanson, J. 2017. lp2normal - A normalization tool for extended logic programs. In *LPNMR*, volume 10377 of *LNCS*, 222–228.

Cabalar, P., and Son, T. C., eds. 2013. *LPNMR*, volume 8148 of *LNCS*.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Ricca, F.; and Schaub, T. 2012. ASP-CORE-2 input language format.

Cuteri, B.; Dodaro, C.; Ricca, F.; and Schüller, P. 2017. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *Theory and Practice of Logic Programming* 17(5-6):780–799.

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 *JELIA*, volume 7519 of *LNCS*, 480–483.

de Cat, B.; Denecker, M.; Bruynooghe, M.; and Stuckey, P. J. 2015. Lazy model expansion: Interleaving grounding with search. *Journal of Artificial Intelligence Research* 52:235–286.

Eén, N., and Sörensson, N. 2006. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2(1-4):1–26.

Eiter, T.; Ianni, G.; and Krennwallner, T. 2009. Answer set programming: A primer. In *Reasoning Web*, volume 5689 of *LNCS*, 40–110.

Eiter, T.; Kaminski, T.; and Weinzierl, A. 2017. Lazy-grounding for answer set programs with external source access. In *IJCAI*, 1015–1022.

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

Falkner, A. A.; Friedrich, G.; Haselböck, A.; Schenner, G.; and Schreiner, H. 2016. Twenty-five years of successful application of constraint technologies at Siemens. *AI Magazine* 37(4):67–80.

Ferraris, P. 2011. Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic* 12(4):25:1–25:40.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2009. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *ICLP*, 250–264.

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

Gebser, M.; Kaufmann, B.; and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence* 187:52–89.

Gebser, M.; Maratea, M.; and Ricca, F. 2017. The sixth answer set programming competition. *Journal of Artificial Intelligence Research* 60:41–95.

Gelfond, M., and Zhang, Y. 2014. Vicious circle principle and logic programs with aggregates. *Theory and Practice of Logic Programming* 14(4-5):587–601.

Greco, S. 1999. Dynamic programming in datalog with aggregates. *IEEE Trans. Knowl. Data Eng.* 11(2):265–283.

Hölldobler, S.; Manthey, N.; and Steinke, P. 2012. A compact encoding of pseudo-Boolean constraints into SAT. In *KI*, volume 7526 of *LNCS*, 107–118.

Janhunen, T., and Niemelä, I. 2011. Compact translations of non-disjunctive answer set programs to propositional clauses. In *Gelfond Festscrift*, volume 6565 of *LNCS*, 111–130.

Lefèvre, C.; Beatrix, C.; Stephan, I.; and Garcia, L. 2017. Asperix, a first-order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming* 17(3):266–310.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2002. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7:499–562.

Liu, L., and Truszczynski, M. 2006. Properties and applications of programs with monotone and convex constraints. *Journal of Artificial Intelligence Research* 27:299–334.

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

Polleres, A.; Frühstück, M.; Schenner, G.; and Friedrich, G. 2013. Debugging non-ground ASP programs with choice rules, cardinality and weight constraints. In Cabalar and Son (2013), 452–464.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181–234.

Weinzierl, A. 2017. Blending lazy-grounding and CDNL search for answer-set solving. In *LPNMR*, volume 10377 of *LNCS*, 191–204.

Woltran, S. 2004. Characterizations for relativized notions of equivalence in answer set programming. In *JELIA*, volume 3229 of *LNCS*, 161–173.