Grounding Recursive Aggregates: Preliminary Report

Martin Gebser[1,3], Roland Kaminski[3], and Torsten Schaub[2,3]*

[1] Aalto University, HIIT
[2] INRIA Rennes
[3] University of Potsdam

Abstract. Problem solving in Answer Set Programming consists of two steps, a first grounding phase, systematically replacing all variables by terms, and a second solving phase computing the stable models of the obtained ground program. An intricate part of both phases is the treatment of aggregates, which are popular language constructs that allow for expressing properties over sets. In this paper, we elaborate upon the treatment of aggregates during grounding in `gringo` series 4. Consequently, our approach is applicable to grounding based on semi-naive database evaluation techniques. In particular, we provide a series of algorithms detailing the treatment of recursive aggregates and illustrate this by a running example.

1 Introduction

Modern grounders like (the one in) `dlv` [1] or `gringo` [2] are based on semi-naive database evaluation techniques [3, 4] for avoiding duplicate work during grounding. Grounding is seen as an iterative bottom-up process guided by the successive expansion of a program's Herbrand base, that is, the set of variable-free atoms constructible from the signature of the program at hand. During this process, a ground rule is only produced if its positive body atoms belong to the current Herbrand base, in which case its head atom is added to the current Herbrand base. The basic idea of semi-naive database evaluation is to focus this process on the new atoms generated at each iteration in order to avoid reproducing the same ground rules. This idea is based on the observation that the production of a new ground rule relies on the existence of an atom having been new at the previous iteration. Accordingly, a ground rule is only produced if its positive body contains at least one atom produced at the last iteration.

In what follows, we show how a grounding framework relying upon semi-naive database evaluation techniques can be extended to incorporate recursive aggregates. An example of such an aggregate is shown in Table 1, giving an encoding of the *Company Controls Problem* [5]: A company X controls a company Y, if X directly or indirectly controls more than 50% of the shares of Y. The aggregate sum^+ implements summation over positive integers. Notably, it takes part in the recursive definition of *controls*/2 in Table 1. A corresponding problem instance is given in Table 2. Note that a systematic instantiation of the four variables in Table 1 with the eight constants in Table 2 results in 64 ground rules. However, taken together, the encoding and the instance are equivalent to the program in Table 3, which consists of four ground rules only. In fact, all liter-

* Affiliated with Simon Fraser University, Canada, and IIIS Griffith University, Australia.

$$\begin{aligned}
\mathit{controls}(X,Y) \leftarrow \mathit{sum}^+\{S : \mathit{owns}(X,Y,S);&\\
S,Z : \mathit{controls}(X,Z), \mathit{owns}(Z,Y,S)\} > \overline{50}&\\
\wedge\ \mathit{company}(X) \wedge \mathit{company}(Y) \wedge X \neq Y&
\end{aligned}$$

Table 1. Company Controls Encoding

$\mathit{company}(c_1).$	$\mathit{company}(c_2).$	$\mathit{company}(c_3).$	$\mathit{company}(c_4).$
$\mathit{owns}(c_1,c_2,\overline{60}).$	$\mathit{owns}(c_1,c_3,\overline{20}).$	$\mathit{owns}(c_2,c_3,\overline{35}).$	$\mathit{owns}(c_3,c_4,\overline{51}).$

Table 2. Company Controls Instance

als in Table 3 can even be evaluated in view of the problem instance, which moreover allows us to evaluate the aggregate atoms, so that the grounding of the above company controls instance boils down to the four facts $\mathit{controls}(c_1,c_2)$, $\mathit{controls}(c_3,c_4)$, $\mathit{controls}(c_1,c_3)$, and $\mathit{controls}(c_1,c_4)$.

Accordingly, the goal of this paper is to elaborate upon the efficient computation of the relevant grounding of programs with recursive aggregates. Section 2 starts with recalling the formal preliminaries from [6]. Section 3 provides basic grounding algorithms (cf. [1]), paving the way for the more sophisticated algorithms addressing recursive aggregates in Section 4. We summarize our contribution and relate it to the state of the art in Section 5. The developed approach is implemented in `gringo` series 4.

2 Formal Preliminaries

This section recalls the formal preliminaries regarding the syntax and semantics of `gringo`'s input language, developed in [6].

2.1 Syntax

Alphabet. We consider numerals, (symbolic) constants, variables, and aggregate names, along with the symbols

$$\neq \quad < \quad > \quad \leq \quad \geq\,^{4} \tag{1}$$
$$\bot \quad \sim \quad \wedge \quad \vee \quad \leftarrow \tag{2}$$
$$, \quad ; \quad : \quad (\quad) \quad \{ \quad \} \tag{3}$$

Numerals are strings of numbers optionally preceded with a minus symbol. Constants are strings of letters, underscores, and numbers starting with a lowercase letter. Variables are strings of letters, underscores, and numbers starting with an uppercase letter.[5]

Terms. Numerals, constants, and variables are terms. Given a constant f and a term tuple $\boldsymbol{t}$, $f(\boldsymbol{t})$ is a term as well. A variable-free term is said to be ground.

[4] Equality is not included here because it is treated specially in `gringo`; a description is beyond the scope of this paper.
[5] We use _ to denote anonymous variables, i.e., each _ stands for unique variable.

$$
\begin{aligned}
\mathit{controls}(c_1, c_2) \leftarrow\ & \mathit{sum}^+\{\overline{60} : \mathit{owns}(c_1, c_2, \overline{60})\} > \overline{50} \\
& \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_2) \wedge c_1 \neq c_2 \\
\mathit{controls}(c_3, c_4) \leftarrow\ & \mathit{sum}^+\{\overline{51} : \mathit{owns}(c_3, c_4, \overline{51})\} > \overline{50} \\
& \wedge \mathit{company}(c_3) \wedge \mathit{company}(c_4) \wedge c_3 \neq c_4 \\
\mathit{controls}(c_1, c_3) \leftarrow\ & \mathit{sum}^+\{\overline{20} : \mathit{owns}(c_1, c_3, \overline{20}); \\
& \qquad \overline{35}, c_2 : \mathit{controls}(c_1, c_2), \mathit{owns}(c_2, c_3, \overline{35})\} > \overline{50} \\
& \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_3) \wedge c_1 \neq c_3 \\
\mathit{controls}(c_1, c_4) \leftarrow\ & \mathit{sum}^+\{\overline{51}, c_3 : \mathit{controls}(c_1, c_3), \mathit{owns}(c_3, c_4, \overline{51})\} > \overline{50} \\
& \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_4) \wedge c_1 \neq c_4
\end{aligned}
$$

Table 3. Relevant Grounding of Company Controls

Interpretation of numerals and aggregates. A numeral $\overline{n}$ corresponds to the integer n. There is a total order on all ground terms extending that on numerals, that is, for any integers m and n, $\overline{m} \leq \overline{n}$ if $m \leq n$.

For a ground term tuple $\boldsymbol{t}$, $\mathit{weight}(\boldsymbol{t})$ is n, if the first element of $\boldsymbol{t}$ is a numeral of form $\overline{n}$, otherwise it is 0.

Each aggregate name α is associated with a function $\widehat{\alpha}$ from the set of sets of ground term tuples into the set of ground terms. Given a set T of ground term tuples, we consider the aggregate names/functions defined by

- $\widehat{\mathit{sum}}(T) = \overline{\Sigma_{\boldsymbol{t} \in T} \mathit{weight}(\boldsymbol{t})}$, if the subset of tuples with non-zero weights is finite, and $\overline{0}$ otherwise;
- $\widehat{\mathit{sum}^+}(T) = \overline{\Sigma_{\boldsymbol{t} \in T, \mathit{weight}(\boldsymbol{t})>0} \mathit{weight}(\boldsymbol{t})}$, if the subset of tuples with positive weights is finite, and ∞ otherwise.[6]

Atoms and literals. Symbolic atoms have the form $p(\boldsymbol{t})$ where p is a constant and $\boldsymbol{t}$ is a term tuple. Comparison atoms have form $u_1 \prec u_2$ where u_1 and u_2 are terms. We use atom $\bot$ to denote a comparison atom that is false (e.g., $0 > 0$), and use $\top$ analogously. Simple literals have form a or ${\sim}a$ where a is a symbolic or comparison atom.

Aggregate atoms have form

$$\alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots ; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec s \tag{4}$$

where

- $n \geq 0$
- α is an aggregate name
- each $\boldsymbol{t}_i$ is a term tuple
- each $\boldsymbol{L}_i$ is a tuple of simple literals
- $\prec$ is one of the symbols (1)
- s is a term, also called guard

[6] $\overline{n} < \infty$ holds for any numeral $\overline{n}$.

Finally, literals have the form a or $\sim a$ where a is either a symbolic, comparison, or aggregate atom.[7]

Rules and programs. Rules are of form $h \leftarrow l_1 \wedge \cdots \wedge l_n$, where $n \geq 0$, h is a symbolic atom, and each l_i is a literal. A program is a finite set of rules.

Miscellaneous definitions. We use the following projection functions on rules.

– $\mathrm{head}(r) = h$ in a rule r of the above form
– $\mathrm{body}(r) = \{l_1, \dots, l_n\}$ in a rule r of the above form
– $\mathrm{body}^+(r) = \{a \in \mathrm{body}(r) \mid a \text{ is a symbolic atom}\}$
– $\mathrm{body}^-(r) = \{a \mid \sim a \in \mathrm{body}(r), a \text{ is a symbolic atom}\}$
– $\mathrm{body}^\pm(r) = \mathrm{body}^+(r) \cup \mathrm{body}^-(r)$

In the following, some body literals are marked. The binary relation $r \dagger l$ holds if the literal $l \in \mathrm{body}(r)$ of rule r is marked. Marked body literals are indicated by $l^\dagger$.

A substitution is a mapping from variables to (ground) terms. We represent substitutions by sets of form $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ where $n \geq 0$, each x_i is a variable, and each t_i is a ground term. A substitution σ of the above form applied to a literal l, written $l\sigma$, replaces all occurrences of variables x_i in l with corresponding terms t_i.

Moreover, we associate in what follows each occurrence of an aggregate in a logic program with a unique identifier. We use α_i, $\boldsymbol{x}_i$, and s_i to refer to the aggregate function, tuple of global variables, and guard of the aggregate occurrence identified by i. Furthermore, $(s_i)^{\boldsymbol{x}_i}_{\boldsymbol{g}}$ refers to the ground guard where the variables listed in tuple $\boldsymbol{x}_i$ are replaced with the corresponding terms in tuple $\boldsymbol{g}$ in s_i.

An aggregate α together with a relation $\prec$ is monotone, if for any sets $T_1 \subseteq T_2$ of ground term tuples and ground term s, we have that $\alpha(T_1) \prec s$ implies $\alpha(T_2) \prec s$.

2.2 Semantics

The semantics of programs rests upon a translation into (infinitary) propositional formulas along with their stable models [7].

Ground simple literals are mapped via τ on propositional atoms as follows.

– $\tau(a) = a$ for (ground) symbolic atom a
– $\tau(t_1 \prec t_2)$ is $\top$, if the relation $\prec$ holds between t_1 and t_2, and $\bot$ otherwise
– $\tau(\sim a) = \neg\tau(a)$ for a literal $\sim a$

Global variables. A variable is global

– in a simple literal, if it occurs in the literal
– in an aggregate literal, if it occurs in the guard
– in a rule, if it is global in the head or a body literal

[7] `gringo` as well as its semantic underpinnings in [6] also allow for double negated literals of form $\sim\sim a$.

Aggregate literals. The translation τ extends to aggregate atoms a as in (4) as follows. An instance of an aggregate element $\boldsymbol{t} : \boldsymbol{L}$ is obtained by substituting all its variables with ground terms. We let $\tau\boldsymbol{L}$ stand for the conjunction of applications of τ to the ground simple literals in $\boldsymbol{L}$.

Let E be the set of all instances of aggregate elements in a. A set $\Delta \subseteq E$ justifies a, if the relation $\prec$ holds between $\widehat{\alpha}\{\boldsymbol{t} \mid (\boldsymbol{t} : \boldsymbol{L}) \in \Delta\}$ and the guard s. Then, τa is the conjunction of formulas $\bigwedge_{(\boldsymbol{t}:\boldsymbol{L})\in\Delta} \tau\boldsymbol{L} \to \bigvee_{(\boldsymbol{t}:\boldsymbol{L})\in E\setminus\Delta} \tau\boldsymbol{L}$ for all sets $\Delta \subseteq E$ that do not justify a.

A negative aggregate literal $\sim a$ is treated analogous to a negative simple literal.

Rules and programs. An instance of a rule r is obtained by substituting all global variables with ground terms. Then, τr is the set of formulas $\tau l_1 \wedge \cdots \wedge \tau l_n \to \tau h$ for all instances $h \leftarrow l_1 \wedge \cdots \wedge l_n$ of rule r, and $\tau P = \bigcup_{r\in P} \tau r$ for a program P.

Stable models. The stable models of a logic program P are the stable models of the (infinitary) propositional formula τP [7].

2.3 Safety and Rule Dependency Graph

A global variable is safe in a rule, if it is bound by a positive symbolic literal in the rule body. A non-global variable is safe in an aggregate element, if it is bound by a positive symbolic literal in the corresponding aggregate element. A rule is safe, if all its variables are safe. A program is safe, if all its rules are safe. In what follows, we consider safe programs only.

The *rule dependency graph* $G = (V, E)$ of a (normal) logic program P is a directed graph such that $V = P$ and $E = \{(r_1, r_2) \in V \times V \mid l \in \mathrm{body}^{\pm}(r_2), \mathrm{head}(r_1) \text{ unifies } l\}$.[8] The positive rule dependency graph G^+ is defined similarly but considers edges induced by positive literals only ($l \in \mathrm{body}^+(r_2)$).

3 Basic Grounding Algorithms

This section provides some basic algorithms underlying semi-naive evaluation based grounding (see also [1]). All of them apply to normal logic programs and are thus independent of the treatment of recursive aggregates described in the next section.

We illustrate the basic algorithms by means of a Hamiltonian cycle example[9] using the graph in Figure 1. This graph is represented by the problem instance in Table 4. The actual problem encoding is given in (5) to (13) below. The resulting Hamiltonian cycle is expressed through instances of predicate *path*/2; a detailed discussion of such encodings can be found in [8–10].

[8] Unification assumes that variables in r_1 and r_2 are distinct, even if they have the same name.
[9] `https://en.wikipedia.org/wiki/Hamiltonian_path_problem`

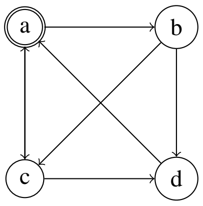


Fig. 1. Hamiltonian Cycle Instance (Graph)

$node(a).$	$node(b).$	$node(c).$
$node(d).$	$start(a).$	$edge(a,b).$
$edge(a,c).$	$edge(b,c).$	$edge(b,d).$
$edge(c,a).$	$edge(c,d).$	$edge(d,a).$

Table 4. Hamiltonian Cycle Instance

$$path(X,Y) \leftarrow edge(X,Y) \land {\sim}omit(X,Y) \tag{5}$$
$$omit(X,Y) \leftarrow edge(X,Y) \land {\sim}path(X,Y) \tag{6}$$
$$\leftarrow path(X,Y) \land path(X',Y) \land X < X' \tag{7}$$
$$\leftarrow path(X,Y) \land path(X,Y') \land Y < Y' \tag{8}$$
$$on\_path(Y) \leftarrow path(X,Y) \land path(Y,Z) \tag{9}$$
$$\leftarrow node(X) \land {\sim}on\_path(X) \tag{10}$$
$$reach(X) \leftarrow start(X) \tag{11}$$
$$reach(Y) \leftarrow reach(X) \land path(X,Y) \tag{12}$$
$$\leftarrow node(X) \land {\sim}reach(X) \tag{13}$$

3.1 Analyzing Logic Programs

The function `Analyze` given in Algorithm 1 takes a logic program P, classifies occurrences of recursive symbolic atoms (A_{r}), and groups rules into components suitable for successive grounding. The classification of atoms can be used to apply on-the-fly simplifications in the following algorithms (cf. Algorithm 3).

`Analyze` first determines the strongly connected components of the program's dependency graph (Lines 2-3). This graph contains dependencies induced by both positive and negative literals. The outer loop (Lines 5-5) iterates over its components in topological order.[10] Each component is then further refined in terms of its positive dependency graph (Lines 6-7).

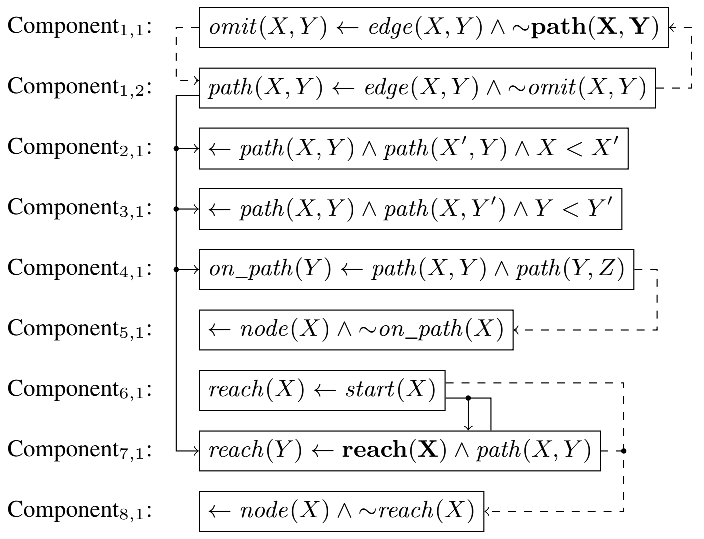
The set A_{r} of recursive symbolic atoms is determined in Line 9. These are all body literals whose atom unifies with the head of a rule in the current or a following component. Finally, the refined component together with its recursive atoms is appended to the list L in Line 10. This list is the result of the algorithm returned in Line 11.

Figure 2 shows the dependency graph of the encoding given in (5) to (13). Positive edges are depicted with solid lines, negative ones with dashed lines. Recursive atoms are typeset in bold. The negative edge from Component$_{1,2}$ to Component$_{1,1}$ is due to the fact that $path(X,Y)$ in the negative body of (6) unifies with $path(X,Y)$ in the head of (5). Furthermore, the occurrence of $path(X,Y)$ in (6) is recursive because it induces an edge from a later component in the topological ordering at hand. In contrast to positive literals, the recursiveness of negative literals depends on the topological ordering.

[10] A component C_1 precedes C_2 when there is an edge (r_1, r_2) with $r_1 \in C_1$ and $r_2 \in C_2$.

```
function Analyze($P$)
    let $G$ be the dependency graph of $P$
        $S$ be the strongly connected components of $G$
    $L \leftarrow []$
    foreach $C$ in $S$ do
        let $G^+$ be the positive dependency graph of $C$
            $S^+$ be the strongly connected components of $G^+$
        foreach $C^+$ in $S^+$ do
            let $A_\mathrm{r} = \{a \in \mathrm{body}^\pm(r_2) \mid r_1 \in P, r_2 \in C^+, \mathrm{head}(r_1) \text{ unifies } a\}$
            $(L, P) \leftarrow (L + [(C^+, A_\mathrm{r})], P \setminus C^+)$
    return $L$
```

Algorithm 1: Analyze Logic Programs for Grounding



Component$_{1,1}$: $omit(X,Y) \leftarrow edge(X,Y) \wedge {\sim}\mathbf{path(X,Y)}$
Component$_{1,2}$: $path(X,Y) \leftarrow edge(X,Y) \wedge {\sim}omit(X,Y)$
Component$_{2,1}$: $\leftarrow path(X,Y) \wedge path(X',Y) \wedge X < X'$
Component$_{3,1}$: $\leftarrow path(X,Y) \wedge path(X,Y') \wedge Y < Y'$
Component$_{4,1}$: $on\_path(Y) \leftarrow path(X,Y) \wedge path(Y,Z)$
Component$_{5,1}$: $\leftarrow node(X) \wedge {\sim}on\_path(X)$
Component$_{6,1}$: $reach(X) \leftarrow start(X)$
Component$_{7,1}$: $reach(Y) \leftarrow \mathbf{reach(X)} \wedge path(X,Y)$
Component$_{8,1}$: $\leftarrow node(X) \wedge {\sim}reach(X)$

Fig. 2. Hamiltonian Cycle Dependency Graph

For instance, $omit(X,Y)$ would be recursive in the topological order obtained by exchanging Component$_{1,1}$ and Component$_{1,2}$. Regarding Component$_{7,1}$, the occurrence of $reach(X)$ in the body of (12) is recursive because it unifies with the head of the same rule. Accordingly, it induces a self-loop in the dependency graph.

3.2 Preparing Components for Grounding

The function `Prepare` sets up the rules in a component C for grounding w.r.t. its recursive atoms A_r. To this end, it adds one of the subscripts n, o, or a to the predicate names of the atoms in the positive rule bodies of a given component.[11] These subscripts

[11] The alphabet in Section 2 does not allow for predicate names with subscripts. During grounding, we temporarily extend this alphabet with such predicate names.

```
1  function Prepare($C, A_\mathrm{r}$)
2      $L \leftarrow \emptyset$
3      foreach $r$ in $C$ do
4          $D \leftarrow \emptyset$
5          let $S = \mathrm{body}^+(r) \cap A_\mathrm{r}$
6          foreach $p(\boldsymbol{x})$ in $S$ do
7              $L \leftarrow L \cup \left\{ \begin{array}{l} \mathrm{head}(r) \leftarrow \bigwedge_{q(\boldsymbol{y}) \in D} q_\mathrm{o}(\boldsymbol{y}) \wedge p_\mathrm{n}(\boldsymbol{x}) \\ \qquad \wedge \bigwedge_{q(\boldsymbol{y}) \in \mathrm{body}^+(r) \setminus (D \cup \{p(\boldsymbol{x})\})} q_\mathrm{a}(\boldsymbol{y}) \\ \qquad \wedge \bigwedge_{l \in \mathrm{body}(r) \setminus \mathrm{body}^+(r)} l \end{array} \right\}$
8              $D \leftarrow D \cup \{p(\boldsymbol{x})\}$
9          if $S = \emptyset$ then
10             $L \leftarrow L \cup \left\{ \begin{array}{l} \mathrm{head}(r) \leftarrow \bigwedge_{p(\boldsymbol{x}) \in \mathrm{body}^+(r)} p_\mathrm{n}(\boldsymbol{x}) \\ \qquad \wedge \bigwedge_{l \in \mathrm{body}(r) \setminus \mathrm{body}^+(r)} l \end{array} \right\}$
11     return $L$
```

Algorithm 2: Prepare Components

indicate *new*, *old*, and *all* atoms belonging to the current materialization of the Herbrand base. In turn, they are used in the course of semi-naive database evaluation to avoid duplicate work when grounding a component w.r.t. an expanding Herbrand base.

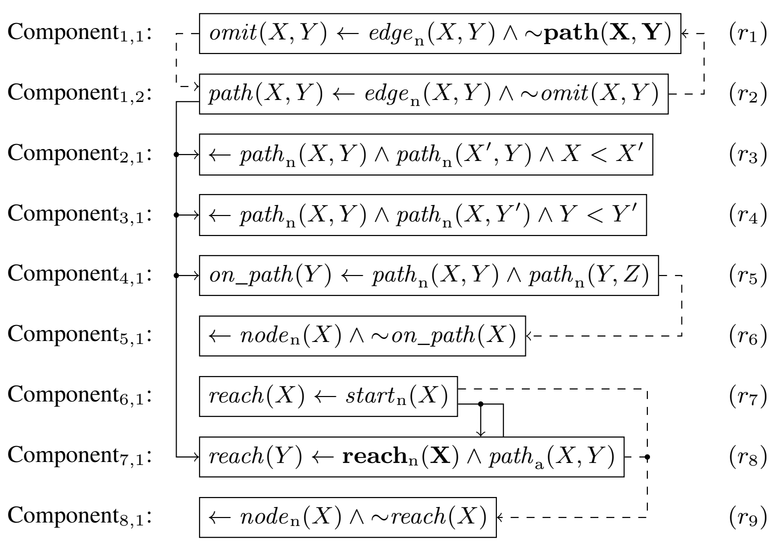
The loop in Lines 3-3 iterates over the rules in the component at hand. Each such rule r is expanded into a set of rules (loop in Lines 6-6) w.r.t. the recursive atoms in its body (Line 5). In the first row of Line 7, predicate names of recursive atoms already considered (set D) receive subscript o, and the predicate name of the recursive atom $p(\boldsymbol{x})$ receives subscript n. In the second row, the recursive atoms not yet considered as well as non-recursive atoms of the positive body receive subscript a. Finally, in the third row, the remaining body literals are kept unmodified. If there are no recursive atoms (Line 9), then subscript n is added to all positive body elements (first row of Line 10). As in the case with recursive atoms, the remaining body literals are kept unmodified (second row).

The result of preparing all components of the dependency graph in Figure 2 is given in Figure 3. All rules but r_8 contain only non-recursive positive body literals, which are adorned with subscript n. Unlike this, the non-recursive positive body literal $\mathit{path}(X, Y)$ in r_8 is adorned with a, while only the recursive one, $\mathit{reach}(X)$, receives subscript n. Since there is only one recursive body atom, only one rule is generated.

3.3 Grounding Rules

The rule grounding algorithm relies upon two auxiliary functions. First, function order returns a *safe body order* of a rule body.[12] A safe body order of a body $\{b_1, \dots, b_n\}$ is a tuple $(b_1, \dots, b_n)$ such that $\{b_1, \dots, b_i\}$ is safe for each $1 \leq i \leq n$. For example,

[12] The runtime of instantiation algorithms is sensitive to the chosen body order. In the context of ASP, heuristics for ordering body literals have been studied in [11].



Component$_{1,1}$: $\boxed{omit(X,Y) \leftarrow edge_{\mathrm{n}}(X,Y) \wedge {\sim}\mathbf{path}(\mathbf{X},\mathbf{Y})}$ (r_1)

Component$_{1,2}$: $\boxed{path(X,Y) \leftarrow edge_{\mathrm{n}}(X,Y) \wedge {\sim}omit(X,Y)}$ (r_2)

Component$_{2,1}$: $\boxed{\leftarrow path_{\mathrm{n}}(X,Y) \wedge path_{\mathrm{n}}(X',Y) \wedge X < X'}$ (r_3)

Component$_{3,1}$: $\boxed{\leftarrow path_{\mathrm{n}}(X,Y) \wedge path_{\mathrm{n}}(X,Y') \wedge Y < Y'}$ (r_4)

Component$_{4,1}$: $\boxed{on\_path(Y) \leftarrow path_{\mathrm{n}}(X,Y) \wedge path_{\mathrm{n}}(Y,Z)}$ (r_5)

Component$_{5,1}$: $\boxed{\leftarrow node_{\mathrm{n}}(X) \wedge {\sim}on\_path(X)}$ (r_6)

Component$_{6,1}$: $\boxed{reach(X) \leftarrow start_{\mathrm{n}}(X)}$ (r_7)

Component$_{7,1}$: $\boxed{reach(Y) \leftarrow \mathbf{reach}_{\mathrm{n}}(\mathbf{X}) \wedge path_{\mathrm{a}}(X,Y)}$ (r_8)

Component$_{8,1}$: $\boxed{\leftarrow node_{\mathrm{n}}(X) \wedge {\sim}reach(X)}$ (r_9)

Fig. 3. Prepared Hamiltonian Cycle Encoding

$(p(X), {\sim}q(X))$ is a safe body order, while $({\sim}q(X), p(X))$ is not. Second, given a symbolic atom a, a substitution σ, and a set A of ground atoms, function $\mathrm{matches}(a, \sigma, A)$ returns the set of *matches* for a in A w.r.t. σ. A match is a $\subseteq$-minimal substitution σ' such that $a\sigma' \in A$ and $\sigma \subseteq \sigma'$. For instance, $\mathrm{matches}(p(X,Y), \{Y \mapsto a\}, \{p(a,a), p(b,b), p(c,a)\})$ yields $\{X \mapsto a, Y \mapsto a\}$ and $\{X \mapsto c, Y \mapsto a\}$.

With both functions at hand, we assemble the basic algorithm `GroundRule` for grounding individual rules in Algorithm 3. Note that the original rule r' along with its marking $\dagger$ can be ignored in our context; they are only relevant when treating aggregates in Section 4.[13] The algorithm proceeds along the safe body order determined in Line 20. If no body literals remain, a ground rule is generated in Line 7 provided that its head is not among the established facts. Moreover, if the rule in focus has become a fact, its head is added to A_{f} in Line 8. The remainder constitutes a case analysis upon the type of the left-most body literal. If b_1 is a positive body literal, an instance of b_1 is added in turn to the (partial) ground body B for each match of b_1. However, this is only done if the literal is not marked and the instance does not yet belong to the established facts. If b_1 is a negative body literal, the instance obtained by applying the current substitution is added to the ground body. Again, this is only done if the literal is not marked, and the literal is recursive or there is already a derivation for it. Substitutions where the instance is a fact are skipped altogether. Finally, comparison literals are directly evaluated and rule instantiation is only pursued if the test was successful.

For illustration, we trace in Figure 4 the application of `GroundRule` to rule r_1, viz. '$omit(X,Y) \leftarrow edge_{\mathrm{n}}(X,Y) \wedge {\sim}path(X,Y)$', from Figure 3. Figure 4 also gives

[13] This is the rule before the application of function `Prepare`. That is, the positive body literals of r' are free of subscripts o, n, and a. The conditions $r' \not\dagger p(\boldsymbol{x})$ and $r' \not\dagger b_1$ in Line 11 or 15, respectively, are tautological in this section.

```
function GroundRule($r, A_\mathrm{r}, A_\mathrm{n}, A_\mathrm{o}, A_\mathrm{a}, A_\mathrm{f}$)
    let $r'$ be the original version of rule $r$
    $G \leftarrow \emptyset$
    function GroundRule'($B, (b_1, \dots, b_n), \sigma$)
        if $n = 0$ then                                                    // rule instance
            if $\mathrm{head}(r)\sigma \notin A_\mathrm{f}$ then
                $G \leftarrow G \cup \{\mathrm{head}(r)\sigma \leftarrow B\}$
                $A_\mathrm{f} \leftarrow A_\mathrm{f} \cup \{\mathrm{head}(r)\sigma \mid B = \emptyset\}$
        else if $b_1 = p_x(\boldsymbol{x})$ for $x \in \{\mathrm{o}, \mathrm{n}, \mathrm{a}\}$ then       // positive literals
            foreach $\sigma' \in \mathrm{matches}(p(\boldsymbol{x}), \sigma, A_x)$ do
                $B' \leftarrow B \cup \{p(\boldsymbol{x})\sigma' \mid r' \not\vdash p(\boldsymbol{x}), p(\boldsymbol{x})\sigma' \notin A_\mathrm{f}\}$
                GroundRule'($B', (b_2, \dots, b_n), \sigma'$)
        else if $b_1 = {\sim}a$ then                                        // negative literals
            if $a\sigma \notin A_\mathrm{f}$ then
                $B' \leftarrow B \cup \{b_1\sigma \mid r' \not\vdash b_1, a \in A_\mathrm{r}$ or $a\sigma \in A_\mathrm{a}\}$
                GroundRule'($B', (b_2, \dots, b_n), \sigma$)
        else                                                               // comparison atoms
            if $b_1\sigma$ is true then
                GroundRule'($B, (b_2, \dots, b_n), \sigma$)
    GroundRule'($\emptyset, \mathrm{order}(\mathrm{body}(r)), \emptyset$)
    return $(G, A_\mathrm{f})$
```

Algorithm 3: Grounding Rules

$$\begin{tikzcd}
\mathit{edge}_\mathrm{n}(X,Y) & {\sim}\mathit{path}(X,Y) & \mathit{omit}(X,Y) \\
\mathit{edge}(a,b) \arrow[r] \arrow[d, dash] & \underline{{\sim}\mathit{path}(a,b)} \arrow[r] & \mathit{omit}(a,b) \\
\mathit{edge}(a,c) \arrow[r] \arrow[d, dash] & \underline{{\sim}\mathit{path}(a,c)} \arrow[r] & \mathit{omit}(a,c) \\
\mathit{edge}(b,c) \arrow[r] \arrow[d, dash] & \underline{{\sim}\mathit{path}(b,c)} \arrow[r] & \mathit{omit}(b,c) \\
\mathit{edge}(b,d) \arrow[r] \arrow[d, dash] & \underline{{\sim}\mathit{path}(b,d)} \arrow[r] & \mathit{omit}(b,d) \\
\mathit{edge}(c,a) \arrow[r] \arrow[d, dash] & \underline{{\sim}\mathit{path}(c,a)} \arrow[r] & \mathit{omit}(c,a) \\
\mathit{edge}(c,d) \arrow[r] \arrow[d, dash] & \underline{{\sim}\mathit{path}(c,d)} \arrow[r] & \mathit{omit}(c,d) \\
\mathit{edge}(d,a) \arrow[r] & \underline{{\sim}\mathit{path}(d,a)} \arrow[r] & \mathit{omit}(d,a)
\end{tikzcd}$$

$$A_\mathrm{r} = \{\mathit{path}(X,Y)\}$$

$$A_\mathrm{f} = \left\{\begin{array}{l} \mathit{edge}(a,b), \mathit{edge}(a,c), \\ \mathit{edge}(b,c), \mathit{edge}(b,d), \\ \mathit{edge}(c,a), \mathit{edge}(c,d), \\ \mathit{edge}(d,a), \dots \end{array}\right\}$$

$$A_\mathrm{o} = \emptyset \qquad A_\mathrm{n} = A_\mathrm{f} \qquad A_\mathrm{a} = A_\mathrm{f}$$

Fig. 4. Call to GroundRule($r_1, A_\mathrm{r}, A_\mathrm{n}, A_\mathrm{o}, A_\mathrm{a}, A_\mathrm{f}$)

the contents of the respective sets of atoms (when tackling the very first component in Figure 3). The header in Figure 4 contains the ordered body followed by the rule head. Starting with the first positive body literal $\mathit{edge}_\mathrm{n}(X,Y)$ results in eight distinct matches of $\mathit{edge}(X,Y)$ in A_n. The atoms resulting from a set of matches are connected with undirected edges in Figure 4. Note that none of the instances of $\mathit{edge}(X,Y)$ is added to

```
1  function Ground($P, A_{\mathrm{f}}$)
2      $(P_{\mathrm{g}}, A_{\mathrm{a}}) \leftarrow (\emptyset, A_{\mathrm{f}})$
3      foreach $(C, A_{\mathrm{r}})$ in Analyze($P$) do
4          $(A_{\mathrm{n}}, A_{\mathrm{o}}) \leftarrow (A_{\mathrm{a}}, \emptyset)$
5          repeat
6              $A_{\vartriangle} \leftarrow \emptyset$
7              foreach $r$ in Prepare($C, A_{\mathrm{r}}$) do
8                  $(P'_{\mathrm{g}}, A_{\mathrm{f}}) \leftarrow$ GroundRule($r, A_{\mathrm{r}}, A_{\mathrm{n}}, A_{\mathrm{o}}, A_{\mathrm{a}}, A_{\mathrm{f}}$)
9                  $(A_{\vartriangle}, P_{\mathrm{g}}) \leftarrow (A_{\vartriangle} \cup \{\mathrm{head}(r_{\mathrm{g}}) \mid r_{\mathrm{g}} \in P'_{\mathrm{g}}\}, P_{\mathrm{g}} \cup P'_{\mathrm{g}})$
10             $(A_{\mathrm{n}}, A_{\mathrm{o}}, A_{\mathrm{a}}) \leftarrow (A_{\vartriangle} \setminus A_{\mathrm{a}}, A_{\mathrm{a}}, A_{\vartriangle} \cup A_{\mathrm{a}})$
11         until $A_{\mathrm{n}} = \emptyset$ or $\{r \in C \mid \mathrm{body}^{+}(r) \cap A_{\mathrm{r}} \neq \emptyset\} = \emptyset$
12     return $P_{\mathrm{g}}$
```

Algorithm 4: Grounding Logic Programs

the (empty) body, since they are all found to be facts. Looking at the trace for the first match, we observe that GroundRule' is next called with the empty body, the singleton $(\sim path(X,Y))$, and substitution $\{X \mapsto a, Y \mapsto b\}$. Given that the atom $path(a,b)$ is not a fact and recursive, no simplifications apply, as indicated by underlining, and the instance is added to the body. The following call with body $\{\sim path(a,b)\}$, the empty tuple, and the same substitution results in the ground rule '$omit(a,b) \leftarrow \sim path(a,b)$'. Analogously, the other six matches result in further instances of r_1.

3.4 Grounding Logic Programs

The above functions are put together in Algorithm 4 for grounding entire (normal) logic programs. The function Ground takes a partition of a program into genuine rules P and atoms A_{f} stemming from facts, and returns (upon termination) a set of ground instantiated rules P_{g}. The latter is incrementally constructed by following the topological order of components determined by Analyze. Then, in turn, each adorned rule in the prepared component is instantiated via GroundRule. The loop in Lines 5-3 is executed only once whenever the component is free of recursive positive body literals, and otherwise until no new (head) atoms are forthcoming. This is accomplished by manipulating the following sets of atoms:

- A_{a} the set of all relevant atoms up to the current grounding step,
- $A_{\mathrm{n}} \subseteq A_{\mathrm{a}}$ the set of atoms atoms newly instantiated in the previous grounding step,
- $A_{\mathrm{o}} = A_{\mathrm{a}} \setminus A_{\mathrm{n}}$ the set of atoms that are not new w.r.t. the previous step,
- $A_{\vartriangle}$ the set of atoms resulting from the current grounding step, and
- $A_{\mathrm{f}} \subseteq A_{\mathrm{a}} \cup A_{\vartriangle}$ the set of atoms having a corresponding fact in P_{g}.

The set A_{a} comprises the relevant Herbrand base when the algorithm terminates.

For illustration, let us trace Ground in Figure 5 when grounding the last but one component from Figure 3. To be more precise, this deals with the prepared version of Component$_{7,1}$ containing rule r_8 only, viz. '$reach(Y) \leftarrow reach_{\mathrm{n}}(X) \wedge path_{\mathrm{a}}(X,Y)$'. The recursive nature of this rule results in three iterations of the loop in Lines 5-3.

$reach_{\mathrm{n}}(X)$		$path_{\mathrm{a}}(X,Y)$		$reach(Y)$	$A_{\mathrm{r}} = \{reach(X)\}$	**1**
$reach(a)$	$\longrightarrow$	$\underline{path(a,b)}$	$\longrightarrow$	$reach(b)$	$A_{\mathrm{f}}^1 = \{reach(a), \dots\}$	**1.1**
		$\underline{path(a,c)}$	$\longrightarrow$	$reach(c)$	$A_{\mathrm{o}}^1 = \emptyset$	
					$A_{\mathrm{n}}^1 = \left\{\begin{matrix} path(a,b), path(a,c), \\ path(b,c), path(b,d), \\ path(c,a), path(c,d), \\ path(d,a), reach(a), \dots \end{matrix}\right\}$	
					$A_{\mathrm{a}}^1 = A_{\mathrm{n}}^1$	
$\underline{reach(b)}$	$\longrightarrow$	$\underline{path(b,c)}$	$\longrightarrow$	$reach(c)$	$A_{\mathrm{f}}^2 = A_{\mathrm{f}}^1$	**1.2**
		$\underline{path(b,d)}$	$\longrightarrow$	$reach(d)$	$A_{\mathrm{o}}^2 = A_{\mathrm{a}}^1$	
$\underline{reach(c)}$	$\longrightarrow$	$\underline{path(c,a)}$	$\longrightarrow$	$reach(a)$	$A_{\mathrm{n}}^2 = \{reach(b), reach(c)\}$	
		$\underline{path(c,d)}$	$\longrightarrow$	$reach(d)$	$A_{\mathrm{a}}^2 = A_{\mathrm{a}}^1 \cup A_{\mathrm{n}}^2$	
$\underline{reach(d)}$	$\longrightarrow$	$\underline{path(d,a)}$	$\longrightarrow$	$reach(a)$	$A_{\mathrm{f}}^3 = A_{\mathrm{f}}^2$	**1.3**
					$A_{\mathrm{o}}^3 = A_{\mathrm{a}}^2$	
					$A_{\mathrm{n}}^3 = \{reach(d)\}$	
					$A_{\mathrm{a}}^3 = A_{\mathrm{a}}^2 \cup A_{\mathrm{n}}^3$	

Fig. 5. Grounding Component$_{7,1}$

Accordingly, we index the atom sets in Figure 5 to reflect their state in the respective iteration. Moreover, we only provide the parts of A_{o}, A_{n}, A_{a}, and A_{f} that are relevant to grounding r_8. Otherwise, conventions follow the ones in Figure 4.

At the first iteration, the atom $reach(a)$ (obtained from grounding Component$_{6,1}$) is used to obtain rule instances

$$reach(b) \leftarrow path(a,b)$$
$$reach(c) \leftarrow path(a,c)$$

Note that $reach(a)$ is removed from both rule bodies because it belongs to the established facts. Moreover, this iteration yields the new atoms $reach(b)$ and $reach(c)$, which are used in the next iteration to obtain the four rule instances

$$\begin{aligned} reach(c) &\leftarrow reach(b) \land path(b,c) \\ reach(d) &\leftarrow reach(b) \land path(b,d) \\ reach(a) &\leftarrow reach(c) \land path(c,a) \\ reach(d) &\leftarrow reach(c) \land path(c,d) \end{aligned}$$

Unlike above, no simplifications can be performed because no facts are involved. The iteration brings about a single new atom, $reach(d)$, which yields the rule instance

$$reach(a) \leftarrow reach(d) \land path(d,a)$$

```
1  function Rewrite($P$)
2      $Q \leftarrow \emptyset$
       // in the loop below, $\diamond \in \{\epsilon, \sim\}$ stands for the sign of the aggregate literal
3      foreach $r$ in $P$ with $a \in \mathrm{body}(r)$, $a = \diamond\alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \dots ; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec s$ do
4          let $i$ be a unique identifier
5              $\boldsymbol{x}$ be the global variables in $a$
6              $B(\boldsymbol{L}) = \bigwedge_{l \in \mathrm{body}(r) \setminus \boldsymbol{L}, l \text{ is a simple literal}} l^{\dagger}$
7          replace occurrence $a$ in $P$ with $\diamond aggr_i(\boldsymbol{x})$
           $Q \leftarrow Q \cup \{accu_i(\boldsymbol{x}, neutral) \leftarrow \widehat{\alpha}(\emptyset) \prec s \wedge B(\emptyset)\}$
8                  $\cup \{accu_i(\boldsymbol{x}, tuple(\boldsymbol{t}_j)) \leftarrow \bigwedge_{l \in \boldsymbol{L}_j} l \wedge B(\boldsymbol{L}_j) \mid 1 \leq j \leq n\}$
                   $\cup \{aggr_i(\boldsymbol{x}) \leftarrow accu_i(\boldsymbol{x}, \_) \wedge \bot\}$
9      return $P \cup Q$
```

Algorithm 5: Rewrite Logic Programs

This iteration produces no new atoms and ends the instantiation of Component$_{7,1}$.

The other components are grounded analogously but within a single iteration due to their lack of recursive positive body literals. This is enforced by the second stop criterion in Line 3 of Algorithm 4.

4 Grounding Recursive Aggregates

Having laid the foundations of grounding normal logic programs, we now continue with the treatment of recursive aggregates. The idea is to translate aggregate atoms into normal logic programs, roughly one rule per aggregate element, and then to reuse the basic grounding machinery as much as possible. In addition, some aggregate-specific propagation takes place. At the end, the resulting aggregate instances are re-assembled from the corresponding rules.

4.1 Rewriting Logic Programs with Aggregates

The function Rewrite given in Algorithm 5 takes as input a logic program (possibly with recursive aggregates) and rewrites it into a normal logic program with additional predicates capturing aggregates and aggregate elements.[14]

Each aggregate occurrence is replaced with an atom of form $aggr_i(\boldsymbol{x})$ in Line 7, where i is a unique identifier associated with the aggregate occurrence and $\boldsymbol{x}$ are the global variables occurring in the aggregate. The idea is that each atom over predicate $aggr_i$ in the grounding of the rewriting corresponds to a ground aggregate, which is substituted for the atom in the final grounding.

To represent aggregate elements like $\boldsymbol{t}_j : \boldsymbol{L}_j$, auxiliary rules defining atoms of form $accu_i(\boldsymbol{x}, t)$ are added in Line 8, where $\boldsymbol{x}$ are the global variables as above and t is the tuple that is aggregated (or the special constant $neutral$). Here, the idea is to inspect

[14] We assume that predicates $aggr_i$ and $accu_i$ are not used elsewhere in the program.

$$\begin{aligned}
\mathit{controls}(X,Y) &\leftarrow \mathit{aggr}_1(X,Y) \wedge B\\
\mathit{accu}_1(X,Y,\mathit{neutral}) &\leftarrow \overline{0} > \overline{50} \wedge B^\dagger\\
\mathit{accu}_1(X,Y,\mathit{tuple}(S)) &\leftarrow \mathit{owns}(X,Y,S) \wedge B^\dagger\\
\mathit{accu}_1(X,Y,\mathit{tuple}(S,Z)) &\leftarrow \mathit{controls}(X,Z) \wedge \mathit{owns}(Z,Y,S) \wedge B^\dagger\\
\mathit{aggr}_1(X,Y) &\leftarrow \mathit{accu}_1(X,Y,\_) \wedge \bot
\end{aligned}$$

where $B = \mathit{company}(X) \wedge \mathit{company}(Y) \wedge X \neq Y$

Table 5. Rewritten Company Controls Encoding

the grounding for rules with atoms over accu_i in the head. If enough such atoms are accumulated to satisfy an aggregate, then corresponding atoms over aggr_i are added to the Herbrand base to further ground the program.

The first rule in Line 8 handles the special case that the aggregate is satisfied for an empty set of tuples (e.g., anti-monotone aggregates like $\mathit{sum}^+\{\boldsymbol{t} : \boldsymbol{L}\} \leq s$). Its body contains a comparison literal that checks whether the empty aggregate is satisfied. Furthermore, we have to make sure that the rule is safe so that it can be instantiated. For this purpose, all simple literals of the rule in which the aggregate i occurs are added to the rule body (via function B). Hence, if the original rule is safe, the auxiliary rule is also safe because global variables are bound by positive symbolic literals only. Furthermore, literals responsible for binding global variables are marked (via †).

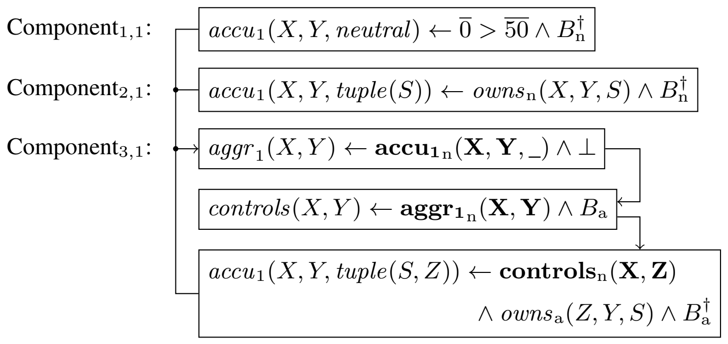
The second set of rules in Line 8 is in charge of accumulating tuples of aggregate elements. The rule body contains the literals of the condition of the aggregate element as well as marked literals necessary for ensuring the rule's safety. Remember that the resulting ground rules represent ground aggregate atoms, where the marking is used to distinguish literals not belonging to the conditions of reconstituted aggregate elements.

Finally, one last rule is added in Line 8 for ensuring that the dependencies induced by the aggregate are kept intact. Since this rule contains $\bot$, it never produces instances though.

The result of rewriting the company controls encoding from Table 1 is given in Table 5. The global variables in the (single) aggregate are X and Y, which occur first in all atoms over aggr_1/2 and accu_1/3. Since the empty aggregate is not satisfied, the rule accumulating the $\mathit{neutral}$ tuple never produces any instances (and could in principle be dropped from the rewriting).

4.2 Analyzing and Preparing Logic Programs with Aggregates

Figure 6 captures the result of function `Analyze` with `Prepare` called for each component of the rewritten company controls encoding in Table 5. The rules in Component$_{1,1}$ and Component$_{2,1}$ depend on facts only, and thus both induce a singleton component. Component$_{3,1}$ contains the remaining rules. The aggregate of the company controls encoding is recursive in this component in view of the atom $\mathit{controls}(X,Z)$ in its second aggregate element. Note that not all aggregate elements are involved in this recursion, given that direct shares are accumulated via the rule in Component$_{2,1}$.


Component$_{1,1}$:
$accu_1(X, Y, neutral) \leftarrow \overline{0} > \overline{50} \land B_n^\dagger$
Component$_{2,1}$:
$accu_1(X, Y, tuple(S)) \leftarrow owns_n(X, Y, S) \land B_n^\dagger$
Component$_{3,1}$:
$aggr_1(X, Y) \leftarrow \mathbf{accu_1}_n(\mathbf{X}, \mathbf{Y}, \_) \land \bot$
$controls(X, Y) \leftarrow \mathbf{aggr_1}_n(\mathbf{X}, \mathbf{Y}) \land B_a$
$accu_1(X, Y, tuple(S, Z)) \leftarrow \mathbf{controls}_n(\mathbf{X}, \mathbf{Z}) \land owns_a(Z, Y, S) \land B_a^\dagger$

where $B_x = company_x(X) \land company_x(Y) \land X \neq Y$

Fig. 6. Dependency Graph for Company Controls Program

```
1  function Propagate($I, r, A_a, A_f$)
2      $A_\Delta \leftarrow \emptyset$
3      foreach $i, \boldsymbol{g}$ where $i \in I$ and $accu_i(\boldsymbol{g}, t) \in A_a$ do
4          let $T_f = \{\boldsymbol{t} \mid accu_i(\boldsymbol{g}, tuple(\boldsymbol{t})) \in A_f, \boldsymbol{t}$ is relevant for $\alpha_i\}$
5              $T_a = \{\boldsymbol{t} \mid accu_i(\boldsymbol{g}, tuple(\boldsymbol{t})) \in A_a, \boldsymbol{t}$ is relevant for $\alpha_i\}$
6          if exists $T_f \subseteq T \subseteq T_a$ where $\widehat{\alpha}_i(T) \prec_i (s_i)_{\boldsymbol{g}}^{\boldsymbol{x}_i}$ is true then
7              if ($aggregate\ i\ is\ monotone$ and $\widehat{\alpha}_i(T_f) \prec_i (s_i)_{\boldsymbol{g}}^{\boldsymbol{x}_i}$)
8                or (not $r$ and $T_a \setminus T_f = \emptyset$) then
9                  $A_f \leftarrow A_f \cup \{aggr_i(\boldsymbol{g})\}$
10             $A_\Delta \leftarrow A_\Delta \cup \{aggr_i(\boldsymbol{g})\}$
11     return $(A_\Delta, A_f)$
```

Algorithm 6: Propagation of Aggregates

4.3 Propagating Aggregates

The function `Propagate` inspects the partial grounding of an aggregate instance in view of its grounded aggregate elements. To this end, it checks atoms over predicate $accu_i$ obtained during grounding. The loop in Lines 3-3 iterates over the given aggregate indices I and tuples of global variables stored in atoms over predicate $accu_i$ appearing among the atoms in A_a. Whenever there are enough tuples captured by such atoms to satisfy the corresponding aggregate, `Propagate` adds atoms over predicate $aggr_i$ to A_Δ for further instantiation. While Line 4 collects tuples that are necessarily accumulated by the aggregate function, Line 5 gathers tuples whose conditions can possibly hold. Also note that the relevance check skips tuples that do not change the result of an aggregate function.[15] For sum aggregates, this amounts to excluding zero-weight tuples by stipulating $weight(\boldsymbol{t}) \neq 0$. Given these sets of tuples, Line 6 checks whether the aggregate can be satisfied using the tuples accumulated so far. For sum ag-

[15] For non-recursive aggregates, where the flag r is false, Line 8 checks whether $T_a \setminus T_f$ is empty. This is why only relevant tuples are gathered.

```
1 function Assemble($P_\mathrm{g}$)
2     foreach $\mathit{aggr}_i(\boldsymbol{g})$ occurring in $P_\mathrm{g}$ do
          // below, $\mathrm{body}(r)$ is assumed to convert to a tuple of literals
3         let $E = \{\boldsymbol{t} : \mathrm{body}(r) \mid r \in P_\mathrm{g}, \mathrm{head}(r) = \mathit{accu}_i(\boldsymbol{g}, \mathit{tuple}(\boldsymbol{t}))\}$
4         replace all occurrences of $\mathit{aggr}_i(\boldsymbol{g})$ in $P_\mathrm{g}$ with $\alpha_i(E) \prec_i (s_i)^{\boldsymbol{x}_i}_{\boldsymbol{g}}$
5     remove all rules with atoms over $\mathit{accu}_i$ in the head from $P_\mathrm{g}$
6     return $P_\mathrm{g}$
```

Algorithm 7: Assembling Aggregates

gregates, this can be tested by adding up the weights of factual tuples and, on the one hand, the negative weights to obtain a minimum, min, or the positive weights to obtain a maximum, max. Then, depending on the relation, the aggregate is satisfiable

– if $\overline{max} \prec (s_i)^{\boldsymbol{x}_i}_{\boldsymbol{g}}$ is true for $\prec \in \{\geq, >\}$,
– if $\overline{min} \prec (s_i)^{\boldsymbol{x}_i}_{\boldsymbol{g}}$ is true for $\prec \in \{\leq, <\}$, or
– if $\overline{min} \prec (s_i)^{\boldsymbol{x}_i}_{\boldsymbol{g}}$ or $\overline{max} \prec (s_i)^{\boldsymbol{x}_i}_{\boldsymbol{g}}$ is true for $\prec \in \{\neq\}$.

If the test in Line 6 succeeds, the ground aggregate atom is added to the new atoms in Line 10. In addition, given a non-recursive or monotone aggregate, the corresponding ground aggregate atom is added to A_f whenever the aggregate is found to be true. At this point, a non-recursive aggregate is true, if all its elements are facts (Line 8), and a monotone aggregate is true, if enough facts to satisfy the aggregate have been accumulated (Line 7). Finally, the sets of new and factual atoms are returned in Line 11.

4.4 Assembling Aggregates

After `Rewrite` has decomposed aggregate atoms into normal program rules, `Assemble` given in Algorithm 7 reconstructs their grounded counterparts from the rewritten ground program. That is, all occurrences of atoms of form $\mathit{aggr}_i(\boldsymbol{g})$ are replaced by their corresponding aggregates. In doing so, the aggregate elements are reconstructed from rules with head atoms $\mathit{accu}_i(\boldsymbol{g}, \mathit{tuple}(\boldsymbol{t}))$ in Line 3, where an element consists of the term tuple $\boldsymbol{t}$ along with the condition expressed by the rule body.[16] The actual replacement takes place in Line 4, followed by the deletion of obsolete rules in Line 5. Finally, the reconstructed ground program is returned in Line 6.

4.5 Grounding Logic Programs with Aggregates

Algorithm 8 extends the `Ground` function in Algorithm 4 to logic program with aggregates. To this end, the extended `Ground` function uses algorithms `Rewrite`, `Propagate`, and `Assemble` from the previous subsections. The changes in the algorithm are highlighted with a gray background, while other parts are left untouched.

The first change is in Line 3, where function `Rewrite` is called to turn the logic program P into a normal logic program before calling `Analyze`. Then, Lines 4 and 5

[16] Recall that marked literals, added for safety, are stripped off by `GroundRule` in Algorithm 3.

```
1  function Ground($P, A_\mathrm{f}$)
2    $(P_\mathrm{g}, A_\mathrm{a}) \leftarrow (\emptyset, A_\mathrm{f})$
3    foreach $(C, A_\mathrm{r})$ in Analyze(Rewrite($P$)) do
4      let $I = \{i \mid aggr_i$ occurs in a rule head in $C\}$
5        $I_\mathrm{r} = \{i \mid r \in C, \mathrm{head}(r) = accu_i(\boldsymbol{x}, t), a \in \mathrm{body}^+(r) \cap A_\mathrm{r}, r \not\dagger a\}$
6      $(A_\mathrm{n}, A_\mathrm{o}) \leftarrow (A_\mathrm{a}, \emptyset)$
7      repeat
8        $A_\Delta \leftarrow \emptyset$
9        foreach $r$ in Prepare($C, A_\mathrm{r}$) do
10         $(P'_\mathrm{g}, A_\mathrm{f}) \leftarrow$ GroundRule($r, A_\mathrm{r}, A_\mathrm{n}, A_\mathrm{o}, A_\mathrm{a}, A_\mathrm{f}$)
11         $(A_\Delta, P_\mathrm{g}) \leftarrow (A_\Delta \cup \{\mathrm{head}(r_\mathrm{g}) \mid r_\mathrm{g} \in P'_\mathrm{g}\}, P_\mathrm{g} \cup P'_\mathrm{g})$
12       if $A_\Delta \subseteq A_\mathrm{a}$ then
13         $(A_\Delta, A_\mathrm{f}) \leftarrow$ Propagate($I \setminus I_\mathrm{r}$, false, $A_\mathrm{a}, A_\mathrm{f}$)
14       if $A_\Delta \subseteq A_\mathrm{a}$ then
15         $(A_\Delta, A_\mathrm{f}) \leftarrow$ Propagate($I \cap I_\mathrm{r}$, true, $A_\mathrm{a}, A_\mathrm{f}$)
16       $(A_\mathrm{n}, A_\mathrm{o}, A_\mathrm{a}) \leftarrow (A_\Delta \setminus A_\mathrm{a}, A_\mathrm{a}, A_\Delta \cup A_\mathrm{a})$
17     until $A_\mathrm{n} = \emptyset$ or $\{r \in C \mid \mathrm{body}^+(r) \cap A_\mathrm{r} \neq \emptyset\} = \emptyset$
18   return Assemble($P_\mathrm{g}$)
```

Algorithm 8: Grounding Logic Programs with Aggregates

are added, just before the loop in charge of grounding each component. Here, all aggregate indices that have to be propagated during the instantiation of a component are collected. First, all indices for which a rule with $aggr_i$ in the head appears in the component are gathered in I. Remember that these rules do not contribute instances because $\perp$ belongs to their bodies (cf. third row in Line 8 of Algorithm 5). Instead, `Propagate` is adding atoms over $aggr_i$ to A_Δ. Second, in Line 5, aggregate indices associated with recursive aggregates are collected, where the (positive) recursion involves some aggregate element indicated by a rule with head atom $accu_i(\boldsymbol{x}, t)$, yet without considering the auxiliary body part marked with †.

The collected indices are in Lines 12-15 used to propagate the corresponding aggregates in the current component. Propagation of aggregates is triggered whenever no more new atoms are obtained in the grounding loop in Lines 9-9. First, non-recursive aggregates $I \setminus I_\mathrm{r}$ are instantiated. At this point, if there is at least one ground atom of form $accu_i(\boldsymbol{g}, t) \in A_\mathrm{a}$, all aggregate elements of the corresponding aggregate (uniquely determined by the terms $\boldsymbol{g}$ for global variables) have been gathered. The aggregate can thus be propagated, and function `Propagate` can apply additional simplifications (cf. Line 8 in Algorithm 6). Afterwards, recursive aggregates $I \cap I_\mathrm{r}$ are propagated in Line 15. In this case, we cannot assume that all aggregate elements have already been accumulated, and propagation can thus not use all of the simplifications applicable to non-recursive aggregates, where the distinction is implemented by setting the second argument of `Propagate` to **true**. Finally, in Line 18, aggregates are reconstructed from the intermediate grounding by calling function `Assemble`.

$\overline{0} > \overline{50} \quad c^{\dagger}_{\mathrm{n}}(X) \quad c^{\dagger}_{\mathrm{n}}(Y) \quad X \neq Y \quad a_1(X,Y,n)$	**1**
$\times$	**1.1**
$o_{\mathrm{n}}(X,Y,S) \quad c^{\dagger}_{\mathrm{n}}(X) \quad c^{\dagger}_{\mathrm{n}}(Y) \quad X \neq Y \quad a_1(X,Y,t(S))$	**2**
$o(c_1,c_2,\overline{60}) \longrightarrow c(c_1) \longrightarrow c(c_2) \longrightarrow c_1 \neq c_2 \longrightarrow a_1(c_1,c_2,t(\overline{60}))$	**2.1**
$o(c_1,c_3,\overline{20}) \longrightarrow c(c_1) \longrightarrow c(c_3) \longrightarrow c_1 \neq c_3 \longrightarrow a_1(c_1,c_3,t(\overline{20}))$	
$o(c_2,c_3,\overline{35}) \longrightarrow c(c_2) \longrightarrow c(c_3) \longrightarrow c_2 \neq c_3 \longrightarrow a_1(c_2,c_3,t(\overline{35}))$	
$o(c_3,c_4,\overline{51}) \longrightarrow c(c_3) \longrightarrow c(c_4) \longrightarrow c_3 \neq c_4 \longrightarrow a_1(c_3,c_4,t(\overline{51}))$	
$\perp \quad {a_1}_{\mathrm{n}}(X,Y,\_) \quad g_1(X,Y)$	**3**
${g_1}_{\mathrm{n}}(X,Y) \quad c^{\dagger}_{\mathrm{a}}(X) \quad c^{\dagger}_{\mathrm{a}}(Y) \quad X \neq Y \quad r(X,Y)$	
$r_{\mathrm{n}}(X,Z) \quad o_{\mathrm{a}}(Z,Y,S) \quad c^{\dagger}_{\mathrm{a}}(X) \quad c^{\dagger}_{\mathrm{a}}(Y) \quad X \neq Y \quad a_1(X,Y,t(S,Z))$	
`Propagate`: $\{g_1(c_1,c_2), g_1(c_3,c_4)\}$	**3.1**
$\times$	**3.2**
$g_1(c_1,c_2) \longrightarrow c(c_1) \longrightarrow c(c_2) \longrightarrow c_1 \neq c_2 \longrightarrow r(c_1,c_2)$	
$g_1(c_3,c_4) \longrightarrow c(c_3) \longrightarrow c(c_4) \longrightarrow c_3 \neq c_4 \longrightarrow r(c_3,c_4)$	
$\times$	
$\times$	**3.3**
$\times$	
$r(c_1,c_2) \longrightarrow o(c_2,c_3,\overline{35}) \rightarrow c(c_1) \longrightarrow c(c_3) \longrightarrow c_1 \neq c_3 \rightarrow a_1(c_1,c_3,t(\overline{35},c_2))$	
$r(c_3,c_4) \longrightarrow \times$	
`Propagate`: $\{g_1(c_1,c_3)\}$	**3.4**
$\times$	**3.5**
$g_1(c_1,c_3) \longrightarrow c(c_1) \longrightarrow c(c_3) \longrightarrow c_1 \neq c_3 \longrightarrow r(c_1,c_3)$	
$\times$	
$\times$	**3.6**
$\times$	
$r(c_1,c_3) \longrightarrow o(c_3,c_4,\overline{51}) \rightarrow c(c_1) \longrightarrow c(c_4) \longrightarrow c_1 \neq c_4 \rightarrow a_1(c_1,c_4,t(\overline{51},c_3))$	
`Propagate`: $\{g_1(c_1,c_4)\}$	**3.7**
$\times$	**3.8**
$g_1(c_1,c_4) \longrightarrow c(c_1) \longrightarrow c(c_4) \longrightarrow c_1 \neq c_4 \longrightarrow r(c_1,c_4)$	
$\times$	
$\times$	**3.9**
$\times$	
$r(c_1,c_4) \longrightarrow \times$	

$a_1 = \mathit{accu}_1,\ g_1 = \mathit{aggr}_1,\ r = \mathit{controls},\ c = \mathit{company},\ o = \mathit{owns},\ n = \mathit{neutral},\ t = \mathit{tuple}$

Fig. 7. Grounding Company Controls

Figure 7 traces the whole grounding process of the rewritten company controls encoding given in Figure 6. The grounding of each component is separated by a horizontal double line, and the instantiation of a component is shown analogously to Figure 5. Due to lack of space, we refrain from giving sets A_r, A_n, A_o, and A_a. For each component, indicated by its number on the right, the contained rules are listed first, followed by grounding iterations for the component in focus. Each iteration is separated by a solid line and indexed with an iteration number on the right, where (instances of) the three rules in Component$_{3,1}$ are separated by dashed lines. The symbol $\times$ indicates that a body literal does not match, which corresponds to the case that the `GroundRule`' function in Algorithm 3 backtracks.

The grounding of Component$_{1,1}$ produces no rule instances because the comparison atom $\overline{0} > \overline{50}$ is false. By putting this literal first in the safe body order, the remaining rule body can be completely ignored. Next, in the grounding of Component$_{2,1}$, direct shares given by facts over $owns$/3 are accumulated, where the obtained atoms over $accu_1$/3 are classified as facts as well. Then, we trace the grounding of Component$_{3,1}$. In the first iteration, none of the rules produces instances because there are no atoms over *controls*/2 and $aggr_1$/2 yet. Hence, aggregate propagation is triggered, resulting in aggregate atoms $aggr_1(c_1, c_2)$ and $aggr_1(c_3, c_4)$, for which enough shares have been accumulated upon grounding Component$_{2,1}$. Note that, since the aggregate is monotone, both atoms are established as facts. In the second iteration, the newly obtained aggregate atoms are used to instantiate the second rule of the component, leading to new atoms over *controls*/2. Observe that, by putting $aggr_1(X, Y)$ first in the safe body order, `GroundRule` can instantiate the rule without backtracking. In the third iteration, the newly obtained atoms over *controls*/2 yield $accu_1(c_1, c_3, t(\overline{35}, c_2))$ via an instance of the third rule of the component, which in turn leads to the aggregate atom $aggr_1(c_1, c_3)$. The following iterations proceed in a similar fashion until no new tuples are accumulated and the grounding loop terminates. Confined to the original predicate *controls*/2, the instantiation generates four atoms, $controls(c_1, c_2)$, $controls(c_3, c_4)$, $controls(c_1, c_3)$, and $controls(c_1, c_4)$, all of which are produced as facts.

Note that the utilized safe body order affects the amount of backtracking in rule instantiation [11]. One particular strategy used in `gringo` is to prefer recursive atoms with subscript n when ordering a rule body. As seen in the grounding of Component$_{3,1}$ above, this helps to avoid backtracking upon generating new rule instances. Furthermore, for the company controls encoding, simplifications ensure that the program is evaluated to facts. In general, this is guaranteed for programs with stratified negation and monotone aggregates only [12].

5 Discussion

We presented an algorithmic framework for grounding logic programs based on semi-naive database evaluation techniques. Our framework, which is implemented in `gringo` series 4, constitutes the first approach capturing full-fledged aggregates under Ferraris' semantics [13, 6]. While semi-naive evaluation techniques trace back to the field of database systems [3, 4], their introduction to grounding in ASP was pioneered by the `dlv` system [14], laying out basic semi-naive grounding algorithms (cf. [1]) similar to those in Section 3. Given this proximity, our grounding techniques for handling recursive aggregates could be adopted within `dlv`, which is so far restricted to stratified aggregates. Other grounding approaches are pursued in `gidl` [15], `lparse` [16], and earlier versions of `gringo` [17, 18, 2]. The latter two also support recursive (convex) aggregates but are limited by the necessity to bind non-global variables by domain predicates, given that programs have to be ω- [16] or λ-restricted [17], respectively. Unlike this, our approach merely relies on the safety condition and no further restriction is imposed on the input language.

Regarding the implementation, our approach aims at reusing existing grounding techniques for (normal) logic programs. To this end, programs with aggregates are rewritten, and conventional semi-naive evaluation is extended with a propagation step for aggregates. Eventually, ground aggregates are reconstructed from the obtained rule instances in a post-processing step. While the present paper considered sum and sum^+ aggregates only, our approach is applicable to any aggregate function. In fact, $count$, min, and max aggregates are also supported in `gringo` series 4, and it is easily amenable to further aggregates (by extending the `Propagate` function).

Acknowledgments This work was funded by AoF (grant 251170), DFG (grants SCHA 550/8 and 550/9), as well as DAAD and AoF (joint project 57071677 / 279121).

References

1. Faber, W., Leone, N., Perri, S.: The intelligent grounder of DLV. [19] 247–264
2. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. In Delgrande, J., Faber, W., eds.: Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11). Springer-Verlag (2011) 345–351
3. Ullman, J.: Principles of Database and Knowledge-Base Systems. Computer Science Press (1988)
4. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
5. Mumick, I., Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates. In McLeod, D., Sacks-Davis, R., Schek, H., eds.: Proceedings of the Sixteenth International Conference on Very Large Data Bases (VLDB'90). Morgan Kaufmann Publishers (1990) 264–277
6. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. Theory and Practice of Logic Programming **15**(4-5) (2015) 449–463
7. Truszczyński, M.: Connecting first-order ASP and the logic FO(ID) through reducts. [19] 543–559
8. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 241–273
9. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In Apt, K., Marek, V., Truszczyński, M., Warren, D., eds.: The Logic Programming Paradigm: A 25-Year Perspective. Springer-Verlag (1999) 375–398
10. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool Publishers (2012)
11. Leone, N., Perri, S., Scarcello, F.: Improving ASP instantiators by join-ordering methods. [20] 280–294
12. Alviano, M., Calimeri, F., Faber, W., Leone, N., Perri, S.: Unfounded sets and well-founded semantics of answer set programs with aggregates. Journal of Artificial Intelligence Research **42** (2011) 487–527

13. Ferraris, P.: Logic programs with propositional connectives and aggregates. ACM Transactions on Computational Logic **12**(4) (2011) 25:1–25:44
14. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
15. Wittocx, J., Mariën, M., Denecker, M.: Grounding FO and FO(ID) with bounds. Journal of Artificial Intelligence Research **38** (2010) 223–269
16. Syrjänen, T.: Omega-restricted logic programs. [20] 267–279
17. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). Springer-Verlag (2007) 266–271
18. Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder gringo. In Erdem, E., Lin, F., Schaub, T., eds.: Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09). Springer-Verlag (2009) 502–508
19. Erdem, E., Lee, J., Lierler, Y., Pearce, D., eds.: Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz. Springer-Verlag (2012)
20. Eiter, T., Faber, W., Truszczyński, M., eds.: Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01). Springer-Verlag (2001)