TPLP **23** (6): 1138–1197, 2023.

doi:10.1017/S1471068422000308 First published online 25 July 2022

On the Foundations of Grounding in Answer Set Programming

ROLAND KAMINSKI and TORSTEN SCHAUB
University of Potsdam, Potsdam, Germany
(*e-mails:* kaminski@cs.uni-potsdam.de, torsten@cs.uni-potsdam.de)

submitted 17 August 2021; revised 16 May 2022; accepted 9 June 2022

Abstract

We provide a comprehensive elaboration of the theoretical foundations of variable instantiation, or grounding, in Answer Set Programming (ASP). Building on the semantics of ASP's modeling language, we introduce a formal characterization of grounding algorithms in terms of (fixed point) operators. A major role is played by dedicated well-founded operators whose associated models provide semantic guidance for delineating the result of grounding along with on-the-fly simplifications. We address an expressive class of logic programs that incorporates recursive aggregates and thus amounts to the scope of existing ASP modeling languages. This is accompanied with a plain algorithmic framework detailing the grounding of recursive aggregates. The given algorithms correspond essentially to the ones used in the ASP grounder *gringo*.

KEYWORDS: answer set programming, grounding theory, grounding algorithms

1 Introduction

Answer Set Programming (ASP; Lifschitz 2002) allows us to address knowledge-intense search and optimization problems in a greatly declarative way due to its integrated modeling, grounding, and solving workflow (Gebser and Schaub 2016; Kaufmann *et al.* 2016). Problems are modeled in a rule-based logical language featuring variables, function symbols, recursion, and aggregates, among others. Moreover, the underlying semantics allows us to express defaults and reachability in an easy way. A corresponding logic program is then turned into a propositional format by systematically replacing all variables by variable-free terms. This process is called *grounding*. Finally, the actual ASP solver takes the resulting propositional version of the original program and computes its answer sets.

Given that both grounding and solving constitute the computational cornerstones of ASP, it is surprising that the importance of grounding has somehow been eclipsed by that of solving. This is nicely reflected by the unbalanced number of implementations. With *lparse* (Syrjänen 2001b), (the grounder in) *dlv* (Faber *et al.* 2012), and *gringo* (Gebser *et al.* 2011), three grounder implementations face dozens of solver implementations, among them *smodels* (Simons *et al.* 2002), (the solver in) *dlv* (Leone *et al.* 2006), *assat* (Lin and Zhao 2004), *cmodels* (Giunchiglia *et al.* 2006), *clasp* (Gebser *et al.* 2012), *wasp* (Alviano *et al.* 2015) just to name the major ones. What caused this imbalance?

One reason may consist in the high expressiveness of ASP's modeling language and the resulting algorithmic intricacy (Gebser *et al.* 2011). Another may lie in the popular viewpoint that grounding amounts to database materialization, and thus that most fundamental research questions have been settled. And finally the semantic foundations of full-featured ASP modeling languages have been established only recently (Harrison *et al.* 2014; Gebser *et al.* 2015a), revealing the semantic gap to the just mentioned idealized understanding of grounding. In view of this, research on grounding focused on algorithm and system design (Faber *et al.* 2012; Gebser *et al.* 2011) and the characterization of language fragments guaranteeing finite propositional representations (Syrjänen 2001b; Gebser *et al.* 2007; Lierler and Lifschitz 2009; Calimeri *et al.* 2008).

As a consequence, the theoretical foundations of grounding are much less explored than those of solving. While there are several alternative ways to characterize the answer sets of a logic program (Lifschitz 2008), and thus the behavior of a solver, we still lack indepth formal characterizations of the input–output behavior of ASP grounders. Although we can describe the resulting propositional program up to semantic equivalence, we have no formal means to delineate the actual set of rules.

To this end, grounding involves some challenging intricacies. First of all, the entire set of systematically instantiated rules is infinite in the worst – yet not uncommon – case. For a simple example, consider the program:

$$\begin{gathered} p(a) \\ p(X) \leftarrow p(f(X)). \end{gathered}$$

This program induces an infinite set of variable-free terms, viz. a, $f(a)$, $f(f(a)), \ldots$, that leads to an infinite propositional program by systematically replacing variable X by all these terms in the second rule, viz.

$$p(a), \quad p(a) \leftarrow p(f(a)), \quad p(f(a)) \leftarrow p(f(f(a))), \quad p(f(f(a))) \leftarrow p(f(f(f(a)))), \quad \ldots$$

On the other hand, modern grounders only produce the fact $p(a)$ and no instances of the second rule, which is semantically equivalent to the infinite program. As well, ASP's modeling language comprises (possibly recursive) aggregates, whose systematic grounding may be infinite in itself. To illustrate this, let us extend the above program with the rule

$$q \leftarrow \#\mathit{count}\{X : p(X)\} = 1 \tag{1}$$

deriving q when the number of satisfied instances of p is one. Analogous to above, the systematic instantiation of the aggregate's element results in an infinite set, viz.

$$\{a : p(a),\ f(a) : p(f(a)),\ f(f(a)) : p(f(f(a))),\ \ldots\}.$$

Again, a grounder is able to reduce the rule in Equation (1) to the fact q since only $p(a)$ is obtained in our example. That is, it detects that the set amounts to the singleton $\{a : p(a)\}$, which satisfies the aggregate. After removing the rule's (satisfied) antecedent, it produces the fact q. In fact, a solver expects a finite set of propositional rules including aggregates over finitely many objects only. Hence, in practice, the characterization of the grounding result boils down to identifying a finite yet semantically equivalent set of rules (whenever possible). Finally, in practice, grounding involves simplifications whose application depends on the ordering of rules in the input. In fact, shuffling a list of propositional rules only affects the order in which a solver enumerates answer sets, whereas

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

shuffling a logic program before grounding may lead to different though semantically equivalent sets of rules. To see this, consider the program:

$$\begin{array}{llll} p(X) \leftarrow \neg q(X) \wedge u(X) & u(1) & u(2) & \\ q(X) \leftarrow \neg p(X) \wedge v(X) & & v(2) & v(3). \end{array}$$

This program has two answer sets; both contain $p(1)$ and $q(3)$, while one contains $q(2)$ and the other $p(2)$. Systematically grounding the program yields the obvious four rules. However, depending upon the order, in which the rules are passed to a grounder, it already produces either the fact $p(1)$ or $q(3)$ via simplification. Clearly, all three programs are distinct but semantically equivalent in sharing the above two answer sets.

Our elaboration of the foundations of ASP grounding rests upon the semantics of ASP's modeling language (Harrison *et al.* 2014; Gebser *et al.* 2015a), which captures the two aforementioned sources of infinity by associating non-ground logic programs with infinitary propositional formulas (Truszczyński 2012). Our main result shows that the stable models of a non-ground input program coincide with the ones of the ground output program returned by our grounding algorithm upon termination. In formal terms, this means that the stable models of the infinitary formula associated with the input program coincide with the ones of the resulting ground program. Clearly, the resulting program must be finite and consist of finitary subformulas only. A major part of our work is thus dedicated to equivalence preserving transformations between ground programs. In more detail, we introduce a formal characterization of grounding algorithms in terms of (fixed point) operators. A major role is played by specific well-founded operators whose associated models provide semantic guidance for delineating the result of grounding. More precisely, we show how to obtain a finitary propositional formula capturing a logic program whenever the corresponding well-founded model is finite, and notably how this transfers to building a finite propositional program from an input program during grounding. The two key instruments accomplishing this are dedicated forms of *program simplification* and *aggregate translation*, each addressing one of the two sources of infinity in the above example. In practice, however, all these concepts are subject to *approximation*, which leads to the order-dependence observed in the last example.

We address an expressive class of logic programs that incorporates recursive aggregates and thus amounts to the scope of existing ASP modeling languages (Gebser *et al.* 2015b). This is accompanied with an algorithmic framework detailing the grounding of recursive aggregates. The given grounding algorithms correspond essentially to the ones used in the ASP grounder *gringo* (Gebser *et al.* 2011). In this way, our framework provides a formal characterization of one of the most widespread grounding systems. In fact, modern grounders like (the one in) *dlv* (Faber *et al.* 2012) or *gringo* (Gebser *et al.* 2011) are based on database evaluation techniques (Ullman 1988; Abiteboul *et al.* 1995). The instantiation of a program is seen as an iterative bottom-up process starting from the program's facts while being guided by the accumulation of variable-free atoms possibly derivable from the rules seen so far. During this process, a ground rule is produced if its positive body atoms belong to the accumulated atoms, in which case its head atom is added as well. This process is repeated until no further such atoms can be added. From an algorithmic perspective, we show how a grounding framework (relying upon database evaluation techniques) can be extended to incorporate recursive aggregates.

Our paper is organized as follows.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Section 2 lays the basic foundations of our approach. We start in Section 2.1 by recalling definitions of (monotonic) operators on lattices; they constitute the basic building blocks of our characterization of grounding algorithms. We then review infinitary formulas along with their stable and well-founded semantics in Sections 2.2, 2.3 and 2.4, respectively. In this context, we explore several operators and define a class of infinitary logic programs that allows us to capture full-featured ASP languages with (recursive) aggregates. Interestingly, we have to resort to concepts borrowed from ID-logic (Bruynooghe *et al.* 2016; Truszczyński 2012) to obtain monotonic operators that are indispensable for capturing iterative algorithms. Notably, the ID-well-founded model can be used for approximating regular stable models. Finally, we define in Section 2.5 our concept of program simplification and elaborate upon its semantic properties. The importance of program simplification can be read off two salient properties. First, it results in a finite program whenever the interpretation used for simplification is finite. And second, it preserves all stable models when simplified with the ID-well-founded model of the program.

Section 3 is dedicated to the formal foundations of component-wise grounding. As mentioned, each rule is instantiated in the context of all atoms being possibly derivable up to this point. In addition, grounding has to take subsequent atom definitions into account. To this end, we extend well-known operators and resulting semantic concepts with contextual information, usually captured by two- and four-valued interpretations, respectively, and elaborate upon their formal properties that are relevant to grounding. In turn, we generalize the contextual operators and semantic concepts to sequences of programs in order to reflect component-wise grounding. The major emerging concept is essentially a well-founded model for program sequences that takes backward and forward contextual information into account. We can then iteratively compute this model to approximate the well-founded model of the entire program. This model-theoretic concept can be used for governing an ideal grounding process.

Section 4 turns to logic programs with variables and aggregates. We align the semantics of such aggregate programs with the one of Ferraris (2011) but consider infinitary formulas (Harrison *et al.* 2014). In view of grounding aggregates, however, we introduce an alternative translation of aggregates that is strongly equivalent to that of Ferraris but provides more precise well-founded models. In turn, we refine this translation to be bound by an interpretation so that it produces finitary formulas whenever this interpretation is finite. Together, the program simplification introduced in Section 2.5 and aggregate translation provide the basis for turning programs with aggregates into semantically equivalent finite programs with finitary subformulas.

Section 5 further refines our semantic approach to reflect actual grounding processes. To this end, we define the concept of an instantiation sequence based on rule dependencies. We then use the contextual operators of Section 3 to define approximate models of instantiation sequences. While approximate models are in general less precise than well-founded ones, they are better suited for on-the-fly grounding along an instantiation sequence. Nonetheless, they are strong enough to allow for completely evaluating stratified programs.

Section 6 lays out the basic algorithms for grounding rules, components, and entire programs and characterizes their output in terms of the semantic concepts developed in the previous sections. Of particular interest is the treatment of aggregates, which are decomposed into dedicated normal rules before grounding, and reassembled afterward.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

This allows us to ground rules with aggregates by means of grounding algorithms for normal rules. Finally, we show that our grounding algorithm guarantees that an obtained finite ground program is equivalent to the original non-ground program.

The previous sections focus on the theoretical and algorithmic cornerstones of grounding. Section 7 refines these concepts by further detailing aggregate propagation, algorithm specifics, and the treatment of language constructs from *gringo*'s input language.

We relate our contributions to the state of the art in Section 8 and summarize it in Section 9.

Although the developed approach is implemented in *gringo* series 4 and 5, their high degree of sophistication make it hard to retrace the algorithms from Section 6. Hence, to ease comprehensibility, we have moreover implemented the presented approach in *μ-gringo*[1] in a transparent way and equipped it with means for retracing the developed concepts during grounding. This can thus be seen as the practical counterpart to the formal elaboration given below. Also, this system may enable some readers to construct and to experiment with own grounder extensions.

This paper draws on material presented during an invited talk at the third workshop on grounding, transforming, and modularizing theories with variables (Gebser *et al.* 2015c).

2 Foundations

2.1 Operators on lattices

This section recalls basic concepts on operators on complete lattices.

A *complete lattice* is a partially ordered set $(L, \leq)$ in which every subset $S \subseteq L$ has a greatest lower bound and a least upper bound in $(L, \leq)$.

An *operator* O on lattice $(L, \leq)$ is a function from L to L. It is *monotone* if $x \leq y$ implies $O(x) \leq O(y)$ for each $x, y \in L$; and it is *antimonotone* if $x \leq y$ implies $O(y) \leq O(x)$ for each $x, y \in L$.

Let O be an operator on lattice $(L, \leq)$. A *prefixed point* of O is an $x \in L$ such that $O(x) \leq x$. A *postfixed point* of O is an $x \in L$ such that $x \leq O(x)$. A *fixed point of* O is an $x \in L$ such that $x = O(x)$, that is, it is both a prefixed and a postfixed point.

Theorem 1 (*Knaster-Tarski; Tarski 1955*)
Let O be a monotone operator on complete lattice $(L, \leq)$. Then, we have the following properties:

(a) Operator O has a least fixed and prefixed point which are identical.
(b) Operator O has a greatest fixed and postfixed point which are identical.
(c) The fixed points of O form a complete lattice.

2.2 Formulas and interpretations

We begin with a propositional signature Σ consisting of a set of atoms. Following Truszczyński (2012), we define the sets $\mathcal{F}_0, \mathcal{F}_1, \dots$ of formulas as follows:

- $\mathcal{F}_0$ is the set of all propositional atoms in Σ,

[1] The *μ-gringo* system is available at `https://github.com/potassco/mu-gringo`.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

- $\mathcal{F}_{i+1}$ is the set of all elements of $\mathcal{F}_i$, all expressions $\mathcal{H}^\wedge$ and $\mathcal{H}^\vee$ with $\mathcal{H} \subseteq \mathcal{F}_i$, and all expressions $F \rightarrow G$ with $F, G \in \mathcal{F}_i$.

The set $\mathcal{F} = \bigcup_{i=0}^{\infty} \mathcal{F}_i$ contains all *(infinitary propositional) formulas* over Σ.

In the following, we use the shortcuts

- $\top = \emptyset^\wedge$ and $\bot = \emptyset^\vee$,
- $\neg F = F \rightarrow \bot$ where F is a formula, and
- $F \wedge G = \{F, G\}^\wedge$ and $F \vee G = \{F, G\}^\vee$ where F and G are formulas.

We say that a formula is *finitary*, if it has a finite number of subformulas.

An occurrence of a subformula in a formula is called *positive*, if the number of implications containing that occurrence in the antecedent is even, and *strictly positive* if that number is zero; if that number is odd the occurrence is *negative*. The sets F^+ and F^- gather all atoms occurring positively or negatively in formula F, respectively; if applied to a set of formulas, both expressions stand for the union of the respective atoms in the formulas. Also, we define $F^\pm = F^+ \cup F^-$ as the set of all atoms occurring in F.

A *two-valued interpretation* over signature Σ is a set I of propositional atoms such that $I \subseteq \Sigma$. Atoms in an interpretation I are considered *true* and atoms in $\Sigma \setminus I$ as *false*. The set of all interpretations together with the $\subseteq$ relation forms a complete lattice.

The satisfaction relation between interpretations and formulas is defined as follows:

- $I \models a$ for atoms a if $a \in I$,
- $I \models \mathcal{H}^\wedge$ if $I \models F$ for all $F \in \mathcal{H}$,
- $I \models \mathcal{H}^\vee$ if $I \models F$ for some $F \in \mathcal{H}$, and
- $I \models F \rightarrow G$ if $I \not\models F$ or $I \models G$.

An interpretation I is a *model* of a set $\mathcal{H}$ of formulas, written $I \models \mathcal{H}$, if it satisfies each formula in the set.

In the following, all atoms, formulas, and interpretations operate on the same (implicit) signature, unless mentioned otherwise.

2.3 Logic programs and stable models

Our terminology in this section keeps following the one of Truszczyński (2012).

The *reduct* F^I of a formula F w.r.t. an interpretation I is defined as:

- $\bot$ if $I \not\models F$,
- a if $I \models F$ and $F = a \in \mathcal{F}_0$,
- $\{G^I \mid G \in \mathcal{H}\}^\wedge$ if $I \models F$ and $F = \mathcal{H}^\wedge$,
- $\{G^I \mid G \in \mathcal{H}\}^\vee$ if $I \models F$ and $F = \mathcal{H}^\vee$, and
- $G^I \rightarrow H^I$ if $I \models F$ and $F = G \rightarrow H$.

An interpretation I is a *stable model* of a formula F if it is among the (set inclusion) minimal models of F^I.

Note that the reduct removes (among other unsatisfied subformulas) all occurrences of atoms that are false in I. Thus, the satisfiability of the reduct does not depend on such atoms, and all minimal models of F^I are subsets of I. Hence, if I is a stable model of F, then it is the only minimal model of F^I.

Sets $\mathcal{H}_1$ and $\mathcal{H}_2$ of infinitary formulas are *equivalent* if they have the same stable models and *classically equivalent* if they have the same models; they are *strongly equivalent* if, for

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

any set $\mathcal{H}$ of infinitary formulas, $\mathcal{H}_1 \cup \mathcal{H}$ and $\mathcal{H}_2 \cup \mathcal{H}$ are equivalent. As shown by Harrison *et al.* (2017), this also allows for replacing a part of any formula with a strongly equivalent formula without changing the set of stable models.

In the following, we consider implications with atoms as consequent and formulas as antecedent. As common in logic programming, they are referred to as rules, heads, and bodies, respectively, and denoted by reversing the implication symbol. More precisely, an *$\mathcal{F}$-program* is set of *rules* of form $h \leftarrow F$ where $h \in \mathcal{F}_0$ and $F \in \mathcal{F}$. We use $H(h \leftarrow F) = h$ to refer to rule *heads* and $B(h \leftarrow F) = F$ to refer to rule *bodies*. We extend this by letting $H(P) \ = \{H(r) \mid r \in P\}$ and $B(P) \ = \{B(r) \mid r \in P\}$ for any program P.

An interpretation I is a model of an $\mathcal{F}$-program P, written $I \models P$, if $I \models B(r) \to H(r)$ for all $r \in P$. The latter is also written as $I \models r$. We define the reduct of P w.r.t. I as $P^I = \{r^I \mid r \in P\}$ where $r^I = H(r) \leftarrow B(r)^I$. As above, an interpretation I is a *stable model* of P if I is among the minimal models of P^I. Just like the original definition of Gelfond and Lifschitz (1988), the reduct of such programs leaves rule heads intact and only reduces rule bodies. (This feature fits well with the various operators defined in the sequel.)

This program-oriented reduct yields the same stable models as obtained by applying the full reduct to the corresponding infinitary formula.

Proposition 2
Let P be an $\mathcal{F}$-program.

Then, the stable models of formula $\{B(r) \to H(r) \mid r \in P\}^\wedge$ are the same as the stable models of program P.

For programs, Truszczyński (2012) introduces in an alternative reduct, replacing each negatively occurring atom with $\bot$, if it is falsified, and with $\top$, otherwise. More precisely, the so-called ID-*reduct* F_I of a formula F w.r.t. an interpretation I is defined as

$$\begin{aligned}
a_I &= a & a_{\overline{I}} &= \top \text{ if } a \in I \\
 & & a_{\overline{I}} &= \bot \text{ if } a \notin I \\
\mathcal{H}^\wedge_I &= \{F_I \mid F \in \mathcal{H}\}^\wedge & \mathcal{H}^\wedge_{\overline{I}} &= \{F_{\overline{I}} \mid F \in \mathcal{H}\}^\wedge \\
\mathcal{H}^\vee_I &= \{F_I \mid F \in \mathcal{H}\}^\vee & \mathcal{H}^\vee_{\overline{I}} &= \{F_{\overline{I}} \mid F \in \mathcal{H}\}^\vee \\
(F \to G)_I &= F_{\overline{I}} \to G_I & (F \to G)_{\overline{I}} &= F_I \to G_{\overline{I}},
\end{aligned}$$

where a is an atom, $\mathcal{H}$ a set of formulas, and F and G are formulas.

The ID-reduct of an $\mathcal{F}$-program P w.r.t. an interpretation I is $P_I = \{r_I \mid r \in P\}$ where $r_I = H(r) \leftarrow B(r)_I$. As with r^I, the transformation of r into r_I leaves the head of r unaffected.

Example 1
Consider the program containing the single rule

$$p \leftarrow \neg\neg p.$$

We get the following reduced programs w.r.t. interpretations $\emptyset$ and $\{p\}$:

$$\begin{aligned}
\{p \leftarrow \neg\neg p\}^\emptyset &= \{p \leftarrow \bot\} & & & \{p \leftarrow \neg\neg p\}^{\{p\}} &= \{p \leftarrow \neg\bot\} \\
\{p \leftarrow \neg\neg p\}_\emptyset &= \{p \leftarrow \neg\neg p\} & &= & \{p \leftarrow \neg\neg p\}_{\{p\}} &= \{p \leftarrow \neg\neg p\}
\end{aligned}$$

Note that both reducts leave the rule's head intact.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Extending the definition of positive occurrences, we define a formula as *(strictly) positive* if all its atoms occur (strictly) positively in the formula. We define an $\mathcal{F}$-program as (strictly) positive if all its rule bodies are (strictly) positive.

For example, the program in Example 1 is positive but not strictly positive because the only body atom p appears in the scope of two antecedents within the rule body $\neg\neg p$.

As put forward by van Emden and Kowalski (1976), we may associate with each program P its *one-step provability operator* T_P, defined for any interpretation X as

$$T_P(X) = \{H(r) \mid r \in P, X \models B(r)\}.$$

Proposition 3 (*Truszczyński 2012*)
Let P be a positive $\mathcal{F}$-program.

Then, the operator T_P is monotone.

Fixed points of T_P are models of P guaranteeing that each contained atom is supported by some rule in P; prefixed points of T_P correspond to the models of P. According to Theorem 1(a), the T_P operator has a least fixed point for positive $\mathcal{F}$-programs. We refer to this fixed point as the *least model* of P and write it as $LM(P)$.

Observing that the ID-reduct replaces all negative occurrences of atoms, any ID-reduct P_I of a program w.r.t. an interpretation I is positive and thus possesses a least model $LM(P_I)$. This gives rise to the following definition of a stable operator (Truszczyński 2012): Given an $\mathcal{F}$-program P, its ID-*stable operator* is defined for any interpretation I as

$$S_P(I) = LM(P_I).$$

The fixed points of S_P are the ID-*stable models* of P.

Note that neither the program reduct P^I nor the formula reduct F^I guarantee (least) models. Also, stable models and ID-stable models do not coincide in general.

Example 2
Reconsider the program from Example 1, comprising rule

$$p \leftarrow \neg\neg p.$$

This program has the two stable models $\emptyset$ and $\{p\}$, but the empty model is the only ID-stable model.

Proposition 4 (*Truszczyński 2012*)
Let P be an $\mathcal{F}$-program.

Then, the ID-stable operator S_P is antimonotone.

No analogous antimonotone operator is obtainable for $\mathcal{F}$-programs by using the program reduct P^I (and for general theories with the formula reduct F^I). To see this, reconsider Example 2 along with its two stable models $\emptyset$ and $\{p\}$. Given that both had to be fixed points of such an operator, it would behave monotonically on $\emptyset$ and $\{p\}$.

In view of this, we henceforth consider exclusively ID-stable operators and drop the prefix "ID". However, we keep the distinction between stable and ID-stable models.

Truszczyński (2012) identifies in a class of programs for which stable models and ID-stable models coincide. The set $\mathcal{N}$ consists of all formulas F such that any implication in F has $\bot$ as consequent and no occurrences of implications in its antecedent. An $\mathcal{N}$-program consists of rules of form $h \leftarrow F$ where $h \in \mathcal{F}_0$ and $F \in \mathcal{N}$.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Proposition 5 (*Truszczyński 2012*)
Let P be an $\mathcal{N}$-program.
Then, the stable and ID-stable models of P coincide.

Note that a positive $\mathcal{N}$-program is also strictly positive.

2.4 Well-founded models

Our terminology in this section follows the one of Truszczyński (2018) and traces back to the early work of Belnap (1977) and Fitting (2002).[2]

We deal with pairs of sets and extend the basic set relations and operations accordingly. Given sets I', I, J', J, and X, we define:

- $(I', J') \mathrel{\bar{\prec}} (I, J)$ if $I' \prec I$ and $J' \prec J$ for $(\bar{\prec}, \prec) \in \{(\sqsubset, \subset), (\sqsubseteq, \subseteq)\}$
- $(I', J') \mathbin{\bar{\circ}} (I, J) = (I' \circ I, J' \circ J)$ for $(\bar{\circ}, \circ) \in \{(\sqcup, \cup), (\sqcap, \cap), (\diagdown, \setminus)\}$
- $(I, J) \mathbin{\bar{\circ}} X = (I, J) \mathbin{\bar{\circ}} (X, X)$ for $\bar{\circ} \in \{\sqcup, \sqcap, \diagdown\}$

A *four-valued interpretation* over signature Σ is represented by a pair $(I, J) \sqsubseteq (\Sigma, \Sigma)$ where I stands for *certain* and J for *possible* atoms. Intuitively, an atom that is

- certain and possible is *true*,
- certain but not possible is *inconsistent*,
- not certain but possible is *unknown*, and
- not certain and not possible is *false*.

A four-valued interpretation (I', J') is more precise than a four-valued interpretation (I, J), written $(I, J) \leq_p (I', J')$, if $I \subseteq I'$ and $J' \subseteq J$. The precision ordering also has an intuitive reading: the more atoms are certain or the fewer atoms are possible, the more precise is an interpretation. The least precise four-valued interpretation over Σ is $(\emptyset, \Sigma)$. As with two-valued interpretations, the set of all four-valued interpretations over a signature Σ together with the relation $\leq_p$ forms a complete lattice. A four-valued interpretation is called *inconsistent* if it contains an inconsistent atom; otherwise, it is called *consistent*. It is *total* whenever it makes all atoms either true or false. Finally, (I, J) is called *finite* whenever both I and J are finite.

Following Truszczyński (2018), we define the ID-*well-founded operator* of an $\mathcal{F}$-program P for any four-valued interpretation (I, J) as

$$W_P(I, J) = (S_P(J), S_P(I)).$$

This operator is monotone w.r.t. the precision ordering $\leq_p$. Hence, by Theorem 1(b), W_P has a least fixed point, which defines the ID-*well-founded model* of P, also written as $WM(P)$. In what follows, we drop the prefix "ID" and simply refer to the ID-well-founded model of a program as its well-founded model. (We keep the distinction between stable and ID-stable models.)

Any well-founded model (I, J) of an $\mathcal{F}$-program P satisfies $I \subseteq J$.

Lemma 6
Let P be an $\mathcal{F}$-Program.
Then, the well-founded model $WM(P)$ of P is consistent.

[2] The interested reader is referred to the tutorial by Truszczyński (2018) for further details.

Example 3

Consider program P_3 consisting of the following rules:

$$\begin{aligned} &a\\ &b \leftarrow a\\ &c \leftarrow \neg b\\ &d \leftarrow c\\ &e \leftarrow \neg d. \end{aligned}$$

We compute the well-founded model of P_3 in four iterations starting from $(\emptyset, \Sigma)$:

$$\begin{array}{lrr} 1. & S_P(\Sigma) = \{a, b\} & S_P(\emptyset) = \{a, b, c, d, e\}\\ 2. & S_P(\{a, b, c, d, e\}) = \{a, b\} & S_P(\{a, b\}) = \{a, b, e\}\\ 3. & S_P(\{a, b, e\}) = \{a, b, e\} & S_P(\{a, b\}) = \{a, b, e\}\\ 4. & S_P(\{a, b, e\}) = \{a, b, e\} & S_P(\{a, b, e\}) = \{a, b, e\}. \end{array}$$

The left and right column reflect the certain and possible atoms computed at each iteration, respectively. We reach a fixed point at Step 4. Accordingly, the well-founded model of P_3 is $(\{a, b, e\}, \{a, b, e\})$.

Unlike general $\mathcal{F}$-programs, the class of $\mathcal{N}$-programs warrants the same stable and ID-stable models for each of its programs. Unfortunately, $\mathcal{N}$-programs are too restricted for our purpose (for instance, for capturing aggregates in rule bodies[3]). To this end, we define a more general class of programs and refer to them as $\mathcal{R}$-programs. Although ID-stable models of $\mathcal{R}$-programs may differ from their stable models (see below), their well-founded models encompass both stable and ID-stable models. Thus, well-founded models can be used for characterizing stable model-preserving program transformations. In fact, we see in Section 2.5 that the restriction of $\mathcal{F}$- to $\mathcal{R}$-programs allows us to provide tighter semantic characterizations of program simplifications.

We define $\mathcal{R}$ to be the set of all formulas F such that implications in F have no further occurrences of implications in their antecedents. Then, an $\mathcal{R}$*-program* consists of rules of form $h \leftarrow F$ where $h \in \mathcal{F}_0$ and $F \in \mathcal{R}$. As with $\mathcal{N}$-programs, a positive $\mathcal{R}$-program is also strictly positive.

Our next result shows that (ID-)well-founded models can be used for approximating (regular) stable models of $\mathcal{R}$-programs.

Theorem 7

Let P be an $\mathcal{R}$-program and (I, J) be the well-founded model of P.

If X is a stable model of P, then $I \subseteq X \subseteq J$.

Example 4

Consider the $\mathcal{R}$-program P_4:[4]

$$\begin{array}{lcl} c \leftarrow (b \rightarrow a) & \qquad & a \leftarrow b\\ a \leftarrow c & & b \leftarrow a. \end{array}$$

[3] Ferraris' semantics (Ferraris 2011) of aggregates introduces implications, which results in rules beyond the class of $\mathcal{N}$-programs.

[4] The choice of the body $b \rightarrow a$ is not arbitrary since it can be seen as representing the aggregate $\#\text{sum}\{1 : a, -1 : b\} \geq 0$.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Observe that $\{a, b, c\}$ is the only stable model of P_4, the program does not have any ID-stable models, and the well-founded model of P_4 is $(\emptyset, \{a, b, c\})$. In accordance with Theorem 7, the stable model of P_4 is enclosed in the well-founded model.

Note that the ID-reduct handles $b \to a$ the same way as $\neg b \vee a$. In fact, the program obtained by replacing

$$c \leftarrow (b \to a)$$

with

$$c \leftarrow \neg b \vee a$$

is an $\mathcal{N}$-program and has neither stable nor ID-stable models.

Further, note that the program in Example 2 is not an $\mathcal{R}$-program, whereas the one in Example 3 is an $\mathcal{R}$-program.

2.5 Program simplification

In this section, we define a concept of program simplification relative to a four-valued interpretation and show how its result can be characterized by the semantic means from above. This concept has two important properties. First, it results in a finite program whenever the interpretation used for simplification is finite. And second, it preserves all (regular) stable models of $\mathcal{R}$-programs when simplified with their well-founded models.

Definition 1

Let P be an $\mathcal{F}$-program, and (I, J) be a four-valued interpretation.

We define the simplification of P w.r.t. (I, J) as

$$P^{(I,J)} = \{r \in P \mid J \models B(r)_I\}.$$

For simplicity, we drop parentheses and we write $P^{I,J}$ instead of $P^{(I,J)}$ whenever clear from context.

The program simplification $P^{I,J}$ acts as a filter eliminating inapplicable rules that fail to satisfy the condition $J \models B(r)_I$. That is, first, all negatively occurring atoms in $B(r)$ are evaluated w.r.t. the certain atoms in I and replaced accordingly by $\bot$ and $\top$, respectively. Then, it is checked whether the reduced body $B(r)_I$ is satisfiable by the possible atoms in J. Only in this case, the rule is kept in $P^{I,J}$. No simplifications are applied to the remaining rules. This is illustrated in Example 5 below.

Note that $P^{I,J}$ is finite whenever (I, J) is finite.

Observe that for an $\mathcal{F}$-program P the head atoms in $P^{I,J}$ correspond to the result of applying the provability operator of program P_I to the possible atoms in J, that is, $H(P^{I,J}) = T_{P_I}(J)$.

Our next result shows that programs simplified with their well-founded model maintain this model.

Theorem 8

Let P be an $\mathcal{F}$-program and (I, J) be the well-founded model of P.

Then, P and $P^{I,J}$ have the same well-founded model.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Example 5

In Example 3, we computed the well-founded model $(\{a,b,e\},\{a,b,e\})$ of P_3. With this, we obtain the simplified program $P_3' = P_3^{\{a,b,e\},\{a,b,e\}}$ after dropping $c \leftarrow \neg b$ and $d \leftarrow c$:

$$\begin{aligned} &a \\ &b \leftarrow a \\ &e \leftarrow \neg d. \end{aligned}$$

Next, we check that the well-founded model of P_3' corresponds to the well-founded model of P_3:

$$\begin{array}{lrr} 1. & S_{P_3'}(\Sigma) = \{a,b\} & S_{P_3'}(\emptyset) = \{a,b,e\} \\ 2. & S_{P_3'}(\{a,b,e\}) = \{a,b,e\} & S_{P_3'}(\{a,b\}) = \{a,b,e\} \\ 3. & S_{P_3'}(\{a,b,e\}) = \{a,b,e\} & S_{P_3'}(\{a,b,e\}) = \{a,b,e\}. \end{array}$$

We observe that it takes two applications of the well-founded operator to obtain the well-founded model. This could be reduced to one step if atoms false in the well-founded model would be removed from the negative bodies by the program simplification. Keeping them is a design decision with the goal to simplify notation in the following.

The next series of results further elaborates on semantic invariants guaranteed by our concept of program simplification. The first result shows that it preserves all stable models between the sets used for simplification.

Theorem 9

Let P be an $\mathcal{F}$-program, and I, J, and X be two-valued interpretations.

If $I \subseteq X \subseteq J$, then X is a stable model of P iff X is a stable model of $P^{I,J}$.

As a consequence, we obtain that $\mathcal{R}$-programs simplified with their well-founded model also maintain stable models.

Corollary 10

Let P be an $\mathcal{R}$-program and (I,J) be the well-founded model of P.

Then, P and $P^{I,J}$ have the same stable models.

For instance, the $\mathcal{R}$-program in Example 3 and its simplification in Example 5 have the same stable model. Unlike this, the program from Example 2 consisting of rule $p \leftarrow \neg\neg p$ induces two stable models, while its simplification w.r.t. its well-founded model $(\emptyset,\emptyset)$ yields an empty program admitting the empty stable model only.

Note that given an $\mathcal{R}$-program with a finite well-founded model, we obtain a semantically equivalent finite program via simplification. As detailed in the following sections, grounding algorithms only compute approximations of the well-founded model. However, as long as the approximation is finite, we still obtain semantically equivalent finite programs. This is made precise by the next two results showing that any program between the original and its simplification relative to its well-founded model preserves the well-founded model, and that this extends to all stable models for $\mathcal{R}$-programs.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Theorem 11

Let P and Q be $\mathcal{F}$-programs, and (I, J) be the well-founded model of P.

If $P^{I,J} \subseteq Q \subseteq P$, then P and Q have the same well-founded models.

Corollary 12

Let P and Q be $\mathcal{R}$-programs, and (I, J) be the well-founded model of P.

If $P^{I,J} \subseteq Q \subseteq P$, then P and Q are equivalent.

3 Splitting

One of the first steps during grounding is to group rules into components suitable for successive instantiation. This amounts to splitting a logic program into a sequence of subprograms. The rules in each such component are then instantiated with respect to the atoms possibly derivable from previous components, starting with some component consisting of facts only. In other words, grounding is always performed relative to a set of atoms that provide a context. Moreover, atoms found to be true or false can be used for on-the-fly simplifications.

Accordingly, this section parallels the above presentation by extending the respective formal concepts with contextual information provided by atoms in a two- and four-valued setting. We then assemble the resulting concepts to enable their consecutive application to sequences of subprograms. Interestingly, the resulting notion of splitting allows for more fine-grained splitting than the traditional concept (Lifschitz and Turner 1994) since it allows us to partition rules in an arbitrary way. In view of grounding, we show that once a program is split into a sequence of programs, we can iteratively compute an approximation of the well-founded model by considering in turn each element in the sequence.

In what follows, we append letter "C" to names of interpretations having a contextual nature.

To begin with, we extend the one-step provability operator accordingly.

Definition 2

Let P be an $\mathcal{F}$-program and IC be a two-valued interpretation.

For any two-valued interpretation I, we define the *one-step provability operator of P relative to IC* as

$$T_P^{IC}(I) = T_P(IC \cup I).$$

A prefixed point of T_P^{IC} is a also a prefixed point of T_P. Thus, each prefixed point of T_P^{IC} is a model of P but not vice versa.

To see this, consider program $P = \{a \leftarrow b\}$. We have $T_P(\emptyset) = \emptyset$ and $T_P^{\{b\}}(\emptyset) = \{a\}$. Hence, $\emptyset$ is a (pre)fixed point of T_P but not of $T_P^{\{b\}}$ since $\{a\} \not\subseteq \emptyset$. The set $\{a\}$ is a prefixed point of both operators.

Proposition 13

Let P be a positive program, and IC and J be two valued interpretations.

Then, the operators T_P^{IC} and $T_P^{\cdot}(J)$ are both monotone.

We use Theorems 1 and 13 to define a contextual stable operator.

Definition 3

Let P be an $\mathcal{F}$-program and IC be a two-valued interpretation.

For any two-valued interpretation J, we define the *stable operator relative* to IC, written $S_P^{IC}(J)$, as the least fixed point of $T_{P_J}^{IC}$.

While the operator is antimonotone w.r.t. its argument J, it is monotone regarding its parameter IC.

Proposition 14

Let P be an $\mathcal{F}$-program, and IC and J be two-valued interpretations.

Then, the operators S_P^{IC} and $S_P^{\cdot}(J)$ are antimonotone and monotone, respectively.

By building on the relative stable operator, we next define its well-founded counterpart. Unlike above, the context is now captured by a four-valued interpretation.

Definition 4

Let P be an $\mathcal{F}$-program and (IC, JC) be a four-valued interpretation.

For any four-valued interpretation (I, J), we define the *well-founded operator relative* to (IC, JC) as

$$W_P^{(IC,JC)}(I, J) = (S_P^{IC}(J \cup JC), S_P^{JC}(I \cup IC)).$$

As above, we drop parentheses and simply write $W_P^{I,J}$ instead of $W_P^{(I,J)}$. Also, we keep refraining from prepending the prefix "ID" to the well-founded operator along with all concepts derived from it below.

Unlike the stable operator, the relative well-founded one is monotone on both its argument and parameter.

Proposition 15

Let P be an $\mathcal{F}$-program, and (I, J) and (IC, JC) be four-valued interpretations.

Then, the operators $W_P^{IC,JC}$ and $W_P^{\cdot}(I, J)$ are both monotone w.r.t. the precision ordering.

From Theorems 1 and 15, we get that the relative well-founded operator has a least fixed point.

Definition 5

Let P be an $\mathcal{F}$-program and (IC, JC) be a four-valued interpretation.

We define the *well-founded model* of P *relative* to (IC, JC), written $WM^{(IC,JC)}(P)$, as the least fixed point of $W_P^{IC,JC}$.

Whenever clear from context, we keep dropping parentheses and simply write $WM^{I,J}(P)$ instead of $WM^{(I,J)}(P)$.

In what follows, we use the relativized concepts defined above to delineate the semantics and resulting simplifications of the sequence of subprograms resulting from a grounder's decomposition of the original program. For simplicity, we first present a theorem capturing the composition under the well-founded operation, before we give the general case involving a sequence of programs.

Just like suffix C, we use the suffix E (and similarly letter E further below) to indicate atoms whose defining rules are yet to come.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

As in traditional splitting, we begin by differentiating a bottom and a top program. In addition to the input atoms (I, J) and context atoms in (IC, JC), we moreover distinguish a set of external atoms, (IE, JE), which occur in the bottom program but are defined in the top program. Accordingly, the bottom program has to be evaluated relative to $(IC, JC) \sqcup (IE, JE)$ (and not just (IC, JC) as above) to consider what could be derived by the top program. Also, observe that our notion of splitting aims at computing well-founded models rather than stable models.

Theorem 16
Let PB and PT be $\mathcal{F}$-programs, (IC, JC) be a four-valued interpretation, $(I, J) = WM^{IC,JC}(PB \cup PT)$, $(IE, JE) = (I, J) \sqcap (B(PB)^{\pm} \cap H(PT))$, $(IB, JB) = WM^{(IC,JC)\sqcup(IE,JE)}(PB)$, and $(IT, JT) = WM^{(IC,JC)\sqcup(IB,JB)}(PT)$.

Then, we have $(I, J) = (IB, JB) \sqcup (IT, JT)$.

Partially expanding the statements of the two previous result nicely reflects the decomposition of the application of the well-founded founded model of a program:

$$WM^{IC,JC}(PB \cup PT) = WM^{(IC,JC)\sqcup(IE,JE)}(PB) \sqcup WM^{(IC,JC)\sqcup(IB,JB)}(PT).$$

Note that the formulation of the theorem forms the external interpretation (IE, JE), by selecting atoms from the overarching well-founded model (I, J). This warrants the correspondence of the overall interpretations to the union of the bottom and top well-founded model. This global approach is dropped below (after the next example) and leads to less precise composed models.

Example 6
Let us illustrate the above approach via the following program:

$$\begin{aligned} &a && (PB)\\ &b && (PB)\\ &c \leftarrow a && (PT)\\ &d \leftarrow \neg b. && (PT)\end{aligned}$$

The well-founded model of this program relative to $(IC, JC) = (\emptyset, \emptyset)$ is

$$(I, J) = (\{a, b, c\}, \{a, b, c\}).$$

First, we partition the four rules of the program into PB and PT as given above. We get $(IE, JE) = (\emptyset, \emptyset)$ since $B(PB)^{\pm} \cap H(PT) = \emptyset$. Let us evaluate PB before PT. The well-founded model of PB relative to $(IC, JC) \sqcup (IE, JE)$ is

$$(IB, JB) = (\{a, b\}, \{a, b\}).$$

With this, we calculate the well-founded model of PT relative to $(IC, JC) \sqcup (IB, JB)$:

$$(IT, JT) = (\{c\}, \{c\}).$$

We see that the union of $(IB, JB) \sqcup (IT, JT)$ is the same as the well-founded model of $PB \cup PT$ relative to (IC, JC).

This corresponds to standard splitting in the sense that $\{a, b\}$ is a splitting set for $PB \cup PT$ and PB is the "bottom" and PT is the "top" (Lifschitz and Turner 1994).

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Example 7

For a complement, let us reverse the roles of programs PB and PT in Example 6. Unlike above, body atoms in PB now occur in rule heads of PT, that is, $B(PB)^{\pm} \cap H(PT) = \{a, b\}$. We thus get $(IE, JE) = (\{a, b\}, \{a, b\})$. The well-founded model of PB relative to $(IC, JC) \sqcup (IE, JE)$ is

$$(IB, JB) = (\{c\}, \{c\}).$$

And the well-founded model of PT relative to $(IC, JC) \sqcup (IB, JB)$ is

$$(IT, JT) = (\{a, b\}, \{a, b\}).$$

Again, we see that the union of both models is identical to (I, J).

This decomposition has no direct correspondence to standard splitting (Lifschitz and Turner 1994) since there is no splitting set.

Next, we generalize the previous results from two programs to sequences of programs. For this, we let $\mathbb{I}$ be a well-ordered index set and direct our attention to sequences $(P_i)_{i\in\mathbb{I}}$ of $\mathcal{F}$-programs.

Definition 6

Let $(P_i)_{i\in\mathbb{I}}$ be a sequence of $\mathcal{F}$-programs.

We define the *well-founded model* of $(P_i)_{i\in\mathbb{I}}$ as

$$WM((P_i)_{i\in\mathbb{I}}) = \bigsqcup_{i\in\mathbb{I}} (I_i, J_i), \tag{2}$$

where

$$E_i = B(P_i)^{\pm} \cap \bigcup_{i<j} H(P_j), \tag{3}$$

$$(IC_i, JC_i) = \bigsqcup_{j<i} (I_j, J_j), \text{ and} \tag{4}$$

$$(I_i, J_i) = WM^{(IC_i, JC_i) \sqcup (\emptyset, E_i)}(P_i). \tag{5}$$

The well-founded model of a program sequence is itself assembled in (2) from a sequence of well-founded models of the individual subprograms in (5). This provides us with semantic guidance for successive program simplification, as shown below. In fact, proceeding along the sequence of subprograms reflects the iterative approach of a grounding algorithm, one component is grounded at a time. At each stage $i \in \mathbb{I}$, this takes into account the truth values of atoms instantiated in previous iterations, viz. (IC_i, JC_i), as well as dependencies to upcoming components in E_i. Note that unlike Proposition 16, the external atoms in E_i are identified purely syntactically, and the interpretation $(\emptyset, E_i)$ treats them as unknown. Grounding is thus performed under incomplete information and each well-founded model in (5) can be regarded as an over-approximation of the actual one. This is enabled by the monotonicity of the well-founded operator in Proposition 15 that only leads to a less precise result when overestimating its parameter.

Accordingly, the next theorem shows that once we split a program into a sequence of $\mathcal{F}$-programs, we can iteratively compute an approximation of the well-founded model by considering in turn each element in the sequence.

Theorem 17
Let $(P_i)_{i\in\mathbb{I}}$ be a sequence of $\mathcal{F}$-programs.

Then, $\mathit{WM}((P_i)_{i\in\mathbb{I}}) \leq_p \mathit{WM}(\bigcup_{i\in\mathbb{I}} P_i)$.

The next two results transfer Theorem 17 to program simplification by successively simplifying programs with the respective well-founded models of the previous programs.

Theorem 18
Let $(P_i)_{i\in\mathbb{I}}$ be a sequence of $\mathcal{F}$-programs, $(I,J) = \mathit{WM}((P_i)_{i\in\mathbb{I}})$, and E_i, $(\mathit{IC}_i, \mathit{JC}_i)$, and (I_i, J_i) be defined as in (3)–(5).

Then, $P_k^{I,J} \subseteq P_k^{(\mathit{IC}_k,\mathit{JC}_k)\sqcup(I_k,J_k)\sqcup(\emptyset,E_k)} \subseteq P_k$ for all $k \in \mathbb{I}$.

Corollary 19
Let $(P_i)_{i\in\mathbb{I}}$ be a sequence of $\mathcal{R}$-programs, and E_i, $(\mathit{IC}_i, \mathit{JC}_i)$, and (I_i, J_i) be defined as in Equation (3)–Equation (5).

Then, $\bigcup_{i\in\mathbb{I}} P_i$ and $\bigcup_{i\in\mathbb{I}} P_i^{(\mathit{IC}_i,\mathit{JC}_i)\sqcup(I_i,J_i)\sqcup(\emptyset,E_i)}$ have the same well-founded and stable models.

Let us mention that the previous result extends to sequences of $\mathcal{F}$-programs and their well-founded models but not their stable models.

Example 8
To illustrate Theorem 17, let us consider the following programs, P_1 and P_2:

$$a \leftarrow \neg c \qquad (P_1)$$
$$b \leftarrow \neg d \qquad (P_1)$$
$$c \leftarrow \neg b \qquad (P_2)$$
$$d \leftarrow e. \qquad (P_2)$$

The well-founded model of $P_1 \cup P_2$ is

$$(I,J) = (\{a,b\},\{a,b\}).$$

Let us evaluate P_1 before P_2. While no head literals of P_2 occur positively in P_1, the head literals c and d of P_2 occur negatively in rule bodies of P_1. Hence, we get $E_1 = \{c,d\}$ and treat both atoms as unknown while calculating the well-founded model of P_1 relative to $(\emptyset,\{c,d\})$:

$$(I_1,J_1) = (\emptyset,\{a,b\}).$$

We obtain that both a and b are unknown. With this and $E_2 = \emptyset$, we can calculate the well-founded model of P_2 relative to (I_1,J_1):

$$(I_2,J_2) = (\emptyset,\{c\}).$$

We see that because a is unknown, we have to derive c as unknown, too. And because there is no rule defining e, we cannot derive d. Hence, $(I_1,J_1) \sqcup (I_2,J_2)$ is less precise than (I,J) because, when evaluating P_1, it is not yet known that c is true and d is false.

Next, we illustrate the simplified programs according to Theorem 18:

$$a \leftarrow \neg c \qquad\qquad a \leftarrow \neg c \qquad (P_1)$$
$$b \leftarrow \neg d \qquad\qquad b \leftarrow \neg d \qquad (P_1)$$
$$c \leftarrow \neg b. \qquad (P_2)$$

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

The left column contains the simplification of $P_1 \cup P_2$ w.r.t. (I, J) and the right column the simplification of P_1 w.r.t. (I_1, J_1) and P_2 w.r.t. $(I_1, J_1) \sqcup (I_2, J_2)$. Note that $d \leftarrow e$ has been removed in both columns because e is false in both (I, J) and $(I_1, J_1) \sqcup (I_2, J_2)$. But we can only remove $c \leftarrow \neg b$ from the left column because, while b is false in (I, J), it is unknown in $(I_1, J_1) \sqcup (I_2, J_2)$.

Finally, observe that in accordance with Theorem 9 and Corollaries 10 and cor:sequence:simplification:stable, the program $P_1 \cup P_2$ and the two simplified programs have the same stable and well-founded models.

Clearly, the best simplifications are obtained when simplifying with the actual well-founded model of the overall program. This can be achieved for a sequence as well whenever E_i is empty, that is, if there is no need to approximate the impact of upcoming atoms.

Corollary 20

Let $(P_i)_{i\in\mathbb{I}}$ be a sequence of $\mathcal{F}$-programs and E_i be defined as in (3).

If $E_i = \emptyset$ for all $i \in \mathbb{I}$ then $WM((P_i)_{i\in\mathbb{I}}) = WM(\bigcup_{i\in\mathbb{I}} P_i)$.

Corollary 21

Let $(P_i)_{i\in\mathbb{I}}$ be a sequence of $\mathcal{F}$-programs, $(I, J) = WM((P_i)_{i\in\mathbb{I}})$, and E_i, (IC_i, JC_i), and (I_i, J_i) be defined as in (3)–(5).

If $E_i = \emptyset$ for all $i \in \mathbb{I}$, then $P_k^{I,J} = P_k^{(IC_k,JC_k)\sqcup(I_k,J_k)}$ for all $k \in \mathbb{I}$.

Example 9

Next, let us illustrate Corollary 20 on an example. We take the same rules as in Example 8 but use a different sequence:

$$d \leftarrow e \tag{P_1}$$

$$b \leftarrow \neg d \tag{P_1}$$

$$c \leftarrow \neg b \tag{P_2}$$

$$a \leftarrow \neg c. \tag{P_2}$$

Observe that the head literals of P_2 do not occur in the bodies of P_1, that is, $E_1 = B(P_1)^{\pm} \cap H(P_2) = \emptyset$. The well-founded model of P_1 is

$$(I_1, J_1) = (\{b\}, \{b\}).$$

And the well-founded model of P_2 relative to $(\{b\}, \{b\})$ is

$$(I_2, J_2) = (\{a\}, \{a\}).$$

Hence, the union of both models is identical to the well-founded model of $P_1 \cup P_2$.

Next, we investigate the simplified program according to Corollary 21:

$$b \leftarrow \neg d \tag{P_1}$$

$$a \leftarrow \neg c. \tag{P_2}$$

As in Example 8, we delete rule $d \leftarrow e$ because e is false in (I_1, J_1). But this time, we can also remove rule $c \leftarrow \neg b$ because b is true in $(I_1, J_1) \sqcup (I_2, J_2)$.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

4 Aggregate programs

We now turn to programs with aggregates and, at the same time, to programs with variables. That is, we now deal with finite nonground programs whose instantiation may lead to infinite ground programs including infinitary subformulas. This is made precise by Harrison *et al.* (2014) and Gebser *et al.* (2015a) where aggregate programs are associated with infinitary propositional formulas (Truszczyński 2012). However, the primary goal of grounding is to produce a finite set of ground rules with finitary subformulas only. In fact, the program simplification introduced in Section 2.5 allows us to produce an equivalent finite ground program whenever the well-founded model is finite. The source of infinitary subformulas lies in the instantiation of aggregates. We address this below by introducing an aggregate translation bound by an interpretation that produces finitary formulas whenever this interpretation is finite. Together, our concepts of program simplification and aggregate translation provide the backbone for turning programs with aggregates into semantically equivalent finite programs with finitary subformulas.

Our concepts follow the ones of Gebser *et al.* (2015a); the semantics of aggregates is aligned with that of Ferraris (2011) yet lifted to infinitary formulas (Truszczyński 2012; Harrison *et al.* 2014).

We consider a *signature* $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{V})$ consisting of sets of function, predicate, and variable symbols. The sets of variable and function symbols are disjoint. *Function* and *predicate symbols* are associated with non-negative arities. For short, a predicate symbol p of arity n is also written as p/n. In the following, we use lower case strings for function and predicate symbols, and upper case strings for variable symbols. Also, we often drop the term 'symbol' and simply speak of functions, predicates, and variables.

As usual, *terms* over Σ are defined inductively as follows:

- $v \in \mathcal{V}$ is a term and
- $f(t_1, \dots, t_n)$ is a term if $f \in \mathcal{F}$ is a function symbol of arity n and each t_i is a term over Σ.

Parentheses for terms over function symbols of arity 0 are omitted.

Unless stated otherwise, we assume that the set of (zero-ary) functions includes a set of numeral symbols being in a one-to-one correspondence to the integers. For simplicity, we drop this distinction and identify numerals with the respective integers.

An *atom* over signature Σ has form $p(t_1, \dots, t_n)$ where $p \in \mathcal{P}$ is a predicate symbol of arity n and each t_i is a term over Σ. As above, parentheses for atoms over predicate symbols of arity 0 are omitted. Given an atom a over Σ, a *literal* over Σ is either the atom itself or its negation $\neg a$. A literal without negation is called *positive*, and *negative* otherwise.

A *comparison* over Σ has form

$$t_1 \prec t_2, \tag{6}$$

where t_1 and t_2 are terms over Σ and $\prec$ is a relation symbol among $<$, $\leq$, $>$, $\geq$, $=$, and $\neq$.

An *aggregate element* over Σ has form

$$t_1, \dots, t_m : a_1 \wedge \dots \wedge a_n, \tag{7}$$

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

where t_i is a term and a_j is an atom, both over Σ for $0 \leq i \leq m$ and $0 \leq j \leq n$. The terms $t_1, \dots, t_m$ are seen as a tuple, which is empty for $m = 0$; the conjunction $a_1 \wedge \dots \wedge a_n$ is called the *condition* of the aggregate element. For an aggregate element e of form (7), we use $H(e) = (t_1, \dots, t_m)$ and $B(e) = \{a_1, \dots, a_n\}$. We extend both to sets of aggregate elements in the straightforward way, that is, $H(E) = \{H(e) \mid e \in E\}$ and $B(E) = \{B(e) \mid e \in E\}$.

An *aggregate atom* over Σ has form

$$f\{e_1, \dots, e_n\} \prec s, \tag{8}$$

where $n \geq 0$, f is an aggregate name among #count, #sum, #sum$^+$, and #sum$^-$, each e_i is an aggregate element, $\prec$ is a relation symbol among $<$, $\leq$, $>$, $\geq$, $=$, and $\neq$ (as above), and s is a term representing the aggregate's *bound*.

Without loss of generality, we refrain from introducing negated aggregate atoms.[5] We often refer to aggregate atoms simply as aggregates.

An *aggregate program* over Σ is a finite set of *aggregate rules* of form

$$h \leftarrow b_1 \wedge \dots \wedge b_n,$$

where $n \geq 0$, h is an atom over Σ and each b_i is either a literal, a comparison, or an aggregate over Σ. We refer to $b_1, \dots, b_n$ as body literals, and extend functions $H(r)$ and $B(r)$ to any aggregate rule r.

Example 10

An example for an aggregate program is shown below, giving an encoding of the *Company Controls Problem* (Mumick *et al.* 1990): A company X controls a company Y if X directly or indirectly controls more than 50% of the shares of Y.

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

The aggregate #sum$^+$ implements summation over positive integers. Notably, it takes part in the recursive definition of predicate *controls*. In the following, we use an instance with ownership relations between four companies:

$$\begin{array}{cccc} \mathit{company}(c_1) & \mathit{company}(c_2) & \mathit{company}(c_3) & \mathit{company}(c_4) \\ \mathit{owns}(c_1,c_2,60) & \mathit{owns}(c_1,c_3,20) & \mathit{owns}(c_2,c_3,35) & \mathit{owns}(c_3,c_4,51). \end{array}$$

We say that an aggregate rule r is *normal* if its body does not contain aggregates. An aggregate program is normal if all its rules are normal.

A term, literal, aggregate element, aggregate, rule, or program is *ground* whenever it does not contain any variables.

We assume that all ground terms are totally ordered by a relation $\leq$, which is used to define the relations $<$, $>$, $\geq$, $=$, and $\neq$ in the standard way. For ground terms t_1, t_2 and

[5] Grounders like *lparse* and *gringo* replace aggregates systematically by auxiliary atoms and place them in the body of new rules implying the respective auxiliary atom. This results in programs without occurrences of negated aggregates.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

a corresponding relation symbol $\prec$, we say that $\prec$ holds between t_1 and t_2 whenever the corresponding relation holds between t_1 and t_2. Furthermore, $>$, $\geq$, and $\neq$ hold between ∞ and any other term, and $<$, $\leq$, and $\neq$ hold between $-\infty$ and any other term. Finally, we require that integers are ordered as usual.

For defining sum-based aggregates, we define for a tuple $t = t_1, \dots, t_m$ of ground terms the following weight functions:

$$\begin{aligned} w(t) &= \begin{cases} t_1 & \text{if } m > 0 \text{ and } t_1 \text{ is an integer} \\ 0 & \text{otherwise,} \end{cases} \\ w^+(t) &= \max\{w(t), 0\}, \text{ and} \\ w^-(t) &= \min\{w(t), 0\}. \end{aligned}$$

With this at hand, we now define how to apply aggregate functions to sets of tuples of ground terms in analogy to Gebser *et al.* (2015a).

Definition 7

Let T be a set of tuples of ground terms.

We define

$$\#\text{count}(T) = \begin{cases} |T| & \text{if } T \text{ is finite,} \\ \infty & \text{otherwise,} \end{cases}$$

$$\#\text{sum}(T) = \begin{cases} \Sigma_{t \in T} w(t) & \text{if } \{t \in T \mid w(t) \neq 0\} \text{ is finite,} \\ 0 & \text{otherwise,} \end{cases}$$

$$\#\text{sum}^+(T) = \begin{cases} \Sigma_{t \in T} w^+(t) & \text{if } \{t \in T \mid w(t) > 0\} \text{ is finite,} \\ \infty & \text{otherwise, and} \end{cases}$$

$$\#\text{sum}^-(T) = \begin{cases} \Sigma_{t \in T} w^-(t) & \text{if } \{t \in T \mid w(t) < 0\} \text{ is finite,} \\ -\infty & \text{otherwise.} \end{cases}$$

Note that in our setting the application of aggregate functions to infinite sets of ground terms is of theoretical relevance only, since we aim at reducing them to their finite equivalents so that they can be evaluated by a grounder.

A variable is *global* in

- a literal if it occurs in the literal,
- a comparison if it occurs in the comparison,
- an aggregate if it occurs in its bound, and
- a rule if it is global in its head atom or in one of its body literals.

For example, the variables X and Y are global in the aggregate rule in Example 10, while Z and S are neither global in the rule nor the aggregate.

Definition 8

Let r be an aggregate rule.

We define r to be *safe*

- if all its global variables occur in some positive literal in the body of r and
- if all its non-global variables occurring in an aggregate element e of an aggregate in the body of r, also occur in some positive literal in the condition of e.

For instance, the aggregate rule in Example 10 is safe.

Note that comparisons are disregarded in the definition of safety. That is, variables in comparisons have to occur in positive body literals.[6]

An aggregate program is safe if all its rules are safe.

An *instance* of an aggregate rule r is obtained by substituting ground terms for all its global variables. We use $\mathrm{Inst}(r)$ to denote the set of all instances of r and $\mathrm{Inst}(P)$ to denote the set of all ground instances of rules in aggregate program P. An *instance* of an aggregate element e is obtained by substituting ground terms for all its variables. We let $\mathrm{Inst}(E)$ stand for all instances of aggregate elements in a set E. Note that $\mathrm{Inst}(E)$ consists of ground expressions, which is not necessarily the case for $\mathrm{Inst}(r)$. As seen from the first example in the introductory section, both $\mathrm{Inst}(r)$ and $\mathrm{Inst}(E)$ can be infinite.

A literal, aggregate element, aggregate, or rule is *closed* if it does not contain any global variables.

For example, the following rule is an instance of the aggregate rule in Example 10.

$$\begin{aligned}\mathit{controls}(c_1, c_2) \leftarrow \#\mathrm{sum}^+\{S : \mathit{owns}(c_1, c_2, S);& \\ S, Z : \mathit{controls}(c_1, Z), \mathit{owns}(Z, c_2, S)\} > 50& \\ \wedge\, \mathit{company}(c_1) \wedge \mathit{company}(c_2) \wedge c_1 \neq c_2.&\end{aligned}$$

Note that both the rule and its aggregate are closed. It is also noteworthy to realize that the two elements of the aggregate induce an infinite set of instances, among them

$$\begin{aligned}20 : \mathit{owns}(c_1, c_2, 20) \qquad\qquad\qquad \text{and}\\ 35, c_2 : \mathit{controls}(c_1, c_2), \mathit{owns}(c_2, c_3, 35).\end{aligned}$$

We now turn to the semantics of aggregates as introduced by Ferraris (2011) but follow its adaptation to closed aggregates by Gebser *et al.* (2015a): Let a be a closed aggregate of form (8) and E be its set of aggregate elements. We say that a set $D \subseteq \mathrm{Inst}(E)$ of its elements' instances *justifies* a, written $D \triangleright a$, if $f(H(D)) \prec s$ holds.

An aggregate a is *monotone* whenever $D_1 \triangleright a$ implies $D_2 \triangleright a$ for all $D_1 \subseteq D_2 \subseteq \mathrm{Inst}(E)$, and accordingly a is *antimonotone* if $D_2 \triangleright a$ implies $D_1 \triangleright a$ for all $D_1 \subseteq D_2 \subseteq \mathrm{Inst}(E)$.

We observe the following monotonicity properties.

Proposition 22 (*Harrison et al. 2014*)

- Aggregates over functions $\#\mathrm{sum}^+$ and $\#\mathrm{count}$ together with relations $>$ and $\geq$ are monotone.
- Aggregates over functions $\#\mathrm{sum}^+$ and $\#\mathrm{count}$ together with relations $<$ and $\leq$ are antimonotone.
- Aggregates over function $\#\mathrm{sum}^-$ have the same monotonicity properties as $\#\mathrm{sum}^+$ aggregates with the complementary relation.

Next, we give the translation τ from aggregate programs to $\mathcal{R}$-programs, derived from the ones of Ferraris (2011) and Harrison *et al.* (2014):

For a closed literal l, we have

$$\tau(l) = l,$$

[6] In fact, *gringo* allows for variables in some comparisons to guarantee safety, as detailed in Section 7.3.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

for a closed comparison l of form (6), we have

$$\tau(l) = \begin{cases} \top & \text{if } \prec \text{ holds between } t_1 \text{ and } t_2 \\ \bot & \text{otherwise} \end{cases}$$

and for a set L of closed literals, comparisons and aggregates, we have

$$\tau(L) = \{\tau(l) \mid l \in L\}.$$

For a closed aggregate a of form (8) and its set E of aggregate elements, we have

$$\tau(a) = \{\tau(D)^\wedge \to \tau_a(D)^\vee \mid D \subseteq \text{Inst}(E), D \not\models a\}^\wedge, \tag{9}$$

where

$$\begin{aligned} \tau_a(D) &= \tau(\text{Inst}(E) \setminus D) \text{ for } D \subseteq \text{Inst}(E), \\ \tau(D) &= \{\tau(e) \mid e \in D\} \text{ for } D \subseteq \text{Inst}(E), \text{ and} \\ \tau(e) &= \tau(B(e))^\wedge \text{ for } e \in \text{Inst}(E). \end{aligned}$$

For a closed aggregate rule r, we have

$$\tau(r) = \tau(H(r)) \leftarrow \tau(B(r))^\wedge.$$

For an aggregate program P, we have

$$\tau(P) = \{\tau(r) \mid r \in \text{Inst}(P)\}. \tag{10}$$

While aggregate programs like P are finite sets of (non-ground) rules, $\tau(P)$ can be infinite and contain (ground) infinitary expressions. Observe that $\tau(P)$ is an $\mathcal{R}$-program. In fact, only the translation of aggregates introduces $\mathcal{R}$-formulas; rules without aggregates form $\mathcal{N}$-programs.

Example 11
To illustrate Ferraris' approach to the semantics of aggregates, consider a count aggregate a of form

$$\#\text{count}\{X : p(X)\} \geq n.$$

Since the aggregate is non-ground, the set G of its element's instances consists of all $t : p(t)$ for each ground term t.

The count aggregate cannot be justified by any subset D of G satisfying $|\{t \mid t : p(t) \in D\}| < n$, or $D \not\models a$ for short. Accordingly, we have that $\tau(a)$ is the conjunction of all formulas

$$\{p(t) \mid t : p(t) \in D\}^\wedge \to \{p(t) \mid t : p(t) \in (G \setminus D)\}^\vee \tag{11}$$

such that $D \subseteq G$ and $D \not\models a$. Restricting the set of ground terms to the numerals $1, 2, 3$ and letting $n = 2$ results in the formulas

$$\begin{aligned} \top &\to p(1) \vee p(2) \vee p(3), \\ p(1) &\to p(2) \vee p(3), \end{aligned}$$

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

$$p(2) \rightarrow p(1) \vee p(3), \text{ and}$$
$$p(3) \rightarrow p(1) \vee p(2).$$

Note that a smaller number of ground terms than n yields an unsatisfiable set of formulas.

However, it turns out that a Ferraris-style translation of aggregates (Ferraris 2011; Harrison *et al.* 2014) is too weak for propagating monotone aggregates in our ID-based setting. That is, when propagating possible atoms (i.e., the second component of the well-founded model), an ID-reduct may become satisfiable although the original formula is not. So, we might end up with too many possible atoms and a well-founded model that is not as precise as it could be. To see this, consider the following example.

Example 12
For some $m, n \geq 0$, the program $P_{m,n}$ consists of the following rules:

$$p(i) \leftarrow \neg q(i) \qquad \text{for } 1 \leq i \leq m$$
$$q(i) \leftarrow \neg p(i) \qquad \text{for } 1 \leq i \leq m$$
$$r \leftarrow \#\text{count}\{X : p(X)\} \geq n.$$

Given the ground instances G of the aggregate's elements and some two-valued interpretation I, observe that

$$\tau(\#\text{count}\{X : p(X)\} \geq n)_I$$

is classically equivalent to

$$\tau(\#\text{count}\{X : p(X)\} \geq n)_I \vee \{p(t) \in B(G) \mid p(t) \notin I\}^{\vee}. \qquad (12)$$

To see this, observe that the formula obtained via τ for the aggregate in the last rule's body consists of positive occurrences of implications of the form $\mathcal{G}^{\wedge} \rightarrow \mathcal{H}^{\vee}$ where either $p(t) \in \mathcal{G}$ or $p(t) \in \mathcal{H}$. The ID-reduct makes all such implications with some $p(t) \in \mathcal{G}$ such that $p(t) \notin I$ true because their antecedent is false. All of the remaining implications in the ID-reduct are equivalent to $\mathcal{H}^{\vee}$ where $\mathcal{H}$ contains all $p(t) \notin I$. Thus, we can factor out the formula on the right-hand side of (12).

Next, observe that for $1 \leq m < n$, the four-valued interpretation $(I, J) = (\emptyset, H(\tau(P_{m,n})))$ is the well-founded model of $P_{m,n}$:

$$S_{\tau(P_{m,n})}(J) = I \text{ and}$$
$$S_{\tau(P_{m,n})}(I) = J.$$

Ideally, atom r should not be among the possible atoms because it can never be in a stable model. Nonetheless, it is due to the second disjunct in (12).

Note that not just monotone aggregates exhibit this problem. In general, we get for a closed aggregate a with elements E and an interpretation I that

$$\tau(a)_I \text{ is classically equivalent to } \tau(a)_I \vee \{c \in B(\text{Inst}(E)) \mid I \not\models c\}^{\vee}.$$

The second disjunct is undesirable when propagating possible atoms.

To address this shortcoming, we augment the aggregate translation so that it provides stronger propagation. The result of the augmented translation is strongly equivalent to that of the original translation (cf. Proposition 23). Thus, even though we get more precise well-founded models, the stable models are still contained in them.

Definition 9
We define π as the translation obtained from τ by replacing the case of closed aggregates in (9) by the following:

For a closed aggregate a of form (8) and its set E of aggregate elements, we have

$$\pi(a) = \{\tau(D)^{\wedge} \rightarrow \pi_a(D)^{\vee} \mid D \subseteq \text{Inst}(E), D \not\triangleright a\}^{\wedge},$$

where

$$\pi_a(D) = \{\tau(C)^{\wedge} \mid C \subseteq \text{Inst}(E) \setminus D, C \cup D \triangleright a\} \text{ for } D \subseteq \text{Inst}(E).$$

Note that just as τ also π is recursively applied to the whole program.

Let us illustrate the modified translation by revisiting Example 11.

Example 13
Let us reconsider the count aggregate a:

$$\#\text{count}\{X : p(X)\} \geq n.$$

As with $\tau(a)$ in Example 11, $\pi(a)$ yields a conjunction of formulas, one conjunct for each set $D \subseteq \text{Inst}(E)$ satisfying $D \not\triangleright a$ of the form:

$$\{B(e) \mid e \in D\}^{\wedge} \rightarrow \left\{\{B(e) \mid e \in (C \setminus D)\}^{\wedge} \mid C \triangleright a, D \subseteq C \subseteq \text{Inst}(E)\right\}^{\vee}. \tag{13}$$

Restricting again the set of ground terms to the numerals $1, 2, 3$ and letting $n = 2$ results now in the formulas

$$\begin{aligned}
&\top \rightarrow (p(1) \wedge p(2)) \vee (p(1) \wedge p(3)) \vee (p(2) \wedge p(3)) \vee (p(1) \wedge p(2) \wedge p(3)),\\
&p(1) \rightarrow p(2) \vee p(3) \vee (p(2) \wedge p(3)),\\
&p(2) \rightarrow p(1) \vee p(3) \vee (p(1) \wedge p(3)), \text{ and}\\
&p(3) \rightarrow p(1) \vee p(2) \vee (p(1) \wedge p(2)).
\end{aligned}$$

Note that the last disjunct can be dropped from each rule's consequent. And as above, a smaller number of ground terms than n yields an unsatisfiable set of formulas.

The next result ensures that $\tau(P)$ and $\pi(P)$ have the same stable models for any aggregate program P.

Proposition 23
Let a be a closed aggregate.

Then, $\tau(a)$ and $\pi(a)$ are strongly equivalent.

The next example illustrates that we get more precise well-founded models using the strongly equivalent refined translation.

Example 14
Reconsider Program $P_{m,n}$ from Example 12.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

As above, we apply the well-founded operator to program $P_{m,n}$ for $m < n$ and four-valued interpretation $(I, J) = (\emptyset, H(\pi(P_{m,n})))$:

$$S_{\pi(P_{m,n})}(J) = I \text{ and}$$
$$S_{\pi(P_{m,n})}(I) = J \setminus \{r\}.$$

Unlike before, r is now found to be false since it does not belong to $S_{\pi(P_{m,n})}(\emptyset)$.

To see this, we can take advantage of the following proposition.

Proposition 24
Let a be a closed aggregate.

If a is monotone, then $\pi(a)_I$ is classically equivalent to $\pi(a)$ for any two-valued interpretation I.

Note that $\pi(a)$ is a negative formula whenever a is antimonotone; cf. Proposition 36.

Let us briefly return to Example 14. We now observe that $\pi(a)_I = \pi(a)$ for $a = \#\mathrm{count}\{X : p(X)\} \geq n$ and any interpretation I in view of the last proposition. Hence, our refined translation π avoids the problematic disjunct in Equation (12) on the right. By Proposition 23, we can use $\pi(P_{m,n})$ instead of $\tau(P_{m,n})$; both formulas have the same stable models.

Using Proposition 24, we augment the translation π to replace monotone aggregates a by the strictly positive formula $\pi(a)_\emptyset$. That is, we only keep the implication with the trivially true antecedent in the aggregate translation (cf. Section 5).

While π improves on propagation, it may still produce infinitary $\mathcal{R}$-formulas when applied to aggregates. This issue is addressed by restricting the translation to a set of (possible) atoms.

Definition 10
Let J be a two-valued interpretation. We define the translation π_J as the one obtained from τ by replacing the case of closed aggregates in (9) by the following:

For a closed aggregate a of form (8) and its set E of aggregate elements, we have

$$\pi_J(a) = \{\tau(D)^\wedge \to \pi_{a,J}(D)^\vee \mid D \subseteq \mathrm{Inst}(E)|_J, D \not\triangleright a\}^\wedge,$$

where

$$\pi_{a,J}(D) = \{\tau(C)^\wedge \mid C \subseteq \mathrm{Inst}(E)|_J \setminus D, C \cup D \triangleright a\} \text{ and}$$
$$\mathrm{Inst}(E)|_J = \{e \in \mathrm{Inst}(E) \mid B(e) \subseteq J\}.$$

Note that $\pi_J(a)$ is a finitary formula whenever J is finite.

Clearly, π_J also conforms to π except for the restricted translation for aggregates defined above. The next proposition elaborates this by showing that π_J and π behave alike whenever J limits the set of possible atoms.

Theorem 25
Let a be a closed aggregate, and $I \subseteq J$ and $X \subseteq J$ be two-valued interpretations.

Then,

(a) $X \models \pi(a)$ iff $X \models \pi_J(a)$,
(b) $X \models \pi(a)_I$ iff $X \models \pi_J(a)_I$, and
(c) $X \models \pi(a)^I$ iff $X \models \pi_J(a)^I$.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

In view of Proposition 23, this result extends to Ferraris' original aggregate translation (Ferraris 2011; Harrison *et al.* 2014).

The next example illustrates how a finitary formula can be obtained for an aggregate, despite a possibly infinite set of terms in the signature.

Example 15

Let $P_{m,n}$ be the program from Example 12. The well-founded model (I, J) of $\pi(P_{m,n})$ is $(\emptyset, H(\pi(P_{m,n})))$ if $n \leq m$.

The translation $\pi_J(P_{3,2})$ consists of the rules

$$\begin{aligned}
&p(1) \leftarrow \neg q(1), && q(1) \leftarrow \neg p(1),\\
&p(2) \leftarrow \neg q(2), && q(2) \leftarrow \neg p(2),\\
&p(3) \leftarrow \neg q(3), && q(3) \leftarrow \neg p(3), \text{ and}\\
&r \leftarrow \pi_J(\#\text{count}\{X : p(X)\} \geq 2), &&
\end{aligned}$$

where the aggregate translation corresponds to the conjunction of the formulas in Example 13. Note that the translation $\pi(P_{3,2})$ depends on the signature whereas the translation $\pi_J(P_{3,2})$ is fixed by the atoms in J.

Importantly, Proposition 25 shows that given a finite (approximation of the) well-founded model of an $\mathcal{R}$-program, we can replace aggregates with finitary formulas. Moreover, in this case, Theorem 9 and Proposition 23 together indicate how to turn a program with aggregates into a semantically equivalent finite $\mathcal{R}$-program with finitary formulas as bodies. That is, given a finite well-founded model of an $\mathcal{R}$-program, the program simplification from Definition 1 results in a finite program and the aggregate translation from Definition 10 produces finitary formulas only.

This puts us in a position to outline how and when (safe non-ground) aggregate programs can be turned into equivalent finite ground programs consisting of finitary subformulas only. To this end, consider an aggregate program P along with the well-founded model (I, J) of $\pi(P)$. We have already seen in Corollary 10 that $\pi(P)$ and its simplification $\pi(P)^{I,J}$ have the same stable models, just like $\pi(P)^{I,J}$ and its counterpart $\pi_J(P)^{I,J}$ in view of Proposition 25.

Now, if (I, J) is finite, then $\pi(P)^{I,J}$ is finite, too. Seen from the perspective of grounding, the safety of all rules in P implies that all global variables appear in positive body literals. Thus, the number of ground instances of each rule in $\pi(P)^{I,J}$ is determined by the number of possible substitutions for its global variables. Clearly, there are only finitely many possible substitutions such that all positive body literals are satisfied by a finite interpretation J (cf. Definition 1). Furthermore, if J is finite, aggregate translations in $\pi_J(P)^{I,J}$ introduce finitary subformulas only. Thus, in this case, we obtain from P a finite set of rules with finitary propositional formulas as bodies, viz. $\pi_J(P)^{I,J}$, that has the same stable models as $\pi(P)$ (as well as $\tau(P)$, the traditional Ferraris-style semantics of P (Ferraris 2011; Harrison *et al.* 2014)).

An example of a class of aggregate programs inducing finite well-founded models as above consists of programs over a signature with nullary function symbols only. Any such program can be turned into an equivalent finite set of propositional rules with finitary bodies.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

5 Dependency analysis

We now further refine our semantic approach to reflect actual grounding processes. In fact, modern grounders process programs on-the-fly by grounding one rule after another without storing any rules. At the same time, they try to determine certain, possible, and false atoms. Unfortunately, well-founded models cannot be computed on-the-fly, which is why we introduce below the concept of an approximate model. More precisely, we start by defining instantiation sequences of (non-ground) aggregate programs based on their rule dependencies. We show that approximate models of instantiation sequences are underapproximations of the well-founded model of the corresponding sequence of (ground) $\mathcal{R}$-programs, as defined in Section 3. The precision of both types of models coincides on stratified programs. We illustrate our concepts comprehensively at the end of this section in Examples 19 and 20.

To begin with, we extend the notion of positive and negative literals to aggregate programs. For atoms a, we define $a^+ = (\neg a)^- = \{a\}$ and $a^- = (\neg a)^+ = \emptyset$. For comparisons a, we define $a^+ = a^- = \emptyset$. For aggregates a with elements E, we define positive and negative atom occurrences, using Proposition 24 to refine the case for monotone aggregates:

- $a^+ = \bigcup_{e \in E} B(e)$,
- $a^- = \emptyset$ if a is monotone, and
- $a^- = \bigcup_{e \in E} B(e)$ if a is not monotone.

For a set of body literals B, we define $B^+ = \bigcup_{b \in B} b^+$ and $B^- = \bigcup_{b \in B} b^-$, as well as $B^\pm = B^+ \cup B^-$.

We see in the following, that a special treatment of monotone aggregates yields better approximations of well-founded models. A similar case could be made for antimonotone aggregates but had led to a more involved algorithmic treatment.

Inter-rule dependencies are determined via the predicates appearing in their heads and bodies. We define $\text{pred}(a)$ to be the predicate symbol associated with atom a and $\text{pred}(A) = \{\text{pred}(a) \mid a \in A\}$ for a set A of atoms. An aggregate rule r_1 *depends* on another aggregate rule r_2 if $\text{pred}(H(r_2)) \in \text{pred}(B(r_1)^\pm)$. Rule r_1 depends *positively* or *negatively* on r_2 if $\text{pred}(H(r_2)) \in \text{pred}(B(r_1)^+)$ or $\text{pred}(H(r_2)) \in \text{pred}(B(r_2)^-)$, respectively.

For simplicity, we first focus on programs without aggregates in examples and delay a full example with aggregates until the end of the section.

Example 16
Let us consider the following rules from the introductory example:

$$u(1) \tag{r_1}$$

$$p(X) \leftarrow \neg q(X) \wedge u(X) \tag{r_2}$$

$$q(X) \leftarrow \neg p(X) \wedge v(X). \tag{r_3}$$

We first determine the rule heads and positive and negative atom occurrences in rule bodies:

$$\begin{aligned} H(r_1) &= u(1) & B(r_1)^+ &= \emptyset & B(r_1)^- &= \emptyset \\ H(r_2) &= p(X) & B(r_2)^+ &= \{u(X)\} & B(r_2)^- &= \{q(X)\} \\ H(r_3) &= q(X) & B(r_3)^+ &= \{v(X)\} & B(r_3)^- &= \{p(X)\}. \end{aligned}$$

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

With this, we infer the corresponding predicates:

$$
\begin{array}{lll}
\mathrm{pred}(H(r_1)) = u/1 & \mathrm{pred}(B(r_1)^+) = \emptyset & \mathrm{pred}(B(r_1)^-) = \emptyset \\
\mathrm{pred}(H(r_2)) = p/1 & \mathrm{pred}(B(r_2)^+) = \{u/1\} & \mathrm{pred}(B(r_2)^-) = \{q/1\} \\
\mathrm{pred}(H(r_3)) = q/1 & \mathrm{pred}(B(r_3)^+) = \{v/1\} & \mathrm{pred}(B(r_3)^-) = \{p/1\}.
\end{array}
$$

We see that r_2 depends positively on r_1 and that r_2 and r_3 depend negatively on each other. View Figure 1 in Example 17 for a graphical representation of these inter-rule dependencies.

The *strongly connected components* of an aggregate program P are the equivalence classes under the transitive closure of the dependency relation between all rules in P. A strongly connected component P_1 *depends* on another strongly connected component P_2 if there is a rule in P_1 that depends on some rule in P_2. The transitive closure of this relation is antisymmetric.

A strongly connected component of an aggregate program is *unstratified* if it depends negatively on itself or if it depends on an unstratified component. A component is *stratified* if it is not unstratified.

A topological ordering of the strongly connected components is then used to guide grounding.

For example, the sets $\{r_1\}$ and $\{r_2, r_3\}$ of rules from Example 16 are strongly connected components in a topological order. There is only one topological order because r_2 depends on r_1. While the first component is stratified, the second component is unstratified because r_2 and r_3 depend negatively on each other.

Definition 11

We define an *instantiation sequence* for P as a sequence $(P_i)_{i\in\mathbb{I}}$ of its strongly connected components such that $i < j$ if P_j depends on P_i.

Note that the components can always be well ordered because aggregate programs consist of finitely many rules.

The consecutive construction of the well-founded model along an instantiation sequence results in the well-founded model of the entire program.

Theorem 26

Let $(P_i)_{i\in\mathbb{I}}$ be an instantiation sequence for aggregate program P.

Then, $\mathit{WM}((\pi(P_i))_{i\in\mathbb{I}}) = \mathit{WM}(\pi(P))$.

Example 17

The following example shows how to split an aggregate program into an instantiation sequence and gives its well-founded model. Let P be the following aggregate program, extending the one from the introductory section:

$$
\begin{array}{cc}
u(1) & u(2) \\
v(2) & v(3) \\
p(X) \leftarrow \neg q(X) \wedge u(X) & q(X) \leftarrow \neg p(X) \wedge v(X) \\
x \leftarrow \neg p(1) & y \leftarrow \neg q(3).
\end{array}
$$

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

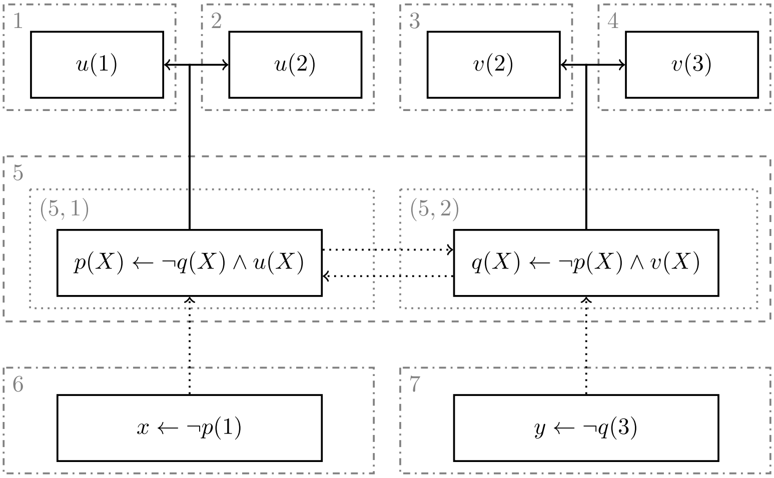

Fig. 1. Rule dependencies for Example 19.

We have already seen how to determine inter-rule dependencies in Example 16. A possible instantiation sequence for program P is given in Figure 1. Rules are depicted in solid boxes. Solid and dotted edges between such boxes depict positive and negative dependencies between the corresponding rules, respectively. Dashed and dashed/dotted boxes represent components in the instantiation sequence (we ignore dotted boxes for now but turn to them in Example 18). The number in the corner of a component box indicates the index in the corresponding instantiation sequence.

For $F = \{u(1), u(2), v(2), v(3)\}$, the well-founded model of $\pi(P)$ is

$$WM(\pi(P)) = (\{p(1), q(3)\}, \{p(1), p(2), q(2), q(3)\}) \sqcup F.$$

By Theorem 26, the ground sequence $(\tau(P_i))_{i\in\mathbb{I}}$ has the same well-founded model as $\pi(P)$:

$$WM((\pi(P))_{i\in\mathbb{I}}) = (\{p(1), q(3)\}, \{p(1), p(2), q(2), q(3)\}) \sqcup F.$$

Note that the set F comprises the facts derived from stratified components. In fact, for stratified components, the set of external atoms (3) is empty. We can use Corollary 20 to confirm that the well founded model (F, F) of sequence $(\pi(P_i))_{1\leq i\leq 4}$ is total. In fact, each of the intermediate interpretations (5) is total and can be computed with just one application of the stable operator. For example, $I_1 = J_1 = \{u(1)\}$ for component P_1.

We further refine instantiation sequences by partitioning each component along its positive dependencies.

Definition 12

Let P be an aggregate program and $(P_i)_{i\in\mathbb{I}}$ be an instantiation sequence for P. Furthermore, for each $i \in \mathbb{I}$, let $(P_{i,j})_{j\in\mathbb{I}_i}$ be an instantiation sequence of P_i considering positive dependencies only.

A *refined instantiation sequence* for P is a sequence $(P_{i,j})_{(i,j)\in\mathbb{J}}$ where the index set $\mathbb{J} = \{(i,j) \mid i \in \mathbb{I}, j \in \mathbb{I}_i\}$ is ordered lexicographically.

We call $(P_{i,j})_{(i,j)\in\mathbb{J}}$ a refinement of $(P_i)_{i\in\mathbb{I}}$.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

We define a component $P_{i,j}$ to be *stratified* or *unstratified* if the encompassing component P_i is stratified or unstratified, respectively.

Examples of refined instantiation sequences are given in Figures 1 and 2.

The advantage of such refinements is that they yield better or equal approximations (cf. Theorem 28 and Example 19). On the downside, we do not obtain that $WM((P_i)_{i\in\mathbb{J}})$ equals $WM(\pi(P))$ for refined instantiation sequences in general.

Example 18

The refined instantiation sequence for program P from Example 17 is given in Figure 1. A dotted box indicates a component in a refined instantiation sequence. Components that cannot be refined further are depicted with a dashed/dotted box. The number or pair in the corner of a component box indicates the index in the corresponding refined instantiation sequence.

Unlike in Example 17, the well-founded model of the refined sequence of ground programs $(\tau(P_{i,j}))_{(i,j)\in\mathbb{J}}$ is

$$WM((\tau(P_{i,j}))_{(i,j)\in\mathbb{J}}) = (\{q(3)\}, \{p(1), p(2), q(2), q(3), x\}) \sqcup F,$$

which is actually less precise than the well-founded model of P. This is because literals over $\neg q(X)$ are unconditionally assumed to be true because their instantiation is not yet available when $P_{5,1}$ is considered. Thus, we get $(I_{5,1}, J_{5,1}) = (\emptyset, \{p(1), p(2)\})$ for the intermediate interpretation (5). Unlike this, the atom $p(3)$ is false when considering component $P_{5,2}$ and $q(3)$ becomes true. In fact, we get $(I_{5,2}, J_{5,2}) = (\{q(3)\}, \{q(2), q(3)\})$. Observe that (I_5, J_5) from above is less precise than $(I_{5,1}, J_{5,1}) \sqcup (I_{5,2}, J_{5,2})$.

We continue with this example below and show in Example 19 that refined instantiation sequences can still be advantageous to get better approximations of well-founded models.

We have already seen in Section 3 that external atoms may lead to less precise semantic characterizations. This is just the same in the non-ground case, whenever a component comprises predicates that are defined in a following component of a refined instantiation sequence. This leads us to the concept of an approximate model obtained by overapproximating the extension of such externally defined predicates.

Definition 13

Let P be an aggregate program, (IC, JC) be a four-valued interpretation, $\mathcal{E}$ be a set of predicates, and P' be the program obtained from P by removing all rules r with $\mathrm{pred}(B(r)^-) \cap \mathcal{E} \neq \emptyset$.

We define the *approximate model of* P *relative* to (IC, JC) as

$$AM_{\mathcal{E}}^{(IC,JC)}(P) = (I, J),$$

where

$$I = S_{\pi(P')}^{IC}(JC) \text{ and}$$
$$J = S_{\pi(P)}^{JC}(IC \cup I).$$

We keep dropping parentheses and simply write $AM_{\mathcal{E}}^{IC,JC}(P)$ instead of $AM_{\mathcal{E}}^{(IC,JC)}(P)$.

The approximate model amounts to an immediate consequence operator, similar to the relative well-founded operator in Definition 4; it refrains from any iterative applications,

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

as used for defining a well-founded model. More precisely, the relative stable operator is applied twice to obtain the approximate model. This is similar to Van Gelder's alternating transformation (Van Gelder 1993). The certain atoms in I are determined by applying the operator to the ground program obtained after removing all rules whose negative body literals comprise externally defined predicates, while the possible atoms J are computed from the entire program by taking the already computed certain atoms in I into account. In this way, the approximate model may result in fewer unknown atoms than the relative well-founded operator when applied to the least precise interpretation (as an easy point of reference). How well we can approximate the certain atoms with the approximate operator depends on the set of external predicates $\mathcal{E}$. When approximating the model of a program P in a sequence, the set $\mathcal{E}$ comprises all negative predicates occurring in P for which possible atoms have not yet been fully computed. This leads to fewer certain atoms obtained from the reduced program, $P' = \{r \in P \mid \mathrm{pred}(B(r)^-) \cap \mathcal{E} = \emptyset\}$, stripped of all rules from P that have negative body literals whose predicates occur in $\mathcal{E}$.

The next theorem identifies an essential prerequisite for an approximate model of a non-ground program to be an underapproximation of the well-founded model of the corresponding ground program.

Theorem 27
Let P be an aggregate program, $\mathcal{E}$ be a set of predicates, and (IC, JC) be a four-valued interpretation.

If $\mathrm{pred}(H(P)) \cap \mathrm{pred}(B(P)^-) \subseteq \mathcal{E}$ then $AM_{\mathcal{E}}^{IC,JC}(P) \leq_p WM^{IC,JC \cup EC}(\pi(P))$ where EC is the set of all ground atoms over predicates in $\mathcal{E}$.

In general, a grounder cannot calculate on-the-fly a well-founded model. Implementing this task efficiently requires an algorithm storing the grounded program, as, for example, implemented in an ASP solver. But modern grounders are able to calculate the stable operator on-the-fly. Thus, an approximation of the well-founded model is calculated. This is where we use the approximate model, which might be less precise than the well-founded model but can be computed more easily.

With the condition of Theorem 27 in mind, we define the approximate model for an instantiation sequence. We proceed similar to Definition 6 but treat in (14) all atoms over negative predicates that have not been completely defined as external.

Definition 14
Let $(P_i)_{i \in \mathbb{I}}$ be a (refined) instantiation sequence for P.

Then, the *approximate model* of $(P_i)_{i \in \mathbb{I}}$ is

$$AM((P_i)_{i \in \mathbb{I}}) = \bigsqcup_{i \in \mathbb{I}} (I_i, J_i),$$

where

$$\mathcal{E}_i = \mathrm{pred}(B(P_i)^-) \cap \mathrm{pred}(\bigcup_{i \leq j} H(P_j)), \tag{14}$$

$$(IC_i, JC_i) = \bigsqcup_{j < i} (I_j, J_j), \text{ and} \tag{15}$$

$$(I_i, J_i) = AM_{\mathcal{E}_i}^{IC_i, JC_i}(P_i). \tag{16}$$

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Note that the underlying approximate model removes rules containing negative literals over predicates in $\mathcal{E}_i$ when calculating certain atoms. This amounts to assuming all ground instances of atoms over $\mathcal{E}_i$ to be possible.[7] Compared to (3), however, this additionally includes recursive predicates in (14). The set $\mathcal{E}_i$ is empty for stratified components.

The next result relies on Theorem 17 to show that an approximate model of an instantiation sequence constitutes an underapproximation of the well-founded model of the translated entire program. In other words, the approximate model of a sequence of aggregate programs (as computed by a grounder) is less precise than the well-founded model of the whole ground program.

Theorem 28
Let $(P_i)_{i\in\mathbb{I}}$ be an instantiation sequence for aggregate program P and $(P_j)_{j\in\mathbb{J}}$ be a refinement of $(P_i)_{i\in\mathbb{I}}$.

Then, $AM((P_i)_{i\in\mathbb{I}}) \leq_p AM((P_j)_{j\in\mathbb{J}}) \leq_p WM(\pi(P))$.

The finer granularity of refined instantiation sequences leads to more precise models. Intuitively, this is because a refinement of a component may result in a series of approximate models, which yield a more precise result than the approximate model of the entire component because in some cases fewer predicates are considered external in (14).

We remark that all instantiation sequences of a program have the same approximate model. However, this does not carry over to refined instantiation sequences because their evaluation is order dependent.

The two former issues are illustrated in Example 19.

The actual value of approximate models for grounding lies in their underlying series of consecutive interpretations delineating each ground program in a (refined) instantiation sequence. In fact, as outlined after Proposition 25, whenever all interpretations (I_i, J_i) in (16) are finite so are the $\mathcal{R}$-programs $\pi_{JC_i \cup J_i}(P_i)^{(IC_i,JC_i)\sqcup(I_i,J_i)}$ obtained from each P_i in the instantiation sequence.

Theorem 29
Let $(P_i)_{i\in\mathbb{I}}$ be a (refined) instantiation sequence of an aggregate program P, and let (IC_i, JC_i) and (I_i, J_i) be defined as in (15) and (16).

Then, $\bigcup_{i\in\mathbb{I}} \pi_{JC_i \cup J_i}(P_i)^{(IC_i,JC_i)\sqcup(I_i,J_i)}$ and $\pi(P)$ have the same well-founded and stable models.

Notably, this union of $\mathcal{R}$-programs is exactly the one obtained by the grounding algorithm proposed in the next section (cf. Theorem 34).

Example 19
We continue Example 18.

The approximate model of the instantiation sequence $(P_i)_{i\in\mathbb{I}}$, defined in Definition 14, is less precise than the well-founded model of the sequence, viz.

$$AM((P_i)_{i\in\mathbb{I}}) = (\emptyset, \{p(1), p(2), q(2), q(3), x, y\}) \sqcup F.$$

[7] To be precise, rules involving aggregates that could in principle derive certain atoms might be removed, too. Here, we are interested in a syntactic criteria that allows us to underapproximate the set of certain atoms.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

This is because we have to use $AM^{F,F}_{\mathcal{E}}(P_5)$ to approximate the well-founded model of component P_5. Here, the set $\mathcal{E} = \{a/1, b/1\}$ determined by Equation (14) forces us to unconditionally assume instances of $\neg q(X)$ and $\neg p(X)$ to be true. Thus, we get $(I_5, J_5) = (\emptyset, \{p(1), p(2), q(2), q(3)\})$ for the intermediate interpretation in (16). This is also reflected in Definition 13, which makes us drop all rules containing negative literals over predicates in $\mathcal{E}$ when calculating true atoms.

In accord with Theorem 28, we approximate the well-founded model w.r.t. the refined instantiation sequence $(P_{i,j})_{(i,j)\in\mathbb{J}}$ and obtain

$$AM((P_{i,j})_{(i,j)\in\mathbb{J}}) = (\{q(3)\}, \{p(1), p(2), q(2), q(3), x\}) \sqcup F,$$

which, for this example, is equivalent to the well-founded model of the corresponding ground refined instantiation sequence and more precise than the approximate model of the instantiation sequence.

In an actual grounder implementation the approximate model is only a byproduct, instead, it outputs a program equivalent to the one in Theorem 29:

$$\begin{array}{ll}
u(1) & u(2) \\
v(2) & v(3) \\
p(1) \leftarrow \neg q(1) \land u(1) & p(2) \leftarrow \neg q(2) \land u(2) \\
q(2) \leftarrow \neg p(2) \land v(2) & q(3) \leftarrow \neg p(3) \land v(3) \\
\quad x \leftarrow \neg p(1). &
\end{array}$$

Note that the rule $y \leftarrow \neg q(3)$ is not part of the simplification because $q(3)$ is certain.

Remark 1

The reason why we use the refined grounding is that we cannot expect a grounding algorithm to calculate the well-founded model for a component without further processing. But at least some consequences should be considered. *Gringo* is designed to ground on-the-fly without storing any rules, so it cannot be expected to compute all possible consequences but it should at least take all consequences from preceding interpretations into account. With the help of a solver, we could calculate the exact well-founded model of a component after it has been grounded.

Whenever an aggregate program is stratified, the approximate model of its instantiation sequence is total (and coincides with the well-founded model of the entire ground program).

Theorem 30

Let $(P_i)_{i\in\mathbb{I}}$ be an instantiation sequence of an aggregate program P such that $\mathcal{E}_i = \emptyset$ for each $i \in \mathbb{I}$ as defined in (14).

Then, $AM((P_i)_{i\in\mathbb{I}})$ is total.

Example 20

The dependency graph of the company controls encoding is given in Figure 2 and follows the conventions in Example 19. Because the encoding only uses positive literals and monotone aggregates, grounding sequences cannot be refined further. Since the program is positive, we can apply Theorem 30. Thus, the approximate model of the grounding

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

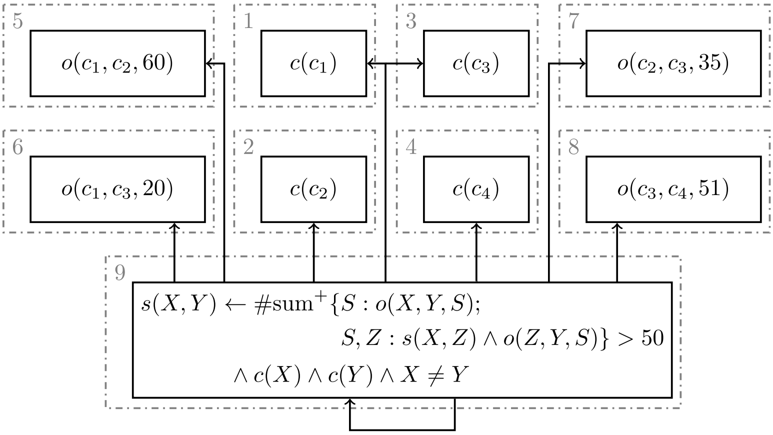
5
$o(c_1, c_2, 60)$
1
$c(c_1)$
3
$c(c_3)$
7
$o(c_2, c_3, 35)$
6
$o(c_1, c_3, 20)$
2
$c(c_2)$
4
$c(c_4)$
8
$o(c_3, c_4, 51)$
9
$s(X, Y) \leftarrow \#\text{sum}^+\{S : o(X, Y, S); S, Z : s(X, Z) \wedge o(Z, Y, S)\} > 50 \wedge c(X) \wedge c(Y) \wedge X \neq Y$

Fig. 2. Rule dependencies for the company controls encoding and instance in Example 10 where $c = \mathit{company}$, $o = \mathit{owns}$, and $s = \mathit{controls}$.

sequence is total and corresponds to the well-founded model of the program. We use the same abbreviations for predicates as in Figure 2. The well-founded model is $(F \cup I, F \cup I)$ where

$$\begin{aligned} F = \{&c(c_1), c(c_2), c(c_3), c(c_4),\\ &o(c_1, c_2, 60), o(c_1, c_3, 20), o(c_2, c_3, 35), o(c_3, c_4, 51)\} \text{ and}\\ I = \{&s(c_1, c_2), s(c_3, c_4), s(c_1, c_3), s(c_1, c_4)\}. \end{aligned}$$

6 Algorithms

This section lays out the basic algorithms for grounding rules, components, and entire programs and characterizes their output in terms of the semantic concepts developed in the previous sections. Of particular interest is the treatment of aggregates, which are decomposed into dedicated normal rules before grounding and reassembled afterward. This allows us to ground rules with aggregates by means of grounding algorithms for normal rules. Finally, we show that our grounding algorithm guarantees that an obtained finite ground program is equivalent to the original non-ground program.

In the following, we refer to terms, atoms, comparisons, literals, aggregate elements, aggregates, or rules as *expressions*. As in the preceding sections, all expressions, interpretations, and concepts introduced below operate on the same (implicit) signature Σ unless mentioned otherwise.

A *substitution* is a mapping from the variables in Σ to terms over Σ. We use ι to denote the identity substitution mapping each variable to itself. A *ground substitution* maps all variables to ground terms or themselves. The result of *applying* a substitution σ to an expression e, written $e\sigma$, is the expression obtained by replacing each variable v in e by $\sigma(v)$. This directly extends to sets E of expressions, that is, $E\sigma = \{e\sigma \mid e \in E\}$.

The *composition* of substitutions σ and θ is the substitution $\sigma \circ \theta$ where $(\sigma \circ \theta)(v) = \theta(\sigma(v))$ for each variable v.

A substitution σ is a *unifier* of a set E of expressions if $e_1\sigma = e_2\sigma$ for all $e_1, e_2 \in E$. In what follows, we are interested in one-sided unification, also called *matching*. A

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

```
function $\texttt{GroundRule}^{I,J}_{r,f,J'}(\sigma, L)$
    if $L \neq \emptyset$ then                                                    // match next
        $(G, l) \leftarrow (\emptyset, \texttt{Select}_\sigma(L))$;
        foreach $\sigma' \in \texttt{Matches}^{I,J}_l(\sigma)$ do
            $G \leftarrow G \cup \texttt{GroundRule}^{I,J}_{r,f,J'}(\sigma', L \setminus \{l\})$;
        return $G$;
    else if $f = \mathbf{t}$ or $B(r\sigma)^+ \not\subseteq J'$ then                // rule instance
        return $\{r\sigma\}$;
    else                                                                           // rule seen
        return $\emptyset$;
```

Algorithm 1: Grounding Rules

substitution σ matches a non-ground expression e to a ground expression g, if $e\sigma = g$ and σ maps all variables not occurring in e to themselves. We call such a substitution the *matcher* of e to g. Note that a matcher is a unique ground substitution unifying e and g, if it exists. This motivates the following definition.

For a (non-ground) expression e and a ground expression g, we define:

$$\texttt{match}(e, g) = \begin{cases} \{\sigma\} & \text{if there is a matcher } \sigma \text{ from } e \text{ to } g \\ \emptyset & \text{otherwise} \end{cases}$$

When grounding rules, we look for matches of non-ground body literals in the possibly derivable atoms accumulated so far. The latter is captured by a four-valued interpretation to distinguish certain atoms among the possible ones. This is made precise in the next definition.

Definition 15

Let σ be a substitution, l be a literal or comparison, and (I, J) be a four-valued interpretation.

We define the set of *matches* for l in (I, J) w.r.t. σ, written $\texttt{Matches}^{I,J}_l(\sigma)$,

for an atom $l = a$ as

$$\texttt{Matches}^{I,J}_a(\sigma) = \{\sigma \circ \sigma' \mid a' \in J, \sigma' \in \texttt{match}(a\sigma, a')\},$$

for a ground literal $l = \neg a$ as

$$\texttt{Matches}^{I,J}_{\neg a}(\sigma) = \{\sigma \mid a\sigma \notin I\}, \text{ and}$$

for a ground comparison $l = t_1 \prec t_2$ as in (6) as

$$\texttt{Matches}^{I,J}_{t_1 \prec t_2}(\sigma) = \{\sigma \mid \prec \text{ holds between } t_1\sigma \text{ and } t_2\sigma\}.$$

In this way, positive body literals yield a (possibly empty) set of substitutions, refining the one at hand, while negative and comparison literals are only considered when ground and then act as a test on the given substitution.

Our function for rule instantiation is given in Algorithm 1. It takes a substitution σ and a set L of literals and yields a set of ground instances of a safe normal rule r, passed

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

as a parameter; if called with the identity substitution and the body literals $B(r)$ of r, it yields ground instances of the rule. The other parameters consist of a four-valued interpretation (I, J) comprising the set of possibly derivable atoms along with the certain ones, a two-valued interpretation J' reflecting the previous value of J, and a Boolean flag f used to avoid duplicate ground rules in consecutive calls to Algorithm 1. The idea is to extend the current substitution in Lines 4–5 until we obtain a ground substitution σ that induces a ground instance $r\sigma$ of rule r. To this end, $\mathtt{Select}_\sigma(L)$ picks for each call some literal $l \in L$ such that $l \in L^+$ or $l\sigma$ is ground. That is, it yields either a positive body literal or a ground negative or ground comparison literal, as needed for computing $\mathtt{Matches}_l^{I,J}(\sigma)$. Whenever an application of $\mathtt{Matches}$ for the selected literal in $B(r)$ results in a non-empty set of substitutions, the function is called recursively for each such substitution. The recursion terminates if at least one match is found for each body literal and an instance $r\sigma$ of r is obtained in Line 8. The set of all such ground instances is returned in Line 6. (Note that we refrain from applying any simplifications to the ground rules and rather leave them intact to obtain more direct formal characterizations of the results of our grounding algorithms.) The test $B(r\sigma)^+ \not\subseteq J'$ in Line 7 makes sure that no ground rules are generated that were already obtained by previous invocations of Algorithm 1. This is relevant for recursive rules and reflects the approach of semi-naive database evaluation (Abiteboul *et al.* 1995).

For characterizing the result of Algorithm 1 in terms of aggregate programs, we need the following definition.

Definition 16

Let P be an aggregate program and (I, J) be a four-valued interpretation.

We define $\operatorname{Inst}^{I,J}(P) \subseteq \operatorname{Inst}(P)$ as the set of all instances g of rules in P satisfying $J \models \pi(B(g))^\wedge_I$.

In terms of the program simplification in Definition 1, an instance g belongs to $\operatorname{Inst}^{I,J}(P)$ iff $H(g) \leftarrow \pi(B(g))^\wedge \in \pi(r)^{I,J}$. Note that the members of $\operatorname{Inst}^{I,J}(P)$ are not necessarily ground, since non-global variables may remain within aggregates; though they are ground for normal rules.

We use Algorithm 1 to iteratively compute ground instances of a rule w.r.t. an increasing set of atoms. The Boolean flag f and the set of atoms J' are used to avoid duplicating ground instances in successive iterations. The flag f is initially set to true to not filter any rule instances. In subsequent iterations, duplicates are omitted by setting the flag to false and filtering rules whose positive bodies are a subset of the atoms J' used in previous iterations. This is made precise in the next result.

Proposition 31

Let r be a safe normal rule and (I, J) be a finite four-valued interpretation.

Then,

(a) $\operatorname{Inst}^{I,J}(\{r\}) = \mathtt{GroundRule}^{I,J}_{r,\mathbf{t},\emptyset}(\iota, B(r))$ and

(b) $\operatorname{Inst}^{I,J}(\{r\}) = \operatorname{Inst}^{I,J'}(\{r\}) \cup \mathtt{GroundRule}^{I,J}_{r,\mathbf{f},J'}(\iota, B(r))$ for all $J' \subseteq J$.

Now, let us turn to the treatment of aggregates. To this end, we define the following translation of aggregate programs to normal programs.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Definition 17

Let P be a safe aggregate program over signature Σ.

Let Σ' be the signature obtained by extending Σ with fresh predicates

$$\alpha_{a,r}/n, \text{ and} \tag{17}$$

$$\epsilon_{a,r}/n \tag{18}$$

for each aggregate a occurring in a rule $r \in P$ where n is the number of global variables in a, and fresh predicates

$$\eta_{e,a,r}/(m+n) \tag{19}$$

for each aggregate element e occurring in aggregate a in rule r where m is the size of the tuple $H(e)$.

We define P^α, P^ϵ, and P^η as normal programs over Σ' as follows.

- Program P^α is obtained from P by replacing each aggregate occurrence a in P with

  $$\alpha_{a,r}(X_1, \dots, X_n), \tag{20}$$

  where $\alpha_{a,r}/n$ is defined as in (17) and $X_1, \dots, X_n$ are the global variables in a.
- Program P^ϵ consists of rules

  $$\epsilon_{a,r}(X_1, \dots, X_n) \leftarrow t \prec b \wedge b_1 \wedge \dots \wedge b_l \tag{21}$$

  for each predicate $\epsilon_{a,r}/n$ as in (18) where $X_1, \dots, X_n$ are the global variables in a, a is an aggregate of form $f\{E\} \prec b$ occurring in r, $t = f(\emptyset)$ is the value of the aggregate function applied to the empty set, and $b_1, \dots, b_l$ are the body literals of r excluding aggregates.
- Program P^η consists of rules

  $$\eta_{e,a,r}(t_1, \dots, t_m, X_1, \dots, X_n) \leftarrow c_1 \wedge \dots \wedge c_k \wedge b_1 \wedge \dots \wedge b_l \tag{22}$$

  for each predicate $\eta_{e,a,r}/m+n$ as in (19) where $(t_1, \dots, t_m) = H(e)$, $X_1, \dots, X_n$ are the global variables in a, $\{c_1, \dots, c_k\} = B(e)$, and $b_1, \dots, b_l$ are the body literals of r excluding aggregates.

Summarizing the above, we translate an aggregate program P over Σ into a normal program P^α along with auxiliary normal rules in P^ϵ and P^η, all over a signature extending Σ by the special-purpose predicates in (17)–(19). In fact, there is a one-to-one correspondence between the rules in P and P^α, so that we get $P = P^\alpha$ and $P^\epsilon = P^\eta = \emptyset$ whenever P is normal.

Example 21

We illustrate the translation of aggregate programs on the company controls example in Example 10. We rewrite the rule

$$\mathit{controls}(X,Y) \leftarrow \overbrace{\#\mathrm{sum}^+\{\underbrace{S : \mathit{owns}(X,Y,S)}_{e_1}; \underbrace{S,Z : \mathit{controls}(X,Z) \wedge \mathit{owns}(Z,Y,S)}_{e_2}\} > 50}^{a} \wedge \mathit{company}(X) \wedge \mathit{company}(Y) \wedge X \neq Y \tag{r}$$

containing aggregate a with elements e_1 and e_2, into rules r_1 to r_4:

$$\begin{aligned} \mathit{controls}(X,Y) \leftarrow{}& \alpha_{a,r}(X,Y) \\ & \wedge \mathit{company}(X) \wedge \mathit{company}(Y) \wedge X \neq Y, \end{aligned} \tag{r_1}$$

$$\begin{aligned} \epsilon_{a,r}(X,Y) \leftarrow{}& 0 > 50 \\ & \wedge \mathit{company}(X) \wedge \mathit{company}(Y) \wedge X \neq Y, \end{aligned} \tag{r_2}$$

$$\begin{aligned} \eta_{e_1,a,r}(S,X,Y) \leftarrow{}& \mathit{owns}(X,Y,S) \\ & \wedge \mathit{company}(X) \wedge \mathit{company}(Y) \wedge X \neq Y, \text{ and} \end{aligned} \tag{r_3}$$

$$\begin{aligned} \eta_{e_2,a,r}(S,Z,X,Y) \leftarrow{}& \mathit{controls}(X,Z) \wedge \mathit{owns}(Z,Y,S) \\ & \wedge \mathit{company}(X) \wedge \mathit{company}(Y) \wedge X \neq Y. \end{aligned} \tag{r_4}$$

We have $P^\alpha = \{r_1\}$, $P^\epsilon = \{r_2\}$, and $P^\eta = \{r_3, r_4\}$.

This example illustrates how possible instantiations of aggregate elements are gathered via the rules in P^η. Similarly, the rules in P^ϵ collect instantiations warranting that the result of applying aggregate functions to the empty set is in accord with the respective bound. In both cases, the relevant variable bindings are captured by the special head atoms of the rules. In turn, groups of corresponding instances of aggregate elements are used in Definition 20 to sanction the derivation of ground atoms of form (20). These atoms are ultimately replaced in P^α with the original aggregate contents.

We next define two functions gathering information from instances of rules in P^ϵ and P^η. In particular, we make precise how groups of aggregate element instances are obtained from ground rules in P^η.

Definition 18

Let P be an aggregate program, and G^ϵ and G^η be subsets of ground instances of rules in P^ϵ and P^η, respectively. Furthermore, let a be an aggregate occurring in some rule $r \in P$ and σ be a substitution mapping the global variables in a to ground terms.

We define

$$\epsilon_{r,a}(G^\epsilon, \sigma) = \bigcup_{g \in G^\epsilon} \mathtt{match}(r_a\sigma, g),$$

where r_a is a rule of form (21) for aggregate occurrence a, and

$$\eta_{r,a}(G^\eta, \sigma) = \{e\sigma\theta \mid g \in G^\eta, e \in E, \theta \in \mathtt{match}(r_e\sigma, g)\},$$

where E are the aggregate elements of a and r_e is a rule of form (22) for aggregate element occurrence e in a.

Given that σ maps the global variables in a to ground terms, $r_a\sigma$ is ground whereas $r_e\sigma$ may still contain local variables from a. The set $\epsilon_{r,a}(G^\epsilon, \sigma)$ has an indicative nature: For an aggregate $a\sigma$, it contains the identity substitution when the result of applying its aggregate function to the empty set is in accord with its bound, and it is empty otherwise. The construction of $\eta_{r,a}(G^\eta, \sigma)$ goes one step further and reconstitutes all ground aggregate elements of $a\sigma$ from variable bindings obtained by rules in G^η. Both functions play a central role below in defining the function `Propagate` for deriving ground aggregate placeholders of form (20) from ground rules in G^ϵ and G^η.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Example 22

We show how to extract aggregate elements from ground instances of rules (r3) and (r4) in Example 21.

Let G^ϵ be empty and G^η be the program consisting of the following rules:

$$\begin{aligned}
\eta_{e_1,a,r}(60, c_1, c_2) \leftarrow{} & \mathit{owns}(c_1, c_2, 60) \\
& \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_2) \wedge c_1 \neq c_2, \\
\eta_{e_1,a,r}(20, c_1, c_3) \leftarrow{} & \mathit{owns}(c_1, c_3, 20) \\
& \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_3) \wedge c_1 \neq c_3, \\
\eta_{e_1,a,r}(35, c_2, c_3) \leftarrow{} & \mathit{owns}(c_2, c_3, 35) \\
& \wedge \mathit{company}(c_2) \wedge \mathit{company}(c_3) \wedge c_2 \neq c_3, \\
\eta_{e_1,a,r}(51, c_3, c_4) \leftarrow{} & \mathit{owns}(c_3, c_4, 51) \\
& \wedge \mathit{company}(c_3) \wedge \mathit{company}(c_4) \wedge c_3 \neq c_4, \text{ and} \\
\eta_{e_2,a,r}(35, c_2, c_1, c_3) \leftarrow{} & \mathit{controls}(c_1, c_2) \wedge \mathit{owns}(c_2, c_3, 35) \\
& \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_3) \wedge c_1 \neq c_3.
\end{aligned}$$

Clearly, we have $\epsilon_{r,a}(G^\epsilon, \sigma) = \emptyset$ for any substitution σ because $G^\epsilon = \emptyset$. This means that aggregate a can only be satisfied if at least one of its elements is satisfiable. In fact, we obtain non-empty sets $\eta_{r,a}(G^\eta, \sigma)$ of ground aggregate elements for four substitutions σ:

$$\begin{aligned}
\eta_{r,a}(G^\eta, \sigma_1) &= \{60 : \mathit{owns}(c_1, c_2, 60)\} && \text{for } \sigma_1 : X \mapsto c_1, Y \mapsto c_2, \\
\eta_{r,a}(G^\eta, \sigma_2) &= \{51 : \mathit{owns}(c_3, c_4, 51)\} && \text{for } \sigma_2 : X \mapsto c_3, Y \mapsto c_4, \\
\eta_{r,a}(G^\eta, \sigma_3) &= \{35, c_2 : \mathit{controls}(c_1, c_2) \wedge \mathit{owns}(c_2, c_3, 35); \\
&\qquad 20 : \mathit{owns}(c_1, c_3, 20)\} && \text{for } \sigma_3 : X \mapsto c_1, Y \mapsto c_3, \text{ and} \\
\eta_{r,a}(G^\eta, \sigma_4) &= \{35 : \mathit{owns}(c_2, c_3, 35)\} && \text{for } \sigma_4 : X \mapsto c_2, Y \mapsto c_3.
\end{aligned}$$

For capturing the result of grounding aggregates relative to groups of aggregate elements gathered via P^η, we restrict their original translation to subsets of their ground elements. That is, while $\pi(a)$ and $\pi_a(\cdot)$ draw in Definition 9 on all instances of aggregate elements in a, their counterparts $\pi_G(a)$ and $\pi_{a,G}(\cdot)$ are restricted to a subset of such aggregate element instances:[8]

Definition 19

Let a be a closed aggregate and of form (8), E be the set of its aggregate elements, and $G \subseteq \mathrm{Inst}(E)$ be a set of aggregate element instances.

We define the translation $\pi_G(a)$ of a w.r.t. G as follows:

$$\pi_G(a) = \{\tau(D)^\wedge \rightarrow \pi_{a,G}(D)^\vee \mid D \subseteq G, D \not\triangleright a\}^\wedge,$$

where

$$\pi_{a,G}(D) = \{\tau(C)^\wedge \mid C \subseteq G \setminus D, C \cup D \triangleright a\}.$$

As before, this translation maps aggregates, possibly including non-global variables, to a conjunction of (ground) $\mathcal{R}$-rules. The resulting $\mathcal{R}$-formula is used below in the definition of functions `Propagate` and `Assemble`.

[8] Note that the restriction to sets of ground aggregate elements is similar to the one to two-valued interpretations in Definition 10.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Example 23

We consider the four substitutions σ_1 to σ_4 together with the sets $G_1 = \eta_{r,a}(G^\eta, \sigma_1)$ to $G_4 = \eta_{r,a}(G^\eta, \sigma_4)$ from Example 22 for aggregate a.

Following the discussion after Proposition 24, we get the formulas

$$\begin{aligned}
\pi_{G_1}(a\sigma_1) &= \mathit{owns}(c_1, c_2, 60),\\
\pi_{G_2}(a\sigma_2) &= \mathit{owns}(c_3, c_4, 51),\\
\pi_{G_3}(a\sigma_3) &= \mathit{controls}(c_1, c_2) \land \mathit{owns}(c_2, c_3, 35) \land \mathit{owns}(c_1, c_3, 20), \text{ and}\\
\pi_{G_4}(a\sigma_4) &= \bot.
\end{aligned}$$

The function `Propagate` yields a set of ground atoms of form (20) that are used in Algorithm 2 to ground rules having such placeholders among their body literals. Each such special atom is supported by a group of ground instances of its aggregate elements.

Definition 20

Let P be an aggregate program, (I, J) be a four-valued interpretation, and G^ϵ and G^η be subsets of ground instances of rules in P^ϵ and P^η, respectively.

We define $\texttt{Propagate}_P^{I,J}(G^\epsilon, G^\eta)$ as the set of all atoms of form $\alpha\sigma$ such that $\epsilon_{r,a}(G^\epsilon, \sigma) \cup G \neq \emptyset$ and $J \models \pi_G(a\sigma)_I$ with $G = \eta_{r,a}(G^\eta, \sigma)$ where α is an atom of form (20) for aggregate a in rule r and σ is a ground substitution for r mapping all global variables in a to ground terms.

An atom $\alpha\sigma$ is only considered if σ warrants ground rules in G^ϵ or G^η, signaling that the application of α to the empty set is feasible when applying σ or that there is a non-empty set of ground aggregate elements of α obtained after applying σ, respectively. If this is the case, it is checked whether the set G of aggregate element instances warrants that $\pi_G(a\sigma)$ admits stable models between I and J.

Example 24

We show how to propagate aggregates using the sets G_1 to G_4 and their associated formulas from Example 23. Suppose that $I = J = F \cup \{\mathit{controls}(c_1, c_2)\}$ using F from Example 20.

Observe that $J \models \pi_{G_1}(a\sigma_1)_I$, $J \models \pi_{G_2}(a\sigma_2)_I$, $J \models \pi_{G_3}(a\sigma_3)_I$, and $J \not\models \pi_{G_4}(a\sigma_4)_I$. Thus, we get $\texttt{Propagate}_P^{I,J}(G^\epsilon, G^\eta) = \{\alpha_{a,r}(c_1, c_2), \alpha_{a,r}(c_1, c_3), \alpha_{a,r}(c_3, c_4)\}$.

The function `Assemble` yields an $\mathcal{R}$-program in which aggregate placeholder atoms of form (20) have been replaced by their corresponding $\mathcal{R}$-formulas.

Definition 21

Let P be an aggregate program, and G^α and G^η be subsets of ground instances of rules in P^α and P^η, respectively.

We define $\texttt{Assemble}(G^\alpha, G^\eta)$ as the $\mathcal{R}$-program obtained from G^α by replacing

- all comparisons by $\top$ and
- all atoms of form $\alpha\sigma$ by the corresponding formulas $\pi_G(a\sigma)$ with $G = \eta_{r,a}(G^\eta, \sigma)$ where α is an atom of form (20) for aggregate a in rule r and σ is a ground substitution for r mapping all global variables in a to ground terms.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

```
function GroundComponent$(P, I, J)$
    $(G^{\alpha}, G^{\epsilon}, G^{\eta}, f, JA, JA', J') \leftarrow (\emptyset, \emptyset, \emptyset, \mathbf{t}, \emptyset, \emptyset, \emptyset)$;
    repeat
        // ground aggregate elements
        $G^{\epsilon} \leftarrow G^{\epsilon} \cup \bigcup_{r \in P^{\epsilon}} \texttt{GroundRule}^{I,J}_{r,f,J'}(\iota, B(r))$;
        $G^{\eta} \leftarrow G^{\eta} \cup \bigcup_{r \in P^{\eta}} \texttt{GroundRule}^{I,J}_{r,f,J'}(\iota, B(r))$;
        // propagate aggregates
        $JA \leftarrow \texttt{Propagate}^{I,J}_{P}(G^{\epsilon}, G^{\eta})$;
        // ground remaining rules
        $G^{\alpha} \leftarrow G^{\alpha} \cup \bigcup_{r \in P^{\alpha}} \texttt{GroundRule}^{I,J \cup JA}_{r,f,J' \cup JA'}(\iota, B(r))$;
        $(f, JA', J', J) \leftarrow (\mathbf{f}, JA, J, J \cup H(G^{\alpha}))$;
    until $J' = J$;
    return Assemble$(G^{\alpha}, G^{\eta})$;
```

Algorithm 2: Grounding Components

Example 25

We show how to assemble aggregates using the sets G_1 to G_3 for aggregate atoms that have been propagated in Example 24. Therefore, let G^{α} be the program consisting of the following rules:

$$\begin{aligned}
&\mathit{controls}(c_1, c_2) \leftarrow \alpha_{a,r}(c_1, c_2) \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_2) \wedge c_1 \neq c_2,\\
&\mathit{controls}(c_3, c_4) \leftarrow \alpha_{a,r}(c_3, c_4) \wedge \mathit{company}(c_3) \wedge \mathit{company}(c_4) \wedge c_3 \neq c_4, \text{ and}\\
&\mathit{controls}(c_1, c_3) \leftarrow \alpha_{a,r}(c_1, c_3) \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_3) \wedge c_1 \neq c_3.
\end{aligned}$$

Then, program $\texttt{Assemble}(G^{\alpha}, G^{\eta})$ consists of the following rules:

$$\begin{aligned}
&\mathit{controls}(c_1, c_2) \leftarrow \pi_{G_1}(a\sigma_1) \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_2) \wedge \top,\\
&\mathit{controls}(c_3, c_4) \leftarrow \pi_{G_2}(a\sigma_2) \wedge \mathit{company}(c_3) \wedge \mathit{company}(c_4) \wedge \top, \text{ and}\\
&\mathit{controls}(c_1, c_3) \leftarrow \pi_{G_3}(a\sigma_3) \wedge \mathit{company}(c_1) \wedge \mathit{company}(c_3) \wedge \top.
\end{aligned}$$

The next result shows how a (non-ground) aggregate program P is transformed into a (ground) $\mathcal{R}$-program $\pi_J(P)^{I,J}$ in the context of certain and possible atoms (I, J) via the interplay of grounding P^{ϵ} and P^{η}, deriving aggregate placeholders from their ground instances G^{ϵ} and G^{η}, and finally replacing them in G^{α} by the original aggregates' contents.

Proposition 32

Let P be an aggregate program, (I, J) be a finite four-valued interpretation, $G^{\epsilon} = \mathrm{Inst}^{I,J}(P^{\epsilon})$, $G^{\eta} = \mathrm{Inst}^{I,J}(P^{\eta})$, $JA = \texttt{Propagate}^{I,J}_{P}(G^{\epsilon}, G^{\eta})$, and $G^{\alpha} = \mathrm{Inst}^{I,J \cup JA}(P^{\alpha})$.

Then,

(a) $\texttt{Assemble}(G^{\alpha}, G^{\eta}) = \pi_J(P)^{I,J}$ and
(b) $H(G^{\alpha}) = T_{\pi(P)_I}(J)$.

Property (b) highlights the relation of the possible atoms contributed by G^{α} to a corresponding application of the immediate consequence operator. In fact, this is the first of three such relationships between grounding algorithms and consequence operators.

```
function Ground($P$)
    let $(P_i)_{i\in\mathbb{I}}$ be a refined instantiation sequence for $P$;
    $(F, G) \leftarrow (\emptyset, \emptyset)$;
    foreach $i \in \mathbb{I}$ do
        let $P'_i$ be the program obtained from $P_i$ as in Definition 13;
        $F \leftarrow F \cup \texttt{GroundComponent}(P'_i, H(G), H(F))$;
        $G \leftarrow G \cup \texttt{GroundComponent}(P_i, H(F), H(G))$;
    return $G$;
```

Algorithm 3: Grounding Programs

Let us now turn to grounding components of instantiation sequences in Algorithm 2. The function `GroundComponent` takes an aggregate program P along with two sets I and J of ground atoms. Intuitively, P is a component in a (refined) instantiation sequence and I and J form a four-valued interpretation (I, J) comprising the certain and possible atoms gathered while grounding previous components (although their roles get reversed in Algorithm 3). After variable initialization, `GroundComponent` loops over consecutive rule instantiations in P^α, P^ϵ, and P^η until no more possible atoms are obtained. In this case, it returns in Line 10 the $\mathcal{R}$-program obtained from G^α by replacing all ground aggregate placeholders of form (20) with the $\mathcal{R}$-formula corresponding to the respective ground aggregate. The body of the loop can be divided into two parts: Lines 4–6 deal with aggregates and Lines 7 and 8 care about grounding the actual program. In more detail, Lines 4 and 5 instantiate programs P^ϵ and P^η, whose ground instances, G^ϵ and G^η, are then used in Line 6 to derive ground instances of aggregate placeholders of form (20). The grounded placeholders are then added via variable JA to the possible atoms J when grounding the actual program P^α in Line 7, where J' and JA' hold the previous value of J and JA, respectively. For the next iteration, J is augmented in Line 8 with all rule heads in G^α and the flag f is set to false. Recall that the purpose of f is to ensure that initially all rules are grounded. In subsequent iterations, duplicates are omitted by setting the flag to false and filtering rules whose positive bodies are a subset of the atoms $J' \cup JA'$ used in previous iterations.

While the inner workings of Algorithm 2 follow the blueprint given by Proposition 32. its outer functionality boils down to applying the stable operator of the corresponding ground program in the context of the certain and possible atoms gathered so far.

Proposition 33
Let P be an aggregate program, (IC, JC) be a finite four-valued interpretation, and $J = S^{JC}_{\pi(P)}(IC)$.

Then,

(a) `GroundComponent`(P, IC, JC) terminates iff J is finite.

If J is finite, then

(a) `GroundComponent`$(P, IC, JC) = \pi_{JC\cup J}(P)^{IC, JC\cup J}$ and
(b) $H($`GroundComponent`$(P, IC, JC)) = J$.

Finally, Algorithm 3 grounds an aggregate program by iterating over the components of one of its refined instantiation sequences. Just as Algorithm 2 reflects the application of a stable operator, function `Ground` follows the definition of an approximate model when grounding a component (cf. Definition 13). At first, facts are computed in Line 6 by using the program stripped from rules being involved in a negative cycle overlapping with the present or subsequent components. The obtained head atoms are then used in Line 7 as certain context atoms when computing the ground version of the component at hand. The possible atoms are provided by the head atoms of the ground program built so far, and with roles reversed in Line 6. Accordingly, the whole iteration aligns with the approximate model of the chosen refined instantiation sequence (cf. Definition 14), as made precise next.

Our grounding algorithm computes implicitly the approximate model of the chosen instantiation sequence and outputs the corresponding ground program; it terminates whenever the approximate model is finite.

Theorem 34
Let P be an aggregate program, $(P_i)_{i\in\mathbb{I}}$ be a refined instantiation sequence for P, and (IC_i, JC_i) and (I_i, J_i) be defined as in Equations (15) and (16).

If $(P_i)_{i\in\mathbb{I}}$ is selected by Algorithm 3 in Line 2, then we have that

(a) the call $\texttt{Ground}(P)$ terminates iff $AM((P_i)_{i\in\mathbb{I}})$ is finite, and
(b) if $AM((P_i)_{i\in\mathbb{I}})$ is finite, then $\texttt{Ground}(P) = \bigcup_{i\in\mathbb{I}} \pi_{JC_i\cup J_i}(P_i)^{(IC_i,JC_i)\sqcup(I_i,J_i)}$.

As already indicated by Theorem 29, grounding is governed by the series of consecutive approximate models (I_i, J_i) in (16) delineating the stable models of each ground program in a (refined) instantiation sequence. Whenever each of them is finite, we also obtain a finite grounding of the original program. Note that the entire ground program is composed of the ground programs of each component in the chosen instantiation sequence. Hence, different sequences may result in different overall ground programs.

Most importantly, our grounding machinery guarantees that an obtained finite ground program has the same stable models as the original non-ground program.

Corollary 35 (Main result)
Let P be an aggregate program.

If $\texttt{Ground}(P)$ terminates, then P and $\texttt{Ground}(P)$ have the same well-founded and stable models.

Example 26
The execution of the grounding algorithms on Example 19 is illustrated in Table 1. Each individual table depicts a call to `GroundComponent` where the header above the double line contains the (literals of the) rules to be grounded and the rows below trace how nested calls to `GroundRule` proceed. The rules in the header contain the body literals in the order as they are selected by `GroundRule` with the rule head as the last literal. Calls to `GroundRule` are depicted with vertical lines and horizontal arrows. A vertical line represents the iteration of the loop in Lines 4–5. A horizontal arrow represents a recursive call to `GroundRule` in Line 5. Each row in the table not marked with × corresponds to a ground instance as returned by `GroundRule`. Furthermore, because all components are stratified, we only show the first iteration of the loop in Lines 3–9 of Algorithm 2 as the second iteration does not produce any new ground instances.

Grounding components P_1 to P_4 results in the programs $F = G = \{u(1) \leftarrow \top, u(2) \leftarrow \top, v(2) \leftarrow \top, v(3) \leftarrow \top\}$. Since grounding is driven by the sets of true and possible

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

$P'_{5,1} = \emptyset$

(a) $I_{5,1} = I_4 \cup H(\texttt{GC}(P'_{5,1}, J_4, I_4))$

$u(X)$		$\neg q(X)$		$p(X)$	1
$u(1)$	$\longrightarrow$	$\neg q(1)$	$\longrightarrow$	$p(1)$	1.1
$u(2)$	$\longrightarrow$	$\neg q(2)$	$\longrightarrow$	$p(2)$	

(b) $J_{5,1} = J_4 \cup H(\texttt{GC}(P_{5,1}, I_{5,1}, J_4))$

$v(X)$		$\neg p(X)$		$q(X)$	1
$v(2)$	$\longrightarrow$	$\times$			1.1
$v(3)$	$\longrightarrow$	$\neg p(3)$	$\longrightarrow$	$q(3)$	

(c) $I_{5,2} = I_{5,1} \cup H(\texttt{GC}(P'_{5,2}, J_{5,1}, I_{5,1}))$

$v(X)$		$\neg p(X)$		$q(X)$	1
$v(2)$	$\longrightarrow$	$\neg p(2)$	$\longrightarrow$	$q(2)$	1.1
$v(3)$	$\longrightarrow$	$\neg p(3)$	$\longrightarrow$	$q(3)$	

(d) $J_{5,2} = J_{5,1} \cup H(\texttt{GC}(P_{5,2}, I_{5,2}, J_{5,1}))$

$\neg p(1)$		x	1
$\times$			1.1

(e) $I_6 = I_{5,2} \cup H(\texttt{GC}(P'_6, J_{5,2}, I_{5,2}))$

$\neg p(1)$		x	1
$\neg p(1)$	$\longrightarrow$	x	1.1

(f) $J_6 = J_{5,2} \cup H(\texttt{GC}(P_6, I_6, J_{5,2}))$

$\neg q(3)$	y	1
$\times$		1.1

(g) $I_7 = I_6 \cup H(\texttt{GC}(P'_7, J_6, I_6))$

$\neg q(3)$	y	1
$\times$		1.1

(h) $J_7 = J_6 \cup H(\texttt{GC}(P_7, I_7, J_6))$

Table 1. *Grounding of components $P_{5,1}$, $P_{5,2}$, P_6, and P_7 from Example 19 where* $\texttt{GC} = \texttt{GroundComponent}$

atoms, we focus on the interpretations I_i and J_i where i is a component index in the refined instantiation sequence. We start tracing the grounding starting with $I_4 = J_4 = \{u(1), u(2), v(2), v(3)\}$.

The grounding of $P_{5,1}$ is depicted in Table 1a and 1b. We have $\mathcal{E} = \{b/1\}$ because predicate $b/1$ is used in the head of the rule in $P_{5,2}$. Thus, $\texttt{GroundComponent}(\emptyset, J_4, I_4)$ in Line 6 returns the empty set because $P'_{5,1} = \emptyset$. We get $I_{5,1} = I_4$. In the next line, the algorithm calls $\texttt{GroundComponent}(P_{5,1}, I_{5,1}, J_4)$ and we get $J_{5,1} = \{p(1), p(2)\}$. Note that at this point, it is not known that $q(1)$ is not derivable and so the algorithm does not derive $p(1)$ as a fact.

The grounding of $P_{5,2}$ is given in Table 1c and 1d. This time, we have $\mathcal{E} = \emptyset$ and $P_{5,2} = P'_{5,2}$. Thus, the first call to $\texttt{GroundRule}$ determines $q(3)$ to be true while the second call additionally determines the possible atom $q(2)$.

The grounding of P_6 is illustrated in Table 1e and 1f. Note that we obtain that x is possible because $p(1)$ was not determined to be true.

The grounding of P_7 is depicted in Table 1g and 1h. Note that, unlike before, we obtain that y is false because $q(3)$ was determined to be true.

Furthermore, observe that the choice of the refined instantiation sequence determines the output of the algorithm. In fact, swapping $P_{5,1}$ and $P_{5,2}$ in the sequence would result in x being false and y being possible.

To conclude, we give the grounding of the program as output by *gringo* in Table 2. The grounder furthermore omits the true body literals marked in green.

Example 27

We illustrate the grounding of aggregates on the company controls example in Example 10 using the grounding sequence $(P_i)_{1 \leq i \leq 9}$ and the set of facts F from Example 20. Observe that the grounding of components P_1 to P_8 produces the program $\{a \leftarrow \top \mid a \in F\}$. We

$u(1)$	$u(2)$
$v(2)$	$v(3)$
$p(1) \leftarrow \neg q(1) \wedge u(1)$	$p(2) \leftarrow \neg q(2) \wedge u(2)$
$q(2) \leftarrow \neg p(2) \wedge v(2)$	$q(3) \leftarrow \neg p(3) \wedge v(3)$
$x \leftarrow \neg p(1)$	

Table 2. *Grounding of Example 19 as output by gringo*

focus on how component P_9 is grounded. Because there are no negative dependencies, the components P_9 and P_9' in Line 5 of Algorithm 3 are equal. To ground component P_9, we use the rewriting from Example 21.

The grounding of component P_9 is illustrated in Table 3, which follows the same conventions as in Example 26. Because the program is positive, the calls in Lines 6 and 7 in Algorithm 3 proceed in the same way and we depict only one of them. Furthermore, because this example involves a recursive rule with an aggregate, the header consists of five rows separated by dashed lines processed by Algorithm 2. The first row corresponds to P_9^ϵ grounded in Line 4, the second and third to P_9^η grounded in Line 5, the fourth to aggregate propagation in Line 6, and the fifth to P_9^α grounded in Line 7. After the header follow the iterations of the loop in Lines 3–9. Because the component is recursive, processing the component requires four iterations, which are separated by solid lines in the table. The right-hand side column of the table contains the iteration number and a number indicating which row in the header is processed. The row for aggregate propagation lists the aggregate atoms that have been propagated.

The grounding of rule r_2 in Row 1.1 does not produce any rule instances in any iteration because the comparison $0 > 50$ is false. By first selecting this literal when grounding the rule, the remaining rule body can be completely ignored. Actual systems implement heuristics to prioritize such literals. Next, in the grounding of rule r_3 in Row 1.2, direct shares given by facts over *owns*/3 are accumulated. Because the rule does not contain any recursive predicates, it only produces ground instances in the first iteration. Unlike this, rule r_4 contains the recursive predicate *controls/2*. It does not produce instances in the first iteration in Row 1.3 because there are no corresponding atoms yet. Next, aggregate propagation is triggered in Row 1.4, resulting in aggregate atoms $\alpha_{a,r}(c_1, c_2)$ and $\alpha_{a,r}(c_3, c_4)$, for which enough shares have been accumulated in Row 1.2. Note that this corresponds to the propagation of the sets G_1 and G_2 in Example 24. With these atoms, rule r_1 is instantiated in Row 1.5, leading to new atoms over *controls*/2. Observe that, by selecting atom $\alpha_{a,r}(X, Y)$ first, `GroundRule` can instantiate the rule without backtracking.

In the second iteration, the newly obtained atoms over predicate *controls*/2 yield atom $\eta_{e_1,a,r}(35, c_2, c_1, c_3)$ in Row 2.3, which in turn leads to the aggregate atom $\alpha_{a,r}(c_1, c_3)$ resulting in further instances of r_4. Note that this corresponds to the propagation of the set G_3 in Example 24.

The following iterations proceed in a similar fashion until no new atoms are accumulated and the grounding loop terminates. Note that the utilized selection strategy affects the amount of backtracking in rule instantiation. One particular strategy used in *gringo* is to prefer atoms over recursive predicates. If there is only one such atom, `GroundRule`

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

$0 > \overline{50}$		$c(X)$	$c(Y)$	$X \neq Y$	$\epsilon_{a,r}(X,Y)$	1
$o(X,Y,S)$		$c(X)$	$c(Y)$	$X \neq Y$	$\eta_{e_1,a,r}(S,X,Y)$	2
$s(X,Z)$	$o(Z,Y,S)$	$c(X)$	$c(Y)$	$X \neq Y$	$\eta_{e_2,a,r}(S,Z,X,Y)$	3
`Propagate`						4
$\alpha_{a,r}(X,Y)$		$c(X)$	$c(Y)$	$X \neq Y$	$s(X,Y)$	5
$\times$						1.1
$o(c_1,c_2,60)$	$\longrightarrow$	$c(c_1) \rightarrow$	$c(c_2) \rightarrow$	$c_1 \neq c_2 \longrightarrow$	$\eta_{e_1,a,r}(60,c_1,c_2)$	1.2
$o(c_1,c_3,20)$	$\longrightarrow$	$c(c_1) \rightarrow$	$c(c_3) \rightarrow$	$c_1 \neq c_3 \longrightarrow$	$\eta_{e_1,a,r}(20,c_1,c_3)$	
$o(c_2,c_3,35)$	$\longrightarrow$	$c(c_2) \rightarrow$	$c(c_3) \rightarrow$	$c_2 \neq c_3 \longrightarrow$	$\eta_{e_1,a,r}(35,c_2,c_3)$	
$o(c_3,c_4,51)$	$\longrightarrow$	$c(c_3) \rightarrow$	$c(c_4) \rightarrow$	$c_3 \neq c_4 \longrightarrow$	$\eta_{e_1,a,r}(51,c_3,c_4)$	
$\times$						1.3
$\{\alpha_{a,r}(c_1,c_2),\alpha_{a,r}(c_3,c_4)\}$						1.4
$\alpha_{a,r}(c_1,c_2)$	$\longrightarrow$	$c(c_1) \rightarrow$	$c(c_2) \rightarrow$	$c_1 \neq c_2 \longrightarrow$	$s(c_1,c_2)$	1.5
$\alpha_{a,r}(c_3,c_4)$	$\longrightarrow$	$c(c_3) \rightarrow$	$c(c_4) \rightarrow$	$c_3 \neq c_4 \longrightarrow$	$s(c_3,c_4)$	
$\times$						2.1
$\times$						2.2
$s(c_1,c_2) \rightarrow$	$o(c_2,c_3,35) \longrightarrow$	$c(c_1) \rightarrow$	$c(c_3) \rightarrow$	$c_1 \neq c_3 \rightarrow$	$\eta_{e_1,a,r}(35,c_2,c_1,c_3)$	2.3
$s(c_3,c_4) \longrightarrow \times$						
$\{\alpha_{a,r}(c_1,c_3)\}$						2.4
$\alpha_{a,r}(c_1,c_3)$	$\longrightarrow$	$c(c_1) \rightarrow$	$c(c_3) \rightarrow$	$c_1 \neq c_3 \longrightarrow$	$s(c_1,c_3)$	2.5
$\times$						3.1
$\times$						3.2
$s(c_1,c_3) \rightarrow$	$o(c_3,c_4,51) \longrightarrow$	$c(c_1) \rightarrow$	$c(c_4) \rightarrow$	$c_1 \neq c_4 \rightarrow$	$\eta_{e_1,a,r}(51,c_3,c_1,c_4)$	3.3
$\{\alpha_{a,r}(c_1,c_4)\}$						3.4
$\alpha_{a,r}(c_1,c_4)$	$\longrightarrow$	$c(c_1) \rightarrow$	$c(c_4) \rightarrow$	$c_1 \neq c_4 \longrightarrow$	$s(c_1,c_4)$	3.5
$\times$						4.1
$\times$						4.2
$s(c_1,c_4) \longrightarrow \times$						4.3
$\emptyset$						4.4
$\times$						4.5

Table 3. *Tracing grounding of component P_9 where c = company, o = owns, and s = controls*

can select this atom first and only has to consider newly derived atoms for instantiation. The table is made more compact by applying this strategy. Further techniques are available in the database literature (Ullman 1988) that also work in case of multiple atoms over recursive predicates.

To conclude, we give the ground rules as output by a grounder like *gringo* in Table 4. We omit the translation of aggregates because its main objective is to show correctness of the algorithms. Solvers like *clasp* implement translations or even native handling of aggregates geared toward efficient solving (Gebser *et al.* 2009). Since our example program is positive, *gringo* is even able to completely evaluate the rules to facts omitting true literals from rule bodies marked in green.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

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

Table 4. *Grounding of the company controls problem from Example 10 as output by gringo*

7 Refinements

Up to now, we were primarily concerned by characterizing the theoretical and algorithmic cornerstones of grounding. This section refines these concepts by further detailing aggregate propagation, algorithm specifics, and the treatment of language constructs from *gringo*'s input language.

7.1 *Aggregate propagation*

We used in Section 6 the relative translation of aggregates for propagation, namely, formula $\pi_G(a\sigma)$ in Definition 20, to check whether an aggregate is satisfiable. In this section, we identify several aggregate specific properties that allow us to implement more efficient algorithms to perform this check.

To begin with, we establish some properties that greatly simplify the treatment of (arbitrary) monotone or antimonotone aggregates.

We have already seen in Proposition 24 that $\pi(a)_I$ is classically equivalent to $\pi(a)$ for any closed aggregate a and two-valued interpretation I. Here is its counterpart for antimonotone aggregates.

Proposition 36
Let a be a closed aggregate.

If a is antimonotone, then $\pi(a)_I$ is classically equivalent to $\top$ if $I \models \pi(a)$ and $\bot$ otherwise for any two-valued interpretation I.

Example 28
In Example 24, we check whether the interpretation J satisfies the formulas $\pi_{G_1}(a\sigma_1)_I$ to $\pi_{G_4}(a\sigma_4)_I$.

Using Proposition 24, this boils down to checking $\sum_{e \in G_i, J \models B(e)} H(e) > 50$ for each $1 \le i \le 4$. We get $60 > 50$, $51 > 50$, $55 > 50$, and $35 \not> 50$ for each G_i, which agrees with checking $J \models \pi_{G_i}(a\sigma_i)_I$.

An actual implementation can maintain a counter for the current value of the sum for each closed aggregate instance, which can be updated incrementally and compared with the bound as new instances of aggregate elements are grounded.

Next, we see that such counter based implementations are also possible #sum aggregates using the $<$, $\leq$, $>$, or $\geq$ relations. We restrict our attention to finite interpretations because Proposition 37 is intended to give an idea on how to implement an actual propagation algorithm for aggregates (infinite interpretations would add more special cases). Furthermore, we just consider the case that the bound is an integer here; the aggregate is constant for any other ground term.

Proposition 37
Let I be a finite two-valued interpretation, E be a set of aggregate elements, and b be an integer.

For $T = H(\{e \in \mathrm{Inst}(E) \mid I \models B(e)\})$, we get

(a) $\pi(\#\mathrm{sum}\{E\} \succ b)_I$ is classically equivalent to $\pi(\#\mathrm{sum}^+\{E\} \succ b')$ with $\succ \in \{\geq, >\}$ and $b' = b - \#\mathrm{sum}^-(T)$, and
(b) $\pi(\#\mathrm{sum}\{E\} \prec b)_I$ is classically equivalent to $\pi(\#\mathrm{sum}^-\{E\} \prec b')$ with $\prec \in \{\leq, <\}$ and $b' = b - \#\mathrm{sum}^+(T)$.

The remaining propositions identify properties that can be exploited when propagating aggregates over the $=$ and $\neq$ relations.

Proposition 38
Let I be a two-valued interpretation, E be a set of aggregate elements, and b be a ground term.

We get the following properties:

(a) $\pi(f\{E\} < b)_I \vee \pi(f\{E\} > b)_I$ implies $\pi(f\{E\} \neq b)_I$, and
(b) $\pi(f\{E\} = b)_I$ implies $\pi(f\{E\} \leq b)_I \wedge \pi(f\{E\} \geq b)_I$.

The following proposition identifies special cases when the implications in Proposition 38 are equivalences. Another interesting aspect of this proposition is that we can actually replace #sum aggregates over $=$ and $\neq$ with a conjunction or disjunction, respectively, at the expense of calculating a less precise approximate model. The conjunction is even strongly equivalent to the original aggregate under Ferraris' semantics but not the disjunction.

Proposition 39
Let I and J be two-valued interpretations, f be an aggregate function among #count, $\#\mathrm{sum}^+$, $\#\mathrm{sum}^-$ or #sum, E be a set of aggregate elements, and b be an integer.

We get the following properties:

(a) for $I \subseteq J$, we have $J \models \pi(f\{E\} < b)_I \vee \pi(f\{E\} > b)_I$ iff $J \models \pi(f\{E\} \neq b)_I$, and
(b) for $J \subseteq I$, we have $J \models \pi(f\{E\} = b)_I$ iff $J \models \pi(f\{E\} \leq b)_I \wedge \pi(f\{E\} \geq b)_I$.

The following proposition shows that full propagation of #sum, $\#\mathrm{sum}^+$, or $\#\mathrm{sum}^-$ aggregates over relations $=$ and $\neq$ involves solving the subset sum problem (Martello and Toth 1990). We assume that we propagate w.r.t. some polynomial number of aggregate elements. Propagating possible atoms when using the $=$ relation, that is, when $I \subseteq J$, involves deciding an NP problem and propagating certain atoms when using the $\neq$

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

relation, that is, when $J \subseteq I$, involves deciding a co-NP problem.[9] Note that the decision problem for #count aggregates is polynomial, though.

Proposition 40
Let I and J be finite two-valued interpretations, f be an aggregate function, E be a set of aggregate elements, and b be a ground term.

For $T_I = \{H(e) \mid e \in \text{Inst}(E), I \models B(e)\}$ and $T_J = \{H(e) \mid e \in \text{Inst}(E), J \models B(e)\}$, we get the following properties:

(a) for $J \subseteq I$, we have $J \models \pi(f\{E\} \neq b)_I$ iff there is no set $X \subseteq T_I$ such that $f(X \cup T_J) = b$, and

(b) for $I \subseteq J$, we have $J \models \pi(f\{E\} = b)_I$ and iff there is a set $X \subseteq T_J$ such that $f(X \cup T_I) = b$.

7.2 Algorithmic refinements

The calls in Lines 6 and 7 in Algorithm 3 can sometimes be combined to calculate certain and possible atoms simultaneously. This can be done whenever a component does not contain recursive predicates. In this case, it is sufficient to just calculate possible atoms along with rule instances in Line 7 augmenting Algorithm 1 with an additional check to detect whether a rule instance produces a certain atom. Observe that this condition applies to all stratified components but can also apply to components depending on unstratified components. In fact, typical programs following the generate, define, and test methodology (Lifschitz 2002; Niemelä 2008) of ASP, where the generate part uses choice rules (Simons *et al.* 2002) (see below), do not contain unstratified negation at all. When a grounder is combined with a solver built to store rules and perform inferences, one can optimize for the case that there are no negative recursive predicates in a component. In this case, it is sufficient to compute possible atoms along with their rule instances and leave the computation of certain atoms to the solver. Finally, note that *gringo* currently does not separate the calculation of certain and possible atoms at the expense of computing a less precise approximate model and possibly additional rule instances.

Example 29
For the following example, *gringo* computes atom $p(4)$ as unknown but the algorithms in Section 6 identify it as true.

$$\begin{array}{ll} r(1,4) & p(1) \leftarrow \neg q(1) \\ r(2,3) & q(1) \leftarrow \neg p(1) \\ r(3,1) & p(2) \\ \multicolumn{2}{l}{p(Y) \leftarrow p(X) \wedge r(X,Y).} \end{array}$$

When grounding the last rule, *gringo* determines $p(4)$ to be possible in the first iteration because $p(1)$ is unknown at this point. In the second iteration, it detects that $p(1)$ is a fact but does not use it for grounding again. If there were further rules depending negatively on predicate $p/1$, inapplicable rules might appear in *gringo*'s output.

[9] Note that *clingo*'s grounding algorithm does not attempt to solve these problems in all cases. It simply over- or underapproximates the satisfiability using Proposition 38.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Another observation is that the loop in Algorithm 2 does not produce further rule instances in a second iteration for components without recursive predicates. *Gringo* maintains an index (Garcia-Molina *et al.* 2009) for each positive body literal to speed up matching of literals; whenever none of these indexes, used in rules of the component at hand, are updated, further iterations can be skipped.

Just like *dlv*'s grounder, *gringo* adapts algorithms for semi-naive evaluation from the field of databases. In particular, it works best on linear programs (Abiteboul *et al.* 1995), having at most one positive literal occurrence over a recursive predicate in a rule body. The program in Example 10 for the company controls problem is such a linear program because $controls/2$ is the only recursive predicate. Algorithm 1 can easily be adapted to efficiently ground linear programs by making sure that the recursive positive literal is selected first. We then only have to consider matches that induce atoms not already used for instantiations in previous iterations of the loop in Algorithm 2 to reduce the amount of backtracking to find rule instances. In fact, the order in which literals are selected in Line 3 is crucial for the performance of Algorithm 1. *Gringo* uses an adaptation of the selection heuristics presented by Leone *et al.* (2001) that additionally takes into account recursive predicates and terms with function symbols.

To avoid unnecessary backtracking when grounding general logic programs, *gringo* instantiates rules using an algorithm similar to the improved semi-naive evaluation with optimizations for linear rules (Abiteboul *et al.* 1995).

7.3 Capturing gringo's input language

We presented aggregate programs where rule heads are simple atoms. Beyond that, *gringo*'s input language offers more elaborate language constructs to ease modeling.

A prominent such construct are so-called choice rules (Simons *et al.* 2002). Syntactically, one-element choice rules have the form $\{a\} \leftarrow B$, where a is an atom and B a body. Semantically, such a rule amounts to $a \vee \neg a \leftarrow B$ or equivalently $a \leftarrow \neg\neg a \wedge B$. We can easily add support for grounding choice rules, that is, rules where the head is not a plain atom but an atom marked as a choice, by discarding choice rules when calculating certain atoms and treating them like normal rules when grounding possible atoms. A translation that allows for supporting head aggregates using a translation to aggregate rules and choice rules is given by Gebser *et al.* (2015a). Note that *gringo* implements further refinements to omit deriving head atoms if a head aggregate cannot be satisfied.

Another language feature that can be instantiated in a similar fashion as body aggregates are conditional literals. *Gringo* adapts the rewriting and propagation of body aggregates to also support grounding of conditional literals.

Yet another important language feature are disjunctions in the head of rules (Gelfond and Lifschitz 1991). As disjunctive logic programs, aggregate programs allow us to solve problems from the second level of the polynomial hierarchy. In fact, using Łukasiewicz' theorem (Lukasiewicz 1941), we can write a disjunctive rule of form

$$a \vee b \leftarrow B$$

as the shifted strongly equivalent $\mathcal{R}$-program:

$$\begin{aligned} a &\leftarrow (b \rightarrow a) \wedge B \\ b &\leftarrow (a \rightarrow b) \wedge B. \end{aligned}$$

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

We can use this as a template to design grounding algorithms for disjunctive programs. In fact, *gringo* calculates the same approximate model for the disjunctive rule and the shifted program.

The usage of negation as failure is restricted in $\mathcal{R}$-programs. Note that any occurrence of a negated literal l in a rule body can be replaced by an auxiliary atom a adding rule $a \leftarrow l$ to the program. The resulting program preserves the stable models modulo the auxiliary atoms. This translation can serve as a template for double negation or negation in aggregate elements as supported by *gringo*.

Integrity constraints are a straightforward extension of logic programs. They can be grounded just like normal rules deriving an auxiliary atom that stands for $\bot$. Grounding can be stopped whenever the auxiliary atom is derived as certain. Integrity constraints also allow for supporting negated head atoms, which can be shifted to rule bodies (Janhunen 2001) resulting in integrity constraints, and then treated like negation in rule bodies.

A frequently used convenience feature of *gringo* are term pools (Gebser *et al.* 2015a;b). The grounder handles them by removing them in a rewriting step. For example, a rule of form

$$h(X;Y,Z) \leftarrow p(X;Y), q(Z)$$

is factored out into the following rules:

$$\begin{aligned} h(X,Z) &\leftarrow p(X), q(Z)\\ h(X,Z) &\leftarrow p(Y), q(Z)\\ h(Y,Z) &\leftarrow p(X), q(Z)\\ h(Y,Z) &\leftarrow p(Y), q(Z). \end{aligned}$$

We can then apply the grounding algorithms developed in Section 6.

To deal with variables ranging over integers, *gringo* supports interval terms (Gebser *et al.* 2015a;b). Such terms are handled by a translation to inbuilt range predicates. For example the program

$$h(l..u)$$

for terms l and u is rewritten into

$$h(A) \leftarrow rng(A,l,u)$$

by introducing auxiliary variable A and range atom $rng(A,l,u)$. The range atom provides matches including all substitutions that assign integer values between l and u to A. Special care has to be taken regarding rule safety, the range atom can only provide bindings for variable A but needs variables in the terms l and u to be provided elsewhere.

A common feature used when writing logic programs are terms involving arithmetic expressions and assignments. Both influence which rules are considered safe by the grounder. For example, the rule

$$h(X,Y) \leftarrow p(X+Y,Y) \land X = Y + Y$$

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

is rewritten into

$$h(X,Y) \leftarrow p(A,Y) \wedge X = Y + Y \wedge A = X + Y$$

by introducing auxiliary variable A. The rule is safe because we can match the literals in the order as given in the rewritten rule. The comparison $X = Y + Y$ extends the substitution with an assignment for X and the last comparison serves as a test. *Gringo* does not try to solve complicated equations but supports simple forms like the one given above.

Last but not least, *gringo* does not just support terms in assignments but it also supports aggregates in assignments. To handle such kind of aggregates, the rewriting and propagation of aggregates has to be extended. This is achieved by adding an additional variable to aggregate replacement atoms (20), which is assigned by propagation. For example, the rule

$$\underbrace{h(X,Y) \leftarrow q(X) \wedge \overbrace{Y = \#\mathrm{sum}\{Z : p(X,Z)\}}^{a}}_{r}$$

is rewritten into

$$\begin{aligned}\epsilon_{a,r}(X) &\leftarrow q(X)\\ \eta_{e,a,r}(Z,X) &\leftarrow p(X,Z) \wedge q(X)\\ h(X,Y) &\leftarrow \alpha_{a,r}(X,Y).\end{aligned}$$

Aggregate elements are grounded as before but values for variable Y are computed during aggregate propagation. In case of multiple assignment aggregates, additional care has to to be taken during the rewriting to ensure that the rewritten rules are safe.

8 Related work

This section aims at inserting our contributions into the literature, starting with theoretical aspects over algorithmic ones to implementations.

Splitting for infinitary formulas has been introduced by Harrison and Lifschitz (2016) generalizing results of Janhunen *et al.* (2007) and Ferraris *et al.* (2009). To this end, the concept of an A-stable model is introduced (Harrison and Lifschitz 2016). We obtain the following relationship between our definition of a stable model relative to a set IC and A-stable models: For an $\mathcal{N}$-program P, we have that if X is a stable model of P relative to IC, then $X \cup IC$ is an $(\mathcal{A} \setminus IC)$-stable model of P. Similarly, we get that if X is an A-stable model of P, then $S_P^{X \setminus A}(X)$ is a stable model of P relative to $X \setminus A$. The difference between the two concepts is that we fix atoms IC in our definition while A-stable models allow for assigning arbitrary truth values to atoms in $\mathcal{A} \setminus A$ (Harrison and Lifschitz 2016, Proposition 1). With this, let us compare our handling of program sequences to symmetric splitting (Harrison and Lifschitz 2016). Let $(P_i)_{i \in \mathbb{I}}$ be a refined instantiation sequence of aggregate program P, and $F = \bigcup_{i<j} \pi(P_i)$ and $G = \bigcup_{i \geq j} \pi(P_i)$ for some $j \in \mathbb{I}$ such that $H(F) \neq H(G)$. We can use the infinitary splitting theorem of Harrison and Lifschitz (2016) to calculate the stable model of $F^\wedge \wedge G^\wedge$ through the $H(F)$- and $\mathcal{A} \setminus H(F)$-stable models of $F^\wedge \wedge G^\wedge$. Observe that instantiation sequences do not permit

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

positive recursion between their components and infinite walks are impossible because an aggregate program consists of finitely many rules inducing a finite dependency graph. Note that we additionally require the condition $H(F) \neq H(G)$ because components can be split even if their head atoms overlap. Such a split can only occur if overlapping head atoms in preceding components are not involved in positive recursion.

Next, let us relate our operators to the ones defined by Truszczyński (2012). First of all, it is worthwhile to realize that the motivation of Truszczyński (2012) is to conceive operators mimicking model expansion in ID-logic by adding certain atoms. More precisely, let Φ, St, and Wf stand for the versions of the Fitting, stable, and well-founded operators defined by Truszczyński (2012). Then, we get the following relations to the operators defined in the previous sections:

$$\begin{aligned} St_{P,IC}(J) &= \mathit{lfp}(\Phi_{P,IC}(\cdot, J)) \\ &= \mathit{lfp}(T^{IC}_{P_J}) \cup IC \\ &= S^{IC}_{P}(J) \cup IC. \end{aligned}$$

For the well-founded operator we obtain

$$Wf_{P,IC}(I,J) = W^{IC,IC}_{P}(I,J) \sqcup IC.$$

Our operators allow us to directly calculate the atoms derived by a program. The versions of Truszczyński (2012) always include the input facts in their output and the well-founded operator only takes certain but not possible atoms as input.

In fact, we use operators as Denecker *et al.* (2000) to approximate the well-founded model and to obtain a ground program. While we apply operators to infinitary formulas (resulting from a translation of aggregates) as introduced by Truszczyński (2012), there has also been work on applying operators directly to aggregates. Vanbesien *et al.* (2021) provide an overview. Interestingly, the high complexity of approximating the aggregates pointed out in Proposition 40 has already been identified by Pelov *et al.* (2007).

Simplification can be understood as a combination of unfolding (dropping rules if a literal in the positive body is not among the head atoms of a program, that is, not among the possible atoms) and negative reduction (dropping rules if an atom in the negative body is a fact, that is, the literal is among the certain atoms) (Brass and Dix 1999; Brass *et al.* 2001). Even the process of grounding can be seen as a directed way of applying unfolding (when matching positive body literals) and negative reduction (when matching negative body literals). When computing facts, only rules whose negative body can be removed using positive reduction are considered.

The algorithms of Kemp *et al.* (1991) to calculate well-founded models perform a computation inspired by the alternating sequence to define the well-founded model as Van Gelder (1993). Our work is different in so far as we are not primarily interested in computing the well-founded model but the grounding of a program. Hence, our algorithms stop after the second application of the stable operator (the first to compute certain and the second to compute possible atoms). At this point, a grounder can use algorithms specialized for propositional programs to simplify the logic program at hand. Algorithmic refinements for normal logic programs as proposed by Kemp *et al.* (1991) also apply in our setting.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Last but not least, let us outline the evolution of grounding systems over the last two decades.

The *lparse* (Syrjänen 2001a) grounder introduced domain- or omega-restricted programs (Syrjänen 2001b). Unlike safety, omega-restrictedness is not modular. That is, the union of two omega-restricted programs is not necessarily omega-restricted while the union of two safe programs is safe. Apart from this, *lparse* supports recursive monotone and antimonotone aggregates. However, our company controls encoding in Example 10 is not accepted because it is not omega-restricted. For example, variable X in the second aggregate element needs a domain predicate. Even if we supplied such a domain predicate, *lparse* would instantiate variable X with all terms provided by the domain predicate resulting in a large grounding. As noted by Ferraris and Lifschitz (2005), recursive nonmonotone aggregates (sum aggregates with negative weights) are not supported correctly by *lparse*.

Gringo 1 and 2 add support for lambda-restricted programs (Gebser *et al.* 2007) extending omega-restricted programs. This augments the set of predicates that can be used for instantiation but is still restricted as compared to safe programs. That is, lambda-restrictedness is also not modular and our company controls program is still not accepted. At the time, the development goal was to be compatible to *lparse* but extend the class of accepted programs. Notably, *gringo* 2 adds support for additional aggregates (Gebser *et al.* 2009). Given its origin, *gringo* up to version 4 handles recursive nonmonotone aggregates in the same incorrect way as *lparse*.

The grounder of the *dlv* system has been the first one to implement grounding algorithms based on semi-naive evaluation (Eiter *et al.* 1997). Furthermore, it implements various techniques to efficiently ground logic programs (Leone *et al.* 2001; Faber *et al.* 2001; Perri *et al.* 2007). The $dlv^{\mathcal{A}}$ system is the first *dlv*-based system to support recursive aggregates (Dell'Armi *et al.* 2003), which is nowadays also available in recent versions of *idlv* (Calimeri *et al.* 2017).

Gringo 3 closed up to *dlv* being the first *gringo* version to implement grounding algorithms based on semi-naive evaluation (Gebser *et al.* 2011). The system accepts safe rules but still requires lambda-restrictedness for predicates within aggregates. Hence, our company controls encoding is still not accepted.

Gringo 4 implements grounding of aggregates with algorithms similar to the ones presented in Section 6 (Gebser *et al.* 2015c). Hence, it is the first version that accepts our company controls encoding.

Finally, *gringo* 5 refines the translation of aggregates as proposed by Alviano *et al.* (2015) to properly support nonmonotone recursive aggregates and refines the semantics of pools and undefined arithmetics (Gebser *et al.* 2015a).

Another system with a grounding component is the *idp* system (De Cat *et al.* 2014). Its grounder instantiates a theory by assigning sorts to variables. Even though it supports inductive definitions, it relies solely on the sorts of variables (Wittocx *et al.* 2010) to instantiate a theory. In case of inductive definitions, this can lead to instances of definitions that can never be applied. We believe that the algorithms presented in Section 6 can also be implemented in an *idp* system decreasing the instantiation size of some problems (e.g., the company controls problem presented in Example 10).

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Last but not least, we mention that not all ASP systems follow a two-phase approach of grounding and solving but rather adapt a lazy approach by grounding on-the-fly during solving (Palù *et al.* 2009; Lefèvre *et al.* 2017; Weinzierl *et al.* 2020).

9 Conclusion

We have provided a first comprehensive elaboration of the theoretical foundations of grounding in ASP. This was enabled by the establishment of semantic underpinnings of ASP's modeling language in terms of infinitary (ground) formulas (Harrison *et al.* 2014; Gebser *et al.* 2015a). Accordingly, we start by identifying a restricted class of infinitary programs, namely, $\mathcal{R}$-programs, by limiting the usage of implications. Such programs allow for tighter semantic characterizations than general $\mathcal{F}$-programs, while being expressive enough to capture logic programs with aggregates. Interestingly, we rely on well-founded models (Bruynooghe *et al.* 2016; Truszczyński 2018) to approximate the stable models of $\mathcal{R}$-programs (and simplify them in a stable-models preserving way). This is due do the fact that the (ID-)well-founded-operator enjoys monotonicity, which lends itself to the characterization of iterative grounding procedures. The actual semantics of non-ground aggregate programs is then defined via a translation to $\mathcal{R}$-programs. This setup allows us to characterize the inner workings of our grounding algorithms for aggregate programs in terms of the operators introduced for $\mathcal{R}$-programs. It turns out that grounding amounts to calculating an approximation of the well-founded model together with a ground program simplified with that model. This does not only allow us to prove the correctness of our grounding algorithms but moreover to characterize the output of a grounder like *gringo* in terms of established formal means. To this end, we have shown how to split aggregate programs into components and to compute their approximate models (and corresponding simplified ground programs). The key instruments for obtaining finite ground programs with finitary subformulas have been dedicated forms of program simplification and aggregate translation. Even though, we limit ourselves to $\mathcal{R}$-programs, we capture the core aspects of grounding: a monotonically increasing set of possibly derivable atoms and on-the-fly (ground) rule generation. Additional language features of *gringo*'s input language are relatively straightforward to accommodate by extending the algorithms presented in this paper.

For reference, we implemented the presented algorithms in a prototypical grounder, μ-*gringo*, supporting aggregate programs (see Footnote 1). While it is written to be as concise as possible and not with efficiency in mind, it may serve as a basis for experiments with custom grounder implementations. The actual *gringo* system supports a much larger language fragment. There are some differences compared to the algorithms presented here. First, certain atoms are removed from rule bodies if not explicitly disabled via a command line option. Second, translation π is only used to characterize aggregate propagation. In practice, *gringo* translates ground aggregates to monotone aggregates (Alviano *et al.* 2015). Further translation (Bomanson *et al.* 2014) or even native handling (Gebser *et al.* 2009) of them is left to the solver. Finally, in some cases, *gringo* might produce more rules than the algorithms presented above. This should not affect typical programs. A tighter integration of grounder and solver to further reduce the number of ground rules is an interesting topic of future research.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Supplementary material

To view supplementary material for this article, please visit http://doi.org/10.1017/S1471068422000308.

References

Abiteboul, S., Hull, R. and Vianu, V. 1995. *Foundations of Databases.* Addison-Wesley.

Alviano, M., Dodaro, C., Leone, N. and Ricca, F. 2015. Advances in WASP. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, F. Calimeri, G. Ianni and M. Truszczyński, Eds. Lecture Notes in Artificial Intelligence, vol. 9345. Springer-Verlag, 40–54.

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.

Belnap, N. 1977. A useful four-valued logic. In *Modern Uses of Multiple-Valued Logic*, J. Dunn and G. Epstein, Eds. Reidel.

Bomanson, J., Gebser, M. and Janhunen, T. 2014. Improving the normalization of weight rules in answer set programs. In *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence (JELIA'14)*, E. Fermé and J. Leite, Eds. Lecture Notes in Artificial Intelligence, vol. 8761. Springer-Verlag, 166–180.

Brass, S. and Dix, J. 1999. Semantics of (disjunctive) logic programs based on partial evaluation. *Journal of Logic Programming 40,* 1, 1–46.

Brass, S., Dix, J., Freitag, B. and Zukowski, U. 2001. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming 1,* 5, 497–538.

Bruynooghe, M., Denecker, M. and Truszczyński, M. 2016. ASP with first-order logic and definitions. *AI Magazine 37,* 3, 69–80.

Calimeri, F., Cozza, S., Ianni, G. and Leone, N. 2008. Computable functions in ASP: Theory and implementation. In *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*. Lecture Notes in Computer Science, vol. 5366. Springer-Verlag, 407–424.

Calimeri, F., Fuscà, D., Perri, S. and Zangari, J. 2017. I-DLV: The new intelligent grounder of DLV. *Intelligenza Artificiale 11,* 1, 5–20.

De Cat, B., Bogaerts, B., Bruynooghe, M. and Denecker, M. 2014. Predicate logic as a modelling language: The IDP system. CoRR abs/1401.6312.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N. and Pfeifer, G. 2003. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, G. Gottlob and T. Walsh, Eds. Morgan Kaufmann Publishers, 847–852.

Denecker, M., Marek, V. and Truszczyński, M. 2000. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer Academic Publishers, Dordrecht, 127–144.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G. and Scarcello, F. 1997. A deductive system for nonmonotonic reasoning. In *Proceedings of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, J. Dix, U. Furbach and A. Nerode, Eds. Lecture Notes in Artificial Intelligence, vol. 1265. Springer-Verlag, 363–374.

Faber, W., Leone, N. and Perri, S. 2012. The intelligent grounder of DLV. In *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. Lecture Notes in Computer Science, vol. 7265. Springer-Verlag, 247–264.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Faber, W., Leone, N., Perri, S. and Pfeifer, G. 2001. Efficient instantiation of disjunctive databases. Technical Report DBAI-TR-2001-44, Technische Universität Wien.

Ferraris, P. 2011. Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic 12,* 4, 25.

Ferraris, P., Lee, J., Lifschitz, V. and Palla, R. 2009. Symmetric splitting in the general theory of stable models. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, C. Boutilier, Ed. AAAI/MIT Press, 797–803.

Ferraris, P. and Lifschitz, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5,* 1-2, 45–74.

Fitting, M. 2002. Fixpoint semantics for logic programming: A survey. *Theoretical Computer Science 278,* 1-2, 25–51.

Garcia-Molina, H., Ullman, J. and Widom, J. 2009. *Database Systems: The Complete Book*, 2nd ed. Pearson Education.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V. and Schaub, T. 2015a. Abstract Gringo. *Theory and Practice of Logic Programming 15,* 4-5, 449–463.

Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T. and Thiele, S. 2015b. *Potassco User Guide*, 2nd ed. University of Potsdam.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2009. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09).* Lecture Notes in Computer Science, vol. 5649. Springer-Verlag, 250–264.

Gebser, M., Kaminski, R., König, A. and Schaub, T. 2011. Advances in gringo series 3. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, J. Delgrande and W. Faber, Eds. Lecture Notes in Artificial Intelligence, vol. 6645. Springer-Verlag, 345–351.

Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T. and Thiele, S. 2009. On the input language of ASP grounder gringo. In *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, E. Erdem, F. Lin and T. Schaub, Eds. Lecture Notes in Artificial Intelligence, vol. 5753. Springer-Verlag, 502–508.

Gebser, M., Kaminski, R. and Schaub, T. 2015c. Grounding recursive aggregates: Preliminary report. In *Proceedings of the Third Workshop on Grounding, Transforming, and Modularizing Theories with Variables (GTTV'15)*, M. Denecker and T. Janhunen, Eds.

Gebser, M., Kaufmann, B. and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187-188*, 52–89.

Gebser, M. and Schaub, T. 2016. Modeling and language extensions. *AI Magazine 37,* 3, 33–44.

Gebser, M., Schaub, T. and Thiele, S. 2007. Gringo: A new grounder for answer set programming. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07).* Lecture Notes in Artificial Intelligence, vol. 4483. Springer-Verlag, 266–271.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, R. Kowalski and K. Bowen, Eds. MIT Press, 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

Giunchiglia, E., Lierler, Y. and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning 36,* 4, 345–377.

Harrison, A. and Lifschitz, V. 2016. Stable models for infinitary formulas with extensional atoms. *Theory and Practice of Logic Programming 16,* 5-6, 771–786.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Harrison, A., Lifschitz, V., Pearce, D. and Valverde, A. 2017. Infinitary equilibrium logic and strongly equivalent logic programs. *Artificial Intelligence 246*, 22–33.

Harrison, A., Lifschitz, V. and Yang, F. 2014. The semantics of gringo and infinitary propositional formulas. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, C. Baral, G. De Giacomo and T. Eiter, Eds. AAAI Press.

Janhunen, T. 2001. On the effect of default negation on the expressiveness of disjunctive rules. In *Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*. Lecture Notes in Computer Science, vol. 2173. Springer-Verlag, 93–106.

Janhunen, T., Oikarinen, E., Tompits, H. and Woltran, S. 2007. Modularity aspects of disjunctive stable models. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. Lecture Notes in Artificial Intelligence, vol. 4483. Springer-Verlag, 175–187.

Kaufmann, B., Leone, N., Perri, S. and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Magazine 37,* 3, 25–32.

Kemp, D., Stuckey, P. and Srivastava, D. 1991. Magic sets and bottom-up evaluation of well-founded models. In *Logic Programming, Proceedings of the 1991 International Symposium.* MIT Press, 337–351.

Lefèvre, C., Béatrix, C., Stéphan, 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., Perri, S. and Scarcello, F. 2001. Improving ASP instantiators by join-ordering methods. In *Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*. Lecture Notes in Computer Science, vol. 2173. Springer-Verlag, 280–294.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

Lierler, Y. and Lifschitz, V. 2009. One more decidable class of finitely ground programs. In *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*. Lecture Notes in Computer Science, vol. 5649. Springer-Verlag, 489–493.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138,* 1-2, 39–54.

Lifschitz, V. 2008. Twelve definitions of a stable model. In *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*. Lecture Notes in Computer Science, vol. 5366. Springer-Verlag, 37–51.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *Proceedings of the Eleventh International Conference on Logic Programming.* MIT Press, 23–37.

Lin, F. and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157,* 1-2, 115–137.

Lukasiewicz, J. 1941. Die logik und das grundlagenproblem. *Les Entreties de Zürich sur les Fondaments et la Méthode des Sciences Mathématiques 12,* 6-9, 82–100.

Martello, S. and Toth, P. 1990. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & sons.

Mumick, I., Pirahesh, H. and Ramakrishnan, R. 1990. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases (VLDB'90)*, D. McLeod, R. Sacks-Davis and H. Schek, Eds. Morgan Kaufmann Publishers, 264–277.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press

Niemelä, I. 2008. Answer set programming without unstratified negation. In *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*. Lecture Notes in Computer Science, vol. 5366. Springer-Verlag, 88–92.

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

Pelov, N., Denecker, M. and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming 7,* 3, 301–353.

Perri, S., Scarcello, F., Catalano, G. and Leone, N. 2007. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence 51,* 2-4, 195–228.

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

Syrjänen, T. 2001a. Lparse 1.0 user's manual.

Syrjänen, T. 2001b. Omega-restricted logic programs. In *Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*. Lecture Notes in Computer Science, vol. 2173. Springer-Verlag, 267–279.

Tarski, A. 1955. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics 5*, 285–309.

Truszczyński, M. 2012. Connecting first-order ASP and the logic FO(ID) through reducts. In *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. Lecture Notes in Computer Science, vol. 7265. Springer-Verlag, 543–559.

Truszczyński, M. 2018. An introduction to the stable and well-founded semantics of logic programs. In *Declarative Logic Programming: Theory, Systems, and Applications*, M. Kifer and Y. Liu, Eds. ACM/Morgan & Claypool, 121–177.

Ullman, J. 1988. *Principles of Database and Knowledge-Base Systems*. Computer Science Press.

van Emden, M. and Kowalski, R. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM 23,* 4, 733–742.

Van Gelder, A. 1993. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences 47*, 185–221.

Vanbesien, L., Bruynooghe, M. and Denecker, M. 2021. Analyzing semantics of aggregate answer set programming using approximation fixpoint theory. CoRR abs/2104.14789.

Weinzierl, A., Taupe, R. and Friedrich, G. 2020. Advancing lazy-grounding ASP solving techniques — restarts, phase saving, heuristics, and more. *Theory and Practice of Logic Programming 20,* 5, 609–624.

Wittocx, J., Mariën, M. and Denecker, M. 2010. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research 38*, 223–269.

https://doi.org/10.1017/S1471068422000308 Published online by Cambridge University Press