Axiomatization of Aggregates in Answer Set Programming

Jorge Fandinno*, Zach Hansen*, Yuliya Lierler*

University of Nebraska at Omaha
{jfandinno,zachhansen,ylierler}@unomha.edu

Abstract

The paper presents a characterization of logic programs with *aggregates* based on a many-sorted generalization of operator SM that refers neither to grounding nor to fixpoints. This characterization introduces new function symbols for aggregate operations and aggregate elements, whose meaning can be fixed by adding appropriate axioms to the result of the SM transformation. We prove that for programs without positive recursion through aggregates our semantics coincides with the semantics of the answer set solver `clingo`.

Introduction

Answer set programming (ASP; Lifschitz 2008) is a form of declarative logic programming well-suited to solving knowledge-intensive search problems. Its success relies on the combination of a rich knowledge representation language with efficient solvers for finding solutions to problems expressed in this language (Lifschitz 2019). One of the most useful constructs of this language are *aggregates*: intuitively, these are functions that apply to sets. The semantics of aggregates has been extensively studied in the literature (Simons, Niemelä, and Soininen 2002; Dovier, Pontelli, and Rossi 2003; Pelov, Denecker, and Bruynooghe 2007; Son and Pontelli 2007; Ferraris 2011; Faber, Pfeifer, and Leone 2011; Gelfond and Zhang 2014, 2019; Cabalar et al. 2019). In most cases, papers rely on the idea of *grounding* — a process in which all variables are replaced by variable-free terms. Thus, first a program with variables is transformed into a propositional one, then the semantics of the propositional program is defined. This makes reasoning about programs with variables cumbersome. For instance, it prohibits using first-order theorem provers for verifying properties of programs as advocated by Fandinno et al. (2020).

To the best of our knowledge, only two approaches defined the semantics of aggregates without referring to grounding. Lee, Lifschitz, and Palla (2008) translate certain count aggregates with a ground guard into an existentially quantified first-order formula. Yet, this approach is inapplicable to more general count aggregates as well as to the common sum aggregates. Cabalar et al. (2018) introduce intensional sets as first-class citizens into Quantified Equilibrium Logic (Pearce and Valverde 2005) with partial functions (Cabalar 2011) and provide a formalisation of aggregates that directly corresponds to the idea that aggregates are functions that apply to sets. This approach is truly general: it covers arbitrary aggregates including nested ones. The price for the generality of this formalism is complexity.

Similar to the work by Cabalar et al. (2018), our approach provides a direct formalisation of the idea that aggregates are functions that apply to sets, but it aims to exclusively use the language of classical logic (instead of adding intensional sets as a new construct in the language). To achieve this goal, we assume two restrictions. First, aggregates cannot be nested and second, there cannot exist positive recursion through aggregates. Note that, in practice, solvers cannot process nested aggregates. Regarding the second restriction, solvers may process programs with recursion through aggregates. Yet, different solvers may compute distinct answers for the same input program. For the sake of uncontroversial semantics, the ASP-Core-2 standard does not consider recursion through aggregates (Calimeri et al. 2012).

In this paper, we introduce a translation from logic programs to second-order logic formulas and define the semantics of aggregates using two equivalent characterizations. The expressive power of the answer set semantics cannot be captured by first-order logic, making second-order logic the prime candidate for this task. The first characterization uses a simpler language, where we restrict the considered interpretations at the meta-logic level. The second characterization fixes the meaning of some symbols in this language by providing an axiomatization at the object-level. The first characterization is easier to understand, while the second provides greater mathematical precision. In many cases we can replace second-order formulas by first-order formulas (Ferraris, Lee, and Lifschitz 2011; Lee and Meng 2011). Here we show that for programs with aggregates that apply to finite sets, we can replace the second-order axiomatization of aggregates by a first-order one. This paves the way for using first-order theorem provers to reason about programs with aggregates; to the best of our knowledge this was not yet possible. The restriction to finite aggregates is not a practical limitation as solvers cannot deal with infinite sets. In the studied fragment, our semantics coincide with the semantics of the solver `clingo` (Gebser et al. 2015).

*These authors contributed equally.

Preliminaries

Syntax of programs with aggregates. We assume a *(program) signature* with three countably infinite sets of symbols: *numerals*, *symbolic constants* and *program variables*. We also assume a 1-to-1 correspondence between numerals and integers; the numeral corresponding to an integer n is denoted by $\overline{n}$. *Program terms* are either numerals, symbolic constants, variables or either of the special symbols inf and sup. A program term (or any other expression) is *ground* if it contains no variables. We assume that a total order on ground terms is chosen such that

- inf is its least element and sup is its greatest element,
- for any integers m and n, $\overline{m} < \overline{n}$ iff $m < n$, and
- for any integer n and any symbolic constant c, $\overline{n} < c$.

An *atom* is an expression of the form $p(\mathbf{t})$, where p is a symbolic constant and $\mathbf{t}$ is a list of program terms. A *comparison* is an expression of the form $t \prec t'$, where t and t' are program terms and $\prec$ is one of the *comparison symbols*:

$$= \quad \neq \quad < \quad > \quad \leq \quad \geq \tag{1}$$

An *atomic formula* is either an atom or a comparison. A *basic literal* is an atomic formula possibly preceded by one or two occurrences of *not*. An *aggregate element* has the form

$$t_1, \ldots, t_k : l_1, \ldots, l_m \tag{2}$$

where each t_i ($1 \leq i \leq k$) is a program term and each l_i ($1 \leq i \leq m$) is a basic literal. An *aggregate atom* is of form

$$\#\mathtt{op}\{E\} \prec u \tag{3}$$

where $\mathtt{op}$ is an operation name, E is an aggregate element, $\prec$ is one of the comparison symbols in (1), and u is a program term, called *guard*. We consider operation names names $\mathtt{count}$ and $\mathtt{sum}$. For example, expression $\#\mathtt{sum}\{K, X, Y : in(X, Y), cost(K, X, Y)\} > J$ is an aggregate atom. An *aggregate literal* is an aggregate atom possibly preceded by one or two occurrences of *not*. A *literal* is either a basic literal or an aggregate literal.

A *rule* is an expression of the form

$$Head \text{ :- } B_1, \ldots, B_n, \tag{4}$$

where

- $Head$ is either an atom or symbol $\bot$; we often omit symbol $\bot$ which results in an empty head;
- each B_i ($1 \leq i \leq n$) is a literal.

We call the symbol :- a *rule operator*. We call the left hand side of the rule operator the *head*, the right hand side of the rule operator the *body*. When the head of the rule is an atom we call the rule *normal*. A *program* is a finite set of rules.

We assume that aggregates do not have positive recursion. This is a less restrictive assumption than the one used in the ASP-Core-2 semantics (Calimeri et al. 2012), which requires aggregates have neither positive nor negative recursion. Formally, a *predicate symbol* is a pair p/n, where p is a symbolic constant and n is a nonnegative integer. About a program or another syntactic expression, we say that a predicate symbol p/n *occurs* in it if it contains an atom of the form $p(t_1, \ldots, t_n)$. We say that an occurrence of a predicate symbol p/n in a literal is *strictly positive* if it is not in the scope of negation. For example, literals $not\ r(X, Y, Z)$, $\#\mathtt{sum}\{Y, Z : not\ r(X, Y, Z)\}$ and $not\ \#\mathtt{sum}\{Y, Z : r(X, Y, Z)\}$ contain no strictly positive occurrence of $r/3$. For a program Π, its *(directed predicate) dependency graph* is defined by

1. a set of vertices containing all predicate symbols occurring in Π ,
2. a set of edges containing an edge (h, b) for every normal rule (4) with h being the predicate symbol occurring in the atom $Head$ of the rule and b being any predicate symbol that has a strictly positive occurrence in one of the literals $B_1, \ldots, B_n$ of the rule.

The aggregates in Π *have no positive recursion* if, for every normal rule R of form (4), there is no path in the program's dependency graph from any predicate symbol with a strictly positive occurrence in an aggregate element of one of the literals $B_1, \ldots, B_n$ in R to the predicate symbol occurring in $Head$ of R. For instance, a program consisting of rule $r(X, Y, Z)$:- $q(X, Y, Z), \#\mathtt{sum}\{Y, Z : not\ r(X, Y, Z)\}$ has no aggregate with positive recursion; a program consisting of $r(X, Y, Z)$:- $q(X, Y, Z), \#\mathtt{sum}\{Y, Z :\ r(X, Y, Z)\}$ has an aggregate with positive recursion.

Each operation name $\mathtt{op}$ is associated with a function $\widehat{\mathtt{op}}$ that maps every set of tuples of ground terms to a ground term. If the first member of a tuple $\mathbf{t}$ is a numeral $\overline{n}$ then we say that integer n is the weight of $\mathbf{t}$, otherwise the weight of $\mathbf{t}$ is 0. For any set Δ of tuples of ground terms,

- $\widehat{\mathtt{count}}(\Delta)$ is the numeral corresponding to the cardinality of Δ, if Δ is finite; and sup otherwise.
- $\widehat{\mathtt{sum}}(\Delta)$ is the numeral corresponding to the sum of the weights of all tuples in Δ, if Δ contains finitely many tuples with non-zero weights; and 0 otherwise.[1] If Δ is empty, then $\widehat{\mathtt{sum}}(\Delta) = 0$.

Operator SM for many-sorted signature. Here, we recall the standard definition of many-sorted first-order logic. A signature σ consists of function and predicate constants and a set of sorts. The arity of every function or predicate constant is a tuple of sorts; the arity of function constants is a nonempty tuple. A predicate constant whose arity is the empty tuple is called a *proposition*. We assume that there are infinitely many variables for each sort. Atomic formulas are built similar to the standard unsorted logic with the restriction that in a term $f(t_1, \ldots, t_n)$ (an atom $p(t_1, \ldots, t_n)$, respectively), the sort of term t_i must be a subsort of the i-th argument of f (of p, respectively). In addition, $t_1 = t_2$ is an atomic formula if the sorts of t_1 and t_2 have a common supersort. A many-sorted interpretation I has a nonempty universe $|I|^s$ for each sort s. When sort s_1 is a subsort of sort s_2, an interpretation additionally satisfies the condition $|I|^{s_1} \subseteq |I|^{s_2}$. The notion of satisfaction is analogous

[1] The sum of a set of integers is not always defined. We could choose a special symbol to denote this case, we chose to use 0 following the description of abstract $\mathtt{gringo}$ (Gebser et al. 2015).

to the unsorted case with the restriction that an interpretation maps a term to an element in the universe of its associated sort.

The following symbols are considered to be the logical primitives:

$$\wedge \quad \vee \quad \rightarrow \quad \bot \quad \forall \quad \exists$$

Negation, truth and equivalence are assumed to be abbreviations: $F \rightarrow \bot$ stands for $\neg F$, $\bot \rightarrow \bot$ stands for $\top$, and $(F \rightarrow G) \wedge (G \rightarrow F)$ stands for $F \leftrightarrow G$. The definition of the operator SM for a many-sorted signature is a straightforward generalization of the unsorted case (Ferraris, Lee, and Lifschitz 2011). If p and u are predicate constants or variables of the same arity (note that while the original definition does not account for sort information, here arity refers to both number and sort of the arguments), then $u \leq p$ stands for the formula

$$\forall \mathbf{W}(u(\mathbf{W}) \rightarrow p(\mathbf{W})),$$

where $\mathbf{W}$ is a tuple of distinct object variables matching the arity of p and u. If $\mathbf{p}$ and $\mathbf{u}$ are tuples $p_1, \ldots, p_n$ and $u_1, \ldots, u_n$ of predicate constants or variables such that each p_i and u_i have the same arity, then $\mathbf{u} \leq \mathbf{p}$ stands for the conjunction

$$(u_1 \leq p_1) \wedge \cdots \wedge (u_n \leq p_n),$$

and $\mathbf{u} < \mathbf{p}$ stands for $(\mathbf{u} \leq \mathbf{p}) \wedge \neg(\mathbf{p} \leq \mathbf{u})$. For any many-sorted first-order formula F and a list $\mathbf{p}$ of predicate constants, by $\mathrm{SM}_{\mathbf{p}}[F]$ we denote the second-order formula

$$F \wedge \neg \exists \mathbf{u}\big((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u})\big)$$

where $\mathbf{u}$ is a list of distinct predicate variables $u_1, \ldots, u_n$ of the same length as $\mathbf{p}$, such that the arity of each u_i is the same as the arity of p_i, and $F^*(\mathbf{u})$ is defined recursively:

- $F^* = F$ for any atomic formula F that does not contain members of $\mathbf{p}$,
- $p_i(\mathbf{t})^* = u_i(\mathbf{t})$ for any predicate symbol p_i belonging to $\mathbf{p}$ and any list $\mathbf{t}$ of terms,
- $(F \wedge G)^* = F^* \wedge G^*$
- $(F \vee G)^* = F^* \vee G^*$
- $(F \rightarrow G)^* = (F^* \rightarrow G^*) \wedge (F \rightarrow G)$
- $(\forall x F)^* = \forall x F^*$
- $(\exists x F)^* = \exists x F^*$

If the list $\mathbf{p}$ is empty, then we understand $\mathrm{SM}_{\mathbf{p}}[F]$ as F. For a finite theory Γ, we write $\mathrm{SM}_{\mathbf{p}}[\Gamma]$ to represent $\mathrm{SM}_{\mathbf{p}}[F]$, where F is the conjunction of all formulas in Γ.

Programs with aggregates as many-sorted first-order sentences

In this section we present the translation τ^* that turns a program Π (whose aggregates lack positive recursion) into a first-order sentence with equality over a signature σ_Π of *two sorts*. We start by defining this signature. To do so, we must first introduce the concepts of a global variable and an aggregate symbol.

A variable is said to be *global* in a rule if

1. it occurs in any non-aggregate literal, or
2. it occurs in a guard of any aggregate literal.

For instance, in rule

$$p(X) \text{ :- } q(X), \#\mathtt{sum}\{Y, Z : r(X, Y, Z)\} \geq 1. \tag{5}$$

the only global variable is X.

An *aggregate symbol* is a pair $E/\mathbf{X}$, where E is an aggregate element and $\mathbf{X}$ is a list of variables occurring in E. We say that $E/\mathbf{X}$ occurs in rule R if this rule contains an aggregate literal with the aggregate element E and $\mathbf{X}$ is the list of all variables in E that are global in R. For instance, $Y, Z : r(X, Y, Z)/X$ is the only aggregate symbol occurring in rule (5). We say that $E/\mathbf{X}$ occurs in a program if $E/\mathbf{X}$ occurs in some rule of the program. For the sake of readability we associate each aggregate symbol $E/\mathbf{X}$ with a different name $|E/\mathbf{X}|$.

As stated earlier, the signature σ_Π is defined over *two sorts*. The first sort is called the *program sort*; all program terms are of this sort. The second sort is called the *set sort*; it contains entities that are *sets* (of tuples of object constants of the program sort). We denote the two sorts in an intuitive manner: s_{prg} and s_{set}. For a program Π, signature σ_Π contains:

1. all ground terms as object constants of the program sort;
2. all predicate symbols occurring in Π as predicate constants with all arguments of sort program;
3. the comparison symbols other than equality and inequality as predicate constants of arity $s_{prg} \times s_{prg}$;
4. function constants count and sum of arity $s_{set} \rightarrow s_{prg}$;
5. for each aggregate symbol $E/\mathbf{X}$ occurring in Π, a function constant $\mathit{set}_{|E/\mathbf{X}|}$ of arity $s_{prg} \times \cdots \times s_{prg} \rightarrow s_{set}$. This function symbol takes as many arguments of the program sort as there are variables in $\mathbf{X}$. If $\mathbf{X}$ is the empty list, then $\mathit{set}_{|E/\mathbf{X}|}$ is an object constant.

Intuitively, the result of count is the cardinality of the set passed as an argument; the result of sum is the sum of all elements of the set passed as an argument; and $\mathit{set}_{|E/\mathbf{X}|}(t_1, \ldots, t_k)$ represents the set of elements corresponding to the aggregate element E once all global variables in $\mathbf{X} = X_1, \ldots, X_k$ are replaced by ground terms $t_1, \ldots, t_k$. We formalize these claims below.

As customary in arithmetic we use infix notation in constructing atoms that utilize predicate symbols $>, \geq, <, \leq$. Expression $t_1 \neq t_2$ is considered an abbreviation for the formula $\neg(t_1 = t_2)$. In the following, we use letters X, Y, Z and their variants to denote variables of sort s_{prg} and letter S and its variants to denote variables of sort s_{set}. We use their bold face variants to denote lists of variables of that sort.

We now describe a translation τ^* that converts a program into a finite set of first-order sentences. Given a list $\mathbf{Z}$ of global variables in some rule R, we define $\tau^*_{\mathbf{Z}}$ for all elements of R as follows:

1. for every atomic formula A occurring outside of an aggregate literal, its translation $\tau^*_{\mathbf{Z}} A$ is A itself; $\tau^*_{\mathbf{Z}} \bot$ is $\bot$;
2. for an aggregate atom A of form $\#\mathtt{count}\{E\} \prec u$ or $\#\mathtt{sum}\{E\} \prec u$, its translation $\tau^*_{\mathbf{Z}}$ is the atom

$$\mathit{count}(\mathit{set}_{|E/\mathbf{X}|}(\mathbf{X})) \prec u \text{ or } \mathit{sum}(\mathit{set}_{|E/\mathbf{X}|}(\mathbf{X})) \prec u$$

respectively, where $\mathbf{X}$ is the list of variables in $\mathbf{Z}$ occurring in E;

3. for every (basic or aggregate) literal of the form $not\ A$ its translation $\tau^*_{\mathbf{Z}}(not\ A)$ is $\neg\tau^*_{\mathbf{Z}}A$; for every literal of the form $not\ not\ A$ its translation $\tau^*_{\mathbf{Z}}(not\ not\ A)$ is $\neg\neg\tau^*_{\mathbf{Z}}A$.

We now define the translation τ^* as follows:

4. for every rule R of form (4), its translation τ^*R is the universal closure of the implication

$$\tau^*_{\mathbf{Z}}B_1 \wedge \cdots \wedge \tau^*_{\mathbf{Z}}B_n \rightarrow \tau^*_{\mathbf{Z}}\,Head,$$

where $\mathbf{Z}$ is the list of the global variables of R.

5. for every program Π, its translation $\tau^*\Pi$ is the first-order theory containing τ^*R for each rule R in Π.

For example, the result of applying τ^* to a program consisting of rule (5) and the rules

$$s(X) \text{ :- } q(X), \#\mathtt{sum}\{Y : r(X,Y,Z)\} \geq 1. \tag{6}$$
$$t \text{ :- } \#\mathtt{sum}\{Y,Z : r(X,Y,Z)\} \geq 1. \tag{7}$$
$$q(a).\ q(b).\ q(c). \tag{8}$$
$$r(a,1,a).\ r(b,-1,a).\ r(b,1,a).\ r(b,1,b).\ r(c,0,a). \tag{9}$$

is the first-order theory composed of the universal closure of the following formulas:

$$q(X) \wedge sum(set_{e1}(X)) \geq 1 \ \rightarrow\ p(X) \tag{10}$$
$$q(X) \wedge sum(set_{e2}(X)) \geq 1 \ \rightarrow\ s(X) \tag{11}$$
$$sum(set_{e3}) \geq 1 \ \rightarrow\ t \tag{12}$$
$$q(a)\ \ q(b)\ \ q(c) \tag{13}$$
$$r(a,1,a)\ \ r(b,-1,a)\ \ r(b,1,a)\ \ r(b,1,b)\ \ r(c,0,a) \tag{14}$$

where $e1$ and $e2$ are the names for aggregate symbols $Y,Z : r(X,Y,Z)/X$ and $Y : r(X,Y,Z)/X$, respectively; $e3$ is the name for an aggregate symbol $Y,Z : r(X,Y,Z)$. Note that the aggregate symbols corresponding to names $e1$ and $e2$ have a global variable X. Consequently, function symbols set_{e1} and set_{e2} have arity $s_{prg} \rightarrow s_{set}$. The aggregate symbol corresponding to $e3$ has no global variables. Consequently, set_{e3} is an object constant of sort s_{set}.

Semantics of programs with aggregates. For the sake of clarity, we describe the semantics of programs with aggregates in two steps. We start by assuming some restrictions on the form of interpretations of interest. These interpretations have fixed meanings for the symbols of signature σ_Π introduced in conditions 3-5. In the next section, we remove these restrictions on symbols *count*, *sum* and $set_{|E/\mathbf{X}|}$ and fix their meaning by providing appropriate axioms. In both cases, we assume that the interpretation of the symbolic constants is the identity.

Consider additional notation. For a tuple $\mathbf{X}$ of distinct variables, a tuple $\mathbf{x}$ of ground terms of the same length as $\mathbf{X}$, and an expression α that contains variables from $\mathbf{X}$, $\alpha^{\mathbf{X}}_{\mathbf{x}}$ denotes the expression obtained from α by substituting $\mathbf{x}$ for $\mathbf{X}$. An *agg-interpretation* I is a many-sorted interpretation that satisfies the following *conditions*:

1. the domain $|I|^{s_{prg}}$ is the set containing all ground terms of program sort (or ground program terms, for short);
2. I interprets each ground program term as itself;
3. I interprets predicate symbols $>, \geq, <, \leq$ according to the total order chosen earlier;
4. universe $|I|^{s_{set}}$ is the set of all sets of non-empty tuples that can be formed with elements from $|I|^{s_{prg}}$;
5. if $E/\mathbf{X}$ is an aggregate symbol, where E is an aggregate element of form (2), $\mathbf{Y}$ is the list of all variables occurring in E that are not in $\mathbf{X}$, and $\mathbf{x}$ and $\mathbf{y}$ are lists of ground program terms of the same length as $\mathbf{X}$ and $\mathbf{Y}$ respectively, then $set_{|E/\mathbf{X}|}(\mathbf{x})^I$ is the set of all tuples of form $\langle (t_1)^{\mathbf{XY}}_{\mathbf{xy}}, \ldots, (t_k)^{\mathbf{XY}}_{\mathbf{xy}} \rangle$ such that I satisfies $(l_1)^{\mathbf{XY}}_{\mathbf{xy}} \wedge \cdots \wedge (l_m)^{\mathbf{XY}}_{\mathbf{xy}}$;
6. for term t_{set} of sort s_{set}, $count(t_{set})^I$ is $\widehat{\mathtt{count}}(t^I_{set})$;
7. for term t_{set} of sort s_{set}, $sum(t_{set})^I$ is $\widehat{\mathtt{sum}}(t^I_{set})$;

An agg-interpretation satisfies the standard name assumption for object constants of the program sort, but not for object constants and function constants of the set sort.

We say that an agg-interpretation I is a *stable model* of program Π if it satisfies the second-order sentence $\mathrm{SM}_{\mathbf{p}}[\tau^*\Pi]$ with $\mathbf{p}$ being the list of all predicate symbols occurring in Π (note that this excludes predicate constants for the comparisons $>, \geq, <, \leq$).

In general, ASP solvers do not provide a complete first-order interpretation corresponding to a computed stable model. Rather, they list the set of ground atoms corresponding to it. Formally, for an agg-interpretation I, by $Ans(I)$, we denote the set of ground atoms that are satisfied by I and whose predicate symbol is of sort program. If I is a stable model of Π, we say that $Ans(I)$ is an *answer set* of Π.

For example, take Π_1 to denote a program composed of rules (5-9). Let I be an agg-interpretation over σ_{Π_1} such that

$$\begin{aligned} q^I &= \{a,b,c\} \\ r^I &= \{(a,1,a),(b,-1,a),(b,1,a),(b,1,b),(c,0,a)\}. \end{aligned} \tag{15}$$

Conditions 5 and 7 imply that this agg-interpretation also satisfies the following statements

$$\begin{aligned} &set_{e1}(a)^I = \{(1,a)\} && sum(set_{e1}(a))^I = 1 \\ &set_{e1}(b)^I = \{(-1,a),(1,a),(1,b)\} && sum(set_{e1}(b))^I = 1 \\ &set_{e1}(c)^I = \{(0,a)\} && sum(set_{e1}(c))^I = 0 \\ &set_{e2}(a)^I = \{(1)\} && sum(set_{e2}(a))^I = 1 \\ &set_{e2}(b)^I = \{(-1),(1)\} && sum(set_{e2}(b))^I = 0 \\ &set_{e2}(c)^I = \{(0)\} && sum(set_{e2}(c))^I = 0 \\ &set^I_{e3} = \{(1,a),(-1,a),(1,b),(0,a)\} && sum(set_{e3})^I = 1 \end{aligned} \tag{16}$$

Such an agg-interpretation I is a stable model of program Π_1 when $p^I = \{a,b\}$, $s^I = \{a\}$, and $t^I = \mathbf{true}$. It turns out, this program has a unique answer set

$$\{\, q(a),\ q(b),\ q(c),\ r(a,1,a),\ r(b,-1,a), r(b,1,a),\ r(b,1,b),\ r(c,0,a),\ p(a),\ p(b),\ s(a),\ t \,\}.$$

Axiomatization of Aggregates

In this section we show that conditions 5-7 characterizing agg-interpretations can be removed from the meta-logic level by adding new logical sentences to the theory representing a logic program. This provides higher mathematical rigor and allows us to build object-level proofs to reason about programs with aggregates.

We introduce an extended signature σ^*_Π that expands σ_Π with new symbols and new sorts. The new sorts are s_{int} and s_{tuple} that we refer to as *integer* and *tuple*, respectively. We also assume countably infinite sets of integer and tuple variables (variables of sorts s_{int} and s_{tuple}). We use the letter N and its variants to denote integer variables and the letter T and its variants to denote tuple variables. Letters V, W, and their variants denote variables where the sort is explicitly mentioned.

For a program Π, in addition to the symbols of σ_Π, signature σ^*_Π contains:

- the binary function symbol $+$ of arity $s_{int} \times s_{int} \to s_{int}$ to represent arithmetic addition;
- a function symbol $tuple_k$ that takes k arguments for every natural number k such that an aggregate element of form (2) occurs in Π; its arity is $s_{prg} \times \cdots \times s_{prg} \to s_{tuple}$;
- object constant $\overline{\emptyset}$ of sort s_{set} to represent the empty set;
- the binary predicate symbol $\in$ of arity $s_{tuple} \times s_{set}$ to represent set membership;
- the function symbol rem of arity $s_{set} \times s_{tuple} \to s_{set}$;
- the function symbol $weight$ of arity $s_{tuple} \to s_{int}$.

As customary in mathematics, we use infix notation for the function symbol $+$ and the predicate symbol $\in$. Informally, $tuple_k(t_1, \ldots, t_k)$ is a constructor for the k-tuple containing program terms $t_1, \ldots, t_k$; atomic formula $t_{tuple} \in t_{set}$ holds iff tuple t_{tuple} belongs to set t_{set}; $rem(t_{set}, t_{tuple})$ encodes the set obtained by removing tuple t_{tuple} from set t_{set}; and $weight(t_{tuple})$ encodes the weight of tuple t_{tuple} (recall that the syntactic object t_{tuple} is meant to be interpreted as an object of sort s_{tuple}).

Formally, for this extended signature, we extend the set of *conditions* that an *agg-interpretation* I satisfies:

8. the domain $|I|^{s_{int}}$ is the set of all numerals;
9. I interprets $\overline{m} + \overline{n}$ as $\overline{m+n}$,
10. universe $|I|^{s_{tuple}}$ is the set of all tuples of form $\langle d_1, \ldots, d_m \rangle$ with $m \geq 1$ and each $d_i \in |I|^{s_{prg}}$;
11. I interprets each tuple term of form $tuple_k(t_1, \ldots, t_k)$ as the tuple $\langle t^I_1, \ldots, t^I_k \rangle$.
12. I interprets object constant $\overline{\emptyset}$ as the empty set $\emptyset$;
13. I satisfies $t_1 \in t_2$ iff tuple t^I_1 belongs to set t^I_2;
14. $rem(t_{set}, t_{tuple})^I$ is the set obtained by removing tuple t^I_{tuple} from set t^I_{set}; and
15. $weight(t_{tuple})^I$ is the weight of t^I_{tuple}.

Note that $|I|^{s_{set}}$ is the power set of $|I|^{s_{tuple}}$. Also, each agg-interpretation is extended in a unique way: there is a one-to-one correspondence between the agg-interpretations over σ_Π and σ^*_Π. In the sequel, we identify each agg-interpretation in signature σ_Π with its extension in σ^*_Π.

In the remainder of this section, we show how an agg-interpretation can be "axiomatized" in a theory that interprets symbols for arithmetic, tuples, sets, and program object constants in a standard way. Formally, a first-order interpretation I is called *standard* when it satisfies conditions 1-4 and 8-13. Such an interpretation satisfies the standard name assumption for ground program terms and tuples, the standard interpretation of arithmetic symbols, and the standard interpretation of the set theoretic membership predicate. It does not assign any special meaning to symbols $count$, sum, rem, $weight$, and any of the functions constants of form $set_{|E/\mathbf{X}|}$. It is obvious that every agg-interpretation is also a standard interpretation, but not vice-versa.

We now show that agg-interpretations can be characterized as standard interpretations that satisfy a particular class of sentences. To begin with, consider condition 5 of the agg-interpretation definition. It associates an aggregate symbol $E/\mathbf{X}$, where E has the form (2), with a unique set. We characterize this set with the sentence

$$\forall \mathbf{X}\, T\big(T \in set_{|E/\mathbf{X}|}(\mathbf{X}) \leftrightarrow \exists \mathbf{Y}\, \big(T = tuple_k(t_1, \ldots, t_k) \wedge l_1 \wedge \cdots \wedge l_m\big)\big), \tag{17}$$

where $\mathbf{Y}$ is the list of all the variables occurring in E that are not in $\mathbf{X}$. For instance, recall program Π_1 and the aggregate symbol $Y, Z : r(X, Y, Z)/X$ named $e1$ (introduced in the previous section). For this symbol, sentence (17) has the form

$$\forall XT\big(T \in set_{e1}(X) \leftrightarrow \exists YZ\, \big(T = tuple_2(Y, Z) \wedge r(X, Y, Z)\big)\big)$$

For a standard interpretation I over signature $\sigma^*_{\Pi_1}$ satisfying conditions (15) and this sentence, $set_{e1}(b)^I$ is the set $\{(-1, a), (1, a), (1, b)\}$. This set is identical to the one stated in (16) for an agg-interpretation satisfying conditions (15). This observation hints at a general result:

Proposition 1. *Let I be a standard interpretation. Then, I satisfies condition 5 iff it satisfies sentence* (17) *for every function symbol of form $set_{|E/\mathbf{X}|}$.*

Similarly, the meaning of function symbols rem and $weight$ provided by conditions 14 and 15 of the definition of agg-interpretations can be fixed in standard interpretations using the following sentences:

$$\forall STS'\big(rem(S, T) = S' \leftrightarrow \forall T'\big(T' \in S' \leftrightarrow (T' \in S \wedge T' \neq T)\big)\big) \tag{18}$$

$$\forall N X_2 \ldots X_k\, weight(tuple_k(N, X_2, \ldots, X_k)) = N\big) \tag{19}$$

$$\forall X_1 X_2 \ldots X_k\big((\neg \exists N\, X_1 = N) \rightarrow weight(tuple_k(X_1, X_2, \ldots, X_k)) = 0\big). \tag{20}$$

Proposition 2. *Let I be a standard interpretation. Then,*

- *I satisfies condition 14 iff it satisfies sentence* (18)*; and*
- *I satisfies condition 15 iff it satisfies all sentences of form (19-20).*

Formalizing condition 6 requires determining when a set is finite or not, that is, we need a formula $Finite(t_{set})$ that

holds if and only if the set represented by t_{set} is finite. We can formalize this idea using a second-order formula, which states that *there is a natural number n and an injective function from t_{set} into the set $\{i \in \mathbb{N} \mid i \leq n\}$*. Before formalizing this statement, let us introduce some auxiliary definitions. Given a term t_{set} of sort s_{set} and a function symbol f, we define $\mathit{Injective}(f, t_{set})$ as the formula

$$\forall T_1 T_2 \, (T_1 \in t_{set} \wedge T_2 \in t_{set} \wedge f(T_1) = f(T_2) \rightarrow T_1 = T_2.)$$

Intuitively, formula $\mathit{Injective}(f, t_{set})$ represents the fact that the restriction of function f, whose domain is the set corresponding to t_{set}, is injective. If the image of f is of sort s_{prg} and t_1 and t_2 are also terms of sort s_{prg}, we define $\mathit{Image}(f, t_{set}, t_1, t_2)$ as the formula:

$$\forall T \, (T \in t_{set} \rightarrow t_1 \leq f(T) \wedge f(T) \leq t_2)$$

Formula $\mathit{Image}(f, t_{set}, t_1, t_2)$ holds when the image of the restriction of function f, whose domain is the set corresponding to t_{set}, is between t_1 and t_2. Expression $\mathit{Finite}(t_{set})$ stands for the second-order formula

$$\exists f \, (\mathit{Injective}(f, t_{set}) \wedge \exists N \, \mathit{Image}(f, t_{set}, 0, N))$$

where f is a function variable of arity $s_{tuple} \rightarrow s_{int}$. For a term t_{set} of the set sort, we define formula $\mathit{FiniteCount}(t_{set})$ as

$$\forall T \big(T \in t_{set} \rightarrow \exists N \, \big(\mathit{count}(\mathit{rem}(t_{set}, T)) = N \wedge \mathit{count}(t_{set}) = N + \overline{1} \big) \big)$$

Using these formulas we can formalize condition 6 with the help of the following three sentences:

$$\mathit{count}(\overline{\emptyset}) = \overline{0} \tag{21}$$

$$\forall S \, (\mathit{Finite}(S) \rightarrow \mathit{FiniteCount}(S)) \tag{22}$$

$$\forall S \, (\neg \mathit{Finite}(S) \rightarrow \mathit{count}(S) = \mathit{sup}) \tag{23}$$

Proposition 3. *Let I be an interpretation that satisfies all conditions for being an agg-interpretation except conditions 6 and 7. Then, I satisfies condition 6 iff it satisfies sentences (21-23).*

The axiomatization of aggregates with the operation `sum` is similar, but requires characterizing that the set of tuples with non-zero weight is finite (instead of the set of arbitrary tuples). Given a term t_{set} of sort s_{set} and a function symbol f, we define $\mathit{InjectiveWeight}(f, t_{set})$ as the formula

$$\forall T_1 T_2 \big(T_1 \in t_{set} \wedge T_2 \in t_{set} \wedge \mathit{weight}(T_1) \neq 0 \wedge \mathit{weight}(T_2) \neq 0 \wedge f(T_1) = f(T_2) \rightarrow T_1 = T_2 \big)$$

If the image of f is of sort s_{prg} and t_1 and t_2 are also terms of sort s_{prg}, we define $\mathit{ImageWeight}(f, t_{set}, t_1, t_2)$ as formula

$$\forall T \big(T \in t_{set} \wedge \mathit{weight}(T) \neq 0 \rightarrow t_1 \leq f(T) \wedge f(T) \leq t_2 \big).$$

Expression $\mathit{FiniteWeight}(t_{set})$ stands for the second-order formula

$$\exists f \big(\mathit{InjectiveWeight}(f, t_{set}) \wedge \exists N \, \mathit{ImageWeight}(f, t_{set}, 0, N) \big)$$

where f is a function variable of arity $s_{tuple} \rightarrow s_{int}$. For a term t_{set} of the set sort, we define formula $\mathit{FiniteSum}(t_{set})$ as

$$\forall T \big(T \in t_{set} \rightarrow \exists N (\mathit{sum}(\mathit{rem}(t_{set}, T)) = N \wedge \mathit{sum}(t_{set}) = N + \mathit{weight}(T)) \big)$$

We can define sum to have arity $s_{set} \rightarrow s_{int}$ and simplify the formula that stands for $\mathit{FiniteSum}(t_{set})$ as follows:

$$\forall T \big(T \in t_{set} \rightarrow \mathit{sum}(t_{set}) = \mathit{sum}(\mathit{rem}(t_{set}, T)) + \mathit{weight}(T) \big)$$

Note that a similar simplification cannot be made for count because sometimes it returns sup, which is not of sort int. We also define $\mathit{ZeroWeight}(t_{set})$ as

$$\forall T \, (T \in t_{set} \rightarrow \mathit{weight}(T) = 0)$$

which holds when all members of t_{set} have zero-weight.

Using these formulas we can formalize condition 7 with the help of the following three sentences:

$$\forall S \, \big(\mathit{ZeroWeight}(S) \rightarrow \mathit{sum}(S) = \overline{0} \big) \tag{24}$$

$$\forall S \, (\mathit{FiniteWeight}(S) \rightarrow \mathit{FiniteSum}(S)) \tag{25}$$

$$\forall S \, \big(\neg \mathit{FiniteWeight}(S) \rightarrow \mathit{sum}(S) = \overline{0} \big) \tag{26}$$

In particular, note that (24) entails $\mathit{sum}(\overline{\emptyset}) = \overline{0}$.

Proposition 4. *Let I be an interpretation that satisfies all conditions for being an agg-interpretation except conditions 6 and 7. Then, I satisfies condition 7 iff it satisfies sentences (24-26).*

The theorem below follows directly from Propositions 1– 4. Symbol $\mathbf{p}$ refers to the list of all predicate symbols occurring in Π.

Theorem 1. *A set of ground atoms M is an answer set of a program Π iff there exists some standard model I of $\mathrm{SM}_{\mathbf{p}}[\tau^* \Pi]$ that satisfies all sentences of form (17-26) and $M = \mathit{Ans}(I)$.*

First-order Characterization

There is a wide class of programs without aggregates for which the second-order SM operator can be replaced by a first-order formalization. This includes completion in the case of tight programs (Ferraris, Lee, and Lifschitz 2011) or, more generally, loop formulas (Lee and Meng 2011). The same replacement also works for our translation for programs with aggregates. Yet the resulting formula, when we consider the axiomatization approach, is not a first-order formula due to the quantification over function symbols in formulas $\mathit{Finite}(t_{set})$ and $\mathit{FiniteWeight}(t_{set})$. These formulas are necessary to distinguish between finite and infinite sets. However, in practice, ASP solvers impose restrictions on programs that ensure that programs have finite answer sets and finite aggregates.

Formally, we say that an interpretation I *has finite aggregates* if the set $set_{|E/\mathbf{X}|}(\mathbf{x})^I$ is finite for every aggregate symbol $E/\mathbf{X}$ and any list $\mathbf{x}$ of ground program terms of the same length as $\mathbf{X}$. A program Π *has finite aggregates* if all standard models of $\mathrm{SM}[\tau^* \Pi]$ have finite aggregates. In the rest of this section, we focus on programs with finite aggregates and we disregard how this property is obtained.

Given two terms t_{set}, t'_{set} of the set sort, we define the formula $\mathit{Subset}(t_{set}, t'_{set})$ as

$$\forall T \, \big(T \in t_{set} \rightarrow T \in t'_{set} \big)$$

stating that t_{set} is a subset of t'_{set}. In the case of programs that have finite aggregates, we can replace sentences (22,23;25,26) by the sentences of the form

$$\forall \mathbf{X}\ S\big(\mathit{Subset}(S, set_{|E/\mathbf{X}|}(\mathbf{X})) \rightarrow \mathit{FiniteCount}(S)\big) \quad (27)$$

$$\forall \mathbf{X}\ S\big(\mathit{Subset}(S, set_{|E/\mathbf{X}|}(\mathbf{X})) \rightarrow \mathit{FiniteSum}(S)\big) \quad (28)$$

where $E/\mathbf{X}$ is an aggregate symbol. Intuitively, sentences (27) and (28) have the same meaning as the pairs of sentences (22,23) and (25,26), respectively, but with some restrictions. First, this formalization is appropriate only if the interpretation of $set_{|E/\mathbf{X}|}(\mathbf{x})$ results in a finite set. Furthermore, the interpretation of *count* and *sum* is only fixed for subsets of sets corresponding to terms of the form $set_{|E/\mathbf{X}|}(\mathbf{x})$. Hence, there may be non standard interpretations that satisfy these sentences, which interpret those symbols differently than their intended meaning when applied to other sets. Interestingly, those sets do not correspond to any term in the theory we are interested in. The reason to include all subsets in these sentences is that $\mathit{FiniteCount}(S)$ and $\mathit{FiniteSum}(S)$ recursively refer to some of their subsets. The following result shows that in the case of programs with finite aggregates we can use the introduced first-order axiomatization.

Theorem 2. *A set of ground atoms M is an answer set of some program Π with finite aggregates iff there is some standard model I of $\mathrm{SM}_{\mathbf{p}}[\tau^*\Pi]$ that satisfies all sentences of form (17-21,24,27,28) and $M = Ans(I)$.*

Relation with Abstract Gringo

The abstract gringo semantics of logic programs use a translation which turns a program into a set of infinitary propositional formulas (Gebser et al. 2015). These semantics capture the behavior of the answer set solver `clingo` when it evaluates a program with aggregates. For space reasons, we refer to this work for formal definitions of infinitary propositional formulas and stable models of these formulas.

We now present a simplified version of the abstract gringo translation which is equivalent to the original in the studied fragment. A rule or an aggregate (in a rule) is called *closed* if it has no global variables. An *instance* of a rule R is any rule that can be obtained from R by substituting ground terms for all global variables.

For a closed aggregate element E of form (2) with $\mathbf{Y}$ being the list of non-global variables occurring in it, Ψ_E denotes the set of tuples $\mathbf{y}$ of ground program terms of the same length as $\mathbf{Y}$. Let E be an aggregate atom of form (3), Δ be a subset of Ψ_E and $[\Delta] = \{\mathbf{t}^{\mathbf{Y}}_{\mathbf{y}} \mid \mathbf{y} \in \Delta\}$ with $\mathbf{t}$ being the tuple $\langle t_1, \ldots, t_k\rangle$. Then, Δ *justifies* an aggregate atom if relation $\prec$ holds between $\widehat{\mathtt{count}}([\Delta])$ (resp. $\widehat{\mathtt{sum}}([\Delta])$) and u. For example, if E is aggregate element $3, X, Y : p(X, Y)$, then Ψ_E is the set of all tuples of ground program terms of length 2. If Δ is $\{\langle a, b\rangle, \langle 5, b\rangle\}$, then $[\Delta] = \{\langle 3, a, b\rangle, \langle 3, 5, b\rangle\}$. Thus, Δ justifies aggregate atom $\#\mathtt{sum}\{3, X, Y : p(X, Y)\} \geq 5$, but not $\#\mathtt{sum}\{3, X, Y : p(X, Y)\} \geq 7$.

The abstract gringo translation τ is defined as follows:

1. for every ground atom A, its translation τA is A itself; $\tau\bot$ is $\bot$,
2. for every ground comparison $t_1 \prec t_2$, its translation $\tau(t_1 \prec t_2)$ is $\top$ if the relation $\prec$ holds between terms t_1 and t_2 according to the total order selected above and $\bot$ otherwise;
3. for aggregate atom A of form (3), τA is formula
$$\bigwedge_{\Delta \in \chi} \left(\bigwedge_{\mathbf{y} \in \Delta} \mathbf{l}^{\mathbf{Y}}_{\mathbf{y}} \rightarrow \bigvee_{\mathbf{y} \in \Psi_E \setminus \Delta} \mathbf{l}^{\mathbf{Y}}_{\mathbf{y}} \right) \quad (29)$$
where χ is the set of subsets Δ of Ψ_E that do not justify A, and $\mathbf{l}$ stands for the conjunction $\tau l_1 \wedge \cdots \wedge \tau l_m$;
4. for every (basic or aggregate) literal L of form $not\ A$, its translation τL is $\neg\tau A$; if L is of form $not\ not\ A$, its translation τL is $\neg\neg\tau A$;
5. for every closed rule R of form (4), its translation τR is the implication
$$\tau B_1 \wedge \cdots \wedge \tau B_n \rightarrow \tau \mathit{Head};$$
6. for every non-closed rule R, its translation τR is the conjunction of the result of applying τ to all its instances;
7. for every program Π, its translation $\tau\Pi$ is the infinitary theory containing τR for each rule R in Π.

A set of ground atoms $\mathcal{A}$ is a *gringo answer set* of a program Π if $\mathcal{A}$ is a stable model of $\tau\Pi$ in infinitary propositional logic.

Theorem 3. *The answer sets of any program (whose aggregates have no positive recursion) coincide with its gringo answer sets.*

Conclusions and Future Work

In this paper, we have provided a characterization of the semantics of programs with aggregates that bypasses grounding. This is achieved by introducing a translation from logic programs to many-sorted first-order sentences together with an axiomatization in second-order logic. Interestingly, in the studied fragment (programs whose aggregates have no positive recursion), our semantics coincides with the semantics of the widely used solver `clingo` (Gebser et al. 2015). Furthermore, for many practical programs the second-order axiomatization can be replaced by first-order sentences. This paves the way for the use of first-order theorem provers for reasoning about this class of programs, something that, to the best of our knowledge, was not possible before our characterization. The potential utility of this contribution is best showcased by `anthem`: a proof assistant that relies on the theorem prover `vampire` (Kovács and Voronkov 2013) to check the correctness of `clingo` programs. Currently, this tool can only be applied to programs with*out* aggregates. This paper opens the door to extend this tool to programs that contain aggregates without positive recursion. This is one of the directions for our future research. Another future line of work is to extend our characterization to programs with positive recursion through aggregates. This is something that requires further study as there are several competing semantics. In addition, we plan to investigate how the methodology for constructing formal arguments about the correctness of logic programs as advocated in (Cabalar, Fandinno, and Lierler 2020) can be extended to programs with aggregates.

Acknowledgements

We would like to thank Vladimir Lifschitz for his valuable feedback on multiple iterations of this project. We also gratefully acknowledge the anonymous reviewers whose comments have improved the quality of this paper.

References

Cabalar, P. 2011. Functional answer set programming. *Theory and Practice of Logic Programming*, 11(2-3): 203–233.

Cabalar, P.; Fandinno, J.; Fariñas del Cerro, L.; and Pearce, D. 2018. Functional ASP with Intensional Sets: Application to Gelfond-Zhang Aggregates. *Theory and Practice of Logic Programming*, 18(3-4): 390–405.

Cabalar, P.; Fandinno, J.; and Lierler, Y. 2020. Modular Answer Set Programming as a Formal Specification Language. *Theory and Practice of Logic Programming*, 20: 767–782.

Cabalar, P.; Fandinno, J.; Schaub, T.; and Schellhorn, S. 2019. Gelfond-Zhang aggregates as propositional formulas. *Artificial Intelligence*, 274: 26–43.

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

Dovier, A.; Pontelli, E.; and Rossi, G. 2003. Intensional Sets in CLP. In Palamidessi, C., ed., *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, 284–299. Springer.

Faber, W.; Pfeifer, G.; and Leone, N. 2011. Semantics and Complexity of Recursive Aggregates in Answer Set Programming. *Artificial Intelligence*, 175(1): 278–298.

Fandinno, J.; Lifschitz, V.; Lühne, P.; and Schaub, T. 2020. Verifying Tight Logic Programs with anthem and vampire. *Theory and Practice of Logic Programming*, 5(20): 735–750.

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

Ferraris, P.; Lee, J.; and Lifschitz, V. 2011. Stable models and circumscription. *Artificial Intelligence*, 175(1): 236–263.

Fox, D.; and Gomes, C., eds. 2008. *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*. AAAI Press.

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

Gelfond, M.; and Zhang, Y. 2014. Vicious Circle Principle and Logic Programs with Aggregates. *Theory and Practice of Logic Programming*, 14(4-5): 587–601.

Gelfond, M.; and Zhang, Y. 2019. Vicious Circle Principle, Aggregates, and Formation of Sets in ASP Based Languages. *Artificial Intelligence*, 275: 28–77.

Kovács, L.; and Voronkov, A. 2013. First-Order Theorem Proving and Vampire. In Sharygina, N.; and Veith, H., eds., *Proceedings of the Twenty-fifth International Conference on Computer Aided Verification (CAV'13)*, volume 8044 of *Lecture Notes in Computer Science*, 1–35. Springer-Verlag.

Lee, J.; Lifschitz, V.; and Palla, R. 2008. A Reductive Semantics for Counting and Choice in Answer Set Programming. In (Fox and Gomes 2008), 472–479.

Lee, J.; and Meng, Y. 2011. First-Order Stable Model Semantics and First-Order Loop Formulas. *J. Artif. Intell. Res.*, 42: 125–180.

Lifschitz, V. 2008. What Is Answer Set Programming? In (Fox and Gomes 2008), 1594–1597.

Lifschitz, V. 2019. *Answer Set Programming*. Springer Publishing Company, Incorporated, 1st edition. ISBN 3030246574.

Pearce, D.; and Valverde, A. 2005. A First Order Nonmonotonic Extension of Constructive Logic. *Studia Logica*, 30(2-3): 321–346.

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.

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

Son, T.; and Pontelli, E. 2007. A Constructive Semantic Characterization of Aggregates in Answer Set Programming. *Theory and Practice of Logic Programming*, 7(3): 355–375.