Axiomatization of Aggregates in Answer Set Programming

Jorge Fandinno*, Zachary Hansen*, Yuliya Lierler*

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

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.
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 \(\pi\). Program terms are either numerals, symbolic constants, variables or either of the special symbols \(\text{inf}\) and \(\text{sup}\). 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

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

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

\[ \neq < > \leq \geq \] (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

\[ t_1, \ldots, t_k : l_1, \ldots, l_m \] (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

\[ \#_{\text{op}}\{E\} < u \] (3)

where \(\text{op}\) is an operation name, \(E\) is an aggregate element, \(<\) is one of the comparison symbols in (1), and \(u\) is a program term, called guard. We consider operation names \(\text{count}\) and \(\text{sum}\). For example, expression

\[ \#\text{sum}\{K, X, Y : \text{in}(X, Y), \text{cost}(K, X, Y)\} > J \]

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

\[ \text{Head} : B_1, \ldots, B_n, \] (4)

where

- \(\text{Head}\) 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 \(\rightarrow\) 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 to 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 \(\neg r(X, Y, Z)\), \(\#\text{sum}\{Y, Z : \neg r(X, Y, Z)\}\) and \(\#\text{sum}\{Y, Z : r(X, Y, Z)\}\) contain no strictly positive occurrence of \(r/3\). For a program \(\Pi\), its (directed predicate) dependency graph is defined by

1. a set of vertices containing all predicate symbols occurring in \(\Pi\),
2. a set of edges containing an edge \((h, b)\) for every normal rule (4) with \(b\) being the predicate symbol occurring in the atom \(\text{Head}\) of the rule and \(h\) 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 \(\Pi\) have no positive recursion if, for every normal rule \(\text{R}\) 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 \(\text{R}\) to the predicate symbol occurring in \(\text{Head}\) of \(\text{R}\). For instance, a program consisting of rule

\[ r(X, Y, Z) :- q(X, Y, Z), \#\text{sum}\{Y, Z : \neg r(X, Y, Z)\} \]

has no aggregate with positive recursion; a program consisting of

\[ r(X, Y, Z) :- q(X, Y, Z), \#\text{sum}\{Y, Z : r(X, Y, Z)\} \]

has an aggregate with positive recursion.

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

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

Operator SM for many-sorted signature. Here, we recall the standard definition of many-sorted first-order logic. A signature \(\sigma\) 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 non-empty universe \(|I|^s\) for each sort \(s\). When sort \(s_1\) is a sort of sort \(s_2\), an interpretation additionally satisfies the condition \(|I|^s_1 \subseteq |I|^s_2\). The notion of satisfaction is analogous.

---

1The 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 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:

\[ \land, \lor, \rightarrow, \parallel, \forall, \exists \]

Negation, truth and equivalence are assumed to be abbreviations: \( F \rightarrow \perp \) stands for \( \neg F \), \( \perp \rightarrow \perp \) stands for \( \top \), and \( (F \rightarrow G) \land (G \rightarrow F) \) 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

\[ \forall W(u(W) \rightarrow p(W)) \]

where \( W \) is a tuple of distinct object variables matching the arity of \( p \) and \( u \). If \( p \) and \( u \) 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 \( u \leq p \) stands for the conjunction

\[ (u_1 \leq p_1) \land \cdots \land (u_n \leq p_n) \]

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

\[ F \land \neg \exists u ((u < p) \land F^*(u)) \]

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

1. \( F^* = F \) for any atomic formula \( F \) that does not contain members of \( p \).
2. \( p_i(t)^* = u_i(t) \) for any predicate symbol \( p_i \) belonging to \( p \) and any list \( t \) of terms,
3. \( (F \land G)^* = F^* \land G^* \)
4. \( (F \lor G)^* = F^* \lor G^* \)
5. \( (F \rightarrow G)^* = (F^* \rightarrow G^*) \land (F \rightarrow G) \)
6. \( \forall x F^* = \forall x F^* \)
7. \( \exists x F^* = \exists x F^* \)

If the list \( p \) is empty, then we understand \( \text{SM}_p[F] \) as \( F \). For a finite theory \( \Gamma \), we write \( \text{SM}_p[\Gamma] \) to represent \( \text{SM}_p[F] \), where \( F \) is the conjunction of all formulas in \( \Gamma \).

**Programs With Aggregates as Many-Sorted First-Order Sentences**

In this section we present the translation \( \tau^* \) that turns a program \( \Pi \) (whose aggregates lack positive recursion) into a first-order sentence with equality over a signature \( \sigma_{\Pi} \) 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

\[ p(X) \land q(X), \#\text{sum}(Y, Z : r(X, Y, Z)) \geq 1 \]

the only global variable is \( X \).

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

As stated earlier, the signature \( \sigma_{\Pi} \) 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: \( s_{\text{prg}} \) and \( s_{\text{set}} \). For a program \( \Pi \), signature \( \sigma_{\Pi} \) contains:

1. all ground terms as object constants of the program sort;
2. all predicate symbols occurring in \( \Pi \) as predicate constants with all arguments of sort program;
3. the comparison symbols other than equality and inequality as predicate constants of arity \( s_{\text{prg}} \times s_{\text{prg}} \);
4. function constants \( \text{count} \) and \( \text{sum} \) of arity \( s_{\text{set}} \rightarrow s_{\text{prg}} \);
5. for each aggregate symbol \( E/X \) occurring in \( \Pi \), a function constant \( \text{set}_{E/X} \) of arity \( s_{\text{prg}} \times \cdots \times s_{\text{prg}} \rightarrow s_{\text{set}} \).

This function symbol takes as many arguments of the program sort as there are variables in \( X \). If \( X \) is the empty list, then \( \text{set}_{E/X} \) is an object constant.

Intuitively, the result of \( \text{count} \) is the cardinality of the set passed as an argument; the result of \( \text{sum} \) is the sum of all elements of the set passed as an argument; and \( \text{set}_{E/X}(t_1, \ldots, t_k) \) represents the set of elements corresponding to the aggregate element \( E \) once all global variables in \( X = X_1, \ldots, X_k \) 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 \( s_{\text{prg}} \) and letter \( S \) and its variants to denote variables of sort \( s_{\text{set}} \). We use their bold face variants to denote lists of variables of that sort.

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

1. for every atomic formula \( A \) occurring outside of an aggregate literal, its translation \( \tau^*_Z A \) is \( A \) itself; \( \tau^*_Z \perp \) is \( \perp \);
2. for an aggregate atom \( A \) of form \( \#\text{count}(E) < u \) or \( \#\text{sum}(E) < u \), its translation \( \tau^*_Z \) is the atom

\[ \text{count}(\text{set}_{E/X}(X)) < u \text{ or } \text{sum}(\text{set}_{E/X}(X)) < u \]
respectively, where $X$ is the list of variables in $Z$ occurring in $E$.

3. for every (basic or aggregate) literal of the form $\neg A$ its translation $\tau_2^{\neg A}$ is $\tau_2^A$; for every literal of the form $\neg\neg A$ its translation $\tau_2^{\neg\neg A}$ is $\neg\tau_2^A$.

We now define the translation $\tau^*$ as follows:

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

$$\tau_2^R B_1 \land \cdots \land \tau_2^R B_n \rightarrow \tau_2^\text{Head},$$

where $Z$ is the list of the global variables of $R$.

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

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

$$\begin{align*}
s(X) & : q(X), \# \sum \{ Y : r(X, Y, Z) \} \geq 1. \\
t & : \# \sum \{ Y : r(X, Y, Z) \} \geq 1. \\
q(a), q(b), q(c). \\
\tau(a, 1, a), r(b, -1, a), r(b, 1, a), r(b, 1, b), r(c, 0, a).
\end{align*}$$

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

$$\begin{align*}
q(X) & \land \sum(\text{set}_{c1}(X)) \geq 1 \rightarrow p(X) \\
q(X) & \land \sum(\text{set}_{c2}(X)) \geq 1 \rightarrow s(X) \\
\text{sum}(\text{set}_{c3}) & \geq 1 \rightarrow t \\
q(a) & \land \sum(b) \land \sum(c) \\
\tau(a, 1, a) & \land \tau(b, -1, a) \land \tau(b, 1, a) \land \tau(b, 1, b) \land \tau(c, 0, a)
\end{align*}$$

where $c1$ and $c2$ are the names for aggregate symbols $Y, Z : r(X, Y, Z)/X$ and $Y : r(X, Y, Z)/X$, respectively; $c3$ is the name for an aggregate symbol $Y, Z : r(X, Y, Z)$. Note that the aggregate symbols corresponding to names $c1$ and $c2$ have a global variable $X$. Consequently, function symbols $\text{set}_{c1}$ and $\text{set}_{c2}$ have arity $\text{set}_{\text{prg}} \rightarrow s_{\text{set}}$. The aggregate symbol corresponding to $c3$ has no global variables. Consequently, $\text{set}_{c3}$ is an object constant of sort $s_{\text{set}}$.

**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 $\sigma_{\Pi}$ introduced in conditions 3-5. In the next section, we remove these restrictions on symbols $\text{count}, \sum$, and $\text{set}_{\text{prg}}$, 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 $X$ of distinct variables, a tuple $x$ of ground terms of the same length as $X$, and an expression $\alpha$ that contains variables from $X$, $\alpha^x_X$ denotes the expression obtained from $\alpha$ by substituting $x$ for $X$. An *agg-interpretation* $I$ is a many-sorted interpretation that satisfies the following conditions:

1. the domain $|I|^{\text{prg}}$ 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 $|I|^{\text{set}}$ is the set of all sets of non-empty tuples that can be formed with elements from $|I|^{\text{prg}}$;

5. if $E/X$ is an aggregate symbol, where $E$ is an aggregate element of form (2), $Y$ is the list of all variables occurring in $E$ that are not in $X$, and $x$ and $y$ are lists of ground program terms of the same length as $X$ and $Y$ respectively, then $\text{set}_{\text{prg}}[E/X](X)$ is the set of all tuples of form $(\{t_1\}_{xy}^X, \ldots, \{t_k\}_{xy}^X)$ such that $I$ satisfies $\{l_1\}_{xy}^X \land \cdots \land \{l_m\}_{xy}^X$.

6. for term $t_{\text{set}}$ of sort $s_{\text{set}}$, $\text{count}(t_{\text{set}})^I$ is $\text{count}(t_{\text{set}})^I$;

7. for term $t_{\text{set}}$ of sort $s_{\text{set}}$, $\sum(t_{\text{set}})^I$ is $\sum(t_{\text{set}})^I$;

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 $\Pi$ if it satisfies the second-order sentence $\text{SM}_{\Pi}^{\Pi}$ with $p$ being the list of all predicate symbols occurring in $\Pi$ (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 $\text{Ans}(I)$, 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 $\Pi$, we say that $\text{Ans}(I)$ is an *answer set* of $\Pi$.

For example, take $\Pi_1$ to denote a program composed of rules (5-9). Let $I$ be an agg-interpretation over $\sigma_{\Pi_1}$, such that

$$q^I = \{a, b, c\} \quad r^I = \{(a, 1, a), (b, -1, a), (b, 1, a), (b, 1, b), (c, 0, a)\}.$$ 

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

$$\begin{align*}
\text{set}_{c1}(a)^I & = \{(1, a)\} \quad \text{sum}(\text{set}_{c1}(a))^I = 1 \\
\text{set}_{c1}(b)^I & = \{(-1, a), (1, a), (1, b)\} \quad \text{sum}(\text{set}_{c1}(b))^I = 1 \\
\text{set}_{c2}(c)^I & = \{(0)\} \quad \text{sum}(\text{set}_{c2}(c))^I = 0 \\
\text{set}_{c3} & = \{(1), (1)\} \quad \text{sum}(\text{set}_{c3})^I = 1 \\
\text{set}_{c4} & = \{(1, a), (1, a), (1, b), (0, a)\} \quad \text{sum}(\text{set}_{c4})^I = 1
\end{align*}$$

Such an agg-interpretation $I$ is a stable model of program $\Pi_1$ when $p^I = \{a, b\}$, $s^I = \{a\}$, and $t^I = \text{true}$. It turns out, this program has a unique answer set

$$\{q(a), q(b), q(c), r(a, 1, a), r(b, -1, a), r(b, 1, a), r(b, 1, b), r(c, 0, a), p(b), p(b), s(a), t\}.$$
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 $\sigma$ that expands $\sigma$ with new symbols and new sorts. The new sorts are $s_{int}$ and $stuple$ 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 $stuple$). We use the letter $N$ and its variants to denote integer variables and the letter $T$ and its variants to denote tuple variables. Letters $V, W, \ldots$ and their variants denote variables where the sort is explicitly mentioned.

As customary in mathematics, we use infix notation for the function symbol + and the predicate symbol ∈. Informally, $t_{tuple}(t_1, \ldots, t_k)$ is a constructor for the $k$-tuple containing program terms $t_1, \ldots, t_k$; atomic formula $t_{tuple} \in t_{set}$ holds if 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 $stuple$).

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

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

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 $\sigma$ and $\sigma$. In the sequel, we identify each agg-interpretation in signature $\sigma$ with its extension in $\sigma$. 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 $set_{E/X}$. 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 $E/X$, where $E$ has the form (2), with a unique set. We characterize this set with the sentence

$$\forall X \left( T \in set_{E/X}(X) \leftrightarrow \exists Y \left( \left( T = tuple_k(t_1, \ldots, t_k) \land I_1 \land \cdots \land I_m \right) \right) \right), \quad (17)$$

where $Y$ is the list of all the variables occurring in $E$ that are not in $X$. For instance, recall program $\Pi_3$ and the aggregate symbol $Y, Z : r(X, Y, Z)/X$ named $e_1$ (introduced in the previous section). For this symbol, sentence (17) has the form

$$\forall X T( T \in set_{e_1}(X) \leftrightarrow \exists Y ( T = tuple_k(Y, Z) \land r(X, Y, Z) ) )$$

For a standard interpretation $I$ over signature $\sigma$, satisfying conditions (15) and this sentence, $set_{e_1}(b)^I$ 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 $set_{E/X}$. 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:

$$\forall STS \left( \left( rem(S, T) = S' \leftrightarrow \forall T' ( T' \in S' \leftrightarrow ( T' \in S \land T' \neq T ) ) \right) \right), \quad (18)$$

$$\forall X_1 X_2 \ldots X_k \left( weight(t_{tuple}(N, X_2, \ldots, X_k)) = N \right) \quad (19)$$

$$\forall X_1 X_2 \ldots X_k \left( \left( \neg \exists N ( X_1 = N ) \Rightarrow weight(t_{tuple}(X_1, X_2, \ldots, X_k)) = 0 \right) \right) \quad (20)$$

**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 $\text{Finite}(t_{set})$ 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 $\{ i \in N \mid i \leq n \}$. 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 $\text{Injective}(f, t_{set})$ as the formula

$$\forall T_1 T_2 (T_1 \in t_{set} \land T_2 \in t_{set} \land f(T_1) = f(T_2) \Rightarrow T_1 = T_2.)$$

Intuitively, formula $\text{Injective}(f, t_{set})$ 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 $\text{Image}(f, t_{set}, t_1, t_2)$ as the formula:

$$\forall T ( T \in t_{set} \rightarrow t_1 \leq f(T) \land f(T) \leq t_2 )$$
Formula $\text{Image}(f, t_{set}, t_1, t_2)$ 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 $\text{Finite}(t_{set})$ stands for the second-order formula

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

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

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

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 $\text{ImageWeight}(f, t_{set}, t_1, t_2)$ as formula

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

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

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

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 $\text{FiniteSum}(t_{set})$ as

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

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

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

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

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

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:

$$\forall S (\text{ZeroWeight}(S) \rightarrow \text{sum}(S) = \text{sup})$$

$$\forall S (\text{FiniteWeight}(S) \rightarrow \text{FiniteSum}(S))$$

$$\forall S (\neg \text{FiniteWeight}(S) \rightarrow \text{sum}(S) = \text{sup})$$

In particular, note that (24) entails $\text{sum}(\emptyset) = \text{sup}$.

**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).

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

**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 $\text{Finite}(t_{set})$ and $\text{FiniteWeight}(t_{set})$. 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 $\text{set}_{E = X}(x)^I$ is finite for every aggregate symbol $E / X$ and any list $x$ of ground program terms of the same length as $X$. A program $\Pi$ has finite aggregates if all standard models of $\text{SM}_{\text{pr}}{\tau^*}\Pi$ have finite aggregates. In the rest of this section, we focus on programs with finite aggregates and disregard how this property is obtained.

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

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

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

$$\forall X S (\text{Subset}(S, \text{set}_{E / X}(X)) \rightarrow \text{FiniteCount}(S))$$

$$\forall X S (\text{Subset}(S, \text{set}_{E / X}(X)) \rightarrow \text{FiniteSum}(S))$$

where $E / X$ 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 $\text{set}_{E / X}(x)$ results in a finite set. Furthermore, the interpretation of $\text{count}$ and $\text{sum}$ is only fixed for subsets of sets corresponding to terms of the form $\text{set}_{E / X}(x)$. 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 $\text{FiniteCount}(S)$ and $\text{FiniteSum}(S)$ 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 \( \Pi \) with finite aggregates if and only if there is some standard model \( I \) of \( SM_P[\tau^*\Pi] \) 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 \( Y \) being the list of non-global variables occurring in it, \( \Psi_E \) denotes the set of tuples \( y \) of ground program terms of the same length as \( Y \). Let \( E \) be an aggregate atom of form (3), \( \Delta \) be a subset of \( \Psi_E \) and \( [\Delta] = \{ t^y \mid y \in \Delta \} \) with \( t \) being the tuple \( \langle t_1, \ldots, t_k \rangle \). Then, \( \Delta \) justifies an aggregate atom if relation \( \preceq \) holds between \( \text{count}(\Delta) \) (resp. \( \text{sum}(\Delta) \)) and \( u \). For example, if \( E \) is aggregate element \( 3, X, Y : p(X, Y) \), then \( \Psi_E \) is the set of all tuples of ground program terms of length 2. If \( \Delta \) is \{\(\langle a, b\rangle, \langle 5, b\rangle\} \), then \( [\Delta] = \{3, a, b, \langle 3, 5, b\rangle\} \). Thus, \( \Delta \) justifies aggregate atom \( \#\text{sum}\{3, X, Y : p(X, Y)\} \geq 5 \) but not \( \#\text{sum}\{3, X, Y : p(X, Y)\} \geq 7 \).

The abstract gringo translation \( \tau \) is defined as follows:

1. for every ground atom \( A \), its translation \( \tau A \) 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), \( \tau A \) is formula
   \[
   \bigwedge_{\Delta \in \chi} \left( \bigwedge_{y \in \Delta} \bigwedge_{x \in \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_E \Delta} \bigwedge_{y \in \Psi_


