Arguing Correctness of ASP Programs with Aggregates

Jorge Fandinno[1 [0000−0002−3917−8717]⋆], Zachary Hansen[1 [0000−0002−8447−4048]⋆], and Yuliya Lierler[1 [0000−0002−6146−623X]⋆]

University of Nebraska Omaha, Omaha NE 68182, USA
{jfandinno,zachhansen,ylierler}@unomaha.edu

Abstract. This paper studies the problem of arguing program correctness for logic programs with aggregates in the context of Answer Set Programming. Cabalar, Fandinno, and Lierler (2020) championed a modular methodology for arguing program correctness. We show how a recently proposed many-sorted semantics for logic programs with aggregates allows us to apply their methodology to this type of program. This is illustrated using well-known encodings for the Graph Coloring and Traveling Salesman problems. In particular, we showcase how this modular approach allows us to reuse the proof of correctness of a Hamiltonian Cycle encoding studied in a previous publication when considering the Traveling Salesman program.

Keywords: ASP · Program Verification · Aggregates

1 Introduction

Answer Set Programming (ASP) [12,13] is a well-established Knowledge Representation paradigm for solving (knowledge-intensive) search/optimization problems. Based on logic programming under the *answer set semantics* [11], the ASP methodology relies on devising a logic program so that its answer sets are in one-to-one correspondence to the solutions of the target problem. The fact that this approach is fully declarative positions it as a firm candidate for producing trustworthy Artificial Intelligence (AI) systems, which require, among other qualities, the assessment that those systems produce correct judgments. Given the declarative nature of ASP, it also seems natural to consider an ASP program as a formal specification on its expected solutions [2,6]. This formal specification is usually the first program that an ASP practitioner writes and later refines to achieve higher solving efficiency [1,10]. The equivalence between the formal specification and the refined program can be manually and, in many cases, even automatically checked using existing tools [2,6,14]. Unfortunately, all these approaches deal exclusively with programs without aggregates, which are expressive constructs commonly used in practice.

⋆ These authors contributed equally.

In this paper, we extend the *verification methodology for logic programs* (or, VLP methodology) developed in [2] to programs that contain non-recursive aggregates. This methodology is reviewed in Section 3. Consider the Graph Coloring (GC) problem encoded in Listing 1.1 using a choice rule with cardinality bounds. Here aggregates provide a succinct and convenient way to model the problem. Arguing correctness of this encoding was out of the scope of [2] as a choice rule with cardinality bounds (exemplified by the rule in line 1 of Listing 1.1) is an abbreviation for a pair of rules that includes an integrity constraint with a `count` aggregate [3]. We show how a recent extension of the SM operator to programs with aggregates ([4]) allows us to apply this methodology to programs of this kind. In addition to the GC problem, we also illustrate this VLP

Listing 1.1. Encoding of the graph coloring problem using the ASP language.

```
1 { assign(V,C) : color(C) } = 1 :- vertex(V).
2 :- edge(V1,V2), assign(V1,C), assign(V2,C).
```

methodology on the Traveling Salesman (TS) problem. These two problems are widely-studied and well understood in the ASP community, and they allow us to highlight two different use cases of aggregates. Additionally, the application of the VLP methodology to the TS problem illustrates the benefits of the "modular approach" it advocates. In particular, we are able to reuse a proof of correctness devised for another program — an encoding of the Hamiltonian Cycle problem studied in [2] that forms a subprogram of the TS encoding — as part of the argument of correctness for the TS encoding.

2 Review: Logic Programs via the Many-sorted Approach

We start by reviewing elements of the syntax and semantics of a logic program with aggregates using the SM operator recently developed in [4]. In this approach, a logic program is considered to be an abbreviation for a many-sorted first-order sentence. The semantics of this sentence are characterized by the "agg" models of the second-order sentence obtained from the application of the SM operator to this first-order sentence. We assume familiarity with basic terminology of logic programs and the SM operator. For the sake of brevity, we focus only on the concepts necessary to understand the contributions of this paper. We refer the reader to [4] for details.

2.1 Syntax of Logic Programs with Aggregates

We consider (non-disjunctive) *rules* of the form $Head \leftarrow B_1, \ldots, B_n$ where *Head* is an atom or the symbol $\bot$, and each B is a literal. We typically omit the $\bot$ symbol and instead write a constraint as a rule with an empty head. A *literal* is either a symbolic literal or an aggregate literal. A *symbolic literal* is either an atom or a *comparison* possibly preceded by one or two occurrences of *not*. Similarly, an *aggregate literal* is an aggregate atom possibly preceded by one or two occurrences of *not*. We assume familiarly with the definitions of program

Listing 1.2. Encoding of a Hamiltonian Cycle problem.

```
1 vertex(X) :- edge(X,Y).
2 vertex(X) :- edge(Y,X).
3 { in(X,Y) } :- edge(X,Y).
4 ra(Y) :- in(a,Y).
5 ra(Y) :- in(X,Y), ra(X).
6 :- not ra(X), vertex(X).
7 :- in(X,Y), in(X,Z), Y != Z.
8 :- in(X,Y), in(Z,Y), X != Z.
```

terms, atoms and comparisons and we focus here on describing the syntax of aggregate atoms.

An *aggregate element* is an expression of the form $t_1, \ldots, t_k : l_1, \ldots, l_m$, where each t_i $(1 \leq i \leq k)$ is a program term and each l_i $(1 \leq i \leq m)$ is a symbolic literal. An *aggregate atom* has the form $\texttt{\#op}\{E\} \prec u$, where `op` is an operation name, E is an aggregate element, $\prec$ is a comparison symbol, and u is a program term, called the *guard*. We consider operation names `count` and `sum`. For example, the following two expressions are two aggregate atoms

```
#count{ V,C : assign(V,C), color(C) } = 1          (1)
#sum{ K,X,Y : in(X,Y), cost(K,X,Y) } > J           (2)
```

that will be used in our examples throughout the paper.

A *choice rule* is an expression of the form

$$\{A_0 : A_1, \ldots, A_k\} \prec u \text{ :- } B_1, \ldots, B_n. \tag{3}$$

where each A_i is an atom, each B_i is a literal, $\prec$ is a comparison symbol and u is a numeral; it is understood as an abbreviation for the following pair of rules

$$A_0 \text{ :- } A_1, \ldots A_k,\ B_1, \ldots, B_n,\ \textit{not not}\ A_0. \tag{4}$$

$$\text{:- } B_1, \ldots, B_n,\ \textit{not}\ \texttt{\#count}\{\mathbf{t} : A_0, A_1, \ldots, A_k\} \prec u. \tag{5}$$

where $\mathbf{t}$ is a list of program terms such that A_0 is of the form $p(\mathbf{t})$ for some symbolic constant p. As usual, we allow that "$\prec u$" or "$: A_1, \ldots A_k$," (or both of them) are omitted from choice rules. If "$\prec u$" is omitted, then (5) is omitted; and if "$: A_1, \ldots A_k$" is omitted, then $A_1, \ldots A_k$ is omitted from (4-5).

For instance, rule 3 in Listing 1.2 — capturing the Hamiltonian Cycle encoding used later in the paper as part of the TS encoding — is a choice rule where both elements are omitted and, thus, it is an abbreviation for the rule

```
in(X,Y) :- edge(X,Y), not not in(X,Y).          (6)
```

As another example, rule 1 in Listing 1.1 is a choice rule where both of these elements are present, and it is understood as an abbreviation for rules

```
assign(V,C) :- vertex(V), color(C), not not assign(V,C).          (7)
:- vertex(V), not #count{ V,C : assign(V,C), color(C) } = 1.      (8)
```

A *program* is a finite set of rules.

2.2 From Rules to Many-sorted First-order Formulas

A logic program is understood as many-sorted first-order sentences over a signature σ_{Π} of *two sorts*, one for *program terms* and one for *sets of tuples of program terms*. We name these sorts *program* and *set*, respectively. To define the class of function symbols of the sort set we introduce the concepts of *global variables* and *set symbols*. A variable is said to be *global* in a rule if (i) it occurs in any non-aggregate literal, or (ii) it occurs in a guard of any aggregate literal. A variable that is not global is called *local*. For instance, in rule (8), variable `V` is global and variable `C` is local. In primitive rules, all variables are trivially global. A *set 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,

$$V, C : assign(V, C), color(C)/V \tag{9}$$

is the only set symbol occurring in rule (8). 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 set symbol $E/\mathbf{X}$ with a different name $|E/\mathbf{X}|$.

As stated earlier, for a program Π, we consider a signature σ_{Π} over *two sorts* that contains: (i) all ground terms as object constants of the program sort; (ii) all predicate symbols occurring in Π as predicate constants with all arguments of sort program; (iii) the comparison symbols other than equality and inequality as binary predicate constants whose arguments are of the program sort; (iv) unary function constants *count* and *sum* of sort program whose unique argument is of sort set; and (v) for each set symbol $E/\mathbf{X}$ occurring in Π, a function constant $set_{|E/\mathbf{X}|}$ of the sort set. This function symbol takes as many arguments of the program sort as there are variables in $\mathbf{X}$. If $\mathbf{X}$ is an empty list, then $set_{|E/\mathbf{X}|}$ is an object constant.

We refer to [4] for the precise definition of the translation τ^* that converts a program into a finite set of first-order sentences. For the purposes of this paper it is only necessary to know the result of applying such a translation to the logic programs encoding the GC problem and the TS problem. For instance, consider rule (8). Translation τ^* applied to this rule produces the first-order sentence:

$$\forall V(vertex(V) \wedge \neg count(set_{asg}(V)) = 1 \rightarrow \bot), \tag{10}$$

where *asg* is the name for set symbol (9). The translation of the program in Listing 1.1 is completed by the following two sentences:

$$\forall VC\big(vertex(V) \wedge color(C) \wedge \neg\neg assign(V, C) \rightarrow assign(V, C)\big) \tag{11}$$

$$\forall V1\ V2\ C\big(edge(V1, V2) \wedge assign(V1, C) \wedge assign(V2, C) \rightarrow \bot\big) \tag{12}$$

Sentence (11) corresponds to rule (7). Sentences (10) and (11) together are the translation of rule 1 in Listing 1.1. Formula (12) is the the result of applying translation τ^* to rule 2 in Listing 1.1. We describe the translation of the TS encoding in Section 3.2. From now on, we assume that, unless otherwise made explicit, formulas with free variables stand for their universal closures.

2.3 Semantics via the SM Operator

The SM operator transforms first-order sentences into second-order sentences with equality. Ferraris, Lee and Lifschitz show that programs without aggregates can be considered as abbreviations for first-order sentences to which this operator is applied [7, Section 2.1]. When a set of such sentences Π is transformed by the SM operator into a second-order theory $\mathrm{SM}_{\mathbf{p}}[\Pi]$, the satisfying Herbrand interpretations of $\mathrm{SM}_{\mathbf{p}}[\Pi]$ where $\mathbf{p}$ is the list of all predicates occurring in the program are exactly the stable models of Π as defined by Gelfond and Lifschitz (1988). Fandinno, Hansen, and Lierler (2022) extend this approach to programs with aggregates by relying on a many-sorted generalization of the SM operator. We refer to [4] for the precise definition of the SM operator. For this paper it is enough to understand two properties of this operator, namely, the Splitting and Completion Theorems.

Splitting and modules. The Splitting Theorem in [8] forms one of the foundations of the VLP methodology championed here. This theorem was recently generalised to the two-sorted case [6]; and a look to the proof shows that its generalization to the many-sorted case is straightforward. Let us recall some necessary notation for this result. An occurrence of a predicate symbol in a formula is called *negated* if it belongs to a subformula of the form $F \to \bot$ (often abbreviated as $\neg F$) and *nonnegated* otherwise. An occurrence of an expression in a formula is called *positive* if the number of implications containing that occurrence in the antecedent is even. It is called *strictly positive* if that number is 0. A *rule* of a first-order formula F is a strictly positive occurrence of an implication in F. The *dependency graph* of a formula is a directed graph that: (i) has all intensional predicate symbols as vertices; and (ii) has an edge from p to q if, for some rule $G \to H$ of F, formula G has a positive nonnegated occurrence of q and H has a strictly positive occurrence of p.

Theorem 1. *(**Splitting Theorem**) Let F and G be many-sorted first-order sentences and let $\mathbf{p}$ and $\mathbf{q}$ be two disjoint tuples of distinct predicate symbols such that (i) each strongly connected component of the dependency graph of $F \wedge G$ is a subset either of $\mathbf{p}$ or $\mathbf{q}$; (ii) F does not have strictly positive occurrences of symbols from $\mathbf{q}$; and (iii) G does not have strictly positive occurrences of symbols from $\mathbf{p}$. Then, $\mathrm{SM}_{\mathbf{pq}}[F \wedge G]$ is equivalent to $\mathrm{SM}_{\mathbf{p}}[F] \wedge \mathrm{SM}_{\mathbf{q}}[G]$.*

In the sequel, we often refer to expression $\mathrm{SM}_{\mathbf{p}}[F]$ as a module, whereas list $\mathbf{p}$ of predicate symbols is called *intensional*. The Splitting Theorem tells us how we can, at times, view a module in terms of other modules. When the list of predicate symbols $\mathbf{p}$ is empty, $\mathrm{SM}_{\mathbf{p}}[F]$ is identical to F. Thus, we may refer to any first-order sentence F as a module.

Completion. The theorem on completion presented here forms an important result that allows us, at times, to replace second-order formula (capturing a module) by an equivalent first-order formula. This is important when we construct formal arguments about models of these formulas as the later is easier to understand. Let $\mathbf{p}$ be a list of intensional predicate constants. A rule $G \to H$ is

called *non-disjunctive* if H is an atomic formula or does not contain intensional symbols (i.e., elements of $\mathbf{p}$). We say that $G \to H$ is a *constraint* with respect to $\mathbf{p}$ if H does not contain members of $\mathbf{p}$. About a nondisjunctive rule $G \to H$ we say that it *defines* an intensional symbol p if H is an atomic formula that begins with p. In the following we assume that F is a conjunction of the universal closures of nondisjunctive rules and constraints with respect to $\mathbf{p}$. If the argument sorts of an intensional symbol p are $s_1, \dots, s_n$, and the rules defining p in F are

$$G_i \to p(\mathbf{t}_i) \qquad i = 1, \dots, k,$$

then the *completed definition* of p in F is the sentence

$$\forall \mathbf{V} \left(p(\mathbf{V}) \leftrightarrow \bigvee_{i=1}^{k} \exists \mathbf{U}_i \left(G_i \wedge \mathbf{V} = \mathbf{t}_i \right) \right), \tag{13}$$

where $\mathbf{V}$ is an n-tuple of fresh variables of sorts $s_1, \dots, s_n$, and $\mathbf{U}_i$ is the list of all variables that are free in $G_i \to p(\mathbf{t}_i)$. The expression $\mathbf{V} = \mathbf{t}_i$ here stands for the conjunction of n equalities between the corresponding members of the tuples $\mathbf{V}$ and $\mathbf{t}_i$. The *completion* $\mathrm{COMP}_{\mathbf{p}}[F]$ of F is the conjunction of all completed definitions of all members of $\mathbf{p}$ in F and all constraints of F. The following result immediately follows from the Main Lemma in [5]:

Theorem 2. *If the dependency graph of F is acyclic, then* $\mathrm{SM}_{\mathbf{p}}[F]$ *and* $\mathrm{COMP}_{\mathbf{p}}[F]$ *are equivalent.*

Agg-interpretations. The semantics of aggregates are defined with respect to a particular class of interpretations that we call *agg-interpretations*. Consider the following 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 of the program sort, denoted $|I|^{s_{prg}}$, is the set containing all ground terms of the 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 in [4] (this is the natural interpretation when applied to numerals, but it also apply to symbolic constants);
4. the domain of the set sort, denoted $|I|^{s_{set}}$, 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 a set symbol, where E is an aggregate element, $\mathbf{Y}$ 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 the form $\langle (t_1)^{\mathbf{XY}}_{\mathbf{xy}}, \dots, (t_k)^{\mathbf{XY}}_{\mathbf{xy}} \rangle$ such that I satisfies $(l_1)^{\mathbf{XY}}_{\mathbf{xy}} \wedge \dots \wedge (l_m)^{\mathbf{XY}}_{\mathbf{xy}}$;

6. for $d \in |I|^{s_{set}}$, $count(d)^I$ is is the numeral corresponding to the cardinality of d, if d is finite; and *sup* otherwise.
7. for $d \in |I|^{s_{set}}$, $sum(d)^I$ is the numeral corresponding to the sum of the weights of all tuples in d, if d contains finitely many tuples with non-zero weights; and 0 otherwise.(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 GRINGO [9].) If d is empty, then $\mathtt{sum}(d)^I = 0$.

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

We say that an agg-interpretation I is a $\mathbf{p}$-*stable model* of program Π if it satisfies $\mathrm{SM}_{\mathbf{p}}[\tau^*\Pi]$, where $\mathbf{p}$ is a list of predicate symbols occurring in Π (note that this excludes predicate constants for comparisons $>, \geq, <, \leq$). An agg-interpretation I is a *stable model* of program Π if it is a $\mathbf{p}$-stable model, where $\mathbf{p}$ is the list of *all* predicate symbols occurring in Π. The stable models of a program defined in this way correspond to the answer sets of the abstract GRINGO language [9] when the aggregates have no positive recursion [4] and with the answer sets of ASP-Core-2 [3].

3 Proving the Correctness of Logic Programs

Cabalar, Fandinno and Lierler (2020) developed a methodology for arguing the correctness of answer set programs, partially reproduced below:

Step I: Decompose the informal description of the problem into independent (natural language) statements[1].

Step II: Fix the public predicates used to represent the problem and its solutions.

Step III: Formalize the specification of the statements as a non-ground modular program, possibly introducing auxiliary predicates.

Step IV: Construct an argument (a "metaproof" in natural language) for the correspondence between the constructed program and the informal description of the problem.

An optional fifth step is to construct a formal proof from the constructed program (treated as a specification) to an alternative encoding. Here we consider proving the adherence of the constructed program to the natural language specification. We now put this methodology in practice for the case of the encodings of two problems: Graph Coloring and Traveling Salesman. The considered encodings contain aggregates. The extension of the SM operator applicable to programs with aggregates makes the use of this methodology possible in our context.

[1] In fact, this is also the first step that students are taught in the introduction to modeling in the ASP course taught at the University of Potsdam: https://teaching.potassco.org/

3.1 The Graph Coloring Problem

Step I applied to the GC problem consists in identifying statements

C1 find an assignment from nodes to colors such that
C2 connected nodes do not have the same color.

Formally, an instance of the GC problem is a triple $\langle V, E, C\rangle$, where

- $\langle V, E\rangle$ is a graph with vertices V and edges $E \subseteq V \times V$, and
- C is a set of labels named *colors*.

A solution to the GC problem is

CF1 a function $asg : V \longrightarrow C$ such that
CF2 every edge $(a, b) \in E$ satisfies condition $asg(a) \neq asg(b)$.

Step II consists of choosing the public predicates to represent the problem, in this example: $vertex/1$, $edge/2$, $color/1$, and $assign/2$. **Step III** consists in formalizing the statements from **Step I** as a non-ground modular program Π. The GC problem is a great illustratory example due to the simplicity of its ASP encoding and the fact that each natural language statement is encoded as exactly one rule. In other words, in this example, each rule constitutes its own module. Rule 1 in Listing 1.1 corresponds to the module

$$\mathrm{SM}_{assign}[(10) \wedge (11)] \tag{14}$$

while rule 2 corresponds to the first-order sentence/module (12). Module (14) formalizes statement **CF1**, that is, it ensures that predicate $assign/2$ encodes a function from vertices to colors. Module (12) formalizes statement **CF2**: it ensures that the function encoded by predicate $assign/2$ satisfies the condition of the statement. By the Splitting Theorem, the conjunction of two modules — (12) and (14) — has the same $assign$-stable models as the $assign$-stable models of the conjunction (10) $\wedge$ (11) $\wedge$ (12); recall that this conjunction corresponds to τ^* applied to the GC encoding in Listing 1.1.

We now turn our attention to **Step IV**. To formalise claim **CF1** about module (14), we prove a general result about modules of a similar form. We say that relation $\mathbf{r}$ *encodes* function $f : A \longrightarrow B$ when $\mathbf{r} = \{(a, f(a)) \mid a \in A\}$. Given sets A and B, we can construct a program whose stable models encode all functions from A to B as follows. (By d^* we denote the name of domain element d, that is, an object constant whose interpretation is d.) Let $G(X)$ and $H(Y)$ be two first-order formulas such that $G(d^*)$ is satisfiable iff d belongs to A and $H(d^*)$ is satisfiable iff d belongs to B. Then, $Fun_{A,B}$ is the conjunction of formulas

$$\forall X\big(G(X) \wedge \neg count(set_{fe}(X)) = 1 \rightarrow \bot\big) \tag{15}$$

$$\forall XY\big(G(X) \wedge H(Y) \wedge \neg\neg f(X, Y) \rightarrow f(X, Y)\big) \tag{16}$$

where fe is the name of the set symbol $X, Y : f(X, Y), H(Y)/X$.

Proposition 1. *For an agg-interpretation I and first-order formulas $G(X)$ and $H(Y)$ containing no positive nonnegated occurrences of $f/2$, take*

$$A = \{d \mid d \in |I|^{s_{prg}} \text{ and } I \models G(d^*)\} \text{ and } B = \{d \mid d \in |I|^{s_{prg}} \text{ and } I \models H(d^*)\}.$$

Then, condition $I \models \mathrm{SM}_f[Fun_{A,B}]$ holds iff $(f/2)^I$ encodes a function from A to B.

Proof. If A is empty, then $(f/2)^I$ encodes the empty function. Hence, in the rest of the proof, we assume that A is non-empty. By the Splitting Theorem $\mathrm{SM}_f[(15) \wedge (16)]$ is equivalent to $\mathrm{SM}_f[(16)] \wedge (15)$. By the Completion Theorem, sentence $\mathrm{SM}_f[(16)]$ is equivalent to the first-order sentence

$$\forall XY\Big(f(X,Y) \leftrightarrow G(X) \wedge H(Y) \wedge \neg\neg f(X,Y))\Big). \tag{17}$$

In turn, this sentence is equivalent in first-order logic to

$$\forall XY\big(f(X,Y) \rightarrow G(X) \wedge H(Y)\big). \tag{18}$$

Let F denote the conjunction of (15) and (18), which is equivalent to $\mathrm{SM}_f[Fun_{A,B}]$.
Left-to-right. Assume that $I \models F$. Pick any $a \in A$. Then, $I \models G(a^*)$ and, since $I \models (15)$, it follows that $I \models (count(set_{fe}(a^*)) = 1)$. Hence, $set_{fe}(a^*)^I = \{\langle a, b_a\rangle\}$ for some $b_a \in |I|^{s_{prg}}$ such that $I \models f(a^*, b_a^*) \wedge H(b_a^*)$. Let $\hat{f}$ be the function such that $\hat{f}(a) = b_a$; $\hat{f}$ is a function from A to B encoded by $(f/2)^I$.
Right-to-left. Let $\hat{f}$ be a function from A to B such that $(f/2)^I = \{(a, \hat{f}(a)) \mid a \in A\}$; in other words $(f/2)^I$ encodes $\hat{f}$. Let us show that $I \models F$. First, for any term $a \in |I|^{s_{prg}}$ such that $I \models G(a^*)$, it follows that $a \in A$ and, thus, $set_{fe}(a^*)^I = \{\langle a, \hat{f}(a)\rangle\}$. This implies that $I \models (15)$. Second, for any $a \in |I|^{s_{prg}}$ and $b \in |I|^{s_{prg}}$ such that $I \models f(a^*, b^*)$ it follows by construction that $a \in A$ and $b = \hat{f}(a)$ and $b \in B$. Hence, $I \models G(a^*) \wedge H(b^*)$ and, thus, $I \models (18)$.

Claim **CF2** about module (12) is argued within the proof of the following theorem that can also be seen as a proof of correctness for the GC encoding presented in Listing 1.1.

Theorem 3. *Let I be an agg-interpretation such that $\langle vertex^I, edge^I, color^I\rangle$ forms an instance of the Graph Coloring problem. Then, $I \models (12) \wedge (14)$ iff $(assign/2)^I$ encodes a function that forms a solution to the considered instance.*

Proof. From Proposition 1, we get that $I \models (14)$ iff $(assign/2)^I$ encodes a function $asg : vertex^I \longrightarrow color^I$ such that $(assign/2)^I = \{(a, asg(a)) \mid a \in vertex^I\}$. Sentence (12) is equivalent to

$$\forall V1\ V2\ C1\ C2\big(edge(V1,V2) \wedge assign(V1,C1) \wedge assign(V2,C2) \rightarrow C1 \neq C2\big).$$

This sentence is satisfied by I iff every edge $(a,b) \in edge^I$ satisfies $asg(a) \neq asg(b)$.

The use of the SM operator allows us to argue the correctness of an encoding in isolation from the way its instances are obtained. In Theorem 3, we only implicitly refer to a specific instance of the GC problem by considering an interpretation I such that $\langle vertex^I, edge^I, color^I \rangle$ forms this instance. In practice, to compute a solution for a considered GC instance one has to extend the program corresponding to the GC problem with an encoding of the instance. Such an instance can be represented by a set of facts utilizing predicates chosen for the representation. For example, facts

$$vertex(a).\ vertex(b).\ edge(a,b).\ color(g).\ color(b).\ color(r).$$

encode an instance $\langle \{a,b\}, \{(a,b)\}, \{g,b,r\} \rangle$ of the GC problem. Answer sets of the program in Listing 1.1 extended with these facts will encode the solutions to the specified instance. However, the general approach followed in Theorem 3 actually allows those facts to be generated by a, perhaps very complex, logic program, as long as it does not use the predicate symbols used in the GC encoding other than the ones used to describe the problem instance. We take the same approach of implicit reference to an instance when arguing the correctness of the TS problem.

3.2 The Traveling Salesman Problem

Let us look into the following variant of the Traveling Salesman problem:

> We are given a directed graph with nodes as cities and edges as roads. We assume the presence of a city named "*a*". Each road directly connects a pair of cities, and costs a salesman some time to traverse (time is expressed as an integer value). The salesman may pass each city exactly once. Find: *a route* traversing *all the cities under* a certain *maximum cost* of total time starting and finishing at city *a*.

Formally, an instance of the TS problem is a quadruple $\langle V, E, cst, m \rangle$, where

- $\langle V, E \rangle$ is a graph assuming one vertex in V named a,
- cst is a function from edges E to integers, and
- m is some integer.

A solution to this instance is a subset of edges $P \subseteq E$ such that

T1 P forms a Hamiltonian cycle of graph $\langle V, E \rangle$ and
T2 the following inequality holds

$$\sum_{e \in P} cst(e) \leq m. \tag{19}$$

This constitutes the application of **Step I** to the TS problem. **Step II** consists in choosing the public predicates to represent the problem: $vertex/1$, $edge/2$, $cost/3$, $maxCost/1$, and $in/2$. We say that an agg-interpretation I *encodes* instance $\langle V, E, cst, m \rangle$ if it satisfies the following conditions:

- $(vertex/1)^I = V$ and $(edge/2)^I = E$;
- $(cost/3)^I = \{(c, v_1, v_2) \mid (v_1, v_2) \in E \text{ and } cst\big((v_1, v_2)\big) = c\}$;
- $(maxCost/1)^I = \{m\}$;

Predicate symbol $in/2$ is meant to capture a solution to the TS problem, i.e., an agg-interpretation I encoding an instance of the TS problem also *encodes* a solution P whenever $(in/2)^I = P$. Using the mentioned predicate symbols, we can capture the TS problem by adding the following rule to the encoding of the Hamiltonian Cycle problem in Listing 1.2:

```
:- #sum{ K,X,Y : in(X,Y), cost(K,X,Y) } > J, maxCost(J).
```
(20)

Translation τ^* applied to the rules in Listing 1.2 and rule (20) results in the sentences (recall that we identify formulas below with their universal closures):

$$edge(X,Y) \rightarrow vertex(X) \tag{21}$$
$$edge(Y,X) \rightarrow vertex(X) \tag{22}$$
$$\neg\neg in(X,Y) \wedge edge(X,Y) \rightarrow in(X,Y) \tag{23}$$
$$in(a,Y) \rightarrow ra(Y) \tag{24}$$
$$in(X,Y) \wedge ra(X) \rightarrow ra(Y) \tag{25}$$
$$\neg ra(X) \wedge vertex(X) \rightarrow \bot \tag{26}$$
$$in(X,Y) \wedge in(X,Z) \wedge Y \neq Z \rightarrow \bot \tag{27}$$
$$in(X,Y) \wedge in(Z,Y) \wedge X \neq Z \rightarrow \bot \tag{28}$$
$$maxCost(J) \wedge sum(set_{tp}) > J \rightarrow \bot, \tag{29}$$

where tp is the name of the set symbol

$$K, X, Y : in(X,Y),\ cost(K,X,Y).$$

Note that this set symbol has no global variables in rule (20). By HC we denote the conjunction of sentences (21-28). By **hc** we denote the tuple containing all predicate symbols in HC except $edge/2$ and $vertex/1$. By the Splitting Theorem, $\mathrm{SM}_{\mathbf{hc}}[HC \wedge (29)]$ is equivalent to $\mathrm{SM}_{\mathbf{hc}}[HC] \wedge (29)$ so that $\mathrm{SM}_{\mathbf{hc}}[HC]$ and (29) form modules. The former module, corresponding to a program in Listing 1.2, formalizes statement **T1**; the latter module formalizes statement **T2**. Thus, we have completed **Step III**.

We turn our attention to **Step IV**. Propositions 5 and 8 in [2] prove that module $\mathrm{SM}_{\mathbf{hc}}[HC]$ correctly encodes the Hamiltonian Cycle problem. The proof of that claim used the one-sorted version of the SM operator. It is easy to see that for formulas that include only predicates of one sort, such as HC, there is an immediate correspondence between the one-sorted and the many-sorted models of the formula. Hence, the following is an immediate consequence of Propositions 5 and 8 mentioned above.

Proposition 2. *Let I be an agg-interpretation s.t. $G = \langle vertex^I, edge^I \rangle$ is a graph with $a \in vertex^I$. Then, $I \models \mathrm{SM}_{\mathbf{hc}}[HC]$ iff $(in/2)^I$ forms a Hamiltonian cycle of G.*

All that remains is to prove the following result about sentence (29).

Lemma 1. *Let I be an agg-interpretation that encodes an instance $\langle V, E, cst, m\rangle$ of the Traveling Salesman problem such that $in^I \subseteq edge^I$. Then, $I \models (29)$ iff inequality (19) holds, where $P = (in/2)^I$.*

Proof. Since I is an agg-interpretation that encodes $\langle V, E, cst, m\rangle$ and satisfies $in^I \subseteq edge^I$, it follows that

$$\begin{aligned} set^I_{tp} &= \{\langle c, a, b\rangle \mid \langle a, b\rangle \in in^I \text{ and } \langle c, a, b\rangle \in cost^I\} \\ &= \{\langle c, a, b\rangle \mid \langle a, b\rangle \in in^I \text{ and } \langle a, b\rangle \in edge^I \text{ and } cst(\langle a, b\rangle) = c\} \\ &= \{\langle c, a, b\rangle \mid \langle a, b\rangle \in in^I \text{ and } cst(\langle a, b\rangle) = c\} \end{aligned}$$

Therefore, $sum(set_{tp})^I = \sum_{e \in P} cst(e)$. Finally, since I encodes $\langle V, E, cst, m\rangle$, it follows that $maxCost^I = \{m\}$ and, thus, $I \models (29)$ iff (19) holds.

The following auxiliary lemma follows from the Splitting and Completion Theorems and is due to the presence of sentence (23) in HC. It allows us to complete the argument for the TS problem.

Lemma 2. $\mathrm{SM}_{\mathbf{hc}}[HC] \models \forall XY(in(X, Y) \rightarrow edge(X, Y))$.

Theorem 4. *Let I be an agg-interpretation encoding an instance $\langle V, E, cst, m\rangle$ of the Traveling Salesman problem. Then, $I \models \mathrm{SM}_{\mathbf{hc}}[HC \wedge (29)]$ iff I encodes a solution to the considered instance of the problem.*

Proof. By the Splitting Theorem, $I \models \mathrm{SM}_{\mathbf{hc}}[HC \wedge (29)]$ iff $I \models \mathrm{SM}_{\mathbf{hc}}[HC] \wedge (29)$. Then, from Proposition 2, it follows that the latter holds iff $(in/2)^I$ forms a Hamiltonian cycle of $G = \langle vertex^I, edge^I\rangle$ and $I \models (29)$. Finally, by Lemma 2, we get that $I \models \mathrm{SM}_{\mathbf{hc}}[HC]$ implies $I \models \forall XY(in(X, Y) \rightarrow edge(X, Y))$ and, thus, that $in^I \subseteq edge^I$. Therefore, the result follows from Lemma 1.

Theorem 4 can be seen as a proof of correctness for the TS encoding consisting of rules in Listing 1.2 and rule (20).

4 Conclusions and future work

We have shown how the semantics for programs with aggregates based on a many-sorted extension of the SM operator [4] can be used for arguing correctness of logic programs of this kind. For this we followed a modular methodology [2] and showed how it allows us to reuse the proof of correctness of other programs when they form sub-modules in the encoding of a new problem. One of the limitations of our approach is that it is only applicable to programs where aggregates do not have positive recursion. This limitation is inherited from the semantics for programs with aggregates in which it is based. Although aggregates with positive recursion are rare in practical applications, future work should be directed towards removing this limitation. It will be also interesting to consider programs with weak constraints.

The work by Yuliya Lierler was partially supported by NSF grant 1707371.

References

1. Buddenhagen, M., Lierler, Y.: Performance tuning in answer set programming. In: LPNMR. Lecture Notes in Computer Science, vol. 9345, pp. 186–198. Springer (2015)
2. Cabalar, P., Fandinno, J., Lierler, Y.: Modular answer set programming as a formal specification language. Theory and Practice of Logic Programming **20**(5), 767–782 (2020)
3. Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., Schaub, T.: ASP-Core-2: Input language format (2012), https://www.mat.unical.it/aspcomp2013/ASPStandardization
4. Fandinno, J., Hansen, Z., Lierler, Y.: Axiomatization of aggregates in answer set programming. In: Proceedings of the Thirty-six National Conference on Artificial Intelligence (AAAI'22). AAAI Press (2022)
5. Fandinno, J., Lifschitz, V.: Verification of locally tight programs (2022), http://www.cs.utexas.edu/users/ai-labpub-view.php?PubID=127938
6. Fandinno, J., Lifschitz, V., Lühne, P., Schaub, T.: Verifying tight logic programs with anthem and vampire. Theory and Practice of Logic Programming **20**(5), 735–750 (2020)
7. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artificial Intelligence **175**(1), 236–263 (2011)
8. Ferraris, P., Lee, J., Lifschitz, V., Palla, R.: Symmetric splitting in the general theory of stable models. In: Boutilier, C. (ed.) Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09). pp. 797–803. AAAI/MIT Press (2009)
9. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract Gringo. Theory and Practice of Logic Programming **15**(4-5), 449–463 (2015). https://doi.org/10.1017/S1471068415000150
10. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Challenges in answer set solving. In: Balduccini, M., Son, T. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of his 65th Birthday, Lecture Notes in Computer Science, vol. 6565, pp. 74–90. Springer-Verlag (2011)
11. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88). pp. 1070–1080. MIT Press (1988). https://doi.org/10.1201/b10397-6
12. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K., Marek, V., Truszczyński, M., Warren, D. (eds.) The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer-Verlag (1999)
13. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4), 241–273 (1999)
14. Oetsch, J., Seidl, M., Tompits, H., Woltran, S.: Beyond uniform equivalence between answer-set programs. ACM Trans. Comput. Log. **22**(1), 2:1–2:46 (2021)