Evaluating Answer Set Programming with Non-Convex Recursive Aggregates

Mario Alviano

Department of Mathematics and Computer Science,
University of Calabria, 87036 Rende (CS), Italy
alviano@mat.unical.it

Abstract. Aggregation functions are widely used in answer set programming (ASP) for representing and reasoning on knowledge involving sets of objects collectively. These sets may also depend recursively on the results of the aggregation functions, even if so far the support for such recursive aggregations was quite limited in ASP systems. In fact, recursion over aggregates was restricted to convex aggregates, i.e., aggregates that may have only one transition from false to true, and one from true to false, in this specific order. Recently, such a restriction has been overcome, so that the user can finally use non-convex recursive aggregates in ASP programs, either on purpose or accidentally. A preliminary evaluation of ASP programs with non-convex recursive aggregates is reported in this paper.

Keywords: answer set programming, aggregation functions, non-convex recursive aggregates

1 Introduction

Answer set programming (ASP) is a declarative language for knowledge representation and reasoning [9]. ASP programs are sets of disjunctive logic rules, possibly using default negation under stable model semantics [21, 22]. Several constructs were added to the original, basic language in order to ease the representation of practical knowledge. Of particular interest are aggregate functions [5, 14, 17, 23, 27, 32], which allow for expressing properties on sets of atoms declaratively. In fact, in many ASP programs *functional dependencies* are enforced by means of COUNT aggregates, or equivalently using SUM aggregates; for example, a rule of the following form:

$$\bot \leftarrow R'(\overline{X}),\ \text{SUM}[1, \overline{Y} : R(\overline{X}, \overline{Y}, \overline{Z})] \leq 1$$

constrains relation R to satisfy the functional dependency $\overline{X} \rightarrow \overline{Y}$, where $\overline{X} \cup \overline{Y} \cup \overline{Z}$ is the set of attributes of R, and R' is the projection of R on $\overline{X}$. Aggregate functions are also commonly used in ASP to constrain a nondeterministic guess. For example, in the *knapsack problem* the total weight of the selected items must not exceed a given limit, which can be modeled by the following rule:

$$\bot \leftarrow \text{SUM}[W, O : \mathit{object}(O, W, C), \mathit{in}(O)] \leq \mathit{limit}.$$

Mainstream ASP solvers [15, 20] almost agree on the semantics of aggregates [14, 17], here referred to as F-stable model semantics, even if several valid alternatives were

also considered in the literature [23, 30, 31, 33]. It is interesting to observe that F-stable model semantics was proposed more than a decade ago, providing a reasonable semantics for aggregates also in the recursive case. Indeed, it is based on an extension of the original program reduct, and on a minimality check of the stable model candidate resembling the disjunctive case. Despite this, for many years the implementation of F-stable model semantics was *incomplete*, and recursion over aggregates was restricted to *convex* aggregates [28], the largest class of aggregates for which the common reasoning tasks still belong to the first level of the polynomial hierarchy in the normal case [3]. In fact, convex aggregates may have only one transition from false to true, and one from true to false, in this specific order, a property that guarantees tractability of model checking in the normal case.

However, non-convex aggregations may arise in several contexts while modeling complex knowledge [1, 11, 13], and there are also minimalistic examples that are easily encoded in ASP using recursive non-convex aggregates, while alternative encodings not using aggregates are not so obvious. One of such examples is provided by the Σ_2^P-complete problem called *Generalized Subset Sum* [6]. In this problem, two vectors u and v of integers as well as an integer b are given, and the task is to decide whether the formula $\exists x \forall y (ux + vy \neq b)$ is true, where x and y are vectors of binary variables of the same length as u and v, respectively. For example, for $u = [1, 2]$, $v = [2, 3]$, and $b = 5$, the task is to decide whether the following formula is true: $\exists x_1 x_2 \forall y_1 y_2 (1 \cdot x_1 + 2 \cdot x_2 + 2 \cdot y_1 + 3 \cdot y_2 \neq 5)$. Any natural encoding of such an instance would include an aggregate of the form $\text{SUM}[1 : x_1, 2 : x_2, 2 : y_1, 3 : y_2] \neq 5$. Luckily, a complete implementation of F-stable model semantics for common aggregation functions has been achieved this year by means of a translation combining disjunction and saturation in order to eliminate non-convexity from aggregates [4].

The aim of this paper is to evaluate a few problems that can be encoded in ASP using recursive non-convex aggregates. The tested programs are processed by the rewritings presented in [4], which are implemented in a prototype system written in Python that uses GRINGO and CLASP. In a nutshell, aggregates are represented by specific standard atoms, so that the grounding phase can be delegated to GRINGO [19], and the numeric output of GRINGO is then processed to properly encode aggregates for the subsequent stable model search performed by CLASP [20]. The focus of the paper is on programs using SUM aggregates, even if the tested system also supports several other common aggregation functions such as COUNT, AVG, MIN, MAX, EVEN, and ODD.

2 Background

Let $\mathcal{V}$ be a set of propositional atoms including $\bot$. A propositional literal is an atom possibly preceded by one or more occurrences of the *negation as failure* symbol $\sim$. An aggregate literal, or simply aggregate, is of the following form:

$$\text{SUM}[w_1 : l_1, \ldots, w_n : l_n] \odot b \tag{1}$$

where $n \geq 0$, $b, w_1, \ldots, w_n$ are integers, $l_1, \ldots, l_n$ are propositional literals, and $\odot \in \{<, \leq, \geq, >, =, \neq\}$. (Note that $[w_1 : l_1, \ldots, w_n : l_n]$ is a multiset.) A literal is either a

propositional literal, or an aggregate. A rule r is of the following form:

$$p_1 \vee \cdots \vee p_m \leftarrow l_1 \wedge \cdots \wedge l_n \tag{2}$$

where $m \geq 1$, $n \geq 0$, $p_1, \ldots, p_m$ are propositional atoms, and $l_1, \ldots, l_n$ are literals. The set $\{p_1, \ldots, p_m\} \setminus \{\bot\}$ is referred to as head, denoted by $H(r)$, and the set $\{l_1, \ldots, l_n\}$ is called body, denoted by $B(r)$. A program Π is a finite set of rules. The set of propositional atoms (different from $\bot$) occurring in a program Π is denoted by $At(\Pi)$, and the set of aggregates occurring in Π is denoted by $Ag(\Pi)$.

Example 1. Consider the following program Π_1:

$$x_1 \leftarrow {\sim}{\sim}x_1 \qquad x_2 \leftarrow {\sim}{\sim}x_2 \qquad y_1 \leftarrow \mathit{unequal} \qquad y_2 \leftarrow \mathit{unequal} \qquad \bot \leftarrow {\sim}\mathit{unequal}$$
$$\mathit{unequal} \leftarrow \text{SUM}[1 : x_1, 2 : x_2, 2 : y_1, 3 : y_2] \neq 5$$

As will be clarified after defining the notion of a stable model, Π_1 encodes the instance of Generalized Subset Sum introduced in Section 1. ■

An *interpretation* I is a set of propositional atoms such that $\bot \notin I$. Relation $\models$ is inductively defined as follows:

- for $p \in \mathcal{V}$, $I \models p$ if $p \in I$;
- $I \models {\sim}l$ if $I \not\models l$;
- $I \models \text{SUM}[w_1 : l_1, \ldots, w_n : l_n] \odot b$ if $\sum_{i \in [1..n], I \models l_i} w_i \odot b$;
- for a rule r, $I \models B(r)$ if $I \models l$ for all $l \in B(r)$, and $I \models r$ if $H(r) \cap I \neq \emptyset$ when $I \models B(r)$;
- for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in \Pi$.

For any expression π, if $I \models \pi$, we say that I is a *model* of π, I satisfies π, or π is true in I. In the following, $\top$ will be a shorthand for ${\sim}\bot$, i.e., $\top$ is a literal true in all interpretations.

The *reduct* of a program Π with respect to an interpretation I is obtained by removing rules with false bodies and by fixing the interpretation of all negative literals. More formally, the following function $F(I, \cdot)$ is inductively defined:

- for $p \in \mathcal{V}$, $F(I, p) := p$;
- $F(I, {\sim}l) := \top$ if $I \not\models l$, and $F(I, {\sim}l) := \bot$ otherwise;
- $F(I, \text{SUM}[w_1 : l_1, \ldots, w_n : l_n] \odot b) := \text{SUM}[w_1 : F(I, l_1), \ldots, w_n : F(I, l_n)] \odot b$;
- for a rule r of the form (2), $F(I, r) := p_1 \vee \cdots \vee p_m \leftarrow F(I, l_1) \wedge \cdots \wedge F(I, l_n)$;
- for a program Π, $F(I, \Pi) := \{F(I, r) \mid r \in \Pi, I \models B(r)\}$.

Program $F(I, \Pi)$ is the reduct of Π with respect to I. An interpretation I is a *stable model* of a program Π if $I \models \Pi$ and there is no $J \subset I$ such that $J \models F(I, \Pi)$. Let $SM(\Pi)$ denote the set of stable models of Π.

Example 2. Continuing with Example 1, the models of Π_1, restricted to the atoms in $At(\Pi_1)$, are X, $X \cup \{x_1\}$, $X \cup \{x_2\}$, and $X \cup \{x_1, x_2\}$, where $X = \{\mathit{unequal}, y_1, y_2\}$. Of these, only $X \cup \{x_1\}$ is a stable model. Indeed, the reduct $F(X \cup \{x_1\}, \Pi_1)$ is

$$x_1 \leftarrow \top \qquad y_1 \leftarrow \mathit{unequal} \qquad y_2 \leftarrow \mathit{unequal}$$
$$\mathit{unequal} \leftarrow \text{SUM}[1 : x_1, 2 : x_2, 2 : y_1, 3 : y_2] \neq 5$$

and no strict subset of $X \cup \{x_1\}$ is a model of the above program. On the other hand, the reduct $F(X \cup \{x_2\}, \Pi_1)$ is

$$x_2 \leftarrow \top \qquad y_1 \leftarrow \mathit{unequal} \qquad y_2 \leftarrow \mathit{unequal}$$
$$\mathit{unequal} \leftarrow \text{SUM}[1 : x_1, 2 : x_2, 2 : y_1, 3 : y_2] \neq 5$$

and $\{x_2, y_2\}$ is a model of the above program. Similarly, it can be checked that X and $X \cup \{x_1, x_2\}$ are not stable models of Π_1. ■

An aggregate A is *convex* (in program reducts) if $J \models F(I, A)$ and $L \models F(I, A)$ implies $K \models F(I, A)$, for all $J \subseteq K \subseteq L \subseteq I \subseteq \mathcal{V}$. If A is convex then $I \models A$ and $J \models F(I, A)$ implies $K \models F(I, A)$, for all $J \subseteq K \subseteq I$. Note that aggregate $\text{SUM}[1 : x_1, 2 : x_2, 2 : y_1, 3 : y_2] \neq 5$ from Example 1 is non-convex.

3 Non-Convex Aggregates Elimination

ASP solvers can only process sums of the form (1) in which all numbers are non-negative integers, and the comparison operator $\odot$ is $\geq$. This is due to the numeric format encoding the propositional program produced by the grounder. However, thanks to the rewritings proposed by [4], all sums can be rewritten in the form accepted by current ASP solvers. Following [4], strong equivalences can be used to restrict sums in the input program to only two forms, which are essentially (1) with $\odot \in \{\geq, \neq\}$. These first rewritings are given by means of strong equivalences [16, 25, 34].

Definition 1. *Let $\pi := l_1 \wedge \cdots \wedge l_n$ be a conjunction of literals, for some $n \geq 1$. A pair (J, I) of interpretations such that $J \subseteq I$ is an* SE-model *of π if $I \models \pi$ and $J \models F(I, l_1) \wedge \cdots \wedge F(I, l_n)$. Two conjunctions π, π' are* strongly equivalent*, denoted by $\pi \equiv_{SE} \pi'$, if they have the same SE-models.*

Strong equivalence means that replacing π by π' preserves the stable models of any logic program.

Proposition 1 (Lifschitz et al. 2001; Turner 2003; Ferraris 2005). *Let π, π' be two conjunctions of literals such that $\pi \equiv_{SE} \pi'$. Let Π be a program, and Π' be the program obtained from Π by replacing any occurrence of π by π'. It holds that $\Pi \equiv_{\mathcal{V}} \Pi'$.*

The following strong equivalences can be proven by showing equivalence with respect to models, and by noting that $\sim$ is neither introduced nor eliminated:

(E1) $\text{SUM}[w_1 : l_1, \ldots, w_n : l_n] > b \equiv_{SE} \text{SUM}[w_1 : l_1, \ldots, w_n : l_n] \geq b + 1$;
(E2) $\text{SUM}[w_1 : l_1, \ldots, w_n : l_n] \leq b \equiv_{SE} \text{SUM}[-w_1 : l_1, \ldots, -w_n : l_n] \geq -b$;
(E3) $\text{SUM}[w_1 : l_1, \ldots, w_n : l_n] < b \equiv_{SE} \text{SUM}[w_1 : l_1, \ldots, w_n : l_n] \leq b - 1$;
(E4) $\text{SUM}[w_1 : l_1, \ldots, w_n : l_n] = b \equiv_{SE} \text{SUM}[w_1 : l_1, \ldots, w_n : l_n] \leq b \wedge$
$\text{SUM}[w_1 : l_1, \ldots, w_n : l_n] \geq b$.

For example, (E1) and (E3) are easy to obtain because b is integer by assumption. Similarly, (E4) is immediate by the semantics introduced in Section 2. For (E2), instead, the following equivalences can be observed:

(i) $I \models \text{SUM}[w_1 : l_1, \ldots, w_n : l_n] \leq b$;
(ii) $\sum_{i \in [1..n],\ I \models l_i} w_i \leq b$;
(iii) $\sum_{i \in [1..n],\ I \models l_i} -w_i \geq -b$;
(iv) $I \models \text{SUM}[-w_1 : l_1, \ldots, -w_n : l_n] \geq -b$;

where (iii) above is obtained by multiplying both sides of the inequality (ii) by -1, and the equivalence of (i) and (ii), and of (iii) and (iv), is immediate by the semantics of sums. It is important to observe that the application of (E1)–(E4), from the last to the first, to a program Π gives an equivalent program $pre(\Pi)$ whose aggregates are sums with comparison operators $\geq$ and $\neq$.

Theorem 1. *Let Π be a program. It holds that $\Pi \equiv_{\mathcal{V}} pre(\Pi)$.*

After this preprocessing, the structure of the input program is further simplified by eliminating non-convex aggregates. To ease the presentation, and without loss of generality, hereinafter aggregates are assumed to be of the following form:

$$\begin{aligned} \text{SUM}[& -w_1 : p_1, \ldots, -w_j : p_j, \\ & -w_{j+1} : {\sim}l_{j+1}, \ldots, -w_k : {\sim}l_k, \\ & w_{k+1} : p_{k+1}, \ldots, w_m : p_m, \\ & w_{m+1} : {\sim}l_{m+1}, \ldots, w_n : {\sim}l_n] \odot b \end{aligned} \tag{3}$$

where $n \geq m \geq k \geq j \geq 0$, $w_1, \ldots, w_n$ are positive integers, each p_i is a propositional atom, each l_i is a propositional literal, $\odot \in \{\geq, \neq\}$, and b is an integer. Intuitively, aggregated elements of (3) are partitioned in four sets, namely positive literals with negative weights, negative literals with negative weights, positive literals with positive weights, and negative literals with positive weights.

Let Π be a program whose aggregates are of the form (3). Program $rew(\Pi)$ is obtained from Π by replacing each occurrence of an aggregate of the form (3) by a fresh, hidden propositional atom aux [10, 24]. Moreover, if $\odot$ is $\geq$, then the following rule is added:

$$\begin{aligned} aux \leftarrow \text{SUM}[& w_1 : p_1^F, \ldots, w_j : p_j^F, \\ & w_{j+1} : {\sim}{\sim}l_{j+1}, \ldots, w_k : {\sim}{\sim}l_k, \\ & w_{k+1} : p_{k+1}, \ldots, w_m : p_m, \\ & w_{m+1} : {\sim}l_{m+1}, \ldots, w_n : {\sim}l_n] \geq b + w_1 + \cdots + w_k \end{aligned} \tag{4}$$

where each p_i^F is a fresh, hidden atom associated with the falsity of p_i, for all $i \in [1..j]$, and the following rules are also added to $rew(\Pi)$:

$$p_i^F \leftarrow {\sim}p_i \tag{5}$$
$$p_i^F \leftarrow aux \tag{6}$$
$$p_i \vee p_i^F \leftarrow {\sim}{\sim}aux \tag{7}$$

Similarly, if $\odot$ is $\neq$, then the following rules are added to $rew(\Pi)$:

$$\begin{aligned} aux \leftarrow \text{SUM}[&w_1 : p_1^F, \ldots, w_j : p_j^F, \\ &w_{j+1} : {\sim}{\sim}l_{j+1}, \ldots, w_k : {\sim}{\sim}l_k, \\ &w_{k+1} : p_{k+1}, \ldots, w_m : p_m, \\ &w_{m+1} : {\sim}l_{m+1}, \ldots, w_n : {\sim}l_n] \geq b + 1 + w_1 + \cdots + w_k \end{aligned} \tag{8}$$

$$\begin{aligned} aux \leftarrow \text{SUM}[&w_1 : p_1, \ldots, w_j : p_j, \\ &w_{j+1} : {\sim}l_{j+1}, \ldots, w_k : {\sim}l_k, \\ &w_{k+1} : p_{k+1}^F, \ldots, w_m : p_m^F, \\ &w_{m+1} : {\sim}{\sim}l_{m+1}, \ldots, w_n : {\sim}{\sim}l_n] \geq -b + 1 + w_{k+1} + \cdots + w_n \end{aligned} \tag{9}$$

together with rules (5)–(7) for each new p_i^F. Intuitively, any atom of the form p_i^F introduced by the rewriting must be true whenever p_i is false, but also when aux is true, so to implement what is usually referred to as *saturation* in the literature. Rules (5) and (6) encode such an intuition. Moreover, rule (7) guarantees that at least one between p_i and p_i^F belongs to any model of reducts obtained from interpretations containing aux. It is interesting to observe that when aux belongs to I the satisfaction of the associated aggregate can be tested according to all subsets of I in the reduct $F(\Pi, I)$.

The intuition behind (4) is that an interpretation I satisfies an aggregate of the form (3) such that $\odot$ is $\geq$ if and only if the following inequality is satisfied:

$$\sum_{i=1}^{j} -w_i \cdot I(p_i) + \sum_{i=j+1}^{k} -w_i \cdot I({\sim}l_i) + \sum_{i=k+1}^{m} w_i \cdot I(p_i) + \sum_{i=m+1}^{n} w_i \cdot I({\sim}l_i) \geq b \tag{10}$$

where $I(l) = 1$ if $I \models l$, and $I(l) = 0$ otherwise, for all literals l. Moreover, inequality (10) is satisfied if and only if the following inequality is satisfied:

$$\begin{aligned} \sum_{i=1}^{j} -w_i \cdot I(p_i) + \sum_{i=j+1}^{k} -w_i \cdot I({\sim}l_i) + \sum_{i=k+1}^{m} w_i \cdot I(p_i) + \\ + \sum_{i=m+1}^{n} w_i \cdot I({\sim}l_i) + w_1 + \cdots + w_k \geq b + \sum_{i=1}^{k} w_i \end{aligned} \tag{11}$$

and by distributivity (11) is equivalent to the following inequality:

$$\begin{aligned} \sum_{i=1}^{j} w_i \cdot (1 - I(p_i)) + \sum_{i=j+1}^{k} w_i \cdot (1 - I({\sim}l_i)) + \\ + \sum_{i=k+1}^{m} w_i \cdot I(p_i) + \sum_{i=m+1}^{n} w_i \cdot I({\sim}l_i) \geq b + \sum_{i=1}^{k} w_i. \end{aligned} \tag{12}$$

Note that $1 - I(l) = I({\sim}l)$ for all literals l, and p_i^F is associated with the falsity of p_i, for all $i \in [1..j]$. It is important to observe that negation was not used for positive literals

in order to avoid oversimplifications in program reducts. Indeed, as already explained, for all $i \in [1..j]$, atom p_i^F will be derived true whenever p_i is false, but also when the aggregate is true.

The intuition behind (8)–(9) is similar. Essentially, an aggregate $\text{SUM}(S) \neq b$ of the form (3) is true if and only if either $\text{SUM}(S) \geq b+1$ or $\text{SUM}(S) \leq b-1$ is true, and (E2) is applied to the second aggregate in order to use the previously explained rewriting. Let rew^* denote the composition $rew \circ pre$.

Example 3. Consider again program Π_1 from Example 1. Its rewriting $rew^*(\Pi_1)$ is as follows:

$$
\begin{array}{llll}
x_1 \leftarrow {\sim}{\sim}x_1 & x_2 \leftarrow {\sim}{\sim}x_2 & y_1 \leftarrow unequal & y_2 \leftarrow unequal \quad \bot \leftarrow {\sim}unequal \\
unequal \leftarrow aux & aux \leftarrow \text{SUM}[1 : x_1^F; 2 : x_2^F; 2 : y_1^F; 3 : y_2^F] \geq 4 & & \\
 & aux \leftarrow \text{SUM}[1 : x_1; 2 : x_2; 2 : y_1; 3 : y_2] \geq 6 & & \\
x_1^F \leftarrow {\sim}x_1 & x_1^F \leftarrow aux & x_1 \vee x_1^F \leftarrow {\sim}{\sim}aux & \\
x_2^F \leftarrow {\sim}x_2 & x_2^F \leftarrow aux & x_2 \vee x_2^F \leftarrow {\sim}{\sim}aux & \\
y_1^F \leftarrow {\sim}y_1 & y_1^F \leftarrow aux & y_1 \vee y_1^F \leftarrow {\sim}{\sim}aux & \\
y_2^F \leftarrow {\sim}y_2 & y_2^F \leftarrow aux & y_2 \vee y_2^F \leftarrow {\sim}{\sim}aux &
\end{array}
$$

The only stable model of $rew^*(\Pi_1)$ is $\{x_1, unequal, y_1, y_2, aux, x_1^F, x_2^F, y_1^F, y_2^F\}$. ■

Correctness of the rewriting can be established by slightly adapting the proof by [4].

Theorem 2 (Correctness). *Let Π be a program. It holds that $\Pi \equiv_{At(\Pi)} rew^*(\Pi)$.*

4 Implementation

The rewritings introduced in Section 3 have been implemented in a prototype system written in Python and available at the following URL: `http://alviano.net/software/f-stable-models/`. The prototype accepts an input language whose syntax is almost conformant to ASP Core 2.0 [2]. It is a first-order language, meaning that propositional atoms are replaced by first-order atoms made of a predicate and a list or terms, where each term is an object constant, an object variable, or a composed term obtained by combining a function symbol with other terms. As usual in ASP, all variables are universally quantified, so that the propositional semantics given in Section 2 can be used after a grounding phase that replaces variables by constants in all possible ways.

The only exception to the ASP Core 2.0 format is that sums have to be encoded using the standard predicates f_sum and f_set. Moreover, only positive literals can occur in aggregation sets. In more detail, a sum of the form $\text{SUM}[w_1 : p_1, \ldots, w_n : p_n] \odot b$, where $n \geq 0$, $b, w_1, \ldots, w_n$ are integers, $p_1, \ldots, p_n$ are (first-order) atoms, and $\odot \in \{<, \leq, \geq, >, =, \neq\}$ is encoded by the following first-order atom:

$$f\_sum(id, \mu(\odot), b)$$

where $\mu(\odot)$ equals `"<"`, `"<="`, `">="`, `">"`, `"="`, or `"!="`, and id is an identified for the aggregation set, encoded by the following rules:

$$f\_set(id, w_1, p_1) \leftarrow p_1 \qquad \cdots \qquad f\_set(id, w_n, p_n) \leftarrow p_n$$

where a body p_i ($i \in [1..n]$) can be omitted if p_i has no variables. (It is also possible to extend a body of the above rules in order to further constrain the aggregation set; for example, arithmetic expressions can be used to restrict the selection of atoms in the aggregation sets.)

Example 4. Program Π_1 from Example 1 is encoded as follows:

$$
\begin{array}{ll}
\mathtt{x}_1 :- \mathtt{not\ not\ x}_1. & \mathtt{unequal} :- \mathtt{f\_sum}(\mathtt{uneq}, \mathtt{"!="}, 5). \\
\mathtt{x}_2 :- \mathtt{not\ not\ x}_2. & \mathtt{f\_set}(\mathtt{uneq}, 1, \mathtt{x}_1). \\
\mathtt{y}_1 :- \mathtt{unequal}. & \mathtt{f\_set}(\mathtt{uneq}, 2, \mathtt{x}_2). \\
\mathtt{y}_2 :- \mathtt{unequal}. & \mathtt{f\_set}(\mathtt{uneq}, 2, \mathtt{y}_1). \\
:- \mathtt{not\ unequal}. & \mathtt{f\_set}(\mathtt{uneq}, 3, \mathtt{y}_2).
\end{array}
$$

where not encodes the negation as failure symbol $\sim$, and rules with empty head are integrity constraints, i.e., rules whose head is equivalent to $\bot$.

Alternatively, instances of Generalized Subset Sum can be specified by means of facts involving predicates $exists$, all, and $bound$. For example, the instance above is encoded by the following facts:

$$
\begin{array}{lll}
\mathtt{exists}(\mathtt{x}_1, 1). & \mathtt{all}(\mathtt{y}_1, 2). & \mathtt{bound}(5). \\
\mathtt{exists}(\mathtt{x}_2, 2). & \mathtt{all}(\mathtt{y}_2, 3). &
\end{array}
$$

A program encoding the Generalized Subset Sum problem for instances encoded by these predicates is the following:

$$
\begin{array}{l}
\mathtt{true}(\mathtt{X}, \mathtt{C}) :- \mathtt{exists}(\mathtt{X}, \mathtt{C}),\ \mathtt{not\ not\ true}(\mathtt{X}, \mathtt{C}). \\
\mathtt{true}(\mathtt{X}, \mathtt{C}) :- \mathtt{all}(\mathtt{X}, \mathtt{C}),\ \mathtt{unequal}. \\
:- \mathtt{not\ unequal}. \\
\mathtt{unequal} :- \mathtt{f\_sum}(\mathtt{uneq}, \mathtt{"!="}, \mathtt{B}),\ \mathtt{bound}(\mathtt{B}). \\
\mathtt{f\_set}(\mathtt{uneq}, \mathtt{C}, \mathtt{true}(\mathtt{X}, \mathtt{C})) :- \mathtt{true}(\mathtt{X}, \mathtt{C}).
\end{array}
$$

where X, C, and B are object variables. ■

Given a program encoded as described above, the prototype obtains its propositional version by means of the grounder GRINGO. During the grounding phase, instances of predicate f_sum are considered *external*, i.e., they are assigned the truth value *undefined* in order to prevent their elimination. These instances and those of predicate f_set are identified and mapped in data structures of the prototype, so to have an internal representation of all sums occurring in the propositional program. The rewritings presented in Section 3 are then applied to these sums in order to eliminate any non-convexity. The new sums, and any additional rule introduced by the rewriting process, are added to the propositional program. Finally, the propositional program is printed to the standard output using the numeric format of GRINGO, so that CLASP can be used for computing its F-stable models, which eventually coincide with the F-stable models of the original program because additional atoms are hidden.

5 Experiment

The implemented rewritings were tested on a few domains that can be encoded using recursive sums. One of them is the Generalized Subset Sum problem presented in the introduction, which is of particular relevance in this experiment because its natural encoding in ASP requires a recursive non-convex sum. In fact, an ASP encoding for this problem that does not rely on recursive sums is not available, and therefore in this case the performance of the prototype was compared with an SMT encoding fed into Z3. The other two problems considered in this experiment are k-Clique-Coloring and 2-QBF, Σ_2^p-complete problems whose natural encodings in ASP do not rely on recursive sums. In these two cases, an alternative encoding using recursive sums can be obtained, even if usually paying an overhead on the running time. The aim of the experiment for these two problems is to evaluate such an overhead. All tested instances are available at the following URL: `http://archives.alviano.net/publications/2015/RCRA2015-experiment.zip`.

The experiment was run on an Intel Xeon CPU 2.4 GHz with 16 GB of RAM. CPU and memory usage were limited to 900 seconds and 15 GB, respectively. GRINGO, CLASP, and Z3 were tested with their default settings. Their performances were measured by PYRUNLIM (`http://alviano.net/software/pyrunlim/`). The results are reported in Table 1, where each row reports the number of instances and, for each tested ASP encoding, the number of solved instances, the average execution time and the average memory consumption. Data for Z3 are not reported in the table because it was run only on Generalized Subset Sum, discussed below.

Generalized Subset Sum [6]. Two vectors u and v of integers as well as an integer b are given, and the task is to decide whether the formula $\exists x \forall y (ux + vy \neq b)$ is true, where x and y are vectors of binary variables of the same length as u and v, respectively. For an instance such that $u = u_1, \ldots, u_m$ ($m \geq 1$) and $v = v_1, \ldots, v_n$ ($n \geq 1$) the following ASP encoding was tested (actually, its non-propositional version):

$$
\begin{array}{ll}
x_i \leftarrow \sim\!\sim x_i & \forall i \in [1..m] \\
y_i \leftarrow \mathit{unequal} & \forall i \in [1..n] \\
\bot \leftarrow \sim \mathit{unequal} & \\
\mathit{unequal} \leftarrow \text{SUM}[u_1 : x_1, \ldots, u_m : x_m, v_1 : y_1, \ldots, v_n : y_n] \neq b &
\end{array}
$$

Table 1. Performance of GRINGO+CLASP (number of solved instances; average execution time in seconds; average memory consumption in MB).

			Aggregates			Alternative		
	Benchmark	**inst**	**sol**	**time**	**mem**	**sol**	**time**	**mem**
	Generalized Subset Sum	46	38	1.1	44	n/a	n/a	n/a
	k-Clique Coloring	60	60	177.2	863	60	20.9	205
2-QBF	Preprocessing Track	17	8	64.8	171	9	98.3	171
2-QBF	QBFLib Track	32	1	0.1	101	1	0.1	102
2-QBF	Application Track	48	13	126.1	45	19	22.7	45

As for Z3, the following SMT encoding was tested:

$$\forall y_1 \cdots \forall y_n (\ ite(x_1, u_1, 0) + \cdots + ite(x_m, u_m, 0) + \\ ite(y_1, v_1, 0) + \cdots + ite(y_n, v_n, 0) \neq b)$$

where $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ are Boolean constants and variables, respectively, and $ite(\phi, t_1, t_2)$ is an *if-then-else* expression, i.e., its interpretation is t_1 if ϕ is true, and t_2 otherwise. As reported in the table, the ASP encoding leads to an excellent performance in many cases, with 38 solved instances and an average execution time of around 1.1 seconds. The performance achieved within the SMT encoding is instead less attractive, with only 14 solved instances and an average execution time of around 34.7 seconds. The tested ASP solver is also more efficient in memory, using 44 MB on average, while 148 MB are used by Z3 to solve the SMT instances. The reason of such different performances is that SMT is a more expressive language, allowing arbitrary alternations of quantifies, while in ASP at most one alternation can be simulated by means of saturation techniques. It turns out that ASP solvers can implement more optimized algorithms for problems on the second level of the polynomial hierarchy.

k-Clique-Coloring [29]. Given a graph $G = (V, E)$ with n nodes, and an integer $k \geq 2$, is possible to assign k colors to vertices in V such that each maximal clique K of G contains two vertices of different colors? The tested encoding using non-convex sums is reported below (again, its non-propositional version was actually tested).

$x_c \leftarrow {\sim}{\sim}x_c$	$\forall x \in V,\ \forall c \in [1..k]$
$\bot \leftarrow \text{SUM}[1 : x_1, \ldots, 1 : x_k] \neq 1.$	$\forall x \in V$
$\bot \leftarrow {\sim}saturate$	
$in_x \vee out_x \leftarrow$	$\forall x \in V$
$in_x \leftarrow saturate$	$\forall x \in V$
$out_x \leftarrow saturate$	$\forall x \in V$
$saturate \leftarrow in_x, in_y$	$\forall x, y \in V,\ x \neq y,\ (x, y) \notin E$
$saturate \leftarrow in_x, in_y, x_c, y_d$	$\forall x, y \in V,\ \forall c, d \in [1..k],\ c \neq d$
$saturate \leftarrow out_x, \text{SUM}[n : saturate,$ $-1 : in_{y_1}, \ldots, -1 : in_{y_{n-1}},$ $1 : in_{z_1}, \ldots, 1 : in_{z_j}] \geq 0$	$\forall x \in V,$ where $\{y_1, \ldots, y_{n-1}\} = V \setminus \{x\},$ $\{z_1, \ldots, z_j\} = \{z \mid (x, z) \in E\}$

Intuitively, a color is assigned to each vertex, and the saturation is activated whenever one of the following conditions is verified:

- the guessed K contains two non-adjacent nodes, i.e., K is not a clique;
- the guessed K contains two nodes with different colors;
- there is a vertex $x \in V \setminus K$ such that x is adjacent to all vertices in K, i.e., K is not a maximal clique.

The alternative encoding not using recursive sums is obtained by replacing the last rule above with the following rule:

$$saturate \leftarrow out_x, out_{y_1}, \ldots, out_{y_j} \qquad \forall x \in V$$

where $\{y_1, \ldots, y_j\} = \{y \in V \setminus \{x\} \mid (x, y) \notin E\}$. Intuitively, in this case the third condition leading to saturate is the following:

- there is a vertex $x \in V \setminus K$ such that all vertices in V that are not adjacent to x do not belong to K, i.e., K is not a maximal clique.

For this problem, both encodings lead to solve all tested instances, which are the graphs submitted to the 4th ASP Competition [2] for the Graph Coloring problem. However, the overhead due to the use of recursive non-convex aggregates slows the computation down by a factor of 8, and also the memory consumption is around 4 times higher.

2-QBF. Given a 2-DNF $\exists\overline{x}\forall\overline{y}\phi$, is the formula valid? The tested encoding not using sums is the following:

$$
\begin{array}{ll}
x \leftarrow {\sim}{\sim}x & \forall x \in \overline{x} \\
\bot \leftarrow {\sim}saturate & \\
y^T \vee y^F \leftarrow & \forall y \in \overline{y} \\
y^T \leftarrow saturate & \forall y \in \overline{y} \\
y^F \leftarrow saturate & \forall y \in \overline{y} \\
saturate \leftarrow \mu(l_1), \ldots, \mu(l_n) & \forall l_1 \wedge \cdots \wedge l_n \in \phi,\ n \geq 1
\end{array}
$$

where $\mu(x) = x$ and $\mu(\neg x) = {\sim}x$ for all $x \in \overline{x}$, and $\mu(y) = y^T$ and $\mu(\neg y) = y^F$ for all $y \in \overline{y}$. An equivalent encoding using non-convex sums can be obtained by replacing all rules with y^T or y^F in the head with the following rules:

$$
\begin{array}{ll}
y^T \leftarrow \text{SUM}[1 : saturate, -1 : y^F] \geq 0 & \forall y \in \overline{y} \\
y^F \leftarrow \text{SUM}[1 : saturate, -1 : y^T] \geq 0 & \forall y \in \overline{y}
\end{array}
$$

The tested instances are all the 2-QBF instances in the QBF Gallery 2014 (`http://qbf.satisfiability.org/gallery/results.html`). Also in this case there is an overhead due to the unnatural use of non-convex sums. It impacts significantly on the Application Track, where the difference in terms of solved instances is 6.

6 Related Work

F-stable model semantics [14, 17] is implemented by widely-used ASP solvers [15, 20]. The original definition in [14, 17] is slightly different than the one provided in Section 2. In fact, propositional formulas can be arbitrarily nested in [17], while a more constrained structure is assumed in this paper in order to achieve an efficient implementation. On the other hand, double negation is not permitted in [14], even if it can be simulated by means of auxiliary atoms: a rule $p \leftarrow {\sim}{\sim}p$ can be equivalently encoded by using a fresh atom p^F and the following subprogram: $\{p \leftarrow {\sim}p^F,\ p^F \leftarrow {\sim}p\}$. Similarly, negated literals cannot occur in the aggregates considered by [14] but again can must be encoded by means of auxiliary atoms. Another difference with [14] is on negated aggregates, which are not permitted by the language considered in this paper because [17] and [14] actually assign different semantics to programs with negated aggregates. As a final remark, the reduct of [14] does not remove negated literals from

satisfied bodies, which however are necessarily true in all counter-models because double negation is not allowed.

Techniques to rewrite logic programs with aggregates into equivalent programs with simpler aggregates were investigated in the literature right from the beginning [32]. In particular, rewritings into LPARSE-like programs, which differ from those presented in this paper, were considered in [26]. As a general comment, since disjunction is not considered in [26], all aggregates causing a jump from the first to the second level of the polynomial hierarchy are excluded a priori. This is the case for aggregates of the form $\text{SUM}(S) \neq b$, $\text{AVG}(S) \neq b$, and $\text{COUNT}(S) \neq b$, as first noted by [33], but also for comparators other than $\neq$ when negative weights are involved. In fact, in [26] negative weights are eliminated by a rewriting similar to the one in (4), but negated literals are introduced instead of auxiliary atoms, which may lead to unintuitive results [18]. A different rewriting was presented by [17], whose output are programs with nested expressions, a construct that is not supported by current ASP systems. Other relevant rewriting techniques were proposed in [8, 7], and proved to be quite efficient in practice. However, these rewritings produce aggregate-free programs preserving F-stable models only in the stratified case, or if recursion is limited to convex aggregates. On the other hand, it is interesting to observe that the rewritings of [8, 7] are applicable to the output of the rewritings presented in this paper in order to completely eliminate aggregates, thus preserving F-stable models in general.

Several other stable model semantics were proposed for interpreting logic programs with aggregates. Many of these semantics rely on stability checks that are not based on minimality [30, 31, 33], and therefore the rewritings presented by [4] and recalled in Section 3 cannot be used for these semantics. A more recent proposal is based on a stability check that essentially eliminates aggregates from program reducts [23], and therefore the rewritings by [4] cannot help also in this case. Finally, there are other ASP constructs that are semantically close to aggregates, such as DL [13] and HEX [12] atoms, for interacting with external knowledge bases possibly expressed in different languages; as these constructs cannot be compactly reduced to sums in general, the rewritings by [4] do not apply to these languages as well.

7 Conclusion

ASP takes advantage of several constructs to ease the representation of complex knowledge. Aggregation functions are among the most commonly used constructs in ASP specifications. The rewritings proposed by [4] provide a concrete simplification of the structure of aggregations in input programs, so to improve the efficiency of low-level reasoners. Such rewritings are implemented in a prototype system, presented in this paper, which reported a reasonable performance on benchmarks for which more tailored encodings using disjunction in rule heads exist. More relevant, when such an aggregate-free encoding is unknown or untuitive, for example in the Generalized Subset Sum problem, the rewritings implemented in the prototype are particularly useful. Indeed, in this specific benchmark ASP solving significantly outperforms an alternative encoding in the more expressive language of SMT.

It must be remarked that this is only a preliminary evaluation of recursive non-convex aggregates in ASP. For the future, we plan to collect more encodings for problems that can be easily represented by using recursive non-convex aggregates, so to obtain a more suitable test suite for evaluating the efficiency of ASP solvers in presence of aggregations of this kind. Moreover, we will investigate alternative mappings of common aggregation functions into sums, with the aim of simplifying some of the rewritings by [4]. In particular, concerning ODD and EVEN, the rewritings by [4] are quadratic in size, and hence an interesting question to answer is whether there exist alternative rewritings of these aggregations whose sizes remain linear.

References

1. Abseher, M., Bliem, B., Charwat, G., Dusberger, F., Woltran, S.: Computing secure sets in graphs using answer set programming. In: Inclezan, D., Maratea, M. (eds.) Seventh International Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2014) (2014)
2. Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The fourth answer set programming competition: Preliminary report. In: Cabalar, P., Son, T.C. (eds.) LPNMR. pp. 42–53. LNCS (2013)
3. Alviano, M., Faber, W.: The complexity boundary of answer set programming with generalized atoms under the FLP semantics. In: Cabalar, P., Son, T.C. (eds.) Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8148, pp. 67–72. Springer (2013), http://dx.doi.org/10.1007/978-3-642-40564-8_7
4. Alviano, M., Faber, W., Gebser, M.: Rewriting recursive aggregates in answer set programming: back to monotonicity. Theory and Practice of Logic Programming (2015), to appear
5. Bartholomew, M., Lee, J., Meng, Y.: First-order semantics of aggregates in answer set programming via modified circumscription. In: Logical Formalizations of Commonsense Reasoning, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-06, Stanford, California, USA, March 21-23, 2011. AAAI (2011), http://www.aaai.org/ocs/index.php/SSS/SSS11/paper/view/2472
6. Berman, P., Karpinski, M., Larmore, L.L., Plandowski, W., Rytter, W.: On the complexity of pattern matching for highly compressed two-dimensional texts. J. Comput. Syst. Sci. 65(2), 332–350 (2002), http://dx.doi.org/10.1006/jcss.2002.1852
7. Bomanson, J., Gebser, M., Janhunen, T.: Improving the normalization of weight rules in answer set programs. In: Fermé, E., Leite, J. (eds.) JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8761, pp. 166–180. Springer (2014), http://dx.doi.org/10.1007/978-3-319-11558-0_12
8. Bomanson, J., Janhunen, T.: Normalizing cardinality rules using merging and sorting constructions. Lecture Notes in Computer Science, vol. 8148, pp. 187–199. Springer (2013), http://dx.doi.org/10.1007/978-3-642-40564-8_19
9. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011), http://doi.acm.org/10.1145/2043174.2043195

10. Eiter, T., Tompits, H., Woltran, S.: On solution correspondences in answer set programming. In: Kaelbling, L., Saffiotti, A. (eds.) Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05). pp. 97–102. Professional Book Center (2005)
11. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Conflict-driven ASP solving with external sources. Theory and Practice of Logic Programming 12(4-5), 659–679 (2012), http://dx.doi.org/10.1017/S1471068412000233
12. Eiter, T., Fink, M., Krennwallner, T., Redl, C., Schüller, P.: Efficient hex-program evaluation based on unfounded sets. J. Artif. Intell. Res. (JAIR) 49, 269–321 (2014), http://dx.doi.org/10.1613/jair.4175
13. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artif. Intell. 172(12-13), 1495–1539 (2008), http://dx.doi.org/10.1016/j.artint.2008.04.002
14. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011), http://dx.doi.org/10.1016/j.artint.2010.04.002
15. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the DLV system. Theory and Practice of Logic Programming 8(5-6), 545–580 (2008), http://dx.doi.org/10.1017/S1471068408003323
16. Ferraris, P.: Answer sets for propositional theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05). Lecture Notes in Artificial Intelligence, vol. 3662, pp. 119–131. Springer-Verlag (2005)
17. Ferraris, P.: Logic programs with propositional connectives and aggregates. ACM Trans. Comput. Log. 12(4), 25 (2011), http://doi.acm.org/10.1145/1970398.1970401
18. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming 5(1-2), 45–74 (2005), http://dx.doi.org/10.1017/S1471068403001923
19. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6645, pp. 345–351. Springer (2011), http://dx.doi.org/10.1007/978-3-642-20895-9_39
20. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187, 52–89 (2012), http://dx.doi.org/10.1016/j.artint.2012.04.001
21. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes). pp. 1070–1080. MIT Press (1988)
22. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. 9(3/4), 365–386 (1991), http://dx.doi.org/10.1007/BF03037169
23. Gelfond, M., Zhang, Y.: Vicious circle principle and logic programs with aggregates. Theory and Practice of Logic Programming 14(4-5), 587–601 (2014), http://dx.doi.org/10.1017/S1471068414000222
24. Janhunen, T., Niemelä, I.: Applying visible strong equivalence in answer-set program transformations. In: Erdem, E., Lee, J., Lierler, Y., Pearce, D. (eds.) Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz. Lecture Notes in Computer Science, vol. 7265, pp. 363–379. Springer-Verlag (2012)
25. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2(4), 526–541 (2001)

26. Liu, G., You, J.: Relating weight constraint and aggregate programs: Semantics and representation. Theory and Practice of Logic Programming 13(1), 1–31 (2013), http://dx.doi.org/10.1017/S147106841100038X
27. Liu, L., Pontelli, E., Son, T.C., Truszczynski, M.: Logic programs with abstract constraint atoms: The role of computations. Artif. Intell. 174(3-4), 295–315 (2010), http://dx.doi.org/10.1016/j.artint.2009.11.016
28. Liu, L., Truszczynski, M.: Properties and applications of programs with monotone and convex constraints. J. Artif. Intell. Res. (JAIR) 27, 299–334 (2006), http://dx.doi.org/10.1613/jair.2009
29. Marx, D.: Complexity of clique coloring and related problems. Theor. Comput. Sci. 412(29), 3487–3500 (2011)
30. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and stable semantics of logic programs with aggregates. Theory and Practice of Logic Programming 7(3), 301–353 (2007), http://dx.doi.org/10.1017/S1471068406002973
31. Shen, Y., Wang, K., Eiter, T., Fink, M., Redl, C., Krennwallner, T., Deng, J.: FLP answer set semantics without circular justifications for general logic programs. Artif. Intell. 213, 1–41 (2014), http://dx.doi.org/10.1016/j.artint.2014.05.001
32. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artif. Intell. 138(1-2), 181–234 (2002), http://dx.doi.org/10.1016/S0004-3702(02)00187-X
33. Son, T.C., Pontelli, E.: A constructive semantic characterization of aggregates in answer set programming. Theory and Practice of Logic Programming 7(3), 355–375 (2007), http://dx.doi.org/10.1017/S1471068406002936
34. Turner, H.: Strong equivalence made easy: nested expressions and weight constraints. Theory and Practice of Logic Programming 3(4-5), 609–622 (2003)