Improving the Normalization of Weight Rules in Answer Set Programs*

Jori Bomanson, Martin Gebser**, and Tomi Janhunen

Helsinki Institute for Information Technology HIIT
Department of Information and Computer Science
Aalto University, FI-00076 AALTO, FINLAND

Abstract. Cardinality and weight rules are important primitives in answer set programming. In this context, normalization means the translation of such rules back into normal rules, e.g., for the sake of boosting the search for answers sets. For instance, the normalization of cardinality rules can be based on Boolean circuits that effectively sort or select greatest elements amongst Boolean values. In this paper, we develop further constructions for the normalization of weight rules and adapt techniques that have been previously used to translate pseudo-Boolean constraints into the propositional satisfiability (SAT) problem. In particular, we consider mixed-radix numbers as an efficient way to represent and encode integer weights involved in a weight rule and propose a heuristic for selecting a suitable base. Moreover, we incorporate a scheme for structure sharing in the normalization procedure. In the experimental part, we study the effect of normalizing weight rules on compactness and search performance measured in terms of program size, search time, and number of conflicts.

1 Introduction

Cardinality and *weight* rules [38] are important primitives in answer set programming (ASP) [11]. They enable more compact problem encodings compared to *normal* rules, which formed the first syntax when the *stable model semantics* of rules was originally proposed [22]. Stable models are also called *answer sets*, and the basic intuition of ASP is to capture the solutions of the problem being solved as answer sets of a respective logic program. There are two mainstream approaches to computing answer sets for a logic program given as input. The first is represented by *native* answer set solvers [3, 15, 21, 29, 38], which have direct implementations of extended rule types in their data structures. The alternative, *translation-based* approach aims at transforming rules into other kinds of constraints and using off-the-shelf solver technology such as satisfiability (SAT) [9] solvers and their extensions for the actual search for answer sets (see, e.g., [23, 26, 28, 32, 34]). Regardless of the approach to compute answer sets, the *normalization* [10, 26] of cardinality and weight rules becomes an interesting issue. In this context, this means translating extended rules back into normal rules, e.g., in order to

* The support from the Finnish Centre of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (under grant #251170) is gratefully acknowledged.

** Also affiliated with the University of Potsdam, Germany.

boost the search for answers sets. Normalization is also unavoidable if cardinality and weight constraints are not directly expressible in the language fragment supported by a back-end solver.

Intuitively, a cardinality rule with a *head* atom a, literals $l_1, \ldots, l_n$ in the *body*, and a bound $1 \leq k \leq n$ allows the derivation of a if at least k literals out of $l_1, \ldots, l_n$ can be satisfied by other rules. Existing approaches to normalize cardinality rules exploit translations based on *binary decision diagrams* [16] as well as *Boolean circuits* that effectively *sort* n Boolean values or *select* k greatest elements amongst them [5]. The normalization schemes developed in [26] and [10] introduce of the order of $k \times (n-k)$ or $n \times (\log_2 k)^2$ rules, respectively. The latter scheme is typically more compact and, as suggested by the experimental results in [10], also possibly faster when computing answer sets. Weight rules are similar to cardinality rules but each literal l_i in the body is assigned a positive (integer) weight w_i and, typically, we have that $k \ll \sum_{i=1}^{n} w_i$. The sum of the weights associated with satisfied literals matters when it comes to checking the bound $k \geq 0$ and deriving the head atom a. Literals with different weights bring about extra complexity and obvious asymmetry to the normalization of weight rules. Nevertheless, since cardinality rules form a special case of weight rules ($w_i = 1$ for each l_i), it is to be expected that normalization schemes developed for cardinality rules provide relevant primitives for the normalization of weight rules. Indeed, by introducing suitable auxiliary atoms, a number of rules polynomial in n, $\log_2 k$, and $\log_2(\sum_{i=1}^{n} w_i)$ will be sufficient.

The goal of this paper is to develop further constructions needed in the normalization of weight rules. A natural idea is to adapt techniques that have been previously used to translate pseudo-Boolean constraints into SAT. In particular, the sum of weights associated with satisfied literals is calculated stepwise as in the approach of [16]. In the purely binary case, this means summing up the bits constituting weights, so either 0 or 1, for satisfied literals, while propagating carry bits in increasing order of significance. This is also feasible with *merger* and *sorter* programs developed in [10], as they provide carry bits in a natural way. Since sorter programs consist of merger programs, we use the latter as basic primitives in this paper.

It is also possible to go beyond the base 2 and introduce *mixed-radix bases* to encode integer weights so that the number of digits to be summed gets smaller. In this paper, we propose a heuristic for selecting a suitable base rather than doing a complete search over all alternatives [12]. Moreover, to simplify the check for the bound k, we adopt the idea from [7] and initialize the weight sum calculation with a preselected *tare*. As a consequence, to perform the check it suffices to produce the most significant digit of the sum. Finally, we incorporate a mechanism for structure sharing in the normalization, which composes merger programs in a bottom-up fashion and shares structure whenever possible, while trying to maximize such possibilities.

The paper is organized as follows. Section 2 provides an account of the syntax and semantics of weight constraint programs, as well as a summary of principles for simplifying weight rules before normalization. The basic primitives for the normalization of weight rules, i.e., the merger programs discussed above, are introduced in Section 3, together with sorter programs built on top. The normalizations themselves are then developed in Section 4, where several schemes arise since mixed-radix bases and structure

sharing are used. An experimental evaluation is carried out in Section 5, studying the effects of the new normalization schemes using the state-of-the-art ASP solver CLASP as back end. Related work and conclusions are discussed in Sections 6 and 7.

2 Preliminaries

In what follows, we briefly introduce the syntactic fragments of ASP addressed in this paper, namely *normal logic programs* (NLPs) and *weight constraint programs* (WCPs). Afterwards, we introduce *mixed-radix notation* for encoding finite domain numbers.

Normal logic programs are defined as finite sets of *normal rules* of the form

$$a \leftarrow l_1, \ldots, l_n. \tag{1}$$

where a is a *propositional atom* (or an *atom* for short) and each l_i is a *literal*. Literals are either *positive* or *negative*, i.e., simply atoms 'b' or their *default negations* '$\mathrm{not}\, c$', respectively. Intuitively, the *head* atom a can be derived by the rule (1) whenever positive literals in the body are derivable by other rules in a program but none of the negative literals' atoms is derivable. A *weight rule* allows for a more versatile rule body:

$$a \leftarrow k \leq [l_1 = w_1, \ldots, l_n = w_n]. \tag{2}$$

Each body *literal* l_i in (2) is assigned a weight w_i. The weight w_i is charged if $l_i = b$ is positive and b can be derived or $l_i = \mathrm{not}\, c$ is negative and c cannot be derived. The head a is derived if the sum of satisfied literals' weights is at least k. Also note that *cardinality rules* addressed in [10] are obtained as a special case of (2) when $w_i = 1$ for $1 \leq i \leq n$, and it is customary to omit weights then. Weight constraint programs P are defined as finite sets of normal and/or weight rules. A program P is called *positive* if no negative literals appear in the bodies of its rules.

To introduce the answer set semantics of WCPs, we write $\mathrm{At}(P)$ for the *signature* of a WCP P, i.e., a set of atoms to which all atoms occurring in P belong to. A positive literal $a \in \mathrm{At}(P)$ is *satisfied* in an *interpretation* $I \subseteq \mathrm{At}(P)$ of P, denoted $I \models a$, iff $a \in I$. A negative literal '$\mathrm{not}\, a$' is satisfied in I, denoted $I \models \mathrm{not}\, a$, iff $a \notin I$. The body of (1) is satisfied in I iff $I \models l_1, \ldots, I \models l_n$. Similarly, the body of (2), which contains the weighted literals $l_1 = w_1, \ldots, l_n = w_n$, is satisfied in I iff the *weight sum*

$$\textstyle\sum_{1 \leq i \leq n,\, I \models l_i} w_i \geq k. \tag{3}$$

A rule (1), or alternatively (2), is satisfied in I iff the satisfaction of the body in I implies $a \in I$. An interpretation $I \subseteq \mathrm{At}(P)$ is a (*classical*) *model* of a program P, denoted $I \models P$, iff $I \models r$ for every rule $r \in P$. A model $M \models P$ is $\subseteq$*-minimal* iff there is no $M' \models P$ such that $M' \subset M$. Any positive program P is guaranteed to have a unique minimal model, the *least model* denoted by $\mathrm{LM}(P)$.

For a WCP P and an interpretation $M \subseteq \mathrm{At}(P)$, the *reduct* of P with respect to M, denoted by P^M, contains (i) a positive rule $a \leftarrow b_1, \ldots, b_n$ for each normal rule $a \leftarrow b_1, \ldots, b_n, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_m$ of P such that $M \not\models c_1, \ldots, M \not\models c_m$ [22] and (ii) a weight rule $a \leftarrow k' \leq [b_1 = w_1, \ldots, b_n = w_n]$ for each weight rule

$a \leftarrow k \leq [b_1 = w_1, \ldots, b_n = w_n, \mathrm{not}\, c_1 = w_{n+1}, \ldots, \mathrm{not}\, c_m = w_{n+m}]$ of P, where $k' = \max\{0, k - \sum_{1 \leq i \leq m,\, c_i \notin M} w_{n+i}\}$ is the new lower bound [38]. Given that P^M is positive by definition, an interpretation $M \subseteq \mathrm{At}(P)$ of a WCP P is defined as a *stable model* of P iff $M = \mathrm{LM}(P^M)$ [22, 38]. The set of stable models, also called *answer sets*, of a WCP P is denoted by $\mathrm{SM}(P)$.

Example 1. Consider a WCP P consisting of the following three rules:

$$a \leftarrow 5 \leq [b = 4, \mathrm{not}\, c = 2]. \qquad b \leftarrow 1 \leq [\mathrm{not}\, d = 1]. \qquad c \leftarrow 2 \leq [a = 1, c = 2].$$

Given $M_1 = \{a, b\}$, the reduct P^{M_1} consists of $a \leftarrow 3 \leq [b = 4]$, $b \leftarrow 0 \leq []$, and $c \leftarrow 2 \leq [a = 1, c = 2]$. As $\mathrm{LM}(P^{M_1}) = \{a, b\} = M_1$, M_1 is a stable model of P. But $M_2 = \{a, b, c\}$ is not stable because $\mathrm{LM}(P^{M_2}) = \{b\} \neq M_2$. ■

Since the body of a weight rule (2) can be satisfied by particular subsets of literals, it is to be expected that the normalization of the rule can become a complex operation in the worst case. Thus it makes sense to simplify weight rules before the actual normalization is performed. In the following, we provide a summary of useful principles in this respect. Some of them yield normal rules as by-product of simplification.

1. *Simplify weights*: if the weights $w_1, \ldots, w_n$ in (2) have a *greatest common divisor* (GCD) $d > 1$, replace them by $w_1/d, \ldots, w_n/d$ and the bound k by $\lceil k/d \rceil$.
2. *Normalize directly*: if the sum $s = \sum_{i=1}^{n} w_i \geq k$ but $s - w_i < k$ for each $1 \leq i \leq n$, all body literals are necessary to reach the bound, and (2) can be rewritten as a normal rule (1) by dropping the weights and the bound altogether.
3. *Remove inapplicable rules*: if $\sum_{i=1}^{n} w_i < k$, remove the rule (2) altogether.
4. *Remove dominating literals*: if the body of (2) contains a literal l_i with $w_i \geq k$, add a normal rule $a \leftarrow l_i$ and remove $l_i = w_i$ from the body.

Example 2. Let us reconsider the weight rules from Example 1. The weights of the first rule have the GCD $d = 2$, and the division yields $a \leftarrow 3 \leq [b = 2, \mathrm{not}\, c = 1]$, which can be directly normalized as $a \leftarrow b, \mathrm{not}\, c$. Similarly, the second rule can be directly normalized as $b \leftarrow \mathrm{not}\, d$. The third rule has a dominating literal $c = 2$, which yields a normal rule $c \leftarrow c$. Such a tautological rule can be removed immediately. Since the remainder $c \leftarrow 2 \leq [a = 1]$ is inapplicable, only two normal rules $a \leftarrow b, \mathrm{not}\, c$ and $b \leftarrow \mathrm{not}\, d$ are left, and it is easy to see that $\{a, b\}$ is their unique stable model. ■

A *mixed-radix base* B is a sequence $b_1, \ldots, b_m$ of positive integers. Special cases of such include the binary and decimal bases, $\langle 2, 2, \ldots \rangle$ and $\langle 10, 10, \ldots \rangle$. In this paper, we deal with *finite-length* mixed-radix bases only and refer to them simply as *bases*. The *radices* $b_1, \ldots, b_m$ are indexed from the least significant, b_1, to the most significant, b_m, and we denote the integer at a given *radix position* i by $B(i) = b_i$. The length m is accessed with $|B|$. We define the i^{th} *place value* of B as $\Pi(i) = \prod_{j=1}^{i-1} B(j)$. By w_B^i, we refer to the i^{th} digit of an integer w in B. A *mixed-radix literal* H in base B is a sequence of sequences of literals $H = H_1, \ldots, H_{|B|}$, where each $H_i = h_{i,1}, \ldots, h_{i,n_i}$ captures the i^{th} digit of the encoded value. Any such *literal digit* H_i represents a *unary* digit $v_M(H_i) \in \{0, \ldots, n_i\}$ given by the satisfied literals $h_{i,1}, \ldots, h_{i,j}$ for $0 \leq j \leq n_i$ in a model M. In turn, we write $v_M(H)$ for the value of H calculated

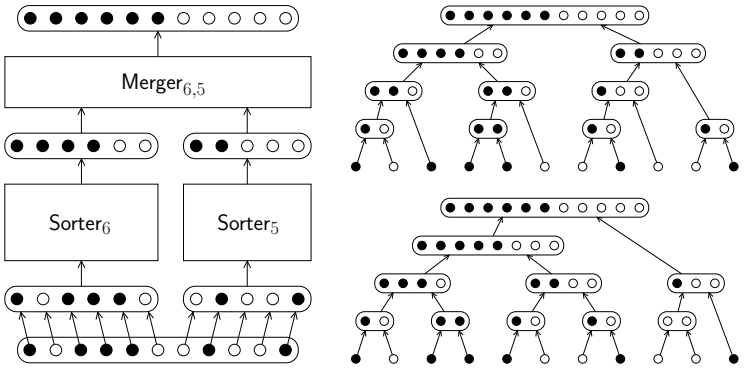


Fig. 1. A recursively constructed merge-sorter on the left, a corresponding unrolled merge-sorter on the top right, and an alternative merge-sorter on the bottom right. On the left and top right, inputs are split approximately in halves. On the bottom right, mergers are mainly laid-out for some power of two many inputs. Other splitting approaches are possible and investigated in Section 4.

as $\sum_{i=1}^{|B|} (v_M(H_i) \times \Pi(i))$. Finally, we distinguish *unique* mixed-radix literals with $n_i = B(i) - 1$ for $1 \leq i \leq |B|$, which represent each value uniquely, whereas *non-unique* mixed-radix literals can generally express a value by several combinations of digits. In the sequel, we make explicit when uniqueness is expected.

3 Merger and Sorter Programs

Sorting operations for, e.g., the normalization of cardinality rules are compactly implementable with the well known *merge-sorting* scheme, illustrated in Figure 1, where an input sequence of literals is recursively split in halves, sorted, and the intermediate results merged. Due to the ordered inputs, merging can be implemented in n^2 rules [7] without introducing any auxiliary atoms. A more efficient alternative that has been successfully applied in ASP [10] is Batcher's odd-even merger [8], which requires of the order of $n \times \log_2 n$ atoms and rules. Furthermore, the variety of options for primitives has been leveraged in practice by parametrizing the decision of when to apply which scheme [1]. For simplicity, we below abbreviate sequences of literals by capital letters. For instance, letting L be $b, \text{not } c$, we write $a \leftarrow 2 \leq [L, \text{not } d]$ as a shorthand for $a \leftarrow 2 \leq [b, \text{not } c, \text{not } d]$. The basic building blocks used in this work are mergers, and in the following we specify the behavior required by them. To this end, we rely on *visible strong equivalence* [10, 27], denoted by $P \equiv_{\text{vs}} P'$ for two programs P and P'.

Definition 1. *Given three sequences $H_1 = h_1, \ldots, h_n$, $H_2 = h_{n+1}, \ldots, h_{n+m}$, and $S = s_1, \ldots, s_{n+m}$ of atoms, we call any NLP P a* merger program*, also referred to by* $\mathsf{Merger}(H_1, H_2, S)$*, if $P \cup Q \equiv_{\text{vs}} \{s_k \leftarrow k \leq [H_1, H_2]. \mid 1 \leq k \leq n+m\} \cup Q$ for $Q = \{h_i \leftarrow h_{i+1}. \mid 1 \leq i < n+m, i \neq n\}$.*

The role of Q in the above definition is to deny interpretations in which H_1 or H_2 is unordered and does not correspond to a unary digit, as presupposed for merging. In order to drop this restriction, a merge-sorter can be conceived as a binary tree with mergers

as inner nodes and literals as leaves, as shown on the right in Figure 1. Starting from trivial sequences at the lowest level, successive merging then yields a sorted output.

Definition 2. *Given a sequence $L = l_1, \ldots, l_n$ of literals and a sequence $S = s_1, \ldots, s_n$ of atoms, we call any NLP P a* sorter program*, also referred to by* $\mathsf{Sorter}(L, S)$*, if* $P \equiv_{\mathrm{vs}} \{s_k \leftarrow k \leq [L]. \mid 1 \leq k \leq n\}$.

Compared to a merger program, a sorter program does not build on preconditions and derives a unary digit representation for an arbitrary sequence of input literals.

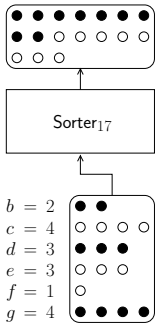
4 Normalizing Weight Rules

In this section, we extend the translation of [7] to normalize WCPs into NLPs. To this end, we decompose normalization into parallel sorting tasks and a sequence of merging tasks. For the former subtasks, we generalize sorting to *weight sorting*.

Example 3. Let us consider a WCP P composed of the single rule

$$a \leftarrow 6 \leq [b = 2, c = 4, d = 3, e = 3, f = 1, g = 4].$$

The NLP realization of $\mathsf{Sorter}_{17}(\langle b, b, c, c, c, c, d, d, d, e, e, e, f, g, g, g, g\rangle, \langle s_1, \ldots, s_{17}\rangle)$ displayed in Figure 2, augmented with the rule $a \leftarrow s_6$, gives a plausible yet unnecessarily large normalization of P. This scheme, implemented via merge-sorting without simplifications by calling `lp2normal2 -ws -r` (cf. Section 5), results in 116 rules. Omitting the `-r` flag enables simplifications and reduces the number of rules to 53. For comparison, the translation described in the sequel leads to 16 rules only. While outcomes like this may seem to discourage unary weight sorting, it still permits compact constructions for rules with small weights and, in particular, cardinality rules.



■ **Fig. 2:** Weight sorting.

Returning to the general translation, we now describe the first constituent, addressing the calculation of a *digit-wise sum* of satisfied input weights in a chosen mixed-radix base B. Given a sequence $L = l_1, \ldots, l_n$ of literals and a sequence $W = w_1, \ldots, w_n$ of weights, we below write $L = W$ as a shorthand for $l_1 = w_1, \ldots, l_n = w_n$. Similarly, $L = W_B^i$ abbreviates $l_1 = (w_1)_B^i, \ldots, l_n = (w_n)_B^i$ for $1 \leq i \leq |B|$, associating the literals in L with the i^{th} digits of their weights in B. Moreover, we refer to a program for weight sorting, such as Sorter_{17} in Example 3, by $\mathsf{WSorter}$.

Definition 3. *Given a sequence $L = W$ of weighted literals and a mixed-radix base B, a* digit-wise sorter *into the non-unique mixed-radix literal H is the program*

$$\mathsf{WDigitwiseSorter}_B(L = W, H) = \bigcup_{i=1}^{|B|} \mathsf{WSorter}(L = W_B^i, H_i). \tag{4}$$

Equation (4) reveals the substeps of decomposing an input expression $L = W$ into digit-wise *bucket expressions* $L = W_B^i$, which are then subject to weight sorting. The result is a potentially non-unique mixed-radix literal H encoding the weight sum. An example of a digit-wise sorter is shown in Figure 3.

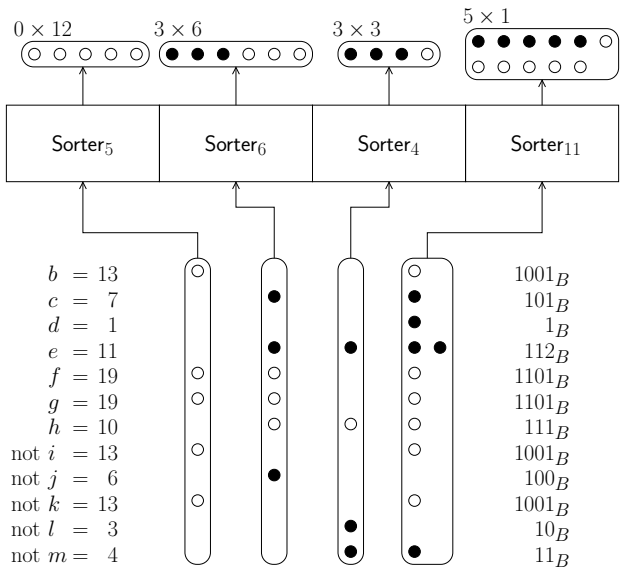


Fig. 3. Structure of a $\mathsf{WDigitwiseSorter}_B$ program for the weighted literals displayed on the left in base $B = 3, 2, 2, 10$. Filled markers designate derivations stemming from the input literals c, d, e, $\mathrm{not}\, j$, $\mathrm{not}\, l$, and $\mathrm{not}\, m$, satisfied in some interpretation M. From right to left, the sorters yield multiples of $\Pi(1) = 1$, $\Pi(2) = 3$, $\Pi(3) = 6$, and $\Pi(4) = 12$. The output mixed-radix literal H represents $v_M(H) = 0 \times 12 + 3 \times 6 + 3 \times 3 + 5 \times 1 = 7 + 1 + 11 + 6 + 3 + 4 = 32$.

The second part of the translation incorporates carries from less to more significant digits in order to derive the weight sum uniquely and accurately. In the following, we denote the sequence $s_d, s_{2d}, \ldots, s_{d\lfloor n/d \rfloor}$, involving every d^{th} element of a literal digit $S = s_1, \ldots, s_n$, by S/d.

Definition 4. *Given a mixed-radix literal $H = H_1, \ldots, H_{|B|}$ in base B, a* carry merger *into the sequence $S = S_1, \ldots, S_{|B|}$ of literal digits, where $S_1 = H_1$, is the program*

$$\mathsf{WCarryMerger}_B(H, S) = \bigcup_{i=2}^{|B|} \mathsf{Merger}(S_{i-1}/B(i-1), H_i, S_i). \tag{5}$$

The intended purpose of the program in (5) is to produce the last digit $S_{|B|}$ of S, while $S_1, \ldots, S_{|B|-1}$ are intermediate results. The role of each merger is to combine carries from S_{i-1} with the unary input digit H_i at position i. To account for the significance gap $B(i-1)$ between S_{i-1} and H_i, the former is divided by $B(i-1)$ in order to extract the carry. An example carry merger is shown in Figure 4.

The digit-wise sorter and carry merger fit together to form a normal program to substitute for a weight rule. To this end, we follow the approach of [7] and first determine a tare t by which both sides of the inequality in (3) are offset. The benefit is that only the most significant digit in B of a weight sum is required for checking the lower bound k. This goal is met by the selection $t = (\lceil k/\Pi(|B|) \rceil \times \Pi(|B|)) - k$. Equipped with this choice for the tare t, we define the following program for weight rule normalization.

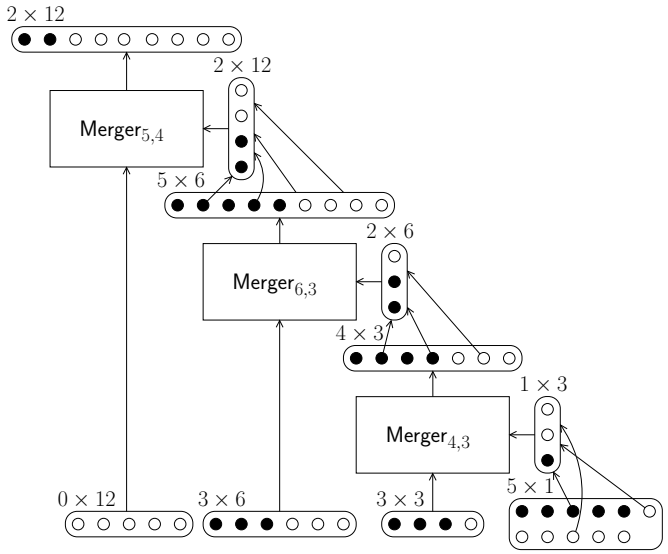


Fig. 4. Structure of a $\mathsf{WCarryMerger}_B$ program for deriving a unique, most significant literal digit $S_{|B|}$ from digit-wise sums H_i in Figure 3. Each merger for $1 < i \leq |B|$ combines H_i with carries, extracted from an intermediate literal digit S_{i-1} by means of the division $S_{i-1}/B(i-1)$. For instance, in view of $B(1) = 3$, every third bit of $S_1 = H_1$ is used as carry for deriving S_2.

Definition 5. *Given a weight rule $a \leftarrow k \leq [L = W]$ and a mixed-radix base B, a* weight sorting network *is the program*

$$\begin{aligned}\mathsf{WSortingNetwork}_{B,k}(L = W, a) \;=\; & \mathsf{WDigitwiseSorter}_B([\, L = W, \top = t\,], H) \\ \cup\; & \mathsf{WCarryMerger}_B(H, S) \\ \cup\; & \{a \leftarrow S_{|B|,\lceil k/\Pi(|B|)\rceil}.\,\}.\end{aligned} \tag{6}$$

In the above, the symbol $\top$ stands for an arbitrary fact, and H and S for auxiliary (hidden) mixed-radix literals capturing the outputs of the utilized subprograms. In view of the tare t, the last rule in (6) merely propagates the truth value of a single output bit from the most significant literal digit $S_{|B|}$ to the original head atom a. Definition 5 readily leads to a weight rule normalization once a base B is picked, and we can then substitute a weight rule (2) with $\mathsf{WSortingNetwork}_{B,k}([l_1 = w_1, \ldots, l_n = w_n], a)$.

The so far presented translation is an ASP variant of the *Global Polynomial Watchdog* [7] encoding, modified to use mixed-radix bases. In what follows, we describe two novel additions to the translation. We give a heuristic for base selection, which is different from the more or less exhaustive methods in [12, 16]. Also, we provide a structure sharing approach to compress the digit-wise sorter component of the translation.

We perform mixed-radix *base selection* for a weight rule (2) by choosing radices one by one from the least to the most significant position. The choices are guided by heuristic estimates of the resulting translation size. In the following, we assume, for simplicity, that $\max\{w_1, \ldots, w_n\} \leq k \leq \sum_{j=1}^{n} w_j$, as guaranteed for example by the simplifications described in Section 2. Furthermore, the order of the size of sorters and

Algorithm 1 Plan Structure Sharing

1: **function** PLAN($L = W, B$)
2: **let** $C \leftarrow \{[\, l_1^{(w_1)^i_B}, \ldots, l_n^{(w_n)^i_B} \,] \mid 1 \leq i \leq |B|\}$
3: **while** $\exists S \in C : \exists x, y \in S : x \neq y$
4: **let** $(x, y) \leftarrow \underset{x,y \in \bigcup C}{\arg\max} \sum_{S \in C} \begin{cases} \#_S(x) \times \#_S(y) & \text{if } x \neq y \\ (\#_S(x) \times (\#_S(x) - 1))/2 & \text{if } x = y \end{cases}$
5: **let** $z \leftarrow [\, x, y \,]$
6: **for each** $S \in C$
7: **let** $j \leftarrow \min\{\#_S(x), \#_S(y)\}$
8: **update** $S \leftarrow \begin{cases} (S \setminus [\, x^j, y^j \,]) \cup [\, z^j \,] & \text{if } x \neq y \\ (S \setminus [\, x^{2\lfloor j/2 \rfloor} \,]) \cup [\, z^{\lfloor j/2 \rfloor} \,] & \text{if } x = y \end{cases}$
9: **return** C

mergers used as primitives is denoted by $\mathrm{z}(n) = n \times (\log_2 n)^2$. To select the i^{th} radix b_i, we consider $B = b_1, \ldots, b_{i-1}, \infty$. Then, in terms of k_B^i, $W_B^i = (w_1)_B^i, \ldots, (w_n)_B^i$, and $s = \sum_{j=1}^n \left((w_j)_B^i \bmod b\right)$, we pick

$$b_i \leftarrow \underset{\substack{b \text{ is prime, } b \leq \max \\ \{2, (w_1)_B^i, \ldots, (w_n)_B^i\}}}{\arg\max} \left(\begin{array}{l} \mathrm{z}(s) + \mathrm{z}\left(n/2 + \min\left\{\lceil s/b \rceil, \lfloor k_B^i / b \rfloor\right\} + 1\right) \\ \qquad + \mathrm{z}\left(3/4 \times n\right) \times \log_2(1/(2 \times n \times b) \times \sum_{j=1}^n (w_j)_B^i) \end{array} \right).$$

The idea of the three addends is to generously estimate the size of primitives directly entailed by the choice of a prime b, the size of immediately following components, and the size of the entire remaining structure. Radices are picked until $\prod_{j=1}^i b_j > \max\{w_1, \ldots, w_n\}$, after which the selection finishes with the base $B = b_1, \ldots, b_{i-1}, \lceil (\sum_{j=1}^n w_j) / \prod_{j=1}^{i-1} b_i \rceil + 1$. In Section 5, we compare the effect of heuristically chosen mixed-radix bases with binary bases having $b_j = 2$ for $1 \leq j < i$.

The digit-wise sorter in (4) consists of sorters that, when implemented via merge-sorting, form a *forest of mergers* on a common set of leaves. The mergers, i.e., inner nodes of the forest, produce sorted sequences based on bucket expressions. This paves the way for structure sharing. Namely, many of these mergers may coincide in terms of their output, and consequently parts of the combined subprograms that would otherwise be replicated can be reused instead. Respective savings are for instance achievable by structural hashing [16].

Our approach advances sharing, taking into account that there is a large degree of freedom in how a single merge-sorter is constructed. In fact, we may choose to split a sequence of input bits at various positions, not only in the middle, as shown on the right in Figure 1. Choices regarding such partitions generally lead to different amounts of reusable, coinciding structure. To this end, we propose Algorithm 1 to greedily expand opportunities for structure sharing. Thereby, we denote a *multiset* S on a set $X = \{x_1, \ldots, x_n\}$ of ground elements with respective *multiplicities* $i_1, \ldots, i_n$ by $[\, x_1^{i_1}, \ldots, x_n^{i_n} \,]$. The multiplicity i_j of $x_j \in X$ is referred to by $\#_S(x_j)$, and x_j is said to have i_j occurrenes in S. The superscript i_j can be omitted from $x_j^{i_j}$ if $i_j = 1$. Furthermore, we write $x \in S$ iff $x \in X$ and $\#_S(x) > 0$. At the beginning of the

algorithm, the bucket expressions $L = W_B^i$ are gathered into a collection C of multisets, where the literals $L = l_1, \ldots, l_n$ form the common ground elements and the digits $W_B^i = (w_1)_B^i, \ldots, (w_n)_B^i$ give the multiplicities for $1 \leq i \leq |B|$. Then, iteratively, pairs (x, y) of elements with heuristically maximal joint occurrences in C are selected to form new multisets z replacing common occurrences of (x, y) in each $S \in C$. The introduced multisets z are in the sequel handled like regular ground elements, and the algorithm proceeds until every $S \in C$ consists of a single multiset. The resulting collection C will generally comprise nested multisets, which we interpret as a directed acyclic graph, intuitively consisting of a number of overlaid trees with the literals $l_1, \ldots, l_n$ as leaves, multisets z as roots, and inner nodes giving rise to mergers.

Example 4. Considering the weighted literals $a = 9, b = 3, c = 7, d = 2, e = 5, f = 4$ and the base $B = 2, 2, 9$, Algorithm 1 yields the following merge-sorter structure:

$$
\begin{aligned}
C &\leftarrow \{[\, a, b, c, e \,], [\, b, c, d \,], [\, a^2, c, e, f \,]\}, \\
C &\leftarrow \{[\, [\, a, e \,], b, c \,], [\, b, c, d \,], [\, a, [\, a, e \,], c, f \,]\}, \\
C &\leftarrow \{[\, [\, a, e \,], [\, b, c \,] \,], [\, [\, b, c \,], d \,], [\, a, [\, a, e \,], c, f \,]\}, \\
&\;\;\vdots \\
C &\leftarrow \{[\, [\, [\, a, e \,], [\, b, c \,] \,] \,], [\, [\, [\, b, c \,], d \,] \,], [\, [\, [\, a, [\, a, e \,] \,], [\, c, f \,] \,] \,]\}.
\end{aligned}
$$

■

5 Experiments

The weight rule normalization techniques described in the previous section are implemented in the translation tool LP2NORMAL2 (v. 1.10).[1] In order to evaluate the effects of normalization, we ran LP2NORMAL2 together with the back-end ASP solver CLASP (v. 3.0.4) [21] on benchmarks stemming from five different domains: Bayesian network structure learning [14, 25], chordal Markov network learning [13], the Fastfood logistics problem [10], and the Incremental scheduling and Nomystery planning tasks from the 4th ASP Competition [4]. The first two domains originally deal with optimization, and we devised satisfiable as well as unsatisfiable decision versions by picking the objective value of an optimum or its decrement (below indicated by the suffixes "Find" and "Prove") as upper bound on solution cost. The other three domains comprise genuine decision problems in which weight constraints restrict the cost of solutions. All experiments were run sequentially on a Linux machine with Intel Xeon E5-4650 CPUs, imposing a CPU time limit of 20 minutes and a memory limit of 3GB RAM per run.

Table 1 provides runtimes in seconds, numbers of constraints, and conflicts reported by CLASP, summing over all instances of a benchmark class and in total, for different weight rule implementations. In the native configuration, weight rules are not normalized but handled internally by CLASP [20]. Different translations by LP2NORMAL2 in the third to sixth column vary in the use of mixed-radix or binary bases as well as the exploitation of structure sharing. Furthermore, results for the Sequential Weight Counter (SWC) normalization scheme, used before in ASP [18] as well as SAT [24],

[1] Available with benchmarks at `http://research.ics.aalto.fi/software/asp`.

# Instances			Mixed		Binary		
↓	Benchmark	Native	Shared	Independent	Shared	Independent	SWC
11	Bayes-Find	202	30	164	246	165	1,721
	# Constraints	34,165	347,450	417,768	325,033	353,381	4,948,058
	# Conflicts	12,277,288	181,957	822,390	1,056,764	868,056	616,930
11	Bayes-Prove	1,391	492	1,316	631	890	2,587
	# Constraints	34,165	344,637	414,967	322,212	350,596	4,947,717
	# Conflicts	52,773,713	1,393,935	3,293,955	1,933,103	3,165,312	1,459,105
11	Markov-Find	2,426	2,770	1,845	2,682	2,966	5,224
	# Constraints	1,580,164	2,176,067	2,296,063	2,309,147	2,436,769	36,699,300
	# Conflicts	1,771,663	1,276,599	1,092,467	1,130,776	1,178,797	318,771
11	Markov-Prove	2,251	3,294	3,428	3,255	3,229	5,402
	# Constraints	1,580,164	2,182,157	2,302,171	2,307,991	2,435,603	36,694,525
	# Conflicts	1,806,525	1,788,800	1,720,270	1,521,272	1,452,042	317,555
38	Fastfood	10,277	12,843	14,156	13,756	13,479	17,867
	# Constraints	928,390	2,880,725	3,640,856	2,826,606	3,667,538	11,860,656
	# Conflicts	122,423,130	47,566,085	42,794,938	44,148,615	49,035,512	8,940,612
12	Inc-Scheduling	257	1,340	1,330	1,481	1,581	
	# Constraints	2,304,166	7,161,226	8,166,527	7,274,513	8,570,210	
	# Conflicts	82,790	127,628	134,987	218,224	173,849	
15	Nomystery	4,907	4,236	3,332	4,290	3,512	4,739
	# Constraints	845,321	1,678,580	2,330,329	1,725,458	2,459,603	5,115,156
	# Conflicts	10,765,572	3,216,072	2,161,566	3,207,353	2,092,378	2,047,501
109	Summary	21,715	25,009	25,576	26,345	25,827	
	# Constraints	7,306,535	16,770,842	19,568,681	17,090,960	20,273,700	
	# Conflicts	201,900,681	55,551,076	52,020,573	53,216,107	57,965,946	
109	Summary	21,715	24,758	26,611	26,524	26,063	
	without	7,306,535	17,279,805	21,632,440	17,665,922	22,358,451	
	simplification	201,900,681	52,264,536	46,809,044	56,247,153	51,814,629	

Table 1. Sums of runtimes, numbers of constraints, and conflicts encountered by CLASP.

are included for comparison. All normalization approaches make use of the weight rule simplifications described in Section 2. The last three rows, however, give accumulated results obtained without simplifications. The summaries exclude the SWC scheme, which works well on "small" weight rules but leads to significant size increases on large ones, as it exceeds the time and memory limits on Incremental scheduling instances.

Considering the benchmark classes in Table 1, normalization has a tremendous effect on the search performance of CLASP for the Bayesian network problems. Although the number of constraints increases roughly by a factor of 10, CLASP encounters about two orders of magnitude fewer conflicts on satisfiable as well as unsatisfiable instances (indicated by "Find" or "Prove"). In particular, we observe advantages due to using mixed-radix bases along with structure sharing. On the Markov network instances, the size increase but also the reduction of conflicts by applying the normalization schemes presented in Section 4 are modest. As a consequence, the runtimes stay roughly the

same as with native weight rule handling by CLASP. Interestingly, the SWC scheme is able to significantly reduce the number of conflicts, yet the enormous size outweighs these gains. The Fastfood instances exhibit similar effects, that is, all normalization approaches lead to a reduction of conflicts, but the increased representation size inhibits runtime improvements. Unlike with the other problems, normalizations even deteriorate search in Incremental scheduling, and the additional atoms and constraints they introduce increase the number of conflicts. With the SWC scheme, the resulting problem size is even prohibitive here. These observations emphasize that the effects of normalization are problem-specific and that care is needed in deciding whether to normalize or not. In fact, normalizations turn again out to be helpful on Nomystery planning instances. Somewhat surprisingly, both with mixed-radix and binary bases, the omission of structure sharing leads to runtime improvements. Given the heuristic nature of structure sharing, it can bring about side-effects, so that it makes sense to keep such techniques optional.

In total, we conclude that the normalization approaches presented in Section 4 are practicable and at eye level with the native handling by CLASP. Although the problem size increases, the additional structure provided by the introduced atoms sometimes boosts search in terms of fewer conflicts, and the basic format of clausal constraints also makes them cheaper to propagate than weight rules handled natively. Advanced techniques like using mixed-radix instead of binary bases as well as structure sharing further improve the solving performance of CLASP on normalized inputs. Finally, taking into account that ASP grounders like GRINGO [19] do not themselves "clean up" ground rules before outputting them, the last three rows in Table 1 also indicate a benefit in terms of numbers of constraints due to simplifying weight rules a priori.

6 Related Work

Extended rule types were introduced in the late 90's [37], at the time when the paradigm of ASP itself was shaping up [30, 33, 35]. The treatment of weight rules varies from solver to solver. Native solvers like CLASP [20], DLV [17], IDP [41], SMODELS [38], and WASP2 [3] (where respective support is envisaged as future work) have internal data structures to handle weight rules. On the other hand, the CMODELS system [23] relies on translation [18] (to nested rules [31]). However, the systematic study of normalization approaches for extended rules was initiated with the LP2NORMAL system [26]. New schemes for the normalization of cardinality rules were introduced in [10], and this paper presents the respective generalizations to weight rules.

Weight rules are closely related to pseudo-Boolean constraints [36], and their normalization parallels translations of pseudo-Boolean constraints into plain SAT. The latter include adder circuits [16, 40], binary decision diagrams [2, 6, 16, 24, 39], and sorting networks [7, 16]. The normalization techniques presented in this paper can be understood as ASP adaptions and extensions of the *Global Polynomial Watchdog* [7] encoding of pseudo-Boolean constraints. Techniques for using mixed-radix bases [12, 16] and structure sharing [2, 16] have also been proposed in the context of SAT translation approaches. However, classical satisfiability equivalence between pseudo-Boolean constraints and their translations into SAT does not immediately carry forward to weight

rules, for which other notions, such as *visible strong equivalence* [27], are needed to account for the stable model semantics. Boolean circuits based on monotone operators yield normalization schemes that preserve stable models in the sense of visible strong equivalence. In particular, this applies to the merger and sorter programs from [10].

7 Conclusions

We presented new ways to normalize weight rules, frequently arising in ASP applications. To this end, we exploit existing translations from pseudo-Boolean constraints into SAT and adapt them for the purpose of transforming weight rules. At the technical level, we use merger and sorter programs from [10] as basic primitives. The normalization schemes based on them combine a number ideas, viz. mixed-radix bases, structure sharing, and tares for simplified bound checking. Such a combination is novel both in the context of ASP as well as pseudo-Boolean satisfiability.

Normalization is an important task in translation-based ASP and, in particular, if a back-end solver does not support cardinality and weight constraints. Our preliminary experiments suggest that normalization does not deteriorate solver performance although the internal representations of logic programs are likely to grow. The decision versions of hard optimization problems exhibit that normalization can even boost the search for answer sets by offering suitable branch points for the underlying branch&bound algorithm. It is also clear that normalization pays off when a rule under consideration forms a corner case (cf. Section 2). For a broad-scale empirical assessment, we have submitted a number of systems exploiting normalization techniques developed in this paper to the 5th ASP Competition (ASPCOMP 2014).

As regards future work, there is a quest for selective normalization techniques that select a scheme on the fly or decide not to normalize, given the characteristics of a weight rule under consideration and suitable heuristics. The current implementation of LP2NORMAL2 already contains such an automatic mode.

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A parametric approach for smaller and better encodings of cardinality constraints. In: CP 2013. Springer (2013) 80–96
2. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Mayer-Eichberger, V.: A new look at BDDs for pseudo-Boolean constraints. Journal of Artificial Intelligence Research **45** (2012) 443–480
3. Alviano, M., Dodaro, C., Ricca, F.: Preliminary report on WASP 2.0. In: NMR 2014. (2014)
4. 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., Wallner, J., Xiao, G.: The fourth answer set programming competition: Preliminary report. In: LPNMR 2013. Springer (2013) 42–53
5. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: A theoretical and empirical study. Constraints **16**(2) (2011) 195–221
6. Bailleux, O., Boufkhad, Y., Roussel, O.: A translation of pseudo Boolean constraints to SAT. Journal on Satisfiability, Boolean Modeling and Computation **2**(1-4) (2006) 191–200
7. Bailleux, O., Boufkhad, Y., Roussel, O.: New encodings of pseudo-Boolean constraints into CNF. In: SAT 2009. Springer (2009) 181–194
8. Batcher, K.: Sorting networks and their applications. In: AFIPS 1968. ACM (1968) 307–314
9. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS (2009)
10. Bomanson, J., Janhunen, T.: Normalizing cardinality rules using merging and sorting constructions. In: LPNMR 2013. Springer (2013) 187–199
11. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54**(12) (2011) 92–103
12. Codish, M., Fekete, Y., Fuhs, C., Schneider-Kamp, P.: Optimal base encodings for pseudo-Boolean constraints. In: TACAS 2011. Springer (2011) 189–204
13. Corander, J., Janhunen, T., Rintanen, J., Nyman, H., Pensar, J.: Learning chordal Markov networks by constraint satisfaction. In: NIPS 2013. Advances in Neural Information Processing Systems **26** (2013) 1349–1357
14. Cussens, J.: Bayesian network learning with cutting planes. In: UAI 2011. AUAI (2011) 153–160
15. De Cat, B., Bogaerts, B., Bruynooghe, M., Denecker, M.: Predicate logic as a modelling language: The IDP system. CoRR **abs/1401.6312** (2014)
16. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2**(1-4) (2006) 1–26
17. 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) (2008) 545–580
18. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming **5**(1-2) (2005) 45–74
19. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Communications **24**(2) (2011) 107–124
20. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: On the implementation of weight constraint rules in conflict-driven ASP solvers. In: ICLP 2009. Springer (2009) 250–264
21. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence **187** (2012) 52–89
22. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP 1988. MIT (1988) 1070–1080

23. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning **36**(4) (2006) 345–377
24. Hölldobler, S., Manthey, N., Steinke, P.: A compact encoding of pseudo-Boolean constraints into SAT. In: KI 2012. Springer (2012) 107–118
25. Jaakkola, T., Sontag, D., Globerson, A., Meila, M.: Learning Bayesian network structure using LP relaxations. In: AISTATS 2010. JMLR (2010) 358–365
26. Janhunen, T., Niemelä, I.: Compact translations of non-disjunctive answer set programs to propositional clauses. In: Gelfond Festschrift. Springer (2011) 111–130
27. Janhunen, T., Niemelä, I.: Applying visible strong equivalence in answer-set program transformations. In: Lifschitz Festschrift. Springer (2012) 363–379
28. Janhunen, T., Niemelä, I., Sevalnev, M.: Computing stable models via reductions to difference logic. In: LPNMR 2009. Springer (2009) 142–154
29. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
30. Lifschitz, V.: Answer set planning. In: ICLP 1999. MIT (1999) 23–37
31. Lifschitz, V., Tang, L., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 369–389
32. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
33. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: A 25-Year Perspective. Springer (1999) 375–398
34. Nguyen, M., Janhunen, T., Niemelä, I.: Translating answer-set programs into bit-vector logic. In: INAP 2011. Springer (2013) 95–113
35. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 241–273
36. Roussel, O., Manquinho, V.: Pseudo-Boolean and cardinality constraints. In: Handbook of Satisfiability. IOS (2009) 695–733
37. Simons, P.: Extending the stable model semantics with more expressive rules. In: LPNMR 1999. Springer (1999) 305–316
38. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
39. Tamura, N., Banbara, M., Soh, T.: PBSugar: Compiling pseudo-Boolean constraints to SAT with order encoding. In: PoS 2013. (2013)
40. Warners, J.: A linear-time transformation of linear inequalities into conjunctive normal form. Information Processing Letters **68**(2) (1998) 63–69
41. Wittocx, J., Denecker, M., Bruynooghe, M.: Constraint propagation for first-order logic and inductive definitions. ACM Transactions on Computational Logic **14**(3) (2013) 17:1–17:45