Debugging non-ground ASP programs with Choice Rules, Cardinality and Weight Constraints

Axel Polleres[1], Melanie Frühstück[1], Gottfried Schenner[1], and Gerhard Friedrich[2]

[1] Siemens AG Österreich, Siemensstraße 90, 1210 Vienna, Austria
[2] Alpen-Adria Universität, Klagenfurt, Austria

Abstract. When deploying Answer Set Programming (ASP) in an industrial context, for instance for (re-)configuration [5], knowledge engineers need debugging support on non-ground programs. Current approaches to ASP debugging, however, do not cover extended modeling features of ASP, such as choice rules, conditional literals, cardinality and weight constraints [13]. To this end, we encode non-ground ASP programs using extended modeling features into normal logic progams; this encoding extends existing encodings for the case of ground programs [4, 10, 11] to the non-ground case. We subsequently deploy this translation on top of an existing ASP debugging approach for non-ground normal logic programs [14]. We have implemented and tested the approach and provide evaluation results.

1 Introduction

Answer Set Programming (ASP), with its intuitive and declarative modeling features – offering the possibility to model knowledge base constraints concisely in the form of non-ground programs plus advanced modeling feature such as choice rules, cardinality constraints and weight constraints [13] – has become an attractive tool for knowledge engineers also in an industrial context. For instance, within the RECONCILE project[3] we deploy ASP for modeling and solving configuration and (re-)configuration problems [5] occurring in practical settings such as in large-scale projects in the railway automation domain.

While in such a context the advanced features in Answer Set Programming (ASP) significantly increase the declarative modeling capabilities of the language, debugging tools that support the full language of ASP are still missing: most approaches for debugging are only able to deal with propositional programs [1, 2, 9, 16, 18], with the exception of Oetsch et al. [14], who developed a meta-program for debugging normal logic programs. Still, this latter approach does not support debugging in the presence of features such as choice rules, cardinality constraints and weight constraints.

Notably, as shown in earlier works, these language constructs do not raise expressivity beyond normal logic programs: Ferraris and Lifschitz [4] have shown how weight constraints can be encoded as nested expressions, while Janhunen and Niemelä [11]

[3] `https://www.cee.siemens.com/web/at/de/corporate/portal/Innovation/InnovationStories/Pages/Reconcile.aspx`

provide a translation of choice rules, cardinality and weight constraints into normal logic programs. In one step of the translation process they show how to transform SMODELS programs to normal programs. Another representation of the translation of choice and cardinality rules to normal logic programs can be found in [10]; their approach is based on [11] introducing more intermediate steps in the translation.

However, all the above mentioned literature focus on propositional ASP programs. In this paper we describe a transformation of non-ground choice rules, as well as cardinality and weight constraints with conditions into non-ground logic programs. Our proposal is mainly based on the structure of rules of Gebser and Schaub [10]. Eventually, we show how to deploy this non-ground embedding for debugging programs using advanced ASP features: based on the non-ground debugging approach by Oetsch et al. [14] for normal logic programs, after applying our translation process, ASP debugging also becomes feasible for programs using more advanced ASP features.

We first introduce the ASP language used herein in Section 2. Then, we extend the translation of [10] to the non-ground case (Section 3). We present an evaluation of this translation, comparing our non-ground embedding to the propositional embedding from [11] in Section 4. Finally, in Section 5 we illustrate how our translation can be embedded into the debugging approach of [14], before we conclude in Section 6.

2 Preliminaries

Syntax A *literal* is an atom that is possibly preceded by the *strong negation* symbol $\neg$. We define a *normal* (non-ground) rule r as

$$h(\overline{x_h}) \leftarrow Body(\overline{X_{Body}}). \tag{1}$$

where $h(\overline{x_h})$ defines the *head* of the rule, i.e. a literal including its vector of parameters $\overline{x_h}$ (variables and constants).[4] The *body* of a rule $Body(\overline{X_{Body}}) = Body^+(\overline{X_{Body}}) \cup Body^-(\overline{X_{Body^-}})$ consists of $Body^+(\overline{X_{Body}})$, the set of all positive body literals, and $Body^-(\overline{X_{Body^-}})$, a set of default-negated body literals (i.e. literals preceded by *not*). X_{Body} denotes the set of all variables occurring in body literals; note that since we assume *safety* all these variables also occur in positive body literals, i.e. more precisely: a rule r of the form (1) is called *safe* if $X_h \subseteq X_{Body}$ and $X_{Body^-} \subseteq X_{Body}$; if the head is omitted then r is called a *constraint*; if the body is empty then r is called a *fact*.

Additionally to normal rules of the form (1), we consider programs expanded with choice rules and cardinality rules, as for instance supported by Potsdam Answer set Solving Collection (Potassco) [7]. Choice rules have the form

$$\{h_1(\overline{x_{h_1}}) : Cond(\overline{Y_{h_1}}), \ldots, h_n(\overline{x_{h_n}}) : Cond(\overline{Y_{h_n}})\} \leftarrow Body(\overline{X_{Body}}). \tag{2}$$

[4] As usual, we denote variables by upper case letters and constants by alphanumeric strings starting with a lower case letter. We further denote mixed vectors of constants and variables by overlined lower case letters (such as $\overline{x}$) whereas, accordingly, we denote the corresponding vector of all variables occurring in $\overline{x}$ by $\overline{X}$ (preserving order) and by X we denote the respective corresponding (unordered) set of variables.

where $h_i(\overline{x_{h_i}}) : Cond(\overline{Y_{h_i}})$ is called a *conditional literal*. The *condition* $Cond(\overline{Y_{h_i}})$ consists of positive literals (with variables in Y_{h_i}) separated by further colons, read as a conjunction, and can be possibly empty.[5]

Further, *cardinality constraints* $l\{h_1(\overline{x_{h_1}}) : Cond(\overline{Y_{h_1}}), \ldots, h_n(\overline{x_{h_n}}) : Cond(\overline{Y_{h_n}})\}u$ where l, u are either numeric constants or variables representing lower and upper bounds are allowed in rule heads and bodies, i.e. w.l.o.g. in rules of the following forms:

$$h(\overline{x_h}) \leftarrow l\{b_1(\overline{x_{b_1}}) : Cond(\overline{Y_{b_1}}), \ldots, b_n(\overline{x_{b_n}}) : Cond(\overline{Y_{b_n}})\}u, Body(\overline{X_{Body}}). \quad (3)$$

$$h(\overline{x_h}) \leftarrow not\ l\{b_1(\overline{x_{b_1}}) : Cond(\overline{Y_{b_1}}), \ldots, b_n(\overline{x_{b_n}}) : Cond(\overline{Y_{b_n}})\}u, Body(\overline{X_{Body}}). \quad (4)$$

$$l\{h_1(\overline{x_{h_1}}) : Cond(\overline{Y_{h_1}}), \ldots, h_n(\overline{x_{h_n}}) : Cond(\overline{Y_{h_n}})\}u \leftarrow Body(\overline{X_{Body}}). \quad (5)$$

We call a set P of *safe* rules of the forms (1)–(5) a *program*: here, we extend the standard notion of safety for rules of the form (1) to rules (2)–(5) as follows: a conditional literal $h_i(\overline{x_{h_i}}) : Cond(\overline{Y_{h_i}})$ within a rule r is *safe* if for all $1 \leq i \leq n$ it holds that $X_{h_i} \subseteq X_{Body} \cup Y_{h_i}$. Accordingly, rules of the forms (2)–(5) are *safe* if (i) they are safe in the standard sense (see above), (ii) all conditional literals are safe, and (iii) bounds l, u are either constants or variables from X_{Body}.[6]

Semantics The *Herbrand universe* HU_P of a program P is the set of all constants appearing in P and the *Herbrand base* HB_P is the set of all ground atoms constructed by predicate symbols in P using constants from HU_P.[7]

As usual in ASP, we define the semantics of a program P in terms of its grounding; the *grounding* of a rule r, $ground(r)$, is defined by the set of ground rules obtained from (i) taking the set of all its ground instantiations, and (ii) replacing each conditional literal $h_i(\overline{x_{h_i}}) : Cond(\overline{Y_{h_i}})$ (within a choice or a cardinality constraint) with the **set** of all possible ground conditional literals obtained from substituting variables with constants from HU_P. Accordingly, we call $ground(P) = \bigcup_{r \in P} ground(r)$, the *grounding* of program P. Note that this procedure covers the two-step instantiation described in [17]: i.e. what they call "global" variables are replaced through step (i) and "local" variables during the expansion in step (ii).

An interpretation $I \subseteq HB_P$ satisfies a ground literal b, written $I \models b$, if $b \in I$. Analogously, $I \models b$ for a ground cardinality constraint $b = l\{h_1 : c_{1,1} : ... : c_{1,m_1}, \ldots, h_n : c_{n,1} : ... : c_{n,m_n}\}u$, if

$$l \leq |\ \{h_i \mid \{h_i, c_{i,1}, ..., c_{i,m_i}\} \subseteq I\}\ | \leq u$$

[5] For simplicity we only consider positive conditional literals and conditions herein; tools of Potassco also allow default negation within conditional literals which we leave to future work. In our formal definitions we also exclude built-ins in conditions (which we allow though in our implementation, cf. Section 5).

[6] Note that we leave out the form where cardinality constraints can be used to assign values to unsafe variables; tools of Potassco also allow cardinality constraints of the form $X = \{h_1(\overline{x_{h_1}}) : Cond(\overline{Y_{h_1}}), \ldots, h_n(\overline{x_{h_n}}) : Cond(\overline{Y_{h_n}})\}$ which assign the cardinality to an (unsafe) variable; we leave this extension to future work.

[7] Note that we assume no "overloading", i.e. each predicate symbol has a fixed arity. This restriction, which is not made in current ASP tools (like those of Potassco), can be easily lifted in a preprocessing step where you replace predicate names occurring in different arities with new unique predicate names per arity, e.g. $p(X), p(X, Y)$ become $p_{/2}(X, Y), p_{/1}(X)$, or alike.

Next, we define the *reduct* r^I of a rule r wrt. $I \subseteq HB_P$ as a set of rules as follows

– if $r : h \leftarrow Body$ is a ground rule where h is a literal, then

$$r^I = \begin{cases} \{h \leftarrow Body^+\} & \text{if there is no } not\ b \in Body^- \text{ with } I \models b. \\ \emptyset & \text{otherwise} \end{cases}$$

– if $r : \{h_1 : c_{1,1} : ... : c_{1,m_1}, \ldots, h_n : c_{n,1} : ... : c_{n,m_n}\} \leftarrow Body$ is a ground choice rule, then r^I is a *set* containing, for each $h_i \in I \cap \{h_1, ..., h_n\}$, the rule

$$r^I_{h_i} = \begin{cases} h_i \leftarrow c_{i,1}, ..., c_{i,m_i}, Body^+ & \text{if there is no } not\ b \in Body^- \text{ with } I \models b. \\ \emptyset & \text{otherwise} \end{cases}$$

Any consistent interpretation I, such that I is a (subset-)minimal model of $P^I = \bigcup_{r \in P} r^I$ is called an *answer set*; likewise, answer sets for a non-ground program P are defined as the answer sets of $ground(P)$.

As in [17], we view rules of the form (5) as syntactic sugar not treated separately in the semantics; we will get back to these in the next section.

As a further extension, *weighted conditional literals* which assign a weight w_i (either a numeric constant or a safe variable, cf. footnote 6) to a conditional literal are allowed in so called *weight constraints* of the form

$$l[h_1(\overline{x_{h_1}}) : Cond(\overline{Y_{h_1}}) = w_1, \ldots, h_n(\overline{x_{h_n}}) : Cond(\overline{Y_{h_n}}) = w_n]u \tag{6}$$

These distinguish from cardinality constraints in that values are summed in a multi-set semantics, i.e. weights w_i, w_j of satisfying ground instances for each conditional literal i and j count separately even if $h_i = h_j$, i.e. when replacing cardinality constraints within rules of the forms (3)–(5) with their weighted counterparts, the upper and lower bounds mean to indicate bounds for sums of *weights* of satisfied instances, rather than counting distinct instances. The semantics of weight constraints extends the semantics for cardinality constraints straightforwardly, formal details of which we omit here for space limitations.

3 Translation to Normal Rules

We translate rules of forms (2)–(5) successively to normal rules in several steps.

Step 1. We first consider choice rules of the form (2). A choice rule can be translated into following rules:

$$\begin{aligned} h_1(\overline{x_{h_1}}) &\leftarrow Body(\overline{X_{Body}}), Cond(\overline{Y_{h_1}}), not\ h'_{r,1}(\overline{x_{h_1}}). \\ h'_{r,1}(\overline{x_{h_1}}) &\leftarrow Body(\overline{X_{Body}}), Cond(\overline{Y_{h_1}}), not\ h_1(\overline{x_{h_1}}). \\ &\ldots \\ h_n(\overline{x_{h_n}}) &\leftarrow Body(\overline{X_{Body}}), Cond(\overline{Y_{h_n}}), not\ h'_{r,n}(\overline{x_{h_n}}). \\ h'_{r,n}(\overline{x_{h_n}}) &\leftarrow Body(\overline{X_{Body}}), Cond(\overline{Y_{h_n}}), not\ h_n(\overline{x_{h_n}}). \end{aligned} \tag{7}$$

where the $h'_{r,i}$ are new predicate symbols, unique to the rule r they appear in (to avoid interferences between the translations of several choice rules).

Step 2. Next, we reduce any rules with cardinality constraints to the form of (3), that is, we rewrite rules of the forms (4)+(5) in such a way that cardinality constraints only appear positively in rule bodies: a cardinality rule r of the form (5) is replaced by

(i) its unconstrained variant, i.e. the translation (according to Step 1) of the choice rule obtained by removing upper and lower bounds l and u; and
(ii) the following pair of rules (where, c_r is a "fresh" predicate symbol):

$$c_r(\overline{X_{Body}}) \leftarrow l\{h_1(\overline{x_{h_1}}) : Cond(\overline{Y_{h_1}}), ..., h_n(\overline{x_{h_n}}) : Cond(\overline{Y_{h_n}})\}u, Body(\overline{X_{Body}}). \quad (8)$$

$$\leftarrow not\ c_r(\overline{X_{Body}}), Body(\overline{X_{Body}}). \quad (9)$$

Similarly, cardinality rules of the form (4) are replaced by the pair of rules

$$c_r(\overline{X_{Body}}) \leftarrow l\{b_1(\overline{x_{b_1}}) : Cond(\overline{Y_{b_1}}), ..., b_n(\overline{x_{b_n}}) : Cond(\overline{Y_{b_n}})\}u, Body(\overline{X_{Body}}). \quad (10)$$

$$h(\overline{x_h}) \leftarrow not\ c_r(\overline{X_{Body}}), Body(\overline{X_{Body}}). \quad (11)$$

Step 3. Finally, cardinality rules of the form (3) – including those of the forms (8)+(10) obtained in the previous step – are translated as follows.

(i) First, we translate the body cardinality constraint to a variant with only lower bounds as follows

$$h(\overline{x_h}) \leftarrow l_r(\overline{X_{Body}}), not\ u_r(\overline{X_{Body}}), Body(\overline{X_{Body}}). \quad (12)$$

$$l_r(\overline{X_{Body}}) \leftarrow l\{b_1(\overline{x_{b_1}}) : Cond(\overline{Y_{b_1}}), ..., b_n(\overline{x_{b_n}}) : Cond(\overline{Y_{b_n}})\}, Body(\overline{X_{Body}}). \quad (13)$$

$$u_r(\overline{X_{Body}}) \leftarrow u + 1\{b_1(\overline{x_{b_1}}) : Cond(\overline{Y_{b_1}}), ..., b_n(\overline{x_{b_n}}) : Cond(\overline{Y_{b_n}})\}, Body(\overline{X_{Body}}). \quad (14)$$

where l_r, u_r are new predicate symbols. Note that $Body(\overline{X_{Body}})$ in rule (12) is not strictly necessary when both a lower and an upper bound are given, but, since both $l_r(\overline{X_{Body}})$ and $not\ u_r(\overline{X_{Body}})$ are optional in this rule, it is necessary to guarentee safety in the absence of the latter. Likewise, rule (13) (and (14), resp.) is only needed in case a lower (or upper, resp.) bound is given.

(ii) Next, we translate rules with a body cardinality constraint with only lower bounds, i.e. rules of the form

$$h(\overline{x_h}) \leftarrow l\{b_1(\overline{x_{b_1}}) : Cond(\overline{Y_{b_1}}), \ldots, b_n(\overline{x_{b_n}}) : Cond(\overline{Y_{b_n}})\}, Body(\overline{X_{Body}}). \quad (15)$$

are translated to

$$h(\overline{x_h}) \leftarrow cnt_r(\overline{X_{Body}}, C), Body(\overline{X_{Body}}), C \geq l. \quad (16)$$

The definition of the new predicate cnt_r is given as follows. We assume a built-in predicate "$<$" defining a total, lexical order for pairs of constants in HU_P. Further, for sequence $\overline{x_{b_i}}$, let $\overline{x_{b_i}}'$ denote the sequence obtained from replacing each variable x occurring in $\overline{x_{b_i}}$ by a fresh variable x'. Lastly, let $\overline{x^{\cup}_{r,i}} = (\overline{x_{b_i}}, \overline{X_{Body}})$, i.e. the concatenation of the two vectors $\overline{x_{b_i}}$ and $\overline{X_{Body}}$ and $\overline{x^{\cup}_{r,i}}' = (\overline{x_{b_i}}', \overline{X_{Body}})$.

We now define the predicate cnt_r by the following auxiliary rules, for each $i \in \{1, \ldots, n\}$

$$val_{r,b_i}(\overline{x^{\cup}_{r,i}}) \leftarrow b_i(\overline{x_{b_i}}), Cond(\overline{Y_{b_i}}), Body(\overline{X_{Body}}). \quad (17)$$

$$exists_{r,b_i}(\overline{X_{Body}}) \leftarrow Body(\overline{X_{Body}}), val_{r,b_i}(\overline{x^{\cup}_{r,i}}). \quad (18)$$

$$exists^{<}_{r,b_i}(\overline{x^{\cup}_{r,i}}) \leftarrow val_{r,b_i}(\overline{x^{\cup}_{r,i}}), val_{r,b_i}(\overline{x^{\cup}_{r,i}}'), \overline{x_{b_i}}' <_{|x_{b_i}|} \overline{x_{b_i}}. \quad (19)$$

$$exists^{>}_{r,b_i}(\overline{x^{\cup}_{r,i}}) \leftarrow val_{r,b_i}(\overline{x^{\cup}_{r,i}}), val_{r,b_i}(\overline{x^{\cup}_{r,i}}'), \overline{x_{b_i}} <_{|x_{b_i}|} \overline{x_{b_i}}'. \quad (20)$$

$$next_{r,b_i}(\overline{x^{\cup}_{r,i}}, \overline{x^{\cup}_{r,i}}') \leftarrow val_{r,b_i}(\overline{x^{\cup}_{r,i}}), val_{r,b_i}(\overline{x^{\cup}_{r,i}}'), \overline{x_{b_i}} <_{|x_{b_i}|} \overline{x_{b_i}}', \quad (21)$$
$$not\ between_{r,b_i}(\overline{x^{\cup}_{r,i}}, \overline{x^{\cup}_{r,i}}').$$

$$between_{r,b_i}(\overline{x^{\cup}_{r,i}}, \overline{x^{\cup}_{r,i}}'') \leftarrow val_{r,b_i}(\overline{x^{\cup}_{r,i}}), val_{r,b_i}(\overline{x^{\cup}_{r,i}}'), val_{r,b_i}(\overline{x^{\cup}_{r,i}}''), \quad (22)$$
$$\overline{x_{b_i}} <_{|x_{b_i}|} \overline{x_{b_i}}', \overline{x_{b_i}}' <_{|x_{b_i}|} \overline{x_{b_i}}''.$$

$$cnt_{r,b_i}(\overline{x^{\cup}_{r,i}}, 1) \leftarrow val_{r,b_i}(\overline{x^{\cup}_{r,i}}), not\ exists^{<}_{r,b_i}(\overline{x^{\cup}_{r,i}}). \quad (23)$$

$$cnt_{r,b_i}(\overline{x^{\cup}_{r,i}}', N+1) \leftarrow next_{r,b_i}(\overline{x^{\cup}_{r,i}}, \overline{x^{\cup}_{r,i}}'), cnt_{r,b_i}(\overline{x^{\cup}_{r,i}}, N). \quad (24)$$

$$cnt'_{r,b_i}(\overline{X_{Body}}, N) \leftarrow cnt_{r,b_i}(\overline{x^{\cup}_{r,i}}, N), not\ exists^{>}_{r,b_i}(\overline{x^{\cup}_{r,i}}) \quad (25)$$

$$cnt'_{r,b_i}(\overline{X_{Body}}, 0) \leftarrow Body(\overline{X_{Body}}), not\ exists_{r,b_i}(\overline{X_{Body}}). \quad (26)$$

where $<_n$ is an auxiliary predicate of arity $2n$ which determines whether the first of two vectors of the same length n is lexicographically smaller than the latter. For $n > 0$, the predicate $<_n$ can be easily defined recursively over the built-in predicate "$<$" in the rules (27)+(28) as follows:

$$(X_1, ..., X_k) <_k (Y_1, ..., Y_k) \leftarrow X_1 < Y_1. \qquad \forall 1 \leq k \leq n \quad (27)$$

$$(X_1, X_2, ..., X_k) <_k (X_1, Y_2, ..., Y_k) \leftarrow (X_2, ..., X_k) <_{k-1} (Y_2, ..., Y_k). \quad \forall 1 < k \leq n \quad (28)$$

Rule (17) "collects" all possible bindings ("values") for variables that make a particular conditional atom b_i true, dependent on a particular body instantiation. The auxiliary rule (18) determines whether a value exists at all for a particular body instantiation; existence of a smaller, or greater, resp., than a prticular value is computed in the auxiliary rules (19)+(20). Rules (21) and (22) define a total order over values, defining a successor predicate ($next$) via the auxiliary information that no value lies in between two consecutive values. The cnt_{r,b_i} predicate then counts all the instantiations that belong to a particular conditional atom b_i, cf. rules (23)+(24). Rule (25) collects, for each b_i and body instantiation, the maximum count in the auxiliary predicates cnt'_{r,b_i}, where rule (26) sets this predicate to 0, in case no actual value exists for the conditional atom b_i. Finally, cnt_r is defined by the following rule which simply sums up all the maximum counts for the respective b_i's.

$$cnt_r(\overline{X_{Body}}, N) \leftarrow cnt'_{r,b_1}(\overline{X_{Body}}, N_1), ..., cnt'_{r,b_m}(\overline{X_{Body}}, N_m), N = N_1 + ... + N_m. \quad (29)$$

where $\{b_1, \ldots, b_m\}$ is the set of distinct predicate names occurring in $\{b_1, \ldots, b_n\}$.

Proposition 1. *The answer sets of a program P and its translation obtained from Steps 1-3 outlined above are in 1-to-1 correspondence.*

While we omit a full proof, we argue that the translation steps outlined above "emulate" semantics as described in Section 2 on non-ground programs, when assuming that HU_P contains apart from explicitly mentioned constants, integers from 0 to a

finitely computable upper bound for instantiating and evaluating N in rules (24),(25), and (29) correctly; state-of-the-art ASP solvers like Potassco deal with such arithmetics appropriately out-of-the-box, which is our main concern when deploying the translation within our debugging use case (cf. Section 5 below).

As a possible optimization, which reduces the number and size of non-ground rules, note that it is possible to equivalently replace rules (19)–(25) with the following rules

$$cnt_{r,b_i}(\overline{x^{\cup}_{r,i}}, 1) \leftarrow val_{r,b_i}(\overline{x^{\cup}_{r,i}}). \tag{23'}$$

$$cnt_{r,b_i}(\overline{x^{\cup}_{r,i}}', N+1) \leftarrow val_{r,b_i}(\overline{x^{\cup}_{r,i}}'), \overline{x_{b_i}} <_{|X_{b_i}|} \overline{x_{b_i}}', cnt_{r,b_i}(\overline{x^{\cup}_{r,i}}, N). \tag{24'}$$

$$cnt'_{r,b_i}(\overline{X_{Body}}, N) \leftarrow cnt_{r,b_i}(\overline{x^{\cup}_{r,i}}, N), not\ nmax_{r,b_i}(\overline{X_{Body}}, N). \tag{25'}$$

$$nmax_{r,b_i}(\overline{X_{Body}}, N-1) \leftarrow cnt_{r,b_i}(\overline{x^{\cup}_{r,i}}, N). \tag{25"}$$

The idea behind this optimization is that, despite getting potentially various derivations for each N per body instance in rules (23')+(24'), there is only one unique *maximum* N derived per body instance, cf. rule (25'), which is the only relevant fact for rule (29), and in consequence for rule (16). Here, the new auxiliary rule (25") is needed to assess that a certain value N is not the maximum count.

Taking this further, the instances of $<_{|X_{b_i}|}$ above can be replaced by a custom comparison predicate $smaller_{r,b_i}$ for each conditional atom b_i. Let k_i denote the arity of b_i, then $smaller_{r,b_i}$ is defined by the following set of rules:

$$\begin{aligned}
&smaller_{r,b_i}(X_1,\ldots,X_{k_i},Y_1,\ldots,Y_{k_i}) \leftarrow X_1 < Y_1,\\
&\qquad val_{r,b_i}(X_1,\ldots,X_{k_i},\overline{X_{Body}}), val_{r,b_i}(Y_1,\ldots,Y_{k_i},\overline{X_{Body}}).\\
&smaller_{r,b_i}(X_1,X_2,\ldots,X_{k_i},X_1,Y_2,\ldots,Y_{k_i}) \leftarrow X_2 < Y_2,\\
&\qquad val_{r,b_i}(X_1,X_2,\ldots,X_{k_i},\overline{X_{Body}}), val_{r,b_i}(X_1,Y_2,\ldots,Y_{k_i},\overline{X_{Body}}).\\
&\qquad\vdots\\
&smaller_{r,b_i}(X_1,\ldots,X_{k_i},X_1,\ldots,X_{k_i-1},Y_{k_i}) \leftarrow X_{k_i} < Y_{k_i},\\
&\qquad val_{r,b_i}(X_1,\ldots,X_{k_i},\overline{X_{Body}}), val_{r,b_i}(X_1,\ldots,X_{k_i-1},Y_{k_i},\overline{X_{Body}}).
\end{aligned} \tag{30}$$

The idea of this definition is that the $smaller_{r,b_i}$ predicate really only compares values relevant for the particular b_i, instead of defining a generic smaller relation between *any* tuples in HU^n_P, which potentially narrows down the size of the grounding.

3.1 Extending the translation by weights

So far, we have only treated "pure" cardinality constraints, involving only conditional atoms with the default weight 1. It is not hard to extend the translation above to arbitrary weight constraints involving weighted conditional literals of the form (6). Firstly, we redefine $\overline{x^{\cup}_{r,i}}$ as follows

$$\overline{x^{\cup}_{r,i}} = (\overline{x_{b_i}}, \overline{X_{Body}}, w_i)$$

i.e. we carry over weights as an additional parameter in our auxiliary predicates. Apart from this change, rules (17)–(22) are modified with respect to the predicate names

val_{r,b_i}, $exists_{r,b_i}$, $exists^{<}_{r,b_i}$, $exists^{>}_{r,b_i}$, $next_{r,b_i}$, and $between_{r,b_i}$ which are now replaced with $val_{r,i}$, $first_{r,i}$, $exists_{r,i}$, $exists^{<}_{r,i}$, $exists^{>}_{r,i}$, $next_{r,i}$, and $between_{r,i}$, respectively. I.e. values are no longer collected "per predicate" b_i, but separately for each weighted conditional literal at position $1 \leq i \leq n$, in order to cater for the multi-set semantics of weight constraints. Secondly, we need to replace the counting rules (23)–(26) and (29) by rules that do summation instead; we use, in analogy to the cnt and cnt' predicates from above a new predicates sum and sum' here:

$$sum_{r,i}(\overline{x^{\cup}_{r,i}}, w_i) \leftarrow val_{r,i}(\overline{x^{\cup}_{r,i}}), not\ exists^{<}_{r,i}(\overline{x^{\cup}_{r,i}}). \tag{31}$$

$$sum_{r,i}(\overline{x^{\cup}_{r,i}}, W + w_i) \leftarrow next_{r,i}(\overline{x^{\cup}_{r,i}}', \overline{x^{\cup}_{r,i}}), sum_{r,i}(\overline{x^{\cup}_{r,i}}', W). \tag{32}$$

$$sum'_{r,i}(\overline{X_{Body}}, W) \leftarrow sum_{r,i}(\overline{x^{\cup}_{r,i}}, W), not\ exists^{>}_{r,i}(\overline{x^{\cup}_{r,i}}). \tag{33}$$

$$sum'_{r,i}(\overline{x^{\cup,0}_{r,i}}, 0) \leftarrow Body(\overline{X_{Body}}), not\ exists_{r,i}(\overline{X_{Body}}). \tag{34}$$

$$sum_r(\overline{X_{Body}}, W) \leftarrow sum'_{r,1}(\overline{X_{Body}}, W_1), ..., sum'_{r,n}(\overline{X_{Body}}, W_n), \quad W = W_1 + ... + W_n. \tag{35}$$

Similar to the predicates cnt and cnt' before, the unique total sum value over all values is collected in the $sum'_{r,i}$ predicates for each i, whereas the $sum_{r,i}$ predicates collect the respective intermediate sums. Note that, due to negative weights, sums are not necessarily monotonically increasing over all values; this prevents, on the one hand, the same optimization as for cnt (cf. rules (23')+(25")) to be applied in the case of weight constraints. On the other hand, the resulting encoding can – assuming that respective arithmetic is supported – deal with negative weights out-of-the-box, i.e. negative weights do not need to be eliminated as in [13].

Finally, rule (16) is analogously replaced by

$$h(\overline{X_h}) \leftarrow sum_r(\overline{X_{Body}}, W), Body(\overline{X_{Body}}), W \geq l. \tag{36}$$

4 Evaluation

Obviously, the additional machinery added in our translation comes at a cost. In order to evaluate how much it affects program size and performance in state of the art solvers, we chose some benchmark problems from the second Answer Set Programming Competition [3] involving cardinality constraints and choices: we took 8 different instances for *graph colouring*, *knight tour*, *hanoi* and *partner units*.

For grounding and solving we used gringo (v. 3.0.5) and clasp (v. 2.1.1) from Potassco. Results are reported in Table 1: each column reports size of the non-ground program ($\#ng$), size of the program after grounding ($\#g$), and evaluation time (t) in seconds (including grounding, translation and solving). We report results for grounding and evaluating the original program ($orig$), our naïve translation (tr), the optimized translation (tr_{opt}) using rules (23')–(30). Additionally, as a reference, we compare our results to first grounding the original program and then applying a ground transformation $tr_{\texttt{lp2normal}}$ to normal programs, using the tool `lp2normal` by Janhunen and Niemelä [11].

Table 1. Total times (in seconds).

Program	Instance*	$orig$	tr	tr_{opt}	$tr_{lp2normal}$
		$\#ng/\#g/t$	$\#ng/\#g/t$	$\#ng/\#g/t$	$\#ng/\#g/t$
Graph Colouring	$1-125$	$1672/6903/1.11$	$1690/23753/20.48$	$1687/20503/2.02$	$1672/10235/0.2$
	$11-130$	$1757/7243/>900$	$1775/22778/>900$	$1772/19653/>900$	$1757/9780/>900$
	$21-135$	$1986/8087/>900$	$2004/25232/>900$	$2001/21857/>900$	$1986/11194/>900$
	$30-135$	$1794/7415/14.24$	$1812/24560/24.71$	$1809/21185/4.51$	$1794/10522/13.44$
	$31-140$	$2039/8315/419.13$	$2057/26095/>900$	$2054/22595/>900$	$2039/11537/283.05$
	$40-140$	$2219/8945/>900$	$2237/26725/>900$	$2234/23225/>900$	$2219/12167/>900$
	$41-145$	$2262/9138/>900$	$2280/27553/>900$	$2277/23928/>900$	$2262/12475/>900$
	$51-150$	$2405/9681/>900$	$2423/28731/>900$	$2420/24981/>900$	$2405/13133/>900$
Knight Tour	$01-8$	$21/1852/0$	$61/23078/0.45$	$55/17794/0.2$	$21/5043/0.01$
	$03-12$	$22/4526/0.01$	$62/68044/6.34$	$56/50975/1.08$	$22/13386/0.04$
	$05-16$	$21/8388/0.03$	$61/136810/44.62$	$55/101314/5.01$	$21/25822/0.08$
	$06-20$	$21/13432/0.05$	$61/229346/232.84$	$55/168760/15.46$	$21/42233/0.12$
	$07-30$	$21/31222/0.14$	$61/564694/>900$	$55/412272/181.31$	$21/100752/0.43$
	$08-40$	$21/56412/0.34$	$61/1048642/>900$	$55/762774/758.63$	$21/184230/0.85$
	$09-46$	$21/75078/0.42$	$61/1410338/>900$	$55/1024440/>900$	$21/246336/1.23$
	$10-50$	$22/89000/0.88$	$62/1681185/>900$	$56/1220267/>900$	$22/292706/1.58$
Hanoi	$09-28$	$104/37323/1.56$	$168/3279898/>900$	$156/1745347/51.24$	$104/52445/4.39$
	$11-30$	$106/40041/13.62$	$170/3514328/>900$	$158/1870177/51.42$	$106/56243/5.74$
	$15-34$	$110/45477/51.56$	$174/3983116/>900$	$162/2119757/441.48$	$110/63839/31.71$
	$16-40$	$100/31886/1.56$	$164/2811325/>900$	$152/1496006/19.15$	$100/44848/2.07$
	$22-60$	$102/33314/0.82$	$166/3175997/633.51$	$154/1683463/26.99$	$102/46472/1.37$
	$36-80$	$106/40041/1.19$	$170/3514364/600.88$	$158/1870217/30.89$	$106/56243/1.11$
	$41-100$	$104/37322/0.48$	$168/3279933/321.64$	$156/1745386/22.45$	$104/52444/0.98$
	$47-120$	$99/30527/1.9$	$163/2694686/845.48$	$151/1434231/16.28$	$99/42949/0.75$
Partner Units	$176-24$	$68/13213/0.61$	$146/162007/40.98$	$131/102667/7.1$	$68/19347/1.07$
	$29-40$	$108/61777/0.13$	$186/1068887/>900$	$171/631427/>900$	$108/79679/6.98$
	$23-30$	$117/40332/0.13$	$195/451513/>900$	$180/277733/8.51$	$117/51127/0.49$
	$207-58$	$136/162537/0.61$	$214/4857458/>900$	$199/2730134/>900$	$136/203258/1.48$
	$204-67$	$141/223931/1.54$	$219/7672436/>900$	$204/4285390/>900$	$141/276403/1.86$
	$175-75$	$290/689087/20.78$	$368/15446057/>900$	$353/8611453/>900$	$290/762711/27.44$
	$52-100$	$254/963749/>900$	$332/36843565/>900$	$317/20137215/>900$	$254/1082289/>900$
	$115-100$	$254/963806/>900$	$332/37214669/>900$	$317/20328419/>900$	$254/1082942/>900$

*) For the instance naming convention, please refer to http://dtai.cs.kuleuven.be/events/ASP-competition/index.shtml.

As expected, the results in Table 1 show that the time that evaluation time rises significantly, which is mainly due to a blowup during grounding. There are certain exceptions, as in our selection one particular graph colouring example where our optimized encoding even outperforms all others. Solving the instances with pre-grounding the original program and using `lp2normal` on the ground instantiated program shows better results, however we couldn't use this approach in our use case of debugging, described in the next section.

5 Debugging with Ouroboros

In our project, we deploy ASP programs for encoding (re-)configuration problems [5], where debugging of the resulting (non-ground) programs became a significant issue in practical use cases. We base our debugger on the approach of Oetsch et al. [14], who developed a meta-program for debugging non-ground programs in ASP. The basic idea of this debugging method is to reify a program P as well as the fully expected interpretation I. Reification means that the program and interpretation are brought onto a meta-level. Finally, the meta program and meta interpretation are fed to an ASP solver. The obtained answer sets explain why I is not an answer set of P.

There are two main explanation classes why an interpretation I is not answer set of P. First, some atoms of the interpretation can form an unfounded loop. A non-empty set L of ground literals is a loop of P iff for each pair $(a, b) \in L$ there is a path from a to b in the positive dependency graph. The length of the path from a to b can be equal to or greater than 0. Additionally, let I, J be interpretations. J is supported by P wrt. I if the grounding of P contains some rule r whose body is satisfied by I and some head atoms of r are included in J, but all head atoms of r that are not included in J are false under I. Moreover, this support is ensured to be external, that means without any reference to the set J itself [14]. If J is not externally supported by P wrt. I, J is called *unfounded* by P with respect to I. In particular, if there is a loop in P that is contained in I but this loop is not externally supported (unfounded) by P with respect to I then I is not an answer set and the debugger program returns the unfounded (sub)set of I. The second type of explanation are unsatisfied rules, that means that instantiations of rules in P are not satisfied by I, in this case the debugger returns the (non-ground) unsatisfied rule(s).

The original meta-program debugger was written for DLV System [12] and can handle non-ground (even disjunctive) logic programs, integer arithmetic $(+, *)$, comparison predicates $(=, \neq, \leq, <, \geq, >)$ and strong negation. We made some minor adaptions to use Potassco, which we deployed throughout our project, where we do not need disjunction but make heavy use of other extended constructs such as choices, cardinality and weight constraints: As a first step, we transformed the meta-program in such a way that the usage of Potassco system [8] for debugging was facilitated. Debugging of programs containing choices, cardinality and weight constraints was enabled straightforwardly by (i) applying our presented translation from Section 3 above to the input, whereas we translate back debugging results from the meta-program, such that they refer back to the original rules with choices and cardinality constraints, whenever a rule occurring from our translation is identified as "buggy". Minor additional adaptions of the meta-program included support for extended integer arithmetic (e.g. to support use of $-$ in rule (25")). To support the debugging process including the translation of cardinality constraints, we extended the `SeaLion` Eclipse plugin [15] (an integrated development environment (IDE) for Answer Set Programming) by the `Ouroboros` plugin[8]. Our extended `Ouroboros` plugin can handle rules with cardinality constraints (and has not yet implemented the translation of weight constraints).

The plugin, including the new transformed and extended meta-program debugger based on [14], can be found at `https://mmdasp.svn.sourceforge.net/svnroot/mmdasp/sealion/trunk/org.mmdasp.sealion.ouroboros/`. To illustrate a simple debugging scenario consider the following example from constraint-based configuration. ASP programmer Lilian wants to assign each thing to exactly one cabinet with the constraint that there should not be more than two things in one cabinet. Her program, P_1, looks as follows:

```
thing(th1). thing(th2). thing(th3).
cabinet(c1). cabinet(c2). cabinet(c3).
1 {cabinetToThing(X, Y) : cabinet(X)} 1 :- thing(Y).
:- 2 {cabinetToThing(X, Y) : thing(Y)}, cabinet(X).
```

[8] Details on the `Ouroboros` plugin can be found in a companion system description [6].

Executing P_1, Lilian gets six answer sets. However, she wonders why there are only answer sets where in each of them one cabinet has exactly one thing. Normally, there should be answer sets where e.g. cabinet $c2$ has two things. So she decides to save the following interpretation – I_1 – as facts, where she replaced `cabinetToThing(c3, th2)` with `cabinetToThing(c2, th2)` to check why it is not an answer set:

```
thing(th1). thing(th2). thing(th3).
cabinet(c1). cabinet(c2). cabinet(c3).
cabinetToThing(c1, th3).
cabinetToThing(c2, th1).
cabinetToThing(c2, th2).
```

Now she creates a debug configuration and selects the program file as well as the adapted interpretation file and chooses the explanation type *Unsatisfiability*. After debugging the explanation says *Guessed rule: :- 2 {cabinetToThing(X, Y) : thing(Y)}, cabinet(X)*. Indeed, investigation of this rule reveals that the lower bound was set wrongly and should be 3 instead of 2.

In the background of this debugging process, the following happens: Let us denote the set of rules containing cardinality constraints or choices from a given program P as $P_{cc} = \{r_{cc_1}, \ldots, r_{cc_n}\}$. Moreover, let $tr(r_{cc_i})$ be the translation of a resp. rule r_{cc_i} according to Section 3. In our case, the two cardinality constraint rules of $P_{1,cc}$ are translated as follows:

```
cabinetTOthing(X, Y) :- thing(Y), cabinet(X), not -cabinetTOthing(X, Y).
-cabinetTOthing(X, Y) :- thing(Y), cabinet(X), not cabinetTOthing(X, Y).
:- not lowerUpperOK_1(Y), thing(Y).
lowerUpperOK_1(Y) :- not upper_1(Y), lower_1(Y), thing(Y).
lower_1(Y) :- cnt_1(Y, CounterC), CounterC >= 1, thing(Y).
upper_1(Y) :- CounterC > 1, cnt_1(Y, CounterC), thing(Y).
val_1_0(X, Y, Y) :- cabinet(X), cabinetTOthing(X, Y), thing(Y).
exists_1_0(Y) :- thing(Y), val_1_0(X, Y, Y).
smaller_1_0(X, Y, X1, Y1) :- val_1_0(X, Y, YBody), val_1_0(X1, Y1, YBody), X < X1.
smaller_1_0(X, Y, X, Y1) :- val_1_0(X, Y, YBody), val_1_0(X, Y1, YBody), Y < Y1.
cnt_1_0(X, Y, YBody, 1) :- val_1_0(X, Y, YBody).
cnt_1_0(X1, Y1, YBody, Ncounter1) :- val_1_0(X1, Y1, YBody),
smaller_1_0(X, Y, X1, Y1), cnt_1_0(X, Y, YBody, Ncounter), Ncounter1 = Ncounter+1.
cntPrime_1_0(YBody, Ncounter) :- cnt_1_0(X, Y, YBody, Ncounter),
not nmax_1_0(YBody, Ncounter).
cntPrime_1_0(Y, 0) :- thing(Y), not exists_1_0(Y).
nmax_1_0(YBody, Ncounter1) :- cnt_1_0(X, Y, YBody, Ncounter), Ncounter1 = Ncounter-1.
cnt_1(YBody, Ncounter0) :- cntPrime_1_0(YBody, Ncounter0).
:- lower_2(X), cabinet(X).
lower_2(X) :- cnt_2(X, CounterC), CounterC >= 2, cabinet(X).
val_2_0(X, Y, X) :- thing(Y), cabinetTOthing(X, Y), cabinet(X).
exists_2_0(X) :- cabinet(X), val_2_0(X, Y, X).
smaller_2_0(X, Y, X1, Y1) :- val_2_0(X, Y, XBody), val_2_0(X1, Y1, XBody), X < X1.
smaller_2_0(X, Y, X, Y1) :- val_2_0(X, Y, XBody), val_2_0(X, Y1, XBody), Y < Y1.
cnt_2_0(X, Y, XBody, 1) :- val_2_0(X, Y, XBody).
cnt_2_0(X1, Y1, XBody, Ncounter1) :- val_2_0(X1, Y1, XBody),
smaller_2_0(X, Y, X1, Y1), cnt_2_0(X, Y, XBody, Ncounter), Ncounter1 = Ncounter+1.
cntPrime_2_0(XBody, Ncounter) :- cnt_2_0(X, Y, XBody, Ncounter),
not nmax_2_0(XBody, Ncounter).
cntPrime_2_0(X, 0) :- cabinet(X), not exists_2_0(X).
nmax_2_0(XBody, Ncounter1) :- cnt_2_0(X, Y, XBody, Ncounter), Ncounter1 = Ncounter-1.
cnt_2(XBody, Ncounter0) :- cntPrime_2_0(XBody, Ncounter0).
```

Since the debugging approach requires a complete interpretation, we first have to extend the interpretation I given for debugging by the newly derivable auxiliary literals introduced in the translation. For this purpose a distinction must be made between

satisfied and unsatisfied (wrt. I) cardinality constraints: if r_{cc_i} involves a cardinality constraint with bounds, then $tr(r_{cc_i})$ contains an integrity constraint (either the rule is a constraint, then see rule (8) or (10) or otherwise see rule (9)); now, if the cardinality constraint is satisfied under the interpretation at hand, solving $tr(r_{cc_i}) \cup I$ yields one answer set that contains the additionally required literals. If a cardinality constraint P_i is not satisfied under the interpretation, then solving $tr(r_{cc_i}) \cup I$ yields no answer set at all. In this case, the original interpretation I is used. Thus, if a cardinality constraint is unsatisfied under I the debugger meta-program will state that rule (17) and rule (26) are unsatisfied.

As another case, some atoms of the interpretation can also form an unfounded loop. Let us consider Lilian's program just with the first cardinality constraint, denoted as P_2:

```
thing(th1). thing(th2).
cabinet(c1). cabinet(c2).
1 {cabinetTOthing(X, Y) : cabinet(X)} 1 :- thing(Y).
```

This program has four answer sets. However, Lilian expects to have some answer sets something like `cabinetTOthing(c3,th3)`, i.e. expects interpretation I_2 to be an desired answer set:

```
thing(th1). thing(th2).
cabinet(c1). cabinet(c2).
cabinetTOthing(c2, th1).
cabinetTOthing(c1, th2).
cabinetTOthing(c3, th3).
```

In this case, the debugging output explains that `cabinetTOthing(c3, th3).` forms an unfounded loop. In particular, there is neither a fact `thing(th3)` nor a fact `cabinet(c3).`

We emphasize that both these kinds of errors – wrong cardinalities, missing facts – occurred in practice in the encodings of our practical configuration settings.

6 Conclusions

We have presented a non-ground embedding of advanced ASP constructs (choices, cardinality and weight constraints) into normal logic programs and demonstrated how this embedding can be used to debug non-ground ASP programs using these constructs in the domain of configuration. While the non-ground embedding allowed us to extend an existing debugging approach for normal non-ground progams [15] relatively straightforwardly, our preliminary evaluation of the non-ground transformation shows that it cannot compete directly with non-ground embeddings as of yet. An investigation of further optimizations, or the possibility to use more efficient ground transformations directly in our debugger are on our agenda for future work.

Acknowledgements. The authors would like to thank Tomi Janhunen for providing advice on how to use the tools from [11] in our evaluation and Jörg Pührer for supporting and giving advice regarding the `Ouroboros` plugin. This work was funded by FFG FIT-IT within the scope of the project RECONCILE (grant number 825071).

References

1. Martin Brain and Marina De Vos. Debugging logic programs under the answer set semantics. In 3rd International Workshop on Answer Set Programming (ASP'05). CEUR Workshop Proceedings (2005) 141–152, 2005.
2. R. Caballero, Y. Garca-Ruiz, and F. Senz-Prez. A theoretical framework for the declarative debugging of datalog programs. In Klaus-Dieter Schewe and Bernhard Thalheim, editors, Semantics in Data and Knowledge Bases, volume 4925 of Lecture Notes in Computer Science, pages 143–159. Springer Berlin / Heidelberg, 2008.
3. Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Mirosław Truszczyński. The second answer set programming competition. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, Logic Programming and Nonmonotonic Reasoning, volume 5753 of Lecture Notes in Computer Science, pages 637–654. Springer Berlin Heidelberg, 2009.
4. Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. Theory Pract. Log. Program., 5(1-2):45–74, 2005.
5. G. Friedrich, A. Ryabokon, A.A. Falkner, A. Haselböck, G. Schenner, and H. Schreiner. (Re)configuration using Answer Set Programming. In IJCAI 2011 Workshop on Configuration, pages 17–25, 2011.
6. Melanie Frühstück, Jörg Pührer, and Gerhard Friedrich. Debugging answer-set programs with Ouroboros – extending the SeaLion plugin. In LPNMR, 2013. In this volume.
7. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo, 2010.
8. Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. AI Commun., 24(2):107–124, April 2011.
9. Martin Gebser, Jörg Pührer, Torsten Schaub, and Hans Tompits. A meta-programming technique for debugging answer-set programs. In Proceedings of the 23rd national conference on Artificial intelligence - Volume 1, AAAI'08, pages 448–453. AAAI Press, 2008.
10. Martin Gebser and Torsten Schaub. Answer set solving in practice, 2011. Available online at `http://www.cs.uni-potsdam.de/~torsten/ijcai11tutorial/asp.pdf`; visited on October 18th 2012.
11. Tomi Janhunen and Ilkka Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In Marcello Balduccini and Tran Cao Son, editors, Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning, volume 6565 of Lecture Notes in Computer Science, pages 111–130. Springer, 2011.
12. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. ACM Trans. Comput. Logic, 7(3):499–562, July 2006.
13. Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In LPNMR, pages 317–331, 1999.
14. Johannes Oetsch, Jörg Pührer, and Hans Tompits. Catching the ouroboros: On debugging non-ground answer-set programs. Theory Pract. Log. Program., 10(4-6):513–529, July 2010.
15. Johannes Oetsch, Jörg Pührer, and Hans Tompits. The sealion has landed: An IDE for answer-set programming. In 25th Workshop on Logic Programming (WLP), 2011.
16. Enrico Pontelli, Tran cao Son, and Omar Elkhatib. Justifications for logic programs under answer set semantics. Theory Pract. Log. Program., 9(1):1–56, January 2009.
17. Tommi Syrjänen. Cardinality constraint programs. In JELIA 2004, volume 3229 of Lecture Notes in Computer Science, pages 187–199. Springer, 2004.
18. Tommi Syrjänen. Debugging inconsistent answer set programs. Proc. NMR., 6:77–83, 2006.