Automated Aggregator — Rewriting Constraints Using Aggregates

Michael A. Dingess
Department of Computer Science, University of Kentucky, United States
madingess@yahoo.com

Mirosław Truszczyński
Department of Computer Science, University of Kentucky, United States
mirek@cs.uky.edu

Abstract

Answer set programming is a leading declarative constraint programming paradigm with wide use for complex knowledge-intensive applications. Modern answer set programming languages support many equivalent ways to model constraints and specifications in a program. However, so far answer set programming has failed to develop systematic methodologies for building representations that would lend well to automated processing. Here we present an automated rewriting system, the `Automated Aggregator`, that given a non-ground logic programs, produces a family of equivalent programs with variant representations having complementary performance in modern answer set programming solvers. We demonstrate this property of complementary performance through experimental analysis and propose the system's use in automated answer set programming solver selection tools.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Software and its engineering → Automatic programming; General and reference → General literature

Keywords and phrases rewriting, ASP, answer, set, programming, agreggation, aggregates, aggregate, equivalence, automated, family, encodings

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

Supplement Material **URL HERE**

Funding This research was supported by the National Science Foundation under grant 1707371
Michael A. Dingess: University of Kentucky
Mirosław Truszczyński: University of Kentucky

Acknowledgements We are grateful to Nicholas Hippen, Brian Hodges, Daniel Houston, Yuliya Lierler, Liu Liu, and Shelby Stocker for bountiful discussions on the topic

1 Introduction

Developers of answer set programming (ASP) solutions often face situations where a sentiment in a logic program could be expressed using multiple different representations, which are all structurally different yet semantically equivalent. Picking the right representation is crucial to designing these solutions because certain representations lend themselves to better performance in modern solvers than other representations. As such, techniques for selecting a particular representation are commonly chosen in ad hoc ways tailored to the needs of the particular application.
In 2011, Gebser et al. [**?**] presented a set of "rules-of-thumb" used by their expert team in manual tuning ASP solutions. These rules include suggestions on program rewritings that often result in substantial performance gains. They perform all their rewritings manually. Buddenhagen and Lierler [**?**] studied the impact of these rewritings on an ASP-based, natural language processor and reported orders of magnitude in gains in memory and time consumption as a result of some program transformations they executed manually.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:**??**
OpenAccess Series in Informatics
OASICS
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Although exploration has been conducted in the area of program optimization through transformations, these transformations provide performance improvements only in the general case and not uniformly for all logic programs candidate for a given transformation. Because of this, numerous combinations of these rewritings may be performed on a single logic program encoding, to produce a *family* of equivalent alternative encodings, which are experimentally tested to determine optimal encoding.

In this document we focus on a particular type of program rewriting, dubbed *Aggregate Equivalence*, and introduce a software for automating the rewriting of a logic program into various forms of this type of rewriting. We then discuss the effects of the Aggregate Equivalence rewriting on encodings submitted to previous years' ASP Competitions in order to produce families of equivalent alternative encodings. Additionally, we demonstrate a complementary performance between encodings in the family and study the results of testing these families of encodings on both the set of corresponding instances used in the encoding's ASP Competition and on instances which we generate ourselves, uniformly and randomly. Finally, we propose future work on the subject.

2 Aggregate Equivalence Rewriting

In this section we describe the *Aggregate Equivalence* rewriting, its input and output forms. We posit an equivalence between the input and output forms of the rewriting. Proofs are relegated to the appendix. Because all forms are equivalent, it is not necessary that the rewriting be one-way, in that one form is designated as the input while the remainder are designated as output. However, our implementation only functions as a one-way rewriting in its current state, so that is why we present the rewriting in this way. The input of the rewriting is a single rule.

2.1 Preliminaries

We consider a rules of the form: $head \leftarrow body$. The head may consist of a single *literal* or be empty, which constitutes a contradiction, making the rule a constraint. The body may contain one or more literals or be empty, which constitutes a fact. Literals are composed of an *atom* and a sign (positive or negative). A negative literal has the form $not\ a$ where a is an atom. A positive literal does not include the not indication. Atoms have the form $p(t_1, \ldots, t_k)$ where p is a predicate symbol of arity k and each t_i is a term. Terms are constants, variables, or expressions of the form $f(t_1, \ldots, t_k)$ where f is a function symbol of arity $k > 0$ and t_i is a term.

We also consider rules containing *aggregate atoms* having the form:

$$s_1 \prec_1 \alpha\{\mathbf{t}_1 : \mathbf{L}_1; \ldots; \mathbf{t}_n : \mathbf{L}_n\} \prec_2 s_2 \tag{1}$$

All $\mathbf{t}_i$ and $\mathbf{L}_i$ form *aggregate elements*, which are non-empty tuples of terms and literals, respectively. α is some operation on the **unique** term tuples $\mathbf{t}_i$ whose corresponding condition $\mathbf{L}_i$ holds. The result of applying α is compared by means of the *comparison predicates* $\prec_1$ and $\prec_2$ to the terms s_1 and s_2, respectively. These comparison predicates may be one of $\{<, \leq, =, \neq\}$. One or both of these comparisons can be omitted.[**?**]

In this paper, we focuse on *counting aggregates*, aggregate atoms for which $\alpha = \#count$.

$$s_1 \prec_1 \#count\{\mathbf{t}_1 : \mathbf{L}_1; \ldots; \mathbf{t}_n : \mathbf{L}_n\} \prec_2 s_2 \tag{2}$$

The *count* operation then counts the number of **unique** term tuples $\mathbf{t}_i$ whose corresponding condition $\mathbf{L}_i$ holds. The result the count function is compared by the *comparison predicates* $\prec_1$ and $\prec_2$ to the terms s_1 and s_2.

Generally, we consider an accordance with the *ASP-Core-2 Input Language Format*.[**?**]

2.2 Input Forms

The aggregate equivalence rewriting takes as input rules of the form:

$$H :- \bigwedge_{1 \leq i \leq b} F(X_i, \mathbf{Y}) \bigwedge_{1 \leq i \leq j \leq b} X_i \neq X_j, G. \tag{3}$$

where

- H is the head of a rule, possible empty (making the rule a constraint).
- $X_{1..b}$ are differing variables, all in the same position in F.
- $\mathbf{Y}$ is a comma-seperated list of variables, identical in variables and variable positions for all F in the rule.
- F is a predicate function of parity $1 + |\mathbf{Y}|$.
- G is the remaining body of the rule, possibly empty.

and the following hold true:

- $b \geq 2$.
- H, G, and $\mathbf{Y}$ have no occurences of variables $X_{1..b}$.
- The terms $\bigwedge_{1 \leq i \leq j \leq b} X_i \neq X_j$ may instead be a continuous chain of comparisons: $\bigwedge_{1 \leq i \leq b-1} X_i < X_{i+1}$ or $\bigwedge_{1 \leq i \leq b-1} X_i > X_{i+1}$.

Note that X need not be in the first position in F, so long as they are all in the corresponding position in F and the other variables in F (if any) are identical for all occurences of F in the rule. Additionally, some other forms logically equivalent to $\bigwedge_{1 \leq i \leq j \leq b} X_i \neq X_j$ are also acceptable.

2.3 Output Forms

The form of the output depends on the size of $\mathbf{Y}$.
When $|\mathbf{Y}| = 0$, we have the output form:

$$H :- b \leq \#count\{X : F(X)\}, G. \tag{4}$$

where

- Head H, integer b, predicate function F, and body literals G are as above.
- Aggregate element $X : F(X)$ follows the form $\mathbf{term} : \mathbf{literal}$ as defined above. The prior, X, is a tuple of one term, which in this case is a variable. The second, $F(X)$, is a literal, which in this case is a function.

When $|\mathbf{Y}| > 0$, we perform *projection* on F to project out the variable X. This gives us an output form consisting of two rules:

$$H :- b \leq \#count\{X : F(X, \mathbf{Y})\}, F'(\mathbf{Y}), G. \tag{5}$$

$$F'(\mathbf{Y}) :- F(X, \mathbf{Y}). \tag{6}$$

where

- Head H, integer b, predicate function F, variables $\mathbf{Y}$, and body literals G are as above.
- Aggregate element $X : F(X, \mathbf{Y})$ again follows the form $\mathbf{term} : \mathbf{literal}$ as above.
- F' is a new predicate function of arity equal to the size of $\mathbf{Y}$, i.e. equal to the arity of the F minus one.

Alternative Output Forms

A number of alternative, logically equivalent output forms are also available, each derived from the previous mentioned output form.

We observe that $b \leq F$ is inherently disjoint to $F < b$. Thus we can restate the original literal of the aggregate atom as a negation with the disjoint of the aggregate atom. We can then express the aggregate literal as follows:

$$not\ \#count\{X : F(X, \mathbf{Y})\} < b \tag{7}$$

Furthermore, we develop a final output form by making two observations of our input language. The first is that we use integer-only arithmetic. The second follows that, under integer arithmetic, the expression $not\ a < b$ for some integers a and b is equivalent to the conjuctive series:

$$\neg(a = -\infty) \wedge \neg(a = -\infty + 1) \wedge ... \wedge \neg(a = b - 2) \wedge \neg(a = b - 1)$$

Therefore, we can restate the aggregate literal in this alternative output form as a conjunctive series of aggregate literals having the form:

$$\begin{aligned} &not\ \#count\{X : F(X, \mathbf{Y})\} = 0, \\ &not\ \#count\{X : F(X, \mathbf{Y})\} = 1, \\ &..., \\ &not\ \#count\{X : F(X, \mathbf{Y})\} = b - 1 \end{aligned} \tag{8}$$

Note that, due to the precise semantics of logic programs, the equivalence of these two alternative logic forms relies on dependencies between the predicate F and predicates in H. More information on this is given in the appendix.

3 The Automated Aggregator System

We now present the *Automated Aggregator* (*AAgg*) software system for performing the Aggregate Equivalence rewriting. The software provides an automated way to detect rules within a given program following the input format (3), and subsequently rewrite those rules into an equivalent output format (4),(7), or (8).

3.1 Usage

The software relies on the `clingo` Python module provided by the Potassco suite.[**?**] The module is written in Python 2.7. As such, Python 2.7 is required to run the Automated Aggregator system. Installation information is provided in the software's README file. The Automated Aggregator is invoked as follows:

$$\begin{aligned} &python\ aagg/main.py\ [-h, --help]\ [-o, --output\ FILENAME]\ [--no-rewrite] \\ &\quad [--confirm-rewrite]\ [--use-anonymous-variable] \\ &\quad [--aggregate-form\ ID]\ [-d, --debug]\ [-r, --run-clingo] \\ &\quad [encoding_1\ encoding_2\ \ldots] \end{aligned} \tag{9}$$

The -h flag lists the help options. The encoding(s) are the filename(s) of the input encoding(s), and the output is the desired name for the output file. If no output filename is given, one is generated based on the first input filename given. When a candidate rule is discovered, the user is shown the

proposed rewrite and prompted for confirmation. If the –no-rewrite flag is given, no prompts are given and no rewriting is performed. If the –confirm-rewrite flag is given, no prompts are given and all possible rewriting is performed. The ID supplied to –aggregate-form argument informs the program which aggregate form to use when performing rewrites: the values 1, 2, and 3 correspond to aggregate forms (8), (9), and (12) *resp.*. The -d debug flag directs the application to operate with verbosity, printing details during the rewriting candidate discovery process and printing some statistics after the application's conclusion. The -r run clingo flag directs the application to run the resulting program through clingo after any rewritings are performed.

Finally, the –use-anonymous-variable flag indicates an additional modification of the output form to performed, to use the anonymous variable '_' in place of the variable X as in the output forms listed above among some other modifications to correct the semantics. We found the performance difference when using the anonymous variable to be negligable. The programs generated when using and when not using the anonymous variable are identical after grounding, so we do not pursue this route further.

By default all boolean flags are disabled and the aggregate-form ID is 1.. At least one input encoding filename must be specified.

3.2 Methodology

The methodology used for discovering rules candidate for rewriting is as follows. The given logic program(s) are parsed by the PyClingo module, generating an abstract syntax tree for each rule. Each such tree is passed to a *transformer* class for preprocessing. After preprocessing, some information is gathered from the program as a whole; specifically, predicate dependencies which determine when output forms (7) and (8) are appropriate. See the appendix for more details. Rules are then passed individually to an *equivalence transformer* class for processing. After processing, and if the requested rewriting is possible and confirmed by the user, the rewritten form of the rule is returned. Otherwise the original rule is returned. All returned rules, rewritten or not, are again aggregated and output to the desired output file location. Optionally, the resultant program is also run using clingo.

When a rule is passed to the equivalence transformer for processing, it first undergoes a process of exploration, which traverses the entirety of the rule's abstract syntax tree, recording comparisons literals between two variables as well as other pertinent information along the way. Theses comparisons are scrutinized to determine whether a subset of the comparisons, which we will denote C, follows the form given in (3) or some equivalent format as detailed in the Input Forms section. This also identifies those variables in the rule which constitute $X_{1..b}$ as in (3). The rule is then analyzed to determine that there are b occurences of some positive literal of function F over the argument X_i and $\mathbf{Y}$ where each $i \in 1..b$ exists at least once in the set of occurences and $\mathbf{Y}$ is the same for each occurence, $\mathbf{Y}$ having no variables from $X_{1..b}$. Let us call this set of b literals satisfying the above mentioned constraints, combined with the subset of comparisons following the form given in (3) or equivalent, as the rule's *counting literals*. Similarly, we denote the variables X_i as the rule's *counting variables*.

After gathering this candidate set of counting literals and counting variables, the equivalence transformer then verifies that the counting variables are not used within literals anywhere in the rule excluding literals within the set of counting literals. If this verification fails, or any of the constraints for the counting literals cannot be satisfied, or no such set of counting variables can be obtained, then the rule is not fit for rewriting. As a result, no rewriting is performed on the rule and the original rule is returned to the transformer.

However, if the verification succeeds, and the constraints for the counting literals can be satisfied, and such a set of counting variables can be obtained, then the rule is fit for rewriting under one final condition. If the requested output form is (4), then the rule is for rewriting and the rewritten rule is

returned to the transformer. If the requested output form is (7) or (8), then in order for the rule to be fit for rewriting, the additional condition must hold true that there is no dependency of the predicate F on predicates in the head of the rule (i.e. in H using the definition given previously). If no such dependency exists, the rule is fit for rewriting and the rewritten rule is returned. Otherwise the original rule is returned. See the appendix for more details.

Lastly, if a rule is found fit for rewriting, but $|\mathbf{Y}| > 0$, then an additional rule is introduced to project out the variable $X_{1..b}$ from the counting literals, and the projected form of the literal is attached to the rule before rewriting is completed. Both the rewritten rule and the additional projection rule are returned to the transformer class to be included in the final program.

Note that the user is first prompted for confirmation for each rewritten rule before the rule is added to the final program. If the user denies a rewriting, then the original rule is used.

3.3 Limitations

Here we discuss the limitations of the Automated Aggregator in its current form.

1. The rewriting is one-directional. In other words, the software will only rewrite rules from the form (3) into rules of the forms (4), (7), and (8). As it stands, it will not rewrite rules given in any of the forms (4), (7), or (8) into rules of any of other form.
2. In the cases when multiple rewritings are possible for a single rule, only one rewriting will be detected and performed. To illustrate, if the form given in (3) occurs twice in one rule over a disjoint set of variables and predicates, where both sets of counting literals considers the other set as part of G and all conditions hold, then only the set of counting literals with the highest number of counting variables is used for rewriting. If both sets contain the same number of counting variables, then one is chosen deterministically.
3. There are some cases in which the software will not recognized a valid rewriting when one exists. Theses cases mostly lay in the many obscure ways of representing a chain of comparisons equivalent to that in (3). However, there are no known cases in which an incorrect rewriting will be proposed when no valid rewriting is possible. More details are given in the software's README document.

4 Experimental Analysis

To download the Automated Aggregator system, visit **URL HERE**. Encodings with which the application was tested, their corresponding instances, and (Python) scripts for driving such tests, are included.

4.1 Automated Aggregator in Practice

The Automated Aggregator system was applied to gringo logic programs submitted to the 2009, 2014, and 2015 Answer Set Programming Competitions. Of the 58 encodings given to the application, 5 contained rules which were candidate for the rewriting described. (Golomb Ruler and Wire Routing from 2009; Steiner Tree from 2014; Steiner Tree and Graceful Graphs for 2015) For each of these encodings, a family of alternative encodings was produced with one encoding per rewrite form. For encodings with multiple rules candidate for rewriting, all rules were rewritten to the same rewrite form.

4.2 Experimental Results

Results were gathered by systematically grounding and solving each instance-encoding pair within families of encodings for each problem type. The grounding time and solving time of instance-encoding pairs were recorded and compared. The machine used for testing contained an Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz with 16GB RAM.

Interestingly, results indicate that no one type of the aggregate form performs uniformly better than any other. The use of the anonymous variable has no impact on performance—this is somewhat expected considering the grounding of the encoding using the anonymous varibale representation will be identical to the grounding of the same form without the anonymous variable. The greater-than aggregate form (1) has performance uniform with the less-than aggregate form (3). While the no aggregate performs uniformly better than any other, the various aggregate forms, discounting the pairings listed previously, indicate vastly different performance on given instances. That is, while one encoding in the family of produced encodings performs poorly on a given instance, another may perform swimmingly on the same instance, in some cases a speedup of two orders of magitude was observed.

5 Future Work

As detailed in the Limitations section, there are a number of areas of improvement for the Automated Aggregator software. Considering the complementary property of the rule forms input and output by the software, expanding the the detection to allow multidirectional rewriting would prove to be the most purposeful improvement. Secondly, expanding the software to detect and perform multiple Aggregate Equivalence rewritings on a single rule may prove useful, though the chance of encountering such a rule is slim when one considers the great size of such a rule. Lastly, expanding the tool to detect more obscure forms of representations is of lowest priority because we expect that answer set program developers will not use intentionally convoluted forms of representations.

Another area of future work is to explore automation for other types of equivalent rewritings for answer set programs. The PROJECTOR system developed by Hippen and Lierler automatically performs projection rewriting on logic programs.[**?**] A similar system, lpopt was developed by Wächter and Biegler in 2006.[**?**] It is worth noting that these two systems seek to improve the performance of a given logic program via rewritings, while the Automated Aggregator system seeks to develop a family of alternative encodings, each with complementary performance.

In addition to automating the process of encoding selection, automating the process of solver algorithm selection and parameter tuning is an ongoing area of research. Liu and Truszczynski studied the effectiveness of machine learning techniques for encoding selection within the Hamiltonian Cycle domain, with some success.[**?**] The most significant future step for this application would be integration with existing automated algorithm selection and tuning tools not only to predict the best solving algorithm and its parameters, but also to predict the best form of the rewriting (if any) for a given logic program, before passing it to the best solver for the job.

6 Appendix: Correctness of Rewritings

Let us consider the following program rule (H may be $\bot$):

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, \boldsymbol{Z}, \boldsymbol{Z'}), Q(\boldsymbol{X}), G(\boldsymbol{Z'}, \boldsymbol{Z''}), \tag{10}$$

where F is a predicate, $\boldsymbol{X}$ a tuple of variables $X_1, \ldots, X_b$ are variables, $Q(\boldsymbol{X})$ is a list of literals over variables $X_1, \ldots, X_b$, and $\boldsymbol{Z}, \boldsymbol{Z'}$ and $\boldsymbol{Z''}$ are three pairwise disjoint tuples of variables, and

disjoint with $\boldsymbol{X}$, and H contains none of X_i.

Let P be a program containing rule (**??**) and let P' be the program obtained from P by replacing that rule with the two rules

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, \boldsymbol{Z}, \boldsymbol{Z'}), Q(\boldsymbol{X}), G(\boldsymbol{Z'}, \boldsymbol{Z''}), F'(\boldsymbol{Z}) \tag{11}$$

$$F'(\boldsymbol{Z}) \leftarrow F(X, \boldsymbol{Z}, \boldsymbol{Z'}), \tag{12}$$

where F' is a predicate not occurring in P.

▶ **Theorem 1.** *The programs P and P' have the same answer sets modulo ground atoms of the form $F'(\boldsymbol{z})$.*

Proof. (Sketch) Consider a ground instance r of rule (**??**) and let a be the first atoms in the body of r (a ground instance of $F(X_1, \boldsymbol{Z}, \boldsymbol{Z'})$. In P' there are ground rules r' and r'' obtained from (**??**) using the same variable instantitation as that used to produce r. Clearly, r contributes to the reduct of $ground(P)$ if and only if r' and r'' contribute to ithe reduct of $ground(P')$. Moreover, if they do, r "fires" in the least model computation if and only if r' fires in the least model computation. It follows that an interpretation I of P is an answer set of P if and only if $I \cup J$ is an answer set of P', where J consists of all atoms $F'(\boldsymbol{z})$ such that $F(x, \boldsymbol{z}, \boldsymbol{z'}) \in I$, for some constant x and a tuple of constants $\boldsymbol{z'}$. □ ◀

Next, we recall the following theorem proved by Lierler [**?**].

▶ **Theorem 2.** *Let H be an atom (H may be $\bot$), G a list of literals, X a variable, $\boldsymbol{Z}$ a tuple of variables, each different from X and each with at least one occurrence in a literal in G, and F a predicate symbol of arity $1 + |\boldsymbol{Z}|$. If b is an integer, and $X_1, \ldots, X_b$ are variables without any occurrence in H and G, then the logic program rule*

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, \boldsymbol{Z}), \bigwedge_{1 \leq i < j \leq b} X_i \neq X_j, G \tag{13}$$

is strongly equivalent to the logic program rule

$$H \leftarrow b \leq \#count\{X : F(X, \boldsymbol{Z})\}, G, \tag{14}$$

where $\bigwedge$ is used to represent a sequence of expressions separated by commas.

The two theorems together provide a way to introduce aggregates. Let P be a program containing a rule of the form

$$H \leftarrow \bigwedge_{1 \leq i \leq b} F(X_i, \boldsymbol{Z}, \boldsymbol{Z'}), \bigwedge_{1 \leq i < j \leq b} X_i \neq X_j, G(\boldsymbol{Z'}, \boldsymbol{Z''}), \tag{15}$$

under the same assumptions about variable tuples $\boldsymbol{X}$, $\boldsymbol{Z}$, $\boldsymbol{Z'}$ and $\boldsymbol{Z''}$ as before. Clearly, this rule is a special case of a rule of the form (**??**). We replace this rule with the rules (**??**) and (**??**), adjusting the rule (**??**) to contain $\bigwedge_{1 \leq i < j \leq b} X_i \neq X_j$ for $Q(\boldsymbol{X})$. This step is only needed if $\boldsymbol{Z}$ is not empty. We then replace the rule (**??**) (or (**??**), if $\boldsymbol{Z}$ is empty) using Theorem **??**. The two theorems imply that resulting program has the same answer sets as P modulo ground atoms $F'(\boldsymbol{z})$. This argument yields a proof of the following result.

Once a program has a rule of the form (**??**), we can often modify it further by exploiting alternative encodings of the aggregate expressions. In particular, under some assumptions about the structure of the program, we can replace a rule (**??**) by

$$H \leftarrow not\ \#count\{X : F(X, \boldsymbol{Z})\} < b, G, \tag{16}$$

where we assume that the variable X is not a variable in $\boldsymbol{Z}$, and that all variables in $\boldsymbol{Z}$ appear in G.

We recall that a partition (P_b, P_t) of a program P is a *splitting* of P if no predicate appearing in the head of a rule from P_t appears in P_b [**?**, **?**]. A well-known result on splitting states that answer set of programs that have a splitting can be described in terms of answer sets of programs that form the splitting.

▶ **Theorem 3.** *Let P be a logic program and let (P_b, P_t) be a splitting of P. For every answer set I_b of P_b, every answer set of the program $P_t \cup I_b$ is an answer set of P. Conversely, for every answer set I of P, there is an answer set I_b of P_b such that I is an answer set of $P_t \cup I_b$.*

This result implies that in programs that have a splitting, a rule in P_t containing in its body an aggregate expression involving only predicates appearing in P_b can be replaced by a rule in which this aggregate expression is replaced by any of its (classically) equivalent forms. We formally state this result for the case of rules of the form (**??**).

▶ **Theorem 4.** *Let P be a logic program and let (P_b, P_t) be a splitting of P. If P_t contains a rule of the form (**??**) and F appears in P_b, then P and the program P' obtained from P by replacing the rule (**??**) by the rule (**??**) have the same answer sets.*

Proof. (Sketch) Let P'_t be the program obtained from P_t by replacing the rule (**??**) by the rule (**??**). It is clear that (P_b, P'_t) is a splitting of P'. Let I be an answer set of P. By Theorem **??**, there is an answer set I_b of P_b such that I is an answer set of $P_t \cup I_b$. In particular, I is an answer set of the program $I_b \cup ground(P_t)$. Let Q be the program obtained by simplifying the bodies of the rules in $ground(P_t)$ as follows. If a conjunct c in the body of a rule in $ground(P_t)$ involves only atoms from the Herbrand base $HB(P_b)$ of P_b, we remove c if $I_b \models c$, and we remove the rule, if $I_b \not\models c$. Because atoms from $HB(P_b)$ do not appear in the heads of the rules in $ground(P_t)$, I is an answer set of $Q \cup I_b$.

We denote by Q' the program obtained by the same simplification process from $ground(P'_t)$. From the definition of P'_t it follows that $Q' = Q$ (indeed, the only difference between P_t and P'_t is in the bodies of some rules, in which an aggregate built entirely from the atoms in $HB(P_b)$ is replaced by a classically equivalent one; thus, both expressions evaluate in the same way under I_b and the contribution of the corresponding rules to Q and Q' in each case is the same). Consequently, I is an answer set of $Q' \cup I_b$. Because atoms from $HB(P_b)$ do not appear in the heads of the rules in $ground(P'_t)$, I is an answer set of $ground(P'_t) \cup I_b$ and, because (P_b, P'_t) is a splitting of P', also an answer set of P'. A similar argument shows that answer sets of P' are also answer sets of P. □ ◀

It is clear that the same argument applies to other similar rewritings, for instance, to the one that replaces the rule (**??**) by the rule

$$H \leftarrow \bigwedge_{0 \leq i < b-1} not\ i = \#count\{X : F(X, \boldsymbol{Z})\}, G, \tag{17}$$

where, as before, we assume that the variable X is not a variable in $\boldsymbol{Z}$, and that all variables in $\boldsymbol{Z}$ appear in G.

References

1 Buddenhagen, M., Lierler, Y.: Performance tuning in answer set programming. In: Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR) (2015)

2 Cilimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; Schlab, T. 2003. ASP-Core-2 Input Language Format. Arxive.

3 Ferraris, P.; Lee, J.; Lifschitz, V.; Palla, R. 2009. Symmetric Splitting in the General Theory of Stable Models. Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI-2009, pp. 797-803.

4 Eiter, T.; Fink, M.; Woltran, S. 2005. Semantical Characterizations and Complexity of Equivalences in Stable Logic Programming. Technical Report INFSYS RR-1843-05-01, Institut für Informationssysteme, Technische Universität Wien, Austria. Accepted for publication in ACM Transactions on Computational Logic.

5 Eiter, T.; Fink, M.; Tompits, H.; Traxler, P.; Woltran, S. 2006. Replacements in Non-Ground Answer-Set Programming. Institut für Informationssysteme, Technische Universität Wien, Austria.

6 Gebser, M.; Kaminski, R.; Kaufmann, B.; Schaub, T. 2011. Challenges in answer set solving. In: Balduccini, M., Son, T. (eds.) Logic Programming, Knowledge Representation, and Non-monotonic Reasoning: Essays in Honor of Michael Gelfond, vol. 6565, pp. 74–90. Springer

7 Gebser, M.; Kaminski, R.; Kaufmann, B.; Lindauer, M.; Ostrowski, M.; Romero, J.; Schaub, T.; Thiele, S.; 2015. Potassco User Guide.

8 Gebser, M.; Kaminski, R.; Kaufmann, B.; Schaub, T.; (2017). Multi-shot ASP solving with clingo. CoRR, abs/1705.09811.

9 Hippen, M.; Lierler, Y. 2018. Automatic Program Rewriting in Non-Ground Answer Set Programs.

10 Lee, J.; Lifschitz, V.; Palla, R. 2008. A reductive semantics for counting and choice in answer set programming. In Proceedings of the AAAI Conference on Artificial Intelligence (AAAI):472–479.

11 Lierler, Y. 2019. Strong Equivalence and Program's Structure in Arguing Essential Equivalence between First-Order Logic Programs.

12 Lifschitz, V.; Pearce, D.; and Valverde, A. 2001. Strongly equivalent logic programs. ACM ToCL 2(4):526-541.

13 Lifshitz, V.; Hudsen, T. 1994. Splitting a Logic Program. Proceedings of the 11th International Conference on Logic Programming, ICLP-1994, pp. 23-37.

14 Liu, L.; Truszczynski, M. 2019. Encoding Selection for Solving Hamiltonian Cycle Problems with ASP.

15 Wächter, A.; Biegler, L. T. 2006. On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming, Mathematical Programming 106(1), pp. 25-57.