# Strong Equivalence and Program's Structure in Arguing Essential Equivalence between First-Order Logic Programs

Yuliya Lierler

**Abstract.** Answer set programming is a prominent declarative programming paradigm used in formulating combinatorial search problems and implementing distinct knowledge representation formalisms. It is common that several related and yet substantially different answer set programs exist for a given problem. Sometimes these encodings may display significantly different performance. Uncovering *precise formal* links between these programs is often important and yet far from trivial. This paper claims the correctness of a number of interesting program rewritings. Notably, they assume *programs with variables* and such important language features as choice, disjunction, and aggregates.

## Introduction

Answer set programming (ASP) is a prominent knowledge representation paradigm with roots in logic programming [2]. It is frequently used for addressing combinatorial search problems. It has also been used to provide implementations and/or translational semantics to other knowledge representation formalisms such as action languages including language $\mathscr{AL}$ [13, Section 8]. In ASP, when a software engineer tackles a problem domain it is a common practice to first develop *a/some* solution/encoding to a problem and then rewrite this solution/encoding iteratively using, for example, a projection technique to gain a better performing encoding [3]. These common processes bring a question to light: What are the formal means to argue the correctness of renewed formulations of the original encoding to a problem or, in other words, to argue that these distinct formulations are essentially the same — in a sense that they capture solutions to the same problem.

It has been long recognized that studying various notions of equivalence between programs under the answer set semantics is of crucial importance. Researchers proposed and studied strong equivalence [18,19], uniform equivalence [4], relativized strong and uniform equivalences [23]. Also, equivalences relative to specified signatures [6,15] were considered. In most of the cases the programs considered for studying the distinct

forms of equivalence are propositional. Works [5,7,19,22,15] are exceptions. These authors consider programs with variables (or, first-order programs). Yet, it is first-order programs that ASP knowledge engineers develop. Thus, theories on equivalence between programs with variables are especially important as they can lead to more direct arguments about properties of programs used in practice. On the one hand, this work can be seen as a continuation of work by Eiter et al. [5], were we consider common program rewritings using more complex dialect of logic programs. On the other hand, it grounds the concept of program's synonymity studied by Pearce and Valverde [22] in a number of practical examples. Namely, we illustrate how formal results on strong equivalence developed earlier and in this work help us to construct precise claims about programs in practice.

In this paper, we systematically study some common rewritings on first-order programs utilized by ASP practitioners. As a running and motivating example that grounds general theoretical presentation of this work into specific context, we consider two formalizations of a planning module given in [13, Section 9]. Namely,

1. a *Plan-choice* formalization that utilizes choice rules and aggregate expressions,
2. a *Plan-disj* formalization that utilizes disjunctive rules.

Such a planning module is meant to be augmented with an ASP representation of a dynamic system description expressed in action language $\mathscr{AL}$. In [13], Gelfond and Kahl formally state in Proposition 9.1.1 that the answer sets of program *Plan-disj* augmented with a given system description encode all the "histories/plans" of a specified length in the transition system captured by the system description. Although both *Plan-choice* and *Plan-disj* programs *intuitively* encode the same knowledge the exact connection between them is not immediate. In fact, these programs (i) do not share the same signature; (ii) use distinct syntactic constructs such as choice, disjunction, aggregates in the specification of a problem. Here, we establish a one-to-one correspondence between the answer sets of these programs on their properties. Thus, the aforementioned formal claim about *Plan-disj* translates into the same claim for *Plan-choice*. It is due to remark that although in [13], Gelfond and Kahl use the word "module" when formalizing a planning domain they utilize this term only informally to refer to a collection of rules responsible for formalizing "planning".

In this paper we use a dialect of ASP language called RASPL-1 [17]. Notably, this language combines choice, aggregate, and disjunction constructs. Its semantics is given in terms of the SM operator, which exemplifies the approach to the semantics of first-order programs that bypasses grounding. Relying on SM-based semantics allows us to refer to earlier work that study the formal properties of first-order programs [9,10] using this operator. We state a sequence of formal results on programs rewritings and/or programs' properties. Some discussed rewritings are well known and frequently used in practice. Often, their correctness is an immediate consequence of well known properties about logic programs (e.g., relation between intuitionistically provable first-order formulas and strongly equivalent programs viewed as such formulas). Other discussed rewritings are far less straightforward and require elaborations on previous theoretical findings about the operator SM. It is well known that propositional head-cycle-free disjunctive programs [1] can be rewritten to nondisjunctive programs by means of simple syntactic transformation. Here we not only generalize this result to the case of first-order programs, but also illustrate that at times we can remove disjunction from parts

of a program even though the program is not head-cycle-free. This result is relevant to local shifting and component-wise shifting discussed in [5] and [16] respectively. We also generalize so called Completion Lemma and Lemma on Explicit Definitions stated in [8,11] for the case of propositional theories and propositional logic programs. These generalizations are applicable to first-order programs.

**Summary.** We view this paper as an important step towards bringing theories about program's equivalence to providing practical solutions in the realm of ASP as it is used by knowledge engineers. A portfolio of formal results on program rewritings stated in this paper can serve as a solid theoretical basis for

- a software system that may automatically produce new variants of logic programs (some of these encodings will often exhibit better performance) by utilizing studied rewritings;
- a proof technique for arguing the correctness of a logic program. This proof technique assumes the existence of a "gold standard" logic program formalizing a problem at hand, in a sense that this gold standard is trusted to produce correct results. A proper portfolio of known program rewritings and their properties equips ASP practitioners with powerful tools to argue that another encoding is essentially the same to the gold standard.

**Paper Outline.** We start this paper by presenting the *Plan-choice* and *Plan-disj* programs. We then introduce a logic program language called RASPL-1 [17]. The semantics of this language is given in terms of the SM operator. We then proceed to the statement of a sequence of formal results on program's rewritings.

## Running Example and Observations

This section presents two ASP formalizations of a domain independent *planning module* given in [13, Section 9]. Such planning module is meant to be augmented with a logic program encoding a system description expressed in action language $\mathscr{AL}$ that represents a domain of interest (in Section 8 of their book [13], Gelfond and Kahl present a sample Blocks World domain representation). Two formalizations of a planning module are stated here almost verbatim. Predicate names $o$ and *sthHpd* intuitively stand for *occurs* and *something_happend*, respectively. We eliminate classical negation symbol by (i) utilizing auxiliary predicates non_o in place of $\neg o$; and (ii) introducing rule $\leftarrow o(A,I), non\_o(A,I)$. This is a standard practice and ASP systems perform the same procedure when processing classical negation symbol $\neg$ occurring in programs (in other words, symbol $\neg$ is treated as a syntactic sugar).

The first formalization called *Plan-choice* follows:

$$success \leftarrow goal(I),\ step(I).$$

$$\leftarrow not\ success.$$

$$\leftarrow o(A,I), non\_o(A,I) \tag{1}$$

$$non\_o(A,I) \leftarrow action(A),\ step(I),\ not\ o(A,I) \tag{2}$$

$$\{o(A,I)\} \leftarrow action(A),\ SG(I) \tag{3}$$

$$\leftarrow 2 \leq \#count\{A : o(A,I)\},\ SG(I). \tag{4}$$

$$\leftarrow not\ 1 \leq \#count\{A : o(A,I)\},\ SG(I) \tag{5}$$

where

$SG(I)$ abbreviates $step(I)$, $not\ goal(I)$, $I \neq n$,

where $n$ is some integer specifying a limit on a length of an allowed plan. One more remark is in order. In [13], Gelfond and Kahl list only a single rule

$$1\{o(A,I) : action(A)\}1 \leftarrow SG(I)$$

in place of rules (3-5). Note that this single rule is an abbreviation for rules (3-5) [12].

The second formalization that we call a *Plan-disj* encoding is obtained from *Plan-choice* by replacing rules (3-5) with the following:

$$o(A,I) \mid non\_o(A,I) \leftarrow action(A),\ SG(I) \tag{6}$$

$$\leftarrow o(A,I),\ o(A',I),\ action(A),\ action(A'),\ A \neq A' \tag{7}$$

$$sthHpd(I) \leftarrow o(A,I) \tag{8}$$

$$\leftarrow not\ sthHpd(I),\ SG(I). \tag{9}$$

It is important to note several facts about the considered planning module encodings. These planning modules are meant to be used with logic programs that capture (i) a domain of interest originally stated as a system description in the action language $\mathscr{AL}$; (ii) a specification of an initial configuration; (iii) a specification of a goal configuration. The process of encoding (i-iii) as a logic program, which we call a *Plan-instance* encoding, follows a strict procedure. As a consequence, some important properties hold about any *Plan-instance*. To state these it is convenient to recall a notion of a simple rule and define a "terminal" predicate.

A *signature* is a set of function and predicate symbols/constants. A function symbol of arity 0 is an *object constant*. A *term* is an object constant, an object variable, or an expression of the form $f(t_1,\ldots,t_m)$, where $f$ is a function symbol of arity $m$ and each $t_i$ is a term. An *atom* is an expression of the form $p(t_1,\ldots,t_n)$ or $t_1 = t_2$, where $p$ is an $n$-ary predicate symbol and each $t_i$ is a term. A *simple body* has the form $a_1,\ldots,a_m,\ not\ a_{m+1},\ldots,\ not\ a_n$ where $a_i$ is an atom and $n$ is possible 0. Expression $a_1,\ldots,a_m$ forms positive part of a body. A *simple* rule has a form $h_1 \mid \cdots \mid h_k \leftarrow Body$ or $\{h_1\} \leftarrow Body$ where $h_i$ is an atom and *Body* is a simple body. We now state a recursive definition of a terminal predicate with respect to a program. Let $i$ be a nonnegative integer. A predicate that occurs only in rules whose body is empty is called *0-terminal*. We call a predicate $i+1$-*terminal* when it occurs only in the heads of simple rules (left hand side of an arrow), furthermore (i) in these rules all predicates occurring in their positive parts of the bodies must be at most *i-terminal* and (ii) at least one of these rules is such that some predicate occurring in its positive part of the body is *i-terminal* . We call any $x$-terminal predicate *terminal*. For example, in program

$$block(b0).\ block(b1).$$
$$loc(X) \leftarrow block(X). \qquad loc(table).$$

*block* is a 0-terminal predicate, *loc* is a 1-terminal predicate; and both predicates are terminal.

We are now ready to state important *Facts* about any possible *Plan-instance* and, consequently, about the considered planning modules

1. Predicate $o$ never occurs in the heads of rules in *Plan-instance*.
2. Predicates $action$ and $step$ are terminal in *Plan-instance* as well as in *Plan-instance* augmented by either *Plan-choice* or *Plan-disj*.
3. By Facts 1 and 2, predicate $o$ is terminal in *Plan-instance* augmented by either *Plan-choice* or *Plan-disj*.
4. Predicate *sthHpd* never occurs in the heads of the rules in *Plan-instance*.

In the remainder of the paper we will ground considered theoretical results by illustrating how they formally support the following *Observations*:

1. In the presence of rule (2) it is safe to add a rule

$$non\_o(A,I) \leftarrow not\ o(A,I),\ action(A),\ SG(I) \tag{10}$$

   into an arbitrary program. By "safe to add/replace" we understand that the resulting program has the same answer sets as the original one.
2. It is safe to replace rule (4) with rule

$$\leftarrow o(A,I),\ o(A',I),\ SG(I),\ A \neq A' \tag{11}$$

   within an arbitrary program.
3. In the presence of rules (1) and (2), it is safe to replace rule (3) with rule

$$o(A,I) \leftarrow not\ non\_o(A,I),\ action(A),\ SG(I) \tag{12}$$

   within an arbitrary program.
4. Given the syntactic features of the *Plan-choice* encoding and any *Plan-instance* encoding, it is safe to replace rule (3) with rule (6). The argument utilizes *Observations* 1 and 3. Fact 4 forms an essential syntactic feature.
5. Given the syntactic features of the *Plan-choice* encoding and any *Plan-instance* encoding, it is safe to replace rule (4) with rule (7). The argument utilizes *Observation* 2, i.e., it is safe to replace rule (4) with rule (11). An essential syntactic feature relies on Fact 1, and the facts that (i) rule (3) is the only one in *Plan-choice*, where predicate $o$ occurs in the head; and (ii) rule (7) differs from (11) only in atoms that are part of the body of (3).
6. By Fact 4 and the fact that *sthHpd* does not occur in any other rule but (9) in *Plan-disj*, the answer sets of the program obtained by replacing rule (5) with rules (8) and (9) are in one-to-one correspondence with the answer sets of the program *Plan-disj* extended with *Plan-instance*.

**Essential Equivalence Between Two Planning Modules:** These *Observations* are sufficient to claim that the answer sets of the *Plan-choice* and *Plan-disj* programs (extended with any *Plan-instance*) are in one-to-one correspondence. We can capture the simple relation between the answer sets of these programs by observing that dropping the atoms whose predicate symbol is *sthHpd* from an answer set of the *Plan-disj* program results in an answer set of the *Plan-choice* program.

## Preliminaries: RASPL-1 Logic Programs, Operator SM, Strong Equivalence

We now review a logic programming language RASPL-1 [17]. This language is sufficient to capture choice, aggregate, and disjunction constructs (as used in *Plan-choice*

and *Plan-disj*). There are distinct and not entirely compatible semantics for aggregate expressions in the literature. We refer the interested reader to the discussion by Lee et al. in [17] on the roots of semantics of aggregates considered in RASPL-1.

An *aggregate expression* is an expression of the form

$$b \leq \#count\{\mathbf{x} : L_1, \ldots, L_k\} \tag{13}$$

($k \geq 1$), where $b$ is a positive integer (*bound*), $\mathbf{x}$ is a list of variables (possibly empty), and each $L_i$ is an atom possibly preceded by *not*. This expression states that there are at least $b$ values of $\mathbf{x}$ such that conditions $L_1, \ldots, L_k$ hold.

A *body* is an expression of the form

$$e_1, \ldots, e_m, not\ e_{m+1}, \ldots, not\ e_n \tag{14}$$

($n \geq m \geq 0$) where each $e_i$ is an aggregate expression or an atom. A rule is an expression of either of the forms

$$a_1 \mid \cdots \mid a_l \leftarrow Body \tag{15}$$

$$\{a_1\} \leftarrow Body \tag{16}$$

($l \geq 0$) where each $a_i$ is an atom, and *Body* is the body in the form (14). When $l = 0$, we identify the head of (15) with symbol $\bot$ and call such a rule a *denial*. When $l = 1$, we call rule (15) a *defining* rule. We call rule (16) a *choice* rule. A *(logic) program* is a set of *rules*. An atom of the form *not* $t_1 = t_2$ is abbreviated by $t_1 \neq t_2$.

It is easy to see that rules in the *Plan-choice* and *Plan-disj* encodings are in the RASPL-1 language.

**Operator SM**

Typically, the semantics of logic programs with variables is given by stating that these rules are an abbreviation for a possibly infinite set of propositional rules. Then the semantics of propositional programs is considered. The SM operator introduced by Ferraris et al. in [9] gives a definition for the semantics of first-order programs bypassing grounding. It is an operator that takes a first-order sentence $F$ and a tuple $\mathbf{p}$ of predicate symbols and produces the second order sentence that we denote by $\mathrm{SM}_{\mathbf{p}}[F]$.

We now review the operator SM. The symbols $\bot, \wedge, \vee, \rightarrow$, $\forall$, and $\exists$ are viewed as primitives. The formulas $\neg F$ and $\top$ are abbreviations for $F \rightarrow \bot$ and $\bot \rightarrow \bot$, respectively. If $p$ and $q$ are predicate symbols of arity $n$ then $p \leq q$ is an abbreviation for the formula $\forall \mathbf{x}(p(\mathbf{x}) \rightarrow q(\mathbf{x}))$, where $\mathbf{x}$ is a tuple of variables of length $n$. If $\mathbf{p}$ and $\mathbf{q}$ are tuples $p_1, \ldots, p_n$ and $q_1, \ldots, q_n$ of predicate symbols then $\mathbf{p} \leq \mathbf{q}$ is an abbreviation for the conjunction $(p_1 \leq q_1) \wedge \cdots \wedge (p_n \leq q_n)$, and $\mathbf{p} < \mathbf{q}$ is an abbreviation for $(\mathbf{p} \leq \mathbf{q}) \wedge \neg(\mathbf{q} \leq \mathbf{p})$. We apply the same notation to tuples of predicate variables in second-order logic formulas. If $\mathbf{p}$ is a tuple of predicate symbols $p_1, \ldots, p_n$ (not including equality), and $F$ is a first-order sentence then $\mathrm{SM}_{\mathbf{p}}[F]$ denotes the second-order sentence

$$F \wedge \neg \exists \mathbf{u}(\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u}),$$

where $\mathbf{u}$ is a tuple of distinct predicate variables $u_1, \ldots, u_n$, and $F^*(\mathbf{u})$ is defined recursively:

– $p_i(\mathbf{t})^*$ is $u_i(\mathbf{t})$ for any tuple $\mathbf{t}$ of terms;
– $F^*$ is $F$ for any atomic formula $F$ that does not contain members of $\mathbf{p}$;[1]
– $(F \wedge G)^*$ is $F^* \wedge G^*$;
– $(F \vee G)^*$ is $F^* \vee G^*$;
– $(F \to G)^*$ is $(F^* \to G^*) \wedge (F \to G)$;
– $(\forall x F)^*$ is $\forall x F^*$;
– $(\exists x F)^*$ is $\exists x F^*$.

Note that if $\mathbf{p}$ is the empty tuple then $\mathrm{SM}_{\mathbf{p}}[F]$ is equivalent to $F$. For intuitions regarding the definition of the SM operator we direct the reader to [9, Sections 2.3, 2.4].

By $\sigma(F)$ we denote the set of all function and predicate constants occurring in first-order formula $F$ (not including equality). We will call this the *signature of* $F$. An interpretation $I$ over $\sigma(F)$ is a $\mathbf{p}$-*stable model* of $F$ if it satisfies $\mathrm{SM}_{\mathbf{p}}[F]$, where $\mathbf{p}$ is a tuple of predicates from $\sigma(F)$. We note that a $\mathbf{p}$-stable model of $F$ is also a model of $F$.

By $\pi(F)$ we denote the set of all predicate constants (excluding equality) occurring in a formula $F$. Let $F$ be a first-order sentence that contains at least one object constant. We call an Herbrand interpretation of $\sigma(F)$ that is a $\pi(F)$-stable model of $F$ *an answer set*.[2] Theorem 1 from [9] illustrates in which sense this definition can be seen as a generalization of a classical definition of an answer set (via grounding and reduct) for typical logic programs whose syntax is more restricted than syntax of programs considered here.

**Semantics of Logic Programs**

From this point on, we view logic program rules as alternative notation for particular types of first-order sentences. We now define a procedure that turns every aggregate, every rule, and every program into a formula of first-order logic, called its *FOL representation*. First, we identify the logical connectives $\wedge$, $\vee$, and $\neg$ with their counterparts used in logic programs, namely, the comma, the disjunction symbol $|$, and connective *not*. This allows us to treat $L_1, \ldots, L_k$ in (13) as a conjunction of literals. The FOL representation of an aggregate expressions of the form $b \leq \#count\{\mathbf{x} : F(\mathbf{x})\}$ follows

$$\exists \mathbf{x^1} \cdots \mathbf{x^b} \Big[ \bigwedge_{1 \leq i \leq b} F(\mathbf{x^i}) \wedge \bigwedge_{1 \leq i < j \leq b} \neg(x^i = x^j) \Big], \tag{17}$$

where $\mathbf{x^1} \cdots \mathbf{x^b}$ are lists of new variables of the same length as $\mathbf{x}$. The FOL representations of logic rules of the form (15) and (16) are formulas

$$\widetilde{\forall}(Body \to a_1 \vee \cdots \vee a_l) \quad \text{and} \quad \widetilde{\forall}(\neg\neg a_1 \wedge Body \to a_1),$$

where each aggregate expression in *Body* is replaced by its FOL representation. Symbol $\widetilde{\forall}$ denotes universal closure.

[1] This includes equality statements and the formula $\bot$.

[2] An Herbrand interpretation of a signature $\sigma$ (containing at least one object constant) is such that its universe is the set of all ground terms of $\sigma$, and every ground term represents itself. An Herbrand interpretation can be identified with the set of ground atoms (not containing equality) to which it assigns the value true.

For example, expression $SG(I)$ stands for formula $step(I) \wedge \neg goal(I) \wedge \neg I = n$ and rules (3) and (5) in the *Plan-choice* encoding have the FOL representation:

$$\widetilde{\forall}\big(\neg\neg o(A,I) \wedge SG(I) \wedge action(A) \rightarrow o(A,I)\big) \tag{18}$$

$$\forall I\big(\neg\exists A[o(A,I)] \wedge SG(I) \rightarrow \bot\big) \tag{19}$$

The FOL representation of rule (4) is the universal closure of the following implication

$$(\exists AA'\big(o(A,I) \wedge o(A',I) \wedge \neg A = A'\big) \wedge SG(I)) \rightarrow \bot.$$

We define a concept of an answer set for logic programs that contain at least one object constant. This is inessential restriction as typical logic programs without object constants are in a sense trivial. In such programs, whose semantics is given via grounding, rules with variables are eliminated during grounding. Let $\Pi$ be a logic program with at least one object constant. (In the sequel we often omit expression "with at least one object constant".) By $\widehat{\Pi}$ we denote its FOL representation. (Similarly, for a body *Body* or a rule $R$, by $\widehat{Body}$ or $\widehat{R}$ we denote their FOL representations.) An *answer set* of $\Pi$ is an answer set of its FOL representation $\widehat{\Pi}$. In other words, an *answer set* of $\Pi$ is an Herbrand interpretation of $\widehat{\Pi}$ that is a $\pi(\widehat{\Pi})$-stable model of $\widehat{\Pi}$, i.e., a model of

$$\mathrm{SM}_{\pi(\widehat{\Pi})}[\widehat{\Pi}]. \tag{20}$$

Sometimes, it is convenient to identify a logic program $\Pi$ with its semantic counterpart (20) so that formal results stated in terms of SM operator immediately translate into the results for logic programs.

**Review: Strong Equivalence**

We restate the definition of strong equivalence given in [9] and recall some of its properties. First-order formulas $F$ and $G$ are *strongly equivalent* if for any formula $H$, any occurrence of $F$ in $H$, and any tuple $\mathbf{p}$ of distinct predicate constants, $\mathrm{SM}_{\mathbf{p}}[H]$ is equivalent to $\mathrm{SM}_{\mathbf{p}}[H']$, where $H'$ is obtained from $H$ by replacing $F$ by $G$. Trivially, any strongly equivalent formulas are such that their stable models coincide (relative to any tuple of predicate constants). In [19], Ferraris et al. show that first-order formulas $F$ and $G$ are strongly equivalent if they are equivalent in **SQHT**$^{=}$ logic — an intermediate logic between classical and intuitionistic logics. We recall that every formula provable in the natural deduction system without the law of the excluded middle ($F \vee \neg F$) is a theorem in intuitionistic logic. Also, every formula provable using natural deduction, where the axiom of the law of the excluded middle ($F \vee \neg F$) is replaced by the weak law of the excluded middle ($\neg F \vee \neg\neg F$), is a theorem of **SQHT**$^{=}$.

The definition of strong equivalence between first-order formulas paves the way to a definition of strong equivalence for logic programs. A logic program $\Pi_1$ is *strongly equivalent* to logic program $\Pi_2$ when for any program $\Pi$,

$$\mathrm{SM}_{\pi(\widehat{\Pi \cup \Pi_1})}[\widehat{\Pi \cup \Pi_1}] \text{ is equivalent to } \mathrm{SM}_{\pi(\widehat{\Pi \cup \Pi_2})}[\widehat{\Pi \cup \Pi_2}].$$

It immediately follows that logic programs $\Pi_1$ and $\Pi_2$ are *strongly equivalent* if first-order formulas $\widehat{\Pi_1}$ and $\widehat{\Pi_2}$ are equivalent in logic of **SQHT**$^{=}$.

We now review an important result about properties of denials.

**Theorem 1 (Theorem 3 [9]).** *For any first-order formulas $F$ and $G$ and arbitrary tuple* $\mathbf{p}$ *of predicate constants, $SM_{\mathbf{p}}[F \wedge \neg G]$ is equivalent to $SM_{\mathbf{p}}[F] \wedge \neg G$.*

As a consequence, $\mathbf{p}$-stable models of $F \wedge \neg G$ can be characterized as the $\mathbf{p}$-stable models of $F$ that satisfy first-order logic formula $\neg G$. Consider any denial $\leftarrow Body$. Its FOL representation has the form $\widetilde{\forall}(Body \rightarrow \bot)$ that is intuitionistically equivalent to formula $\neg\widetilde{\exists}Body$. Thus, Theorem 1 tells us that given any denial of a program it is safe to compute answer sets of a program without this denial and a posteriori verify that the FOL representation of a denial is satisfied.

**Corollary 1.** *Two denials are strongly equivalent if their FOL representations are classically equivalent.*

This corollary is also an immediate consequence of the Replacement Theorem for intuitionistic logic [21, Section 13.1] stated below.

**Replacement Theorem.** *If $F$ is a first-order formula containing a subformula $G$ and $F'$ is the result of replacing that subformula by $G'$ then $\widetilde{\forall}(G \leftrightarrow G')$ intuitionistically implies $F \leftrightarrow F'$.*

## Rewritings

### Rewritings via Pure Strong Equivalence

Strong equivalence can be used to argue the correctness of some program rewritings practiced by ASP software engineers. Here we state several theorems about strong equivalence between programs. *Observations* 1, 2, and 3 are consequences of these results.

We say that body *Body* subsumes body $Body'$ when $Body'$ has the form $Body, Body''$ (note that an order of expressions in a body is immaterial) . We say that a rule $R$ subsumes rule $R'$ when heads of $R$ and $R'$ coincide while body of $R$ subsumes body of $R'$. For example, rule (2) subsumes rule (10).

**Subsumption Rewriting:** Let $\mathbf{R}'$ denote a set of rules subsumed by rule $R$. It is easy to see that formulas $\widehat{R}$ and $\widehat{R} \wedge \widehat{\mathbf{R}'}$ are intuitionistically equivalent. Thus, program composed of rule $R$ and program $\{R\} \cup \mathbf{R}'$ are strongly equivalent. It immediately follows that *Observation* 1 holds. Indeed, rule (2) is strongly equivalent to the set of rules composed of itself and (10). Indeed, rule (2) subsumes rule (10).

**Removing Aggregates:** The following theorem is an immediate consequence of the Replacement Theorem for intuitionistic logic.

**Proposition 1.** *Program*

$$H \leftarrow b \leq \#count\{\mathbf{x} : F(\mathbf{x})\},\ G \tag{21}$$

*is strongly equivalent to program*

$$H \leftarrow \underset{1 \leq i \leq b}{,}\ F(\mathbf{x}^{\mathbf{i}}) \underset{1 \leq i < j \leq b}{,}\ x^i \neq x^j,\ G \tag{22}$$

*where $G$ and $H$ have no occurrences of variables in $\mathbf{x}^{\mathbf{i}}$ $(1 \leq i \leq b)$.*

Proposition 1 shows us that *Observation* 2 is a special case of a more general fact. Indeed, take rules (4) and (11) to be the instances of rules (21) and (22) respectively.

We note that the Replacement Theorem for intuitionistic logic also allows us to immediately conclude the following.

**Corollary 2.** *Program* $H \leftarrow G$ *is strongly equivalent to program* $H \leftarrow G'$ *when* $\widetilde{\forall}(\widehat{G} \leftrightarrow \widehat{G'})$.

Proposition 1 is a special case of this corollary. We could use Corollary 2 to illustrate the correctness of *Observation* 2. Yet, the utility of Proposition 1 is that it can guide syntactic analysis of a program with a goal of equivalent rewriting (for instance, for the sake of performance or clarity). In contrast, Corollary 2 equips us with a general semantic condition that can be utilized in proving the syntactic properties of programs in spirit of Proposition 1.

**Replacing Choice Rule by Defining Rule:** Theorem 2 shows us that *Observation* 3 is an instance of a more general fact.

**Theorem 2.** *Program*

$$\leftarrow p(\mathbf{x}),\ q(\mathbf{x}) \tag{23}$$
$$q(\mathbf{x}) \leftarrow \mathit{not}\ p(\mathbf{x}), F_1 \tag{24}$$
$$\{p(\mathbf{x})\} \leftarrow F_1,\ F_2 \tag{25}$$

*is strongly equivalent to program composed of rules* (23)*,* (24) *and rule*

$$p(\mathbf{x}) \leftarrow \mathit{not}\ q(\mathbf{x}),\ F_1,\ F_2 \tag{26}$$

Indeed, we can derive the former program (its FOL representation) from the latter intuitionistically; and we can derive the later from the former in logic **SQHT**$^{=}$. For the second direction, De Morgan's law $\neg(F \wedge G) \rightarrow \neg F \vee \neg G$ (provable in logic **SQHT**$^{=}$, but not valid intuitionistically) is essential.

To illustrate the correctness of *Observation* 3 by Theorem 2: (i) take rules (1), (2), (3) be the instances of rules (23), (24), (25) respectively, and (ii) rule (12) be the instance of rule (26).

### Useful Rewritings using Structure

In this section, we study rewritings on a program that rely on its structure. We review the concept of a dependency graph used in posing structural conditions on rewritings.

**Review: Predicate Dependency Graph** We present the concept of the predicate dependency graph of a formula following the lines of [10]. An occurrence of a predicate constant, or any other subexpression, in a formula is called *positive* if the number of implications containing that occurrence in the antecedent is even, and *strictly positive* if that number is 0. We say that an occurrence of a predicate constant is *negated* if it belongs to a subformula of the form $\neg F$ (an abbreviation for $F \rightarrow \bot$), and *nonnegated* otherwise.

For instance, in formula (18), predicate constant $o$ has a strictly positive occurrence in the consequence of the implication; whereas the same symbol $o$ has a negated positive occurrence in the antecedent

$$\neg\neg o(A,I) \wedge step(I) \wedge \neg goal(I) \wedge \neg I = n \wedge action(A) \tag{27}$$

of (18). Predicate symbol *action* has a strictly positive non-negated occurrence in (27). The occurrence of predicate symbol *goal* is negated and not positive in (27). The occurrence of predicate symbol *goal* is negated and positive in (18).

An *FOL rule* of a first-order formula $F$ is a strictly positive occurrence of an implication in $F$. For instance, in a conjunction of two formulas (18) and (19) the FOL rules are as follows

$$\neg\neg o(A,I) \wedge SG(I) \wedge action(A) \rightarrow o(A,I) \tag{28}$$

$$\neg\exists A[o(A,I)] \wedge SG(I) \rightarrow \bot. \tag{29}$$

For any first-order formula $F$, the *(predicate) dependency graph* of $F$ relative to the tuple $\mathbf{p}$ of predicate symbols (excluding $=$) is the directed graph that (i) has all predicates in $\mathbf{p}$ as its vertices, and (ii) has an edge from $p$ to $q$ if for some FOL rule $G \rightarrow H$ of $F$

– $p$ has a strictly positive occurrence in $H$, and
– $q$ has a positive nonnegated occurrence in $G$.

We denote such a graph by $DG_{\mathbf{p}}[F]$. For instance, the dependence graph of a conjunction of formulas (18) and (19) relative to all its predicate symbols contains four vertices, namely, $o$, $action$, $step$, and $goal$, and two edges: one from vertex $o$ to vertex $action$ and the other one from $o$ to $step$. Indeed, consider the only two FOL rules (28) and (29) stemming from this conjunction. Predicate constant $o$ has a strictly positive occurrence in the consequent $o(A,I)$ of the implication (28), whereas $action$ and $step$ are the only predicate constants in the antecedent $\neg\neg o(A,I) \wedge SG(I) \wedge action(A)$ of (28) that have positive and nonnegated occurrence in this antecedent. It is easy to see that a FOL rule of the form $G \rightarrow \bot$, e.g., FOL rule (29), does not contribute edges to any dependency graph.

For any logic program $\Pi$, the *dependency graph* of $\Pi$, denoted $DG[\Pi]$, is a directed graph of $\widehat{\Pi}$ relative to the predicates occurring in $\Pi$. For example, let $\Pi$ be composed of two rules (3) and (5). The conjunction of formulas (18) and (19) forms its FOL representation.

**Shifting** We call a logic program *disjunctive* if all its rules have the form (15), where *Body* only contains atoms possibly preceded by *not*. We say that a disjunctive program is *normal* when it does not contain disjunction connective $|$. In [14], Gelfond et al. defined a mapping from a propositional disjunctive program $\Pi$ to a propositional normal program by replacing each rule (15) with $l > 1$ in $\Pi$ by $l$ new rules

$$a_i \leftarrow Body,\ not\ a_1, \ldots not\ a_{i-1}, not\ a_{i+1}, \ldots not\ a_l.$$

They showed that every answer set of the constructed program is also an answer set of $\Pi$. Although the converse does not hold in general, in [1] Ben-Eliyahu and Dechter

showed that the converse holds if $\Pi$ is "head-cycle-free". In [20], Linke et al. illustrated how this property holds about programs with nested expressions that capture choice rules, for instance. Here we generalize these findings further. First, we show that shifting is applicable to first-order programs (that also allow choice rules and aggregates in addition to disjunction). Second, we illustrate that under certain syntactic/structural conditions on a program we may apply shifting "locally" to some rules with disjunction and not others.

For an atom $a$, by $a^0$ we denote its predicate constant. For example $o(A,I)^0 = o$. Let $R$ be a rule of the form (15) with $l > 1$. By $shift_{\mathbf{p}}(R)$ (where $\mathbf{p}$ is a tuple of distinct predicates excluding $=$) we denote the rule

$$\bigmid_{1 \leq i \leq l,\, a_i^0 \in \mathbf{p}} a_i \leftarrow Body \quad , \quad \bigmid_{1 \leq j \leq l,\, a_j^0 \notin \mathbf{p}} not\ a_j. \tag{30}$$

Let $C$ be the set of strongly connected components in the dependency graph of $\Pi$. By $shift(R)$ we denote the new rules $shift_s(R)$ for every $s \in C$ where $s$ has a predicate symbol that occurs in the head of $R$. Consider a sample program $\Pi_{samp}$ composed of two rules with disjunction

$$a \mid b \mid c \leftarrow \qquad\qquad d \mid c \leftarrow$$

and three defining rules

$$a \leftarrow b \qquad b \leftarrow a \qquad e(1). \tag{31}$$

The strongly connected components of program $\Pi_{samp}$ are $\{\{a,b\},\{c\},\{d\},\{e(1)\}\}$. Expression $shift(a \mid b \mid c \leftarrow)$ denotes rules $a \mid b \leftarrow not\ c$ and $c \leftarrow not\ a, not\ b$.

**Theorem 3.** *Let $\Pi$ be a logic program, $\mathbf{R}$ be a set of rules in $\Pi$ of the form* (15) *with $l > 1$. A program constructed from $\Pi$ by replacing each rule $R \in \mathbf{R}$ with $shift(R)$ has the same answer sets as $\Pi$.*

This theorem tells us, for example, that the answer sets of the sample program $\Pi_{samp}$ coincide with the answer sets of three distinct programs composed of rules in (31) and rules in any of the following columns:

| | | |
|---|---|---|
| $a \mid b \leftarrow not\ c$ | $a \mid b \leftarrow not\ c$ | $a \mid b \mid c \leftarrow$ |
| $c \leftarrow not\ a, not\ b$ | $c \leftarrow not\ a, not\ b$ | |
| $d \leftarrow not\ c$ | $d \mid c \leftarrow$ | $d \leftarrow not\ c$ |
| $c \leftarrow not\ d$ | | $c \leftarrow not\ d$ |

To obtain the rules in the first column take $\mathbf{R}$ to consist of the first two rules of $\Pi_{samp}$. To obtain the second column take $\mathbf{R}$ to consist of the first rule of $\Pi_{samp}$. To obtain the last column take $\mathbf{R}$ to consist of the second rule of $\Pi_{samp}$.

We now use Theorem 3 to argue the correctness of *Observation* 4. Let *Plan-choice*$'$ denote a program constructed from the *Plan-choice* encoding by replacing (3) with (6). Let *Plan-choice*$''$ denote a program constructed from the *Plan-choice*, by (i) replacing (3) with (12) and (ii) adding rule (10). Theorem 3 tells us that programs *Plan-choice*$'$ and *Plan-choice*$''$ have the same answer sets. Indeed,

1. take **R** to consist of rule (6) and
2. recall Facts 1, 2, and 3. Given any *Plan-instance* intended to use with *Plan-choice* a program obtained from the union of *Plan-instance* and *Plan-choice*$'$ is such that $o$ is terminal. It is easy to see that any terminal predicate in a program occurs only in the singleton strongly connected components of a program's dependency graph.

Due to *Observations* 1 and 3, the *Plan-choice* encoding has the same answer sets as *Plan-choice*$''$ and consequently the same answer sets as *Plan-choice*$'$. This argument accounts for the proof of *Observation* 4.

**Completion** We now proceed at stating formal results about first-order formulas and their stable models. The fact that we identify logic programs with their FOL representations translates these results to the case of the RASPL-1 programs.

About a first-order formula $F$ we say that it is in *Clark normal form* [9] relative to the tuple/set **p** of predicate symbols if it is a conjunction of formulas of the form

$$\forall \mathbf{x}(G \rightarrow p(\mathbf{x})) \tag{32}$$

one for each predicate $p \in \mathbf{p}$, where $\mathbf{x}$ is a tuple of distinct object variables. We refer the reader to Section 6.1 in [9] for the description of the intuitionistically equivalent transformations that can convert a first-order formula, which is a FOL representation for a RASPL-1 program (without disjunction and denials), into Clark normal form.

The *completion* of a formula $F$ in Clark normal form relative to predicate symbols **p**, denoted by $Comp_{\mathbf{p}}[F]$, is obtained from $F$ by replacing each conjunctive term of the form (32) with $\forall \mathbf{x}(G \leftrightarrow p(\mathbf{x}))$.

The following Corollary is an immediate consequence of Theorem 10 in [9], Theorem 1, and the fact that formula of the form $\widetilde{\forall}(Body \rightarrow \bot)$ is intuitionistically equivalent to formula $\neg\widetilde{\exists} Body$.

**Corollary 3.** *For any formula $G \wedge H$ such that (i) formula $G$ is in Clark normal form relative to* **p** *and $H$ is a conjunction of formulas of the form $\widetilde{\forall}(K \rightarrow \bot)$, the implication*

$$SM_{\mathbf{p}}[G \wedge H] \rightarrow Comp_{\mathbf{p}}[G] \wedge H$$

*is logically valid.*

To illustrate the utility of this result we now construct an argument for the correctness of *Observation* 5. This argument finds one more formal result of use:

**Proposition 2.** *For a program $\Pi$, a first-order formula $F$ such that every answer set of $\Pi$ satisfies $F$, and any two denials $R$ and $R'$ such that $F \rightarrow (\widehat{R} \leftrightarrow \widehat{R'})$, the answer sets of programs $\Pi \cup \{R\}$ and $\Pi \cup \{R'\}$ coincide.*

Consider the *Plan-choice* encoding without denial (4) extended with any *Plan-instance*. We can partition it into two parts: one that contains the denials, denoted by $\Pi_H$, and the remainder, denoted by $\Pi_G$. Recall Fact 1. Following the steps described by Ferraris et al. in [9, Section 6.1], formula $\widehat{\Pi_G}$ turned into Clark normal form relative to the predicate symbols occurring in $\Pi_H \cup \Pi_G$ contains implication (18). The completion of this formula contains equivalence

$$\widetilde{\forall}\big(\neg\neg o(A,I) \wedge SG(I) \wedge action(A) \leftrightarrow o(A,I)\big). \tag{33}$$

By Corollary 3 it follows that any answer set of $\Pi_H \cup \Pi_G$ satisfies formula (33). It is easy to see that an interpretation satisfies (33) and the FOL representation of (11) if and only if it satisfies (33) and the FOL representation of denial (7). Thus, by Proposition 2 program $\Pi_H \cup \Pi_G$ extended with (11) and program $\Pi_H \cup \Pi_G$ extended with (7) have the same answer sets. Recall *Observation* 2 claiming that it is safe to replace denial (4) with denial (11) within an arbitrary program. It follows that program $\Pi_H \cup \Pi_G$ extended with (7) have the same answer sets $\Pi_H \cup \Pi_G$ extended with (4). This concludes the argument for the claim of *Observation* 5.

We now state the last formal results of this paper. The Completion Lemma stated next is essential in proving the Lemma on Explicit Definitions. *Observation* 6 follows immediately from the latter lemma.

**Theorem 4 (Completion Lemma).** *Let $F$ be a first-order formula and $\mathbf{q}$ be a set of predicate constants that do not have positive, nonnegated occurrences in any FOL rule of $F$. Let $\mathbf{p}$ be a set of predicates in $F$ disjoint from $\mathbf{q}$. Let $D$ be a formula in Clark normal form relative to $\mathbf{q}$ so that in every conjunctive term* (32) *of $D$ no occurrence of an element in $\mathbf{q}$ occurs in $G$ as positive and nonnegated. Formula $SM_{\mathbf{pq}}[F \wedge D]$ is equivalent to formulas*

$$SM_{\mathbf{pq}}[F \wedge D] \wedge Comp[D], \tag{34}$$

$$SM_{\mathbf{p}}[F] \wedge Comp[D], \text{ and} \tag{35}$$

$$SM_{\mathbf{pq}}[F \wedge \bigwedge_{q \in \{\mathbf{q}\}} \forall \mathbf{x}\big(\neg\neg q(\mathbf{x}) \rightarrow q(\mathbf{x})\big)] \wedge Comp[D]. \tag{36}$$

For an interpretation $I$ over signature $\Sigma$, by $I_{|\sigma}$ we denote the interpretation over $\sigma \subseteq \Sigma$ constructed from $I$ so that every function or predicate symbol in $\sigma$ is assigned the same value in both $I$ and $I_{|\sigma}$. We call formula $G$ in (32) a *definition* of $p(\mathbf{x})$.

**Theorem 5 (Lemma on Explicit Definitions).** *Let $F$ be a first-order formula, $\mathbf{q}$ be a set of predicate constants that do not occur in $F$, and $\mathbf{p}$ be an arbitrary set of predicate constants in $F$. Let $D$ be a formula in Clark normal form relative to $\mathbf{q}$ so that in every conjunctive term* (32) *of $D$ there is no occurrence of an element in $\mathbf{q}$ in $G$. Then*

*i $M \mapsto M_{|\sigma(F)}$ is a 1-1 correspondence between the models of $SM_{\mathbf{pq}}[F \wedge D]$ and the models $SM_{\mathbf{p}}[F]$, and*

*ii $SM_{\mathbf{pq}}[F \wedge D]$ and $SM_{\mathbf{pq}}[F^{\mathbf{q}} \wedge D]$ are equivalent, where we understand $F^{\mathbf{q}}$ as a formula obtained from $F$ by replacing occurrences of the definitions of $q(\mathbf{x})$ in $D$ with $q(\mathbf{x})$.*

We note that Splitting Theorem from [10], Theorem 2 and Theorem 11 from [9] provide sufficient grounds to carry out the argument for Theorem 4. The proof of item (i) in Theorem 5 relies on Theorem 4 and the fact that the completion of considered formula $D$ in Theorem 5 corresponds to so called explicit definitions in classical logic. The proof of item (ii) utilizes the Replacement Theorem for intuitionistic logic.

It is easy to see that program composed of a single rule

$$p(\mathbf{y}) \leftarrow 1 \leq \#count\{\mathbf{x} : F(\mathbf{x},\mathbf{y})\}$$

and program $p(\mathbf{y}) \leftarrow F(\mathbf{x},\mathbf{y})$ are strongly equivalent. Thus, we can identify rule (8) in the *Plan-disj* encoding with the rule

$$sthHpd(I) \leftarrow 1 \leq \#count\{A : o(A,I)\}. \tag{37}$$

Using this fact and Theorem 5 allows us to support *Observation* 6. Take $F$ to be the FOL representation of *Plan-choice* encoding extended with any *Plan-instance* and $D$ be the FOL representation of (37), $\mathbf{q}$ be composed of a single predicate *sthHpd* and $\mathbf{p}$ be composed of all the predicates in *Plan-choice* and *Plan-instance*.

**Conclusions** This paper lifts several important theoretical results for propositional programs to the case of first-order logic programs. These new formal findings allow us to argue a number of first-order program rewritings to be safe. We illustrate the usefulness of these findings by utilizing them in constructing an argument which shows that the sample programs *Plan-choice* and *Plan-disj* are essentially the same. We believe that these results provide a strong building block for a portfolio of safe rewritings that can be used in creating an automatic tool for carrying these rewritings during program performance optimization phase discussed in Introduction.

*Acknowledgements* We are grateful to Vladimir Lifschitz and Miroslaw Truszczynski for valuable discussions on the subject of this paper. Yuliya Lierler was partially supported by the NSF 1707371 grant.

## References

1. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Annals of Mathematics and Artificial Intelligence **12**, 53–87 (1994)
2. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54(12)**, 92–103 (2011)
3. 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)
4. Eiter, T., Fink, M.: Uniform equivalence of logic programs under the stable model semantics. In: Palamidessi, C. (ed.) Logic Programming. pp. 224–238. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
5. Eiter, T., Fink, M., Tompits, H., Traxler, P., Woltran, S.: Replacements in non-ground answer-set programming. In: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR) (2006)
6. Eiter, T., Tompits, H., Woltran, S.: On solution correspondences in answer-set programming. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). pp. 97–102 (2005)
7. Eiter, T., Traxler, P., Woltran, S.: An implementation for recognizing rule replacements in non-ground answer-set programs. In: Proceedings of European Conference On Logics In Artificial Intelligence (JELIA) (2006)
8. Ferraris, P.: Answer sets for propositional theories. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 119–131 (2005)
9. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artificial Intelligence **175**, 236–263 (2011)
10. Ferraris, P., Lee, J., Lifschitz, V., Palla, R.: Symmetric splitting in the general theory of stable models. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). pp. 797–803. IJCAI press (2009)
11. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming **5**, 45–74 (2005)

12. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract gringo. Theory and Practice of Logic Programming **15**, 449–463 (7 2015). https://doi.org/10.1017/S1471068415000150, http://journals.cambridge.org/article_S1471068415000150
13. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press (2014)
14. Gelfond, M., Lifschitz, V., Przymusińska, H., Truszczyński, M.: Disjunctive defaults. In: Allen, J., Fikes, R., Sandewall, E. (eds.) Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR). pp. 230–237 (1991)
15. Harrison, A., Lierler, Y.: First-order modular logic programs and their conservative extensions. Theory and Practice of Logic programming, 32nd Int'l. Conference on Logic Programming (ICLP) Special Issue (2016)
16. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. In: Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 175–187 (2007)
17. Lee, J., Lifschitz, V., Palla, R.: A reductive semantics for counting and choice in answer set programming. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI). pp. 472–479 (2008)
18. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic **2**, 526–541 (2001)
19. Lifschitz, V., Pearce, D., Valverde, A.: A characterization of strong equivalence for logic programs with variables. In: Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 188–200 (2007)
20. Linke, T., Tompits, H., Woltran, S.: On acyclic and head-cycle free nested logic programs. In: Proceedings of 19th International Conference on Logic Programming (ICLP). pp. 225–239 (2004)
21. Mints, G.: A Short Introduction to Intuitionistic Logic. Kluwer (2000)
22. Pearce, D., Valverde, A.: Synonymous theories and knowledge representations in answer set programming. Journal of Computer and System Sciences **78**(1), 86 – 104 (2012). https://doi.org/https://doi.org/10.1016/j.jcss.2011.02.013, http://www.sciencedirect.com/science/article/pii/S0022000011000420, jCSS Knowledge Representation and Reasoning
23. Woltran, S.: Characterizations for relativized notions of equivalence in answer set programming. In: Alferes, J.J., Leite, J. (eds.) Logics in Artificial Intelligence. pp. 161–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)