TPLP **22** (3): 335–366, 2022.
doi:10.1017/S1471068421000545 First published online 18 March 2022

*Strong Equivalence and Program Structure in Arguing Essential Equivalence between Logic Programs**

YULIYA LIERLER
University of Nebraska, 6001 Dodge St, Omaha, NE 68182, USA
(*e-mail:* ylierler@unomaha.edu)

submitted 15 October 2019; revised 7 September 2021; accepted 15 November 2021;

Abstract

Answer set programming is a prominent declarative programming paradigm used in formulating combinatorial search problems and implementing different knowledge representation formalisms. Frequently, 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 presents formal results carefully relating a number of interesting program rewritings. It also provides the proof of correctness of system PROJECTOR concerned with automatic program rewritings for the sake of efficiency.

KEYWORDS: program analysis, strong, essential equivalence

1 Introduction

Answer set programming (ASP) is a prominent knowledge representation paradigm with roots in logic programming (Brewka *et al*. 2011). 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 languages $\mathscr{B}$ (Gelfond and Lifschitz 1998, Section 5), $\mathscr{C}$ (Lifschitz and Turner 1999), $\mathscr{BC}$ (Lee *et al*. 2013), $\mathscr{C}+$ (Giunchiglia *et al*. 2004; Babb and Lee 2013), and $\mathscr{AL}$ (Gelfond and Kahl 2014, Section 8).

In answer set programming, a given computational problem is represented by a *declarative program*, also called a *problem encoding*, that describes the properties of a solution to the problem. Then, an answer set solver is used to generate answer sets for the program. These answer sets correspond to solutions to the original problem. As answer set programming evolves, new language features come to life providing means to reformulations of original problem encodings. Such new formulations often prove to be more intuitive and/or more concise and/or more efficient. Similarly, when a software engineer

* We are grateful to Pedro Cabalar, Jorge Fandinno, Nicholas Hippen, Vladimir Lifschitz, Miroslaw Truszczynski for valuable discussions on the subject of this paper. We are also thankful to the anonymous reviewers for their help to improve the presentation of the material in the paper. Yuliya Lierler was partially supported by the NSF 1707371 grant.

tackles a problem domain by means of answer set programming, it is a common practice to first develop *a/some* solution to a problem and then rewrite this solution iteratively using such techniques, for example, as projection to gain a better performing encoding (Buddenhagen and Lierler 2015). These common processes bring a scientific question to light: *what are the formal means to argue the correctness of renewed formulations of the original encodings to problems?* In other words, under the assumption that the original encoding to a problem is correct, how can we argue that a related and yet different encoding is also correct? In addition, automated ASP program rewriting systems come to life. Systems LPOPT (Bichler *et al.* 2016) and PROJECTOR (Hippen and Lierler 2019) exemplify such a trend. These tools rewrite an original program into a new one with the goal of improving an ASP solver's performance. Once again, formal means are necessary to claim the correctness of such systems. We note that the last section of this work is devoted to the claim of correctness of system PROJECTOR.

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 (Lifschitz *et al.* 2001; 2007), uniform equivalence (Eiter and Fink 2003), relativized strong and uniform equivalences (Woltran 2004). Another related approach is the study of forgetting (Leite 2017). Also, equivalences relative to specified signatures were considered (Erdoğan and Lifschitz 2004; Eiter *et al.* 2005; Woltran 2008; Harrison and Lierler 2016). In most of the cases the programs considered for studying the distinct forms of equivalence are propositional. Works by Eiter *et al.* (2006a), Eiter *et al.* (2006b), Lifschitz *et al.* (2007), Oetsch and Tompits (2008), Pearce and Valverde (2012), and Harrison and Lierler (2016) are exceptions. These authors consider programs with variables (or, first-order programs). 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.

In this paper, we show how concepts of strong equivalence and so called conservative extension are of use in illustrating that two programs over different signatures and with significantly different structure are "essentially the same" or "essentially equivalent" in a sense that they capture solutions to the same problem. Let us make the concept of an essential equivalence between problem's encodings precise. We use the same notion of the search problem as Brewka et al. Brewka *et al.* (2011). Quoting from their work, a *search problem* P consists of a set of instances with each *instance* I assigned a finite set $S_P(I)$ of solutions. We say that logic program $\Pi_P(\cdot)$ is an encoding of P when for any instance I of this problem, the solutions to I – the elements of set $S_P(I)$ – can be reconstructed from the answer sets of program $\Pi_P(I)$. We say that encodings $\Pi_P(\cdot)$ and $\Pi'_P(\cdot)$ are *essentially equivalent*, when given any instance I of problem P the answer sets of programs $\Pi_P(I)$ and $\Pi'_P(I)$ are in one-to-one correspondence. The paper has two parts. In the *first part*, we consider propositional programs. In the *second part*, we move to the programs with variables. *These parts can be studied separately. The first one is appropriate for* researchers who are not yet deeply familiar with answer set programming theory and are interested in learning formal details. *The second part*[1] *is geared toward*

[1] This part of the paper is a substantially extended version of the paper presented at PADL 2019 (Lierler 2019).

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

answer set programming practitioners providing them with theoretical grounds and tools to assist them in program analysis and formal claims about the developed encodings and their relations. In both of these parts, we utilize running examples stemming from the literature. For instance, for the case of propositional programs, we study two distinct ASP formalizations of action language $\mathscr{C}$. In the case of first-order programs, we study two distinct formalizations of planning modules for action language $\mathscr{AL}$ given in Gelfond and Kahl (2014, Section 9). Namely,

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

In both cases, we identify interesting results.

Paper Outline. The paper is structured as follows. Section 2.1 presents the concepts of (i) a propositional logic program, (ii) strong equivalence between propositional logic programs, and (iii) a propositional logic program being a conservative extension of another one. Section 2.2 introduces a rewriting technique frequently used by ASP developers when a new auxiliary proposition is introduced in order to denote a conjunction of other propositions. Then these conjunctions are safely renamed by the auxiliary atom. We refer to this process as explicit definition rewriting and illustrate its correctness. We continue by reviewing action language $\mathscr{C}$ in Section 2.3, which serves a role of a running example in the first part of the paper. In Section 2.4.1, we present an original, or gold standard, translation of language $\mathscr{C}$ to a logic program. Section 2.4.2 states a modern formalization stemming from the translation of a syntactically restricted fragment of $\mathscr{C}+$. At last, in Section 2.4.3, we showcase how we can argue on the correctness of a modern formalization by stating the formal relation between the original and modern translations of language $\mathscr{C}$. An interested reader may proceed to the Appendix to find the details of the proof of the claim. There, we utilize reasoning by "weak" natural deduction and a formal result on explicit definition rewriting. (A weak natural deduction system is reviewed in the Appendix.) We also note that there is an interesting by-product of our analysis: we establish that $\mathscr{C}+$ can be viewed as a true generalization of the language $\mathscr{C}$ to the case of multivalued fluents.

We start the second part of the paper by presenting the *Plan-choice* and *Plan-disj* programs at the onset of Section 3. We then introduce the logic program language called RASPL-1 in Section 3.2. The semantics of this language is given in terms of the SM operator reviewed in Section 3.2.2. In Section 3.2.3, we review the concept of strong equivalence for first-order programs. Section 3.3 is devoted to a sequence of formal results on program rewritings. One of the findings of this work is lifting the results by Ben-Eliyahu and Dechter (1994) to the first order case. Earlier work claimed that propositional head-cycle-free disjunctive programs 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. Another important contribution of the work is lifting the Completion Lemma and the Lemma on Explicit Definitions stated in Ferraris (2005), Ferraris and Lifschitz (2005) from the case of propositional theories and propositional logic programs to first-order programs. In conclusion, in Section 3.4, we review a frequently used rewriting technique called projection that

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

often produces better performing encodings. We illustrate the utility of the presented theoretical results as they can be used to argue the correctness of distinct versions of projection that also include rules with aggregates. In particular, the last formal result stated in the paper provides a proof of correctness of system PROJECTOR. The Lemma on Explicit Definitions presented here is essential in this argument.

The Appendix provides the proofs for the formal results presented in the paper.

2 Propositional programs

2.1 Traditional logic programs and their equivalences

A *(traditional logic) program* is a finite set of *rules* of the form

$$a_0 \leftarrow a_1, \ldots, a_l, not\ a_{l+1}, \ldots, not\ a_m, not\ not\ a_{m+1}, \ldots, not\ not\ a_n, \tag{1}$$

$(0 \leq l \leq m \leq n)$, where each a_0 is an atom or $\bot$ and each a_i $(1 \leq i \leq n)$ is an atom, $\top$, or $\bot$. The expression containing atoms a_1 through a_n is called the *body* of the rule. Atom a_0 is called a *head*.

We define the *answer sets* of a traditional program Π following Lifschitz *et al.* (1999). We say that a program is *basic*, when it does not contain connective *not*. In other words, a basic program consists of rules

$$a_0 \leftarrow a_1, \ldots, a_l, \tag{2}$$

where each a_0 is an atom or $\bot$ and each a_i $(1 \leq i \leq l)$ is an atom, $\top$, or $\bot$. We say that a set X of atoms *satisfies* rule (2) if it satisfies the implication

$$a_1 \wedge \cdots \wedge a_l \rightarrow a_0.$$

We say that a set X of atoms is an *answer set* of a basic program Π if X is a minimal set among sets satisfying all rules of Π.

A *reduct* of a program Π with respect to a set X of atoms, denoted by Π^X, is constructed as follows. For each rule (1) in Π

1. when *not not* a_i $(m+1 \leq i \leq n)$ is such that $a_i \in X$, replace this expression with $\top$, otherwise replace it with $\bot$,
2. when *not* a_i $(l+1 \leq i \leq m)$ is such that $a_i \in X$, replace this expression with $\bot$, otherwise replace it with $\top$.

It is easy to see that a reduct of a program forms a basic program. We say that a set X of atoms is an *answer set* of a traditional program if it is an answer set for the reduct Π^X.

In the later part of the paper, we present the definition of an answer set for programs with variables by means of operator SM (Ferraris *et al.* 2011). Ferraris *et al.* (2011) show in which sense SM operator captures the semantics of answer sets presented here.

According to Ferraris and Lifschitz (2005) and Ferraris (2005), rules of the form (1) are sufficient to capture the meaning of the choice rule construct commonly used in answer set programming. For instance, the choice rule $\{p\} \leftarrow q$ is understood as the rule

$$p \leftarrow q,\ not\ not\ p.$$

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

We intuitively read this rule as *given* q *atom* p may be *true*. We use choice rule notation in the sequel.

Strong Equivalence Traditional programs Π_1 and Π_2 are *strongly equivalent* (Lifschitz *et al.* 2001) when for every program Π, programs $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ have the same answer sets. In addition to introducing strong equivalence, Lifschitz *et al.* (2001) also illustrated that traditional programs can be associated with the propositional formulas and a question whether the programs are strongly equivalent can be turned into a question whether the respective propositional formulas are equivalent in the logic of here-and-there (HT-logic) (Lukasiewicz 1941), an intermediate logic between classical and intuitionistic logics.

We follow the steps of Lifschitz *et al.* (2001) and identify a rule (1) with the propositional formula

$$a_1 \wedge \cdots \wedge a_l \wedge \neg a_{l+1} \wedge \cdots \wedge \neg a_m \wedge \neg\neg a_{m+1} \wedge \cdots \wedge \neg\neg a_n \rightarrow a_0. \tag{3}$$

Conservative Extensions Harrison and Lierler (2016) defined the notion of a conservative extension for the case of logic programs. Similarly to strong equivalence, it attempts to capture the conditions under which we can rewrite parts of the program and yet guarantee that the resulting program is not different in an essential way from the original one. Conservative extensions allow us to reason about rewritings even when the rules in question have different signatures.

For a program Π, by $atoms(\Pi)$ we denote the set of atoms occurring in Π. Let Π and Π' be programs such that $atoms(\Pi) \subseteq atoms(\Pi')$. We say that program Π' is a *conservative extension* of Π if $X \mapsto X \cap atoms(\Pi)$ is a 1-1 correspondence between the answer sets of Π' and the answer sets of Π. For instance, program

$$\begin{array}{l} \neg q \rightarrow p \\ \neg p \rightarrow q \end{array} \tag{4}$$

is a conservative extension of the program containing the single choice rule

$$\{p\}.$$

Furthermore, given program Π such that (i) it contains rule $\{p\}$ and (ii) $q \notin atoms(\Pi)$, a program constructed from Π by replacing $\{p\}$ with (4) is a conservative extension of Π.

2.2 On explicit definition rewriting

We now turn our attention to a common rewriting technique based on explicit definitions and illustrate its correctness. This technique introduces an auxiliary proposition in order to denote a conjunction of other propositions. Then these conjunctions are safely renamed by the auxiliary atom in the remainder of the program.

We call a formula *basic conjunction* when it is of the form

$$a_1 \wedge \cdots \wedge a_l \wedge \neg a_{l+1} \wedge \cdots \wedge \neg a_m \wedge \neg\neg a_{m+1} \wedge \cdots \wedge \neg\neg a_n, \tag{5}$$

where each a_i $(1 \leq i \leq n)$ is an atom, $\top$, or $\bot$. For example, the body of any rule in a traditional program is a basic conjunction.

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

Let Π be a program, Q be a set of atoms that do not occur in Π. For an atom $q \in Q$, let $def(q)$ denote a basic conjunction (5), where a_i $(1 \leq i \leq n)$ in $atoms(\Pi)$. We say that $def(q)$ is an *explicit definition* of q in terms of Π. By $def(Q)$ we denote a set of formulas $def(q)$ for each atom $q \in Q$. We assume that all these formulas are distinct. Program $\Pi[Q, def(Q)]$ is constructed from Π as follows:

- all occurrences of all formulas $def(q)$ from $def(Q)$ in some body of Π are replaced by respective q,
- for every atom $q \in Q$ a rule of the form

$$def(q) \rightarrow q$$

is added to the program.

For instance, let Π be a program

$$\neg q \rightarrow p$$

and $def(r)$ be a formula $\neg q$, then $\Pi[\{r\}, \{def(r)\}]$ follows

$$r \rightarrow p$$
$$\neg q \rightarrow r$$

The proposition below supports the fact that the latter program is a conservative extension of the former. It is an important claim as although this kind of rewriting is very frequently used in practice to the best of our knowledge this is the first time it has been formally claimed.

Proposition 1
Let Π be a program, Q be a set of atoms that do not occur in Π, and $def(Q)$ be a set composed of explicit definitions for each element in Q in terms of Π. Program $\Pi[Q, def(Q)]$ is a conservative extension of Π.

2.3 Review of action language C

This review of action language $\mathscr{C}$ follows Lifschitz and Turner (1999).

We consider a set σ of propositional symbols partitioned into the fluent names σ^{fl} and the elementary action names σ^{act}. An action is an interpretation of σ^{act}. Here we only consider what Lifschitz and Turner (1999) call *definite* action descriptions so that we only define this special class of $\mathscr{C}$ action descriptions.

Syntactically, a $\mathscr{C}$ *action description* is a set of *static* and *dynamic laws*. *Static laws* are of the form

$$\textbf{caused } l_0 \textbf{ if } l_1 \wedge \cdots \wedge l_m \tag{6}$$

and dynamic laws are of the form

$$\textbf{caused } l_0 \textbf{ if } l_1 \wedge \cdots \wedge l_m \textbf{ after } l_{m+1} \wedge \cdots \wedge l_n, \tag{7}$$

where

- l_0 is either a literal over σ^{fl} or the symbol $\bot$,
- l_i $(1 \leq i \leq m)$ is a literal in σ^{fl},
- l_i $(m+1 \leq i \leq n)$ is a literal in σ, and

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

- conjunctions $l_1 \wedge \cdots \wedge l_m$ and $l_{m+1} \wedge \cdots \wedge l_n$ are possibly empty and understood as $\top$ in this case.

In both laws, the literal l_0 is called the *head*.

Semantically, an action description defines a graph or a transition system. We call nodes of this graph *states* and directed edges *transitions*. We now define these concepts precisely. Consider an action description D. A *state* is an interpretation of σ^{fl} that satisfies implication

$$l_1 \wedge \cdots \wedge l_m \rightarrow l_0$$

for every static law (6) in D. A *transition* is any triple $\langle s, a, s' \rangle$, where s, s' are states and a is an action; s is the *initial* state of the transition, and s' is its *resulting* state. A literal l is *caused* in a transition $\langle s, a, s' \rangle$ if it is

- the head of a static law (6) from D such that s' satisfies $l_1 \wedge \cdots \wedge l_m$, or
- the head of a dynamic law (7) from D such that s' satisfies

$$l_1 \wedge \cdots \wedge l_m$$

and $s \cup a$ satisfies

$$l_{m+1} \wedge \cdots \wedge l_n.$$

A transition $\langle s, a, s' \rangle$ is *causally explained* by D if its resulting state s' is the set of literals caused in this transition.

The *transition system described by an action description* D is the directed graph, which has the states of D as nodes, and which includes an edge from state s to state s' labeled a for every transition $\langle s, a, s' \rangle$ that is causally explained by D.

We now present an example by Lifschitz and Turner (1999) that formalizes *the effects of putting an object in water*. We use this domain as a running example. It uses the fluent names $inWater$ and wet and the elementary action name $putInnWater$. We follow the convention by Lifschitz and Turner (1999) and present states (interpretations) as lists of literals. In the notation introduced by Gelfond and Lifschitz (1998, Section 6), the action description for *water domain* follows[2]

caused wet **if** $inWater$
$putInWater$ **causes** $inWater$
inertial $inWater, \neg inWater, wet, \neg wet$

Written in full this action description contains six laws:

caused wet **if** $inWater$
caused $inWater$ **if** $\top$ **after** $putInWater$
caused $inWater$ **if** $inWater$ **after** $inWater$
caused $\neg inWater$ **if** $\neg inWater$ **after** $\neg inWater$
caused wet **if** wet **after** wet
caused $\neg wet$ **if** $\neg wet$ **after** $\neg wet$

[2] We remark on the keyword **inertial**. It intuitively suggests that a fluent declared to be inertial is such that its value can be changed by actions only. If no actions, which directly or indirectly affect such a fluent, occur then the value of the inertial fluent remains unchanged.

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

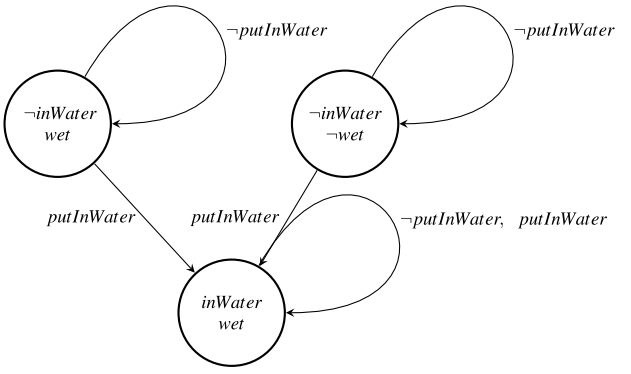

Fig. 1. Transition diagram for Water domain.

The corresponding transition system has 3 states:

$$\neg inWater\ \neg wet,\ \neg inWater\ wet,\ inWater\ wet$$

and 6 causally explained transitions

$$\begin{array}{ll}\langle \neg inWater\ \neg wet, \neg putInWater, \neg inWater\ \neg wet\rangle, & \langle \neg inWater\ \neg wet, putInWater, inWater\ wet\rangle,\\ \langle \neg inWater\ wet, \neg putInWater, \neg inWater\ wet\rangle, & \langle \neg inWater\ wet, putInWater, inWater\ wet\rangle,\\ \langle inWater\ wet, \neg putInWater, inWater\ wet\rangle, & \langle inWater\ wet, putInWater, inWater\ wet\rangle.\end{array} \tag{8}$$

We depict this transition system in Figure 1.

2.4 On relation between the original and modern formalizations of $\mathscr{C}$

We start this section by reviewing the original formalization of action language $\mathscr{C}$ in the language of logic programs under answer set semantics (Lifschitz and Turner 1999). Specifically, Lifschitz and Turner (1999) proposed a translation from an action description D in $\mathscr{C}$ to a logic program $lp_T(D)$ so that the answer sets of this program capture all the "histories" of length T in the transition system specified by D.

Since that original work, languages of answer set programming have incorporated new features such as, for instance, choice rules. At present, these are commonly used by the practitioners of ASP. It is easy to imagine that in a modern formalization of action language $\mathscr{C}$, given a system description D a resulting program will be different from the original $lp_T(D)$. In fact, Babb and Lee (2013) present a translation of an action language $\mathscr{C}+$ (according to (Giunchiglia *et al.* 2004, Section 7.3) $\mathscr{C}$ is *the immediate predecessor* of $\mathscr{C}+$) that utilizes modern language features such as choice rules. In Section 2.4.2, we present this translation for the case of $\mathscr{C}$. In particular, we restrict the language of $\mathscr{C}+$ to Boolean, or two-valued, fluents (in general, $\mathscr{C}+$ permits multivalued fluents). We call this translation $simp_T(D)$. Although, $lp_T(D)$ and $simp_T(D)$ share a lot in common they are substantially different. To begin with, the signatures of these programs are not identical. Also, $simp_T(D)$ utilizes choice rules. The programs $lp_T(D)$ and $simp_T(D)$ are different enough that it is not immediately obvious that their answer sets capture the same entities. There are two ways to argue that the program $simp_T(D)$ is "essentially

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

the same" as program $lp_T(D)$: to illustrate that the answer sets of $simp_T(D)$ capture all the "histories" of length T in the transition system specified by D by relying

1. on the definitions of action language $\mathscr{C}$;
2. on the properties of programs $lp_T(D)$ and $simp_T(D)$ that establish a one-to-one correspondence between their answer sets.

Here we take the second way into consideration. We illustrate how the concepts of strong equivalence and conservative extension together with formal results previously discovered about these prove to be of essence in this argument. The details of this argument are given in the Appendix. Thus, we showcase a proof technique for arguing on 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. It is a common practice in development of answer set programming solutions to obtain a final formalization of a problem by first producing such a gold standard program and then applying a number of rewriting procedures to that program to enhance its performance. The benefits of the proposed method are twofold. First, this methodology can be used by software engineers during a formal analysis of their solutions. Second, we trust that this methodology paves a way for a general framework for arguing correctness of common program rewritings so that they can be automated for the gain of performance. This is a question for investigation in the future.

2.4.1 Review of basic translation

Let D be an action description. Lifschitz and Turner (1999) defined a translation from action description D to a logic program $lp_T(D)$ parametrized with a positive integer T that intuitively represents a time horizon. The remarkable property of logic program $lp_T(D)$ that its answer sets correspond to "histories" – path/trajectories of length T in the transition system described by D.

Recall that by σ^{fl} we denote fluent names of D and by σ^{act} we denote elementary action names of D. Let us construct "complementary" vocabularies to σ^{fl} and σ^{act} as follows

$$\text{-}\sigma^{fl} = \{\text{-}a \mid a \in \sigma^{fl}\}$$

and

$$\text{-}\sigma^{act} = \{\text{-}a \mid a \in \sigma^{act}\}.$$

For a literal l, we define

$$\widehat{l} = \begin{cases} a \text{ if } l \text{ is an atom } a \\ \text{-}a \text{ if } l \text{ is a literal of the form } \neg a \end{cases}$$

and

$$\bar{l} = \begin{cases} \text{-}a \text{ if } l \text{ is an atom } a \\ a \text{ if } l \text{ is a literal of the form } \neg a \end{cases}$$

The language of $lp_T(D)$ has atoms of four kinds:

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

1. fluent atoms–the fluent names of σ^{fl} followed by (t) where $t = 0, \ldots, T$,
2. action atoms–the action names of σ^{act} followed by (t) where $t = 0, \ldots, T-1$,
3. complement fluent atoms–the elements of $\text{-}\sigma^{fl}$ followed by (t) where $t = 0, \ldots, T$,
4. complement action atoms–the elements of $\text{-}\sigma^{act}$ followed by (t) where $t = 0, \ldots, T-1$.

Program $lp_T(D)$ consists of the following rules:

1. for every atom a that is a fluent or action atom of the language of $lp_T(D)$

$$\bot \leftarrow a, \text{-}a \tag{9}$$

and

$$\bot \leftarrow \ not\ a,\ not\ \text{-}a \tag{10}$$

2. for every static law (6) in D, the rules

$$\widehat{l_0}(t) \leftarrow \ not\ \overline{l_1}(t), \ldots,\ not\ \overline{l_m}(t) \tag{11}$$

for all $t = 0, \ldots, T$ (we understand $\widehat{l_0}(t)$ as $\bot$ if l_0 is $\bot$),

3. for every dynamic law (7) in D, the rules

$$\widehat{l_0}(t+1) \leftarrow \ not\ \overline{l_1}(t+1), \ldots,\ not\ \overline{l_m}(t+1), \widehat{\overline{l_{m+1}}}(t), \ldots,\ \widehat{l_n}(t), \tag{12}$$

for all $t = 0, \ldots, T-1$,

4. the rules

$$\begin{array}{l} \text{-}a(0) \leftarrow \ not\ a(0) \\ a(0) \leftarrow \ not\ \text{-}a(0), \end{array} \tag{13}$$

for all fluent names a in σ^{fl} and

5. for every atom a that is an action atom of the language of $lp_T(D)$ the rules

$$\begin{array}{l} \text{-}a \leftarrow \ not\ a \\ a \leftarrow \ not\ \text{-}a. \end{array}$$

Proposition 2 (*Proposition 1 in Lifschitz and Turner (1999)*)
For a set X of atoms, X is an answer set for $lp_T(D)$ if and only if it has the form

$$\left[\bigcup_{t=0}^{T-1} \{\widehat{l}(t) \mid l \in s_t \cup a_t\}\right] \cup \{\widehat{l}(T) \mid l \in s_T\}$$

for some path $\langle s_0, a_0, s_1, \ldots, s_{T-1}, a_{T-1}, s_T\rangle$ in the transition system described by D.

We note that Lifschitz and Turner (1999) presented lp_T translation and Proposition 1 using both default negation *not* and classical negation $\neg$ in the program. Yet, classical negation can always be eliminated from a program by means of auxiliary atoms and additional constraints as it is done here. In particular, auxiliary atoms have the form $\text{-}a(i)$ (where $\text{-}a(i)$ intuitively stands for literal $\neg a(i)$), while the additional constraints have the form (9).

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

To exemplify this translation, consider $\mathscr{C}$ action description (8). Its translation consists of all rules of the form

1. $\bot \leftarrow inWater(t), \text{-}inWater(t)$
   $\bot \leftarrow \ not\ inWater(t),\ not\ \text{-}inWater(t)$
   $\bot \leftarrow wet(t), \text{-}wet(t)$
   $\bot \leftarrow \ not\ wet(t),\ not\ \text{-}wet(t)$
   $\bot \leftarrow putInWater(t),\ not\ \text{-}putInWater(t)$
   $\bot \leftarrow \ not\ putInWater(t),\ not\ \text{-}putInWater(t)$

2. $wet(t) \leftarrow \ not\ \text{-}inWater(t)$

3. $inWater(t+1) \leftarrow \ putInWater(t)$
   $inWater(t+1) \leftarrow \ not\ \text{-}inWater(t+1),$
   $inWater(t)$
   $\text{-}inWater(t+1) \leftarrow \ not\ inWater(t+1),$
   $\text{-}inWater(t)$
   $wet(t+1) \leftarrow \ not\ \text{-}wet(t+1),\ wet(t)$
   $\text{-}wet(t+1) \leftarrow \ not\ wet(t+1), \text{-}wet(t)$

4. $\text{-}inWater(0) \leftarrow \ not\ inWater(0)$
   $inWater(0) \leftarrow \ not\ \text{-}inWater(0)$
   $\text{-}wet(0) \leftarrow \ not\ wet(0)$
   $wet(0) \leftarrow \ not\ \text{-}wet(0)$

5. $\text{-}putInWater(t) \leftarrow \ not\ putInWater(t)$
   $putInWater(t) \leftarrow \ not\ \text{-}putInWater(t)$

2.4.2 Simplified modern translation

As in the previous section, let D be an action description and T a positive integer. In this section, we define a translation from action description D to a logic program $simp_T(D)$ inspired by $lp_T(D)$ and the advances in answer set programming languages. The main property of logic program $simp_T(D)$ is as in case of $lp_T(D)$ that its answer sets correspond to histories captured by the transition system described by D. This translation is a special case of a translation by Babb and Lee (2013) for an action language $\mathscr{C}+$ that is limited to two-valued fluents.

The language of $simp_T(D)$ has atoms of three kinds that coincide with the three first groups (1-3) of atoms identified in the language of $lp_T(D)$.

For a literal l, we define

$$\widetilde{l} = \begin{cases} not\ a \text{ if } l \text{ is a literal of the form } \neg a, \text{ where } a \in \sigma^{act} \\ \widehat{l} \text{ otherwise} \end{cases}$$

Program $simp_T(D)$ consists of the following rules:

1. for every fluent atom a the rules of the form (9) and (10),
2. for every static law (6) in D, $simp_T(D)$ contains the rules of the form

$$\widehat{l_0}(t) \leftarrow \ not\ not\ \widehat{l_1}(t), \ldots,\ not\ not\ \widehat{l_m}(t) \tag{14}$$

for all $t = 0, \ldots, T$[3],

[3] Babb and Lee (2013) allow rules with arbitrary formulas in their bodies so that in place of (14) they consider rule $\widehat{l_0}(t) \leftarrow \ not\ not\ (\widehat{l_1}(t) \wedge \cdots \wedge \widehat{l_m}(t))$. Yet, it is well known that such a rule is strongly equivalent to (14). Furthermore, more answer set solvers allow rules of the form (14) than more general rules considered in Babb and Lee (2013).

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

3. for every dynamic law (7) in D, the rules

$$\widehat{l_0}(t+1) \leftarrow \ not\ not\ \widehat{l_1}(t+1), \ldots,\ not\ not\ \widehat{l_m}(t+1), \widetilde{l_{m+1}}(t), \ldots,\ \widetilde{l_n}(t),$$

for all $t = 0, \ldots, T-1$,

4. the rules

$$\begin{array}{l} \{\text{-}a(0)\} \\ \{a(0)\} \end{array} \tag{15}$$

for all fluent names a in σ^{fl} and

5. for every atom a that is an action atom of the language of $lp_T(D)$, the choice rules $\{a\}$. Here we note that the language $\mathscr{C}$ assumes every action to be exogenous, whereas this is not the case in $\mathscr{C}+$, where it has to be explicitly stated whether an action has this property. Thus, in Babb and Lee (2013) rules of this group only appear for the case of actions that have been stated exogenous.

The $simp_T$ translation of $\mathscr{C}$ action description (8) consists of all rules of the form

1. $\bot \leftarrow inWater(t), \text{-}inWater(t)$ $\bot \leftarrow \ not\ inWater(t),\ not\ \text{-}inWater(t)$ $\bot \leftarrow wet(t), \text{-}wet(t)$ $\bot \leftarrow \ not\ wet(t),\ not\ \text{-}wet(t)$	2. $wet(t) \leftarrow \ not\ not\ inWater(t)$
3. $inWater(t+1) \leftarrow \ putInWater(t)$ $\{inWater(t+1)\} \leftarrow \ inWater(t)$ $\{\text{-}inWater(t+1)\} \leftarrow \ \text{-}inWater(t)$ $\{wet(t+1)\} \leftarrow \ wet(t)$ $\{\text{-}wet(t+1)\} \leftarrow \ \text{-}wet(t)$	4. $\{\text{-}inWater(0)\}$ $\{inWater(0)\}$ $\{\text{-}wet(0)\}$ $\{wet(0)\}$
5. $\{putInWater(t)\}$	

2.4.3 On the relation between programs lp_T and $simp_T$

Proposition 3 stated in this section is the key result of this part of the paper. Its proof outlines the essential steps that we take in arguing that two logic programs lp_T and $simp_T$ formalizing the action language $\mathscr{C}$ are essentially the same. The key claim of the proof is that logic program $lp_T(D)$ is a conservative extension of $simp_T(D)$. Here we only outline the proof, whereas the Appendix of the paper provides a complete proof.

The argument of this claim requires some close attention to groups of rules in the $lp_T(D)$ program. In particular, we establish by means of weak natural deduction that

- the rules in groups 1 and 2 of $lp_T(D)$ are strongly equivalent to the rules in groups 1 and 2 of $simp_T(D)$ and
- the rules in groups 1 and 4 of $lp_T(D)$ are strongly equivalent to the rules in groups 1 and 4 of $simp_T(D)$.

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

Similarly, we show that

- the rules in groups 1 and 3 of $lp_T(D)$ are strongly equivalent to the rules in group 1 of $simp_T(D)$ and the rules structurally similar to rules in group 3 of $simp_T(D)$ and yet not the same.

These arguments allow us to construct a program $lp'_T(D)$, whose answer sets are the same as those of $lp_T(D)$. Program $lp'_T(D)$ is a conservative extension of $simp_T(D)$ due to explicit definition rewriting. Proposition 1 helps us to uncover this fact.

Recall that the language of $simp_T(D)$ includes the action atoms – the action names of σ^{act} followed by (t) where $t = 0, \ldots T-1$. We denote the action atoms by σ_T^{act}.

Proposition 3
For a set X of atoms, X is an answer set for $simp_T(D)$ if and only if the set $X \cup \{\text{-}a \mid a \in \sigma_T^{act} \setminus X\}$ has the form

$$\left[\bigcup_{t=0}^{T-1} \{\widehat{l}(t) \mid l \in s_t \cup a_t\}\right] \cup \{\widehat{l}(T) \mid l \in s_T\}$$

for some path $\langle s_0, a_0, s_1, \ldots, s_{T-1}, a_{T-1}, s_T\rangle$ in the transition system described by D.

2.4.4 Additional concluding remarks: An interesting byproduct

Our work, which illustrates that logic programs $lp_T(D)$ and $simp_T(D)$ are essentially the same, also uncovers a precise formal link between the action description languages $\mathscr{C}$ and $\mathscr{C}+$. The authors of $\mathscr{C}+$ claimed that $\mathscr{C}$ is an immediate predecessor of $\mathscr{C}+$. Yet, to the best of our knowledge the exact formal link between the two languages has not been stated. Thus, earlier one could view $\mathscr{C}+$ as a generalization of $\mathscr{C}$ only *informally* alluding to the fact that $\mathscr{C}+$ allows the same intuitive interpretation of syntactic expressions of $\mathscr{C}$, while generalizing these to allow multivalued fluents in place of Boolean ones. These languages share the same syntactic constructs such as, for example, a dynamic law of the form

$$\textbf{caused } f_0 \textbf{ if } f_1 \wedge \cdots \wedge f_m \textbf{ after } a_1 \wedge \cdots \wedge a_n.$$

that we intuitively read as after the concurrent execution of actions $a_1 \ldots a_n$ the fluent expression f_0 holds in case if fluents expressions $f_1 \ldots f_m$ were the case at the time when aforementioned actions took place. Both languages provide interpretations for such expressions that meet our intuitions of this informal reading. Yet, if one studies the semantics of these languages, it is not trivial to establish a specific formal link between them. For example, the semantics of $\mathscr{C}+$ relies on the concepts of causal theories (Giunchiglia *et al.* 2004). The semantics of $\mathscr{C}$ makes no reference to these theories. Here we recall the translations of $\mathscr{C}$ and $\mathscr{C}+$ to logic programs, whose answer sets correspond to their key semantic objects. We then state the precise relation between the two by means of relating the relevant translations. In conclusion, $\mathscr{C}+$ can be viewed as a true generalization of the language $\mathscr{C}$ to the case of multivalued fluents.

3 Programs with variables

We now proceed toward the second part of the paper devoted to programs with variables. We start by presenting its detailed outline, a new running example and then stating

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

the preliminaries. We conclude with the formal statements on a number of rewriting techniques.

On the one hand, this part of the paper can be seen as a continuation of work by Eiter *et al.* (2006a), where we consider common program rewritings using a more complex dialect of logic programs. On the other hand, this part of the paper grounds the concept of program synonymity studied by Pearce and Valverde (2012) 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 part of the paper, we systematically study some common rewritings on first-order programs utilized by ASP practitioners. We use a running example to ground general theoretical presentation of this work into specific context. In particular, we consider two formalizations of a planning module given in Gelfond and Kahl (2014, Section 9):

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 $\mathscr{AL}$[4]. Gelfond and Kahl (2014) 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. *We note that no formal claim is provided for the* Plan-choice *program.* 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

- do not share the same signature, and
- 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 using their properties. Thus, the aforementioned formal claim about *Plan-disj* translates into the same claim for *Plan-choice*.

Here we use a dialect of the ASP language called RASPL-1 (Lee *et al.* 2008). 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 (Ferraris *et al.* 2011; 2009) using this operator. We state a sequence of formal results on programs rewritings and/or programs properties. Some of these results are geared by our running example and may not appear of great general interest. Yet, we view the proofs of these results as an interesting contribution as they showcase how arguments of correctness of rewritings can be constructed by the practitioners. Also, 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

[4] It is due to remark that although Gelfond and Kahl (2014) use the word "module" when encoding a planning domain, they utilize this term only informally to refer to a collection of rules responsible for formalizing "planning".

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

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 (Ben-Eliyahu and Dechter 1994) can be rewritten to nondisjunctive programs by means of a 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 Eiter *et al.* (2006a) and Janhunen *et al.* (2007), respectively. We also generalize so called Completion Lemma and Lemma on Explicit Definitions stated in Ferraris (2005), Ferraris and Lifschitz (2005) for the case of propositional theories and propositional logic programs. These generalizations are applicable to first-order programs. We conclude by applying the Lemma on Explicit Definitions proved here to argue the correctness of program rewriting system PROJECTOR (Hippen and Lierler 2019).

3.1 Running example and claims

This section presents two ASP formalizations of a domain-independent *planning module* given in Gelfond and Kahl (2014, 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 (Gelfond and Kahl 2014), 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

- utilizing auxiliary predicates non_o in place of $\neg o$; and
- introducing rule $\leftarrow o(A,I), non\_o(A,I)$.

This is a standard practice and ASP systems perform the same rewriting when processing classical negation symbol $\neg$ occurring in programs (in other words, symbol $\neg$ is treated as syntactic sugar).

Let

$$SG(I) \quad \text{abbreviate} \quad step(I),\ not\ goal(I),\ I \neq n,$$

where n is some integer specifying a limit on a length of an allowed plan. The first formalization called *Plan-choice* follows:

$$success \leftarrow goal(I),\ step(I).$$
$$\leftarrow not\ success.$$
$$\leftarrow o(A,I), non\_o(A,I) \tag{16}$$
$$non\_o(A,I) \leftarrow action(A),\ step(I),\ not\ o(A,I) \tag{17}$$
$$\{o(A,I)\} \leftarrow action(A),\ SG(I) \tag{18}$$
$$\leftarrow 2 \leq \#count\{A : o(A,I)\},\ SG(I). \tag{19}$$
$$\leftarrow not\ 1 \leq \#count\{A : o(A,I)\},\ SG(I). \tag{20}$$

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

One more remark is in order. In Gelfond and Kahl (2014), Gelfond and Kahl list only a single rule

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

in place of rules (18-20). Note that this single rule is an abbreviation for rules (18-20) (Gebser *et al.* 2015).

The second formalization that we call a *Plan-disj* encoding is obtained from *Plan-choice* by replacing rules (18-20) with the following:

$$o(A,I) \mid non\_o(A,I) \leftarrow action(A),\ SG(I) \tag{21}$$

$$\leftarrow o(A,I),\ o(A',I),\ action(A),\ action(A'),\ A \neq A' \tag{22}$$

$$sthHpd(I) \leftarrow o(A,I) \tag{23}$$

$$\leftarrow not\ sthHpd(I),\ SG(I). \tag{24}$$

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 the 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 $n \geq 0$. Expression $a_1,\ldots,a_m$ forms the positive part of a body. A *simple* rule has the 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

- in these rules all predicates occurring in their positive parts of the bodies must be at most i-*terminal* and
- at least one of these rules is such that some predicate occurring in its positive part of the body is i-*terminal* .

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

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 use considered theoretical results to illustrate the following *Claims*:

1. In the presence of rule (17) it is safe to add a rule

$$non\_o(A, I) \leftarrow not\ o(A, I),\ action(A),\ SG(I) \tag{25}$$

   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 (19) with rule

$$\leftarrow o(A, I),\ o(A', I),\ SG(I),\ A \neq A' \tag{26}$$

   within an arbitrary program.
3. In the presence of rules (16) and (17), it is safe to replace rule (18) with rule

$$o(A, I) \leftarrow not\ non\_o(A, I),\ action(A),\ SG(I) \tag{27}$$

   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 (18) with rule (21). The argument utilizes *Claims* 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 (19) with rule (22). The argument utilizes *Claim* 2, that is, it is safe to replace rule (19) with rule (26). An essential syntactic feature relies on Fact 1, and the facts that (i) rule (18) is the only one in *Plan-choice*, where predicate o occurs in the head; and (ii) rule (22) differs from (26) only in atoms that are part of the body of (18).
6. By Fact 4 and the fact that *sthHpd* does not occur in any other rule but (24) in *Plan-disj*, the answer sets of the program obtained by replacing rule (20) with rules (23) and (24) 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 *Claims* are sufficient to state 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

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

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.

3.2 Preliminaries: RASPL-1 logic programs, operator SM, strong equivalence

We now review a logic programming language RASPL-1 (Lee *et al.* 2008). 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.* (2008) on the roots of semantics of aggregates considered in RASPL-1.

An *aggregate expression* is an expression of the form

$$b \leq \#count\{\mathbf{x} : L_1, \ldots, L_k\} \tag{28}$$

($k \geq 1$), 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*. We call variables in $\mathbf{x}$ *aggregate* variables. 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

$$e_1, \ldots, e_m, not\ e_{m+1}, \ldots, not\ e_n \tag{29}$$

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

$$a_1 \mid \cdots \mid a_l \leftarrow Body \tag{30}$$

$$\{a_1\} \leftarrow Body \tag{31}$$

($l \geq 0$) where each a_i is an atom, and *Body* is the body in the form (29). When $l = 0$, we identify the head of (30) with symbol $\bot$ and call such a rule a *denial*. When $l = 1$, we call rule (30) a *defining* rule. We call rule (31) a *choice* rule. A *(logic) program* is a set of *rules*. An atom of the form $not\ t_1 = t_2$ is abbreviated by $t_1 \neq t_2$.

3.2.1 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.* (2011) 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

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

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 $\mathbf{p}$;[5]
- $(F \wedge G)^*$ 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 Ferraris *et al.* (2011, 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*.[6] Theorem 1 from (Ferraris *et al.* 2011) 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.

3.2.2 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 (28) as a conjunction of literals.

For an aggregate expressions of the form

$$b \leq \#count\{\mathbf{x} : F(\mathbf{x})\},$$

its FOL representation follows

$$\exists \mathbf{x^1} \cdots \mathbf{x^b}[\bigwedge_{1 \leq i \leq b} F(\mathbf{x^i}) \wedge \bigwedge_{1 \leq i < j \leq b} \neg(x^i = x^j)], \tag{32}$$

where $\mathbf{x^1} \cdots \mathbf{x^b}$ are lists of new variables of the same length as $\mathbf{x}$.

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

[6] An Herbrand interpretation of a signature σ (containing at least one object constant) is such that its universe is the set of all ground terms of σ, 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.

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

The FOL representations of logic rules of the form (30) and (31) are formulas

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

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

For example, expression $SG(I)$ stands for formula $step(I) \wedge \neg goal(I) \wedge \neg I = n$ and rules (18) and (20) in the *Plan-choice* encoding have the FOL representation:

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

$$\forall I\big(\neg\exists A[o(A,I)] \wedge SG(I) \rightarrow \bot\big). \tag{34}$$

The FOL representation of rule (19) is the universal closure of the following implication

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

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 Π 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 head H, a body *Body*, or a rule R, by $\widehat{H}$, $\widehat{Body}$, or $\widehat{R}$ we denote their FOL representations.) An *answer set* of Π is an answer set of its FOL representation $\widehat{\Pi}$. In other words, an *answer set* of Π is an Herbrand interpretation of $\widehat{\Pi}$ that is a $\pi(\widehat{\Pi})$-stable model of $\widehat{\Pi}$, that is, a model of

$$\mathrm{SM}_{\pi(\widehat{\Pi})}[\widehat{\Pi}]. \tag{35}$$

Sometimes, it is convenient to identify a logic program Π with its semantic counterpart (35) so that formal results stated in terms of SM operator immediately translate into the results for logic programs.

3.2.3 Review: Strong equivalence

We restate the definition of strong equivalence for first-order formulas given in Ferraris *et al.* (2011) 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). Lifschitz *et al.* (2007) 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. 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 Π_1 is *strongly equivalent* to logic program Π_2 when for any program Π,

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

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

It immediately follows that logic programs Π_1 and Π_2 are *strongly equivalent* if first-order formulas $\widehat{\Pi_1}$ and $\widehat{\Pi_2}$ are equivalent in logic of $\mathbf{SQHT}^{=}$.

We now review an important result about properties of denials.

Theorem 1 (*Theorem 3 (Ferraris et al. 2011)*)
For any first-order formulas F and G and arbitrary tuple $\mathbf{p}$ of predicate constants, $\mathrm{SM}_{\mathbf{p}}[F \wedge \neg G]$ is equivalent to $\mathrm{SM}_{\mathbf{p}}[F] \wedge \neg G$.

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 $\widetilde{\forall}(\mathit{Body} \rightarrow \bot)$ that is intuitionistically equivalent to formula $\neg\widetilde{\exists}\mathit{Body}$. 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 for first-order formulas (Mints 2000) stated below.

Proposition 4 (*Replacement Theorem II (Mints 2000), Section 13.1*)
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 $F \leftrightarrow F'$.

3.3 Rewritings

3.3.1 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. *Claims* 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 (17) subsumes rule (25).

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 *Claim* 1 holds. Indeed, rule (17) is strongly equivalent to the set of rules composed of itself and (25). Indeed, rule (17) subsumes rule (25).

Removing Aggregates: The following theorem is an immediate consequence of the Replacement Theorem II.

Theorem 2
Program

$$H \leftarrow b \leq \#count\{\mathbf{x} : F(\mathbf{x})\},\ G \tag{36}$$

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

is strongly equivalent to program

$$H \leftarrow \underset{1\leq i\leq b}{,} F(\mathbf{x}^{\mathbf{i}}) \quad \underset{1\leq i<j\leq b}{,} \quad x^i \neq x^j,\ G, \tag{37}$$

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

Theorem 2 shows us that *Claim* 2 is a special case of a more general fact. Indeed, take rules (19) and (26) to be the instances of rules (36) and (37), respectively.

We note that the Replacement Theorem II also allows us to immediately conclude the following.

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

Corollary 2 equips us with a general semantic condition that can be utilized in proving the syntactic properties of programs in spirit of Theorem 2.

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

Theorem 3
Program

$$\leftarrow p(\mathbf{x}),\ q(\mathbf{x}) \tag{38}$$
$$q(\mathbf{x}) \leftarrow not\ p(\mathbf{x}), F_1 \tag{39}$$
$$\{p(\mathbf{x})\} \leftarrow F_1,\ F_2 \tag{40}$$

is strongly equivalent to program composed of rules (38), (39) and rule

$$p(\mathbf{x}) \leftarrow not\ q(\mathbf{x}),\ F_1,\ F_2, \tag{41}$$

where F_1 and F_2 are the expressions of the form (29).

To illustrate the correctness of *Claim* 3 by Theorem 3: (i) take rules (16), (17), (18) be the instances of rules (38), (39), (40), respectively, and (ii) rule (27) be the instance of rule (41).

3.3.2 Useful rewritings using structure

In this subsection, 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 Ferraris *et al.* (2009). 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 (33), predicate constant o has a strictly positive occurrence in the consequence of the implication, whereas the same symbol o has a negated positive

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

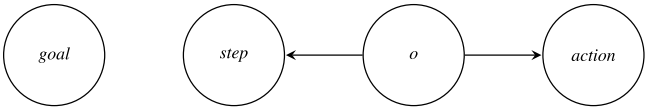

Fig. 2. The predicate dependency graph of a conjunction of formulas (33) and (34).

occurrence in the antecedent

$$\neg\neg o(A, I) \wedge step(I) \wedge \neg goal(I) \wedge \neg I = n \wedge action(A) \tag{42}$$

of (33). Predicate symbol *action* has a strictly positive non-negated occurrence in (42). The occurrence of predicate symbol *goal* is negated and not positive in (42). The occurrence of predicate symbol *goal* is negated and positive in (33).

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 (33) and (34), the FOL rules are as follows

$$\neg\neg o(A, I) \wedge SG(I) \wedge action(A) \rightarrow o(A, I) \tag{43}$$

$$\neg\exists A[o(A, I)] \wedge SG(I) \rightarrow \bot. \tag{44}$$

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, Figure 2 presents the dependence graph of a conjunction of formulas (33) and (34) relative to all its predicate symbols. It 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 (43) and (44) stemming from this conjunction. Predicate constant o has a strictly positive occurrence in the consequent $o(A, I)$ of the implication (43), whereas *action* and *step* are the only predicate constants in the antecedent $\neg\neg o(A, I) \wedge SG(I) \wedge action(A)$ of (43) that have positive and nonnegated occurrence in this antecedent. It is easy to see that a FOL rule of the form $G \rightarrow \bot$, for example, FOL rule (44), does not contribute edges to any dependency graph.

For any logic program Π, the *dependency graph* of Π, denoted $DG[\Pi]$, is a directed graph of $\widehat{\Pi}$ relative to the predicates occurring in Π. For example, let Π be composed of two rules (18) and (20). The conjunction of formulas (33) and (34) forms its FOL representation. Thus, Figure 2 captures its dependency graph $DG[\Pi]$.

Shifting We call a logic program *disjunctive* if all its rules have the form (30), where *Body* only contains atoms possibly preceded by *not*. We say that a disjunctive program is *normal* when it does not contain disjunction connective |. Gelfond *et al.* (1991) defined a mapping from a propositional disjunctive program Π to a propositional normal program by replacing each rule (30) with $l > 1$ in Π by l new rules

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

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

They showed that every answer set of the constructed program is also an answer set of Π. Although the converse does not hold in general, Ben-Eliyahu and Dechter (1994) showed that the converse holds if Π is "head-cycle-free". Linke *et al.* (2004) 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 (which also may contain 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 (30) with $l > 1$. By $shift_{\mathbf{p}}(R)$ (where $\mathbf{p}$ is a tuple of distinct predicates), we denote the rule

$$\bigm| \limits_{1 \leq i \leq l,\, a_i^0 \in \mathbf{p}} a_i \leftarrow Body, \qquad , \limits_{1 \leq j \leq l,\, a_j^0 \notin \mathbf{p}} not\ a_j. \tag{45}$$

Let $\mathscr{P}^R$ be a partition of the set composed of the distinct predicate symbols occurring in the head of rule R. By $shift_{\mathscr{P}^R}(R)$ we denote the set of rules composed of rule $shift_{\mathbf{p}}(R)$ for every member $\mathbf{p}$ of the partition $\mathscr{P}^R$ (order of the elements in $\mathbf{p}$ is immaterial).

For instance, if R_1 denotes a rule

$$a \mid b \mid c \mid d \mid e(1) \leftarrow \tag{46}$$

then ${\mathscr{P}_1}^{R_1} = \{\{a,b\},\{c,d,e\}\}$ and ${\mathscr{P}_2}^{R_1} = \{\{a,b\},\{c\},\{d,e\}\}$ form sample partitions of the described kind. Set $shift_{\mathscr{P}_1^{R_1}}(R_1)$ consists of rules

$$a \mid b \leftarrow not\ c,\ not\ d,\ not\ e(1)$$
$$c \mid d \mid e(1) \leftarrow not\ a,\ not\ b,$$

whereas set $shift_{\mathscr{P}_2^{R_1}}(R_1)$ consists of rules

$$a \mid b \leftarrow not\ c,\ not\ d,\ not\ e(1)$$
$$c \leftarrow not\ a,\ not\ b,\ not\ d,\ not\ e(1)$$
$$d \mid e(1) \leftarrow not\ a,\ not\ b,\ not\ c.$$

Theorem 4
Let Π be a logic program, $\mathbf{R}$ be a set of rules in Π of the form (30) with $l > 1$, and C be the set of strongly connected components in the dependency graph of Π. A program constructed from Π by replacing each rule $R \in \mathbf{R}$ with $shift_{\mathscr{P}^R}(R)$ has the same answer sets as Π if any partition $\mathscr{P}^R$ is such that there are no two distinct members $\mathbf{p}_1$ and $\mathbf{p}_2$ in $\mathscr{P}^R$ so that for some strongly connected component c in C, $c \cap \mathbf{p_1} \neq \emptyset$ and $c \cap \mathbf{p_2} \neq \emptyset$.[7]

Consider a sample program Π_{samp} composed of rule (46), which we denote as R_1, and rules

$$a \leftarrow b$$
$$b \leftarrow a. \tag{47}$$

[7] The statement of this theorem was suggested by Pedro Cabalar and Jorge Fandinno in personal communication on January 17, 2019.

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

The strongly connected components of program Π_{samp} are $\{\{a,b\},\{c\},\{d\},\{e(1)\}\}$. Theorem 4 tells us, for instance, that the answer sets of program Π_{samp} coincide with the answer sets of two distinct programs:

1. a program composed of rules $shift_{\mathscr{P}_1^{R_1}}(R_1)$ and rules (47);
2. a program composed of rules $shift_{\mathscr{P}_2^{R_1}}(R_1)$ and rules (47).

We now use Theorem 4 to argue the correctness of *Claim* 4. Let *Plan-choice*$'$ denote a program constructed from the *Plan-choice* encoding by replacing (18) with (21). Let *Plan-choice*$''$ denote a program constructed from the *Plan-choice*, by (i) replacing (18) with (27) and (ii) adding rule (25). Theorem 4 tells us that programs *Plan-choice*$'$ and *Plan-choice*$''$ have the same answer sets. Indeed,

1. take $\mathbf{R}$ to consist of rule (21) 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 dependency graph.

Due to *Claims* 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 *Claim* 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* (Ferraris *et al.* 2011) relative to the tuple/set $\mathbf{p}$ of predicate symbols if it is a conjunction of formulas of the form

$$\forall \mathbf{x}(G \rightarrow p(\mathbf{x})) \tag{48}$$

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 Ferraris *et al.* (2011) 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 $\mathbf{p}$, denoted by $Comp_{\mathbf{p}}[F]$, is obtained from F by replacing each conjunctive term of the form (48) with

$$\forall \mathbf{x}(G \leftrightarrow p(\mathbf{x})).$$

We now review an important result about properties of completion.

Theorem 5 (*Theorem 10 (Ferraris et al. 2011)*)
For any formula F in Clark normal form and arbitrary tuple $\mathbf{p}$ of predicate constants, formula

$$\mathrm{SM}_{\mathbf{p}}[F] \rightarrow Comp_{\mathbf{p}}[F]$$

is logically valid.

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

The following Corollary is an immediate consequence of this theorem, 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 $\mathbf{p}$ and H is a conjunction of formulas of the form $\widetilde{\forall}(K \rightarrow \bot)$, the implication

$$\mathrm{SM}_{\mathbf{p}}[G \wedge H] \rightarrow Comp_{\mathbf{p}}[G] \wedge H$$

is logically valid.

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

Theorem 6
For a program Π, a first-order formula F such that every answer set of Π 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.

Theorem 1 provides grounds for a straightforward argument for this statement.

Consider the *Plan-choice* encoding without denial (19) extended with any *Plan-instance*. We can partition it into two parts: one that contains the denials, denoted by Π_H, and the remainder, denoted by Π_G. Recall Fact 1. Following the steps described by Ferraris *et al.* (2011, 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 (33). The completion of this formula contains equivalence

$$\widetilde{\forall}\big(\neg\neg o(A,I) \wedge SG(I) \wedge action(A) \leftrightarrow o(A,I)\big). \tag{49}$$

By Corollary 3 it follows that any answer set of $\Pi_H \cup \Pi_G$ satisfies formula (49). It is easy to see that an interpretation satisfies (49) and the FOL representation of (26) if and only if it satisfies (49) and the FOL representation of denial (22). Thus, by Theorem 6 program $\Pi_H \cup \Pi_G$ extended with (26) and program $\Pi_H \cup \Pi_G$ extended with (22) have the same answer sets. Recall *Claim* 2 claiming that it is safe to replace denial (19) with denial (26) within an arbitrary program. It follows that program $\Pi_H \cup \Pi_G$ extended with (22) have the same answer sets $\Pi_H \cup \Pi_G$ extended with (19). This concludes the argument for the claim of *Claim* 5.

We now state the main formal results of the second part of the paper. The Completion Lemma for first-order programs stated next is essential in proving the Lemma on Explicit Definitions for first-order programs. *Claim* 6 follows immediately from the latter lemma.

Theorem 7 (*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 (48) of D no occurrence of an element in $\mathbf{q}$ occurs in G as positive and nonnegated. Formula

$$\mathrm{SM}_{\mathbf{pq}}[F \wedge D] \tag{50}$$

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

is equivalent to formulas

$$\mathrm{SM}_{\mathbf{pq}}[F \wedge D] \wedge Comp[D], \tag{51}$$

$$\mathrm{SM}_{\mathbf{p}}[F] \wedge Comp[D], \text{ and} \tag{52}$$

$$\mathrm{SM}_{\mathbf{pq}}[F \wedge \bigwedge_{q \in \{\mathbf{q}\}} \forall \mathbf{x}\big(\neg\neg q(\mathbf{x}) \rightarrow q(\mathbf{x})\big)] \wedge Comp[D]. \tag{53}$$

This result tells us that **pq**-stable models of $F \wedge D$ are such that they satisfy the classical first-order formula $Comp[D]$. These models also can be characterized as (i) the **p**-stable models of F that satisfy $Comp[D]$, and (ii) the **pq**-stable models of F extended with the counterpart of choice rules for member of **q** that satisfy $Comp[D]$.

For an interpretation I over signature Σ, by $I_{|\sigma}$ we denote the interpretation over $\sigma \subseteq \Sigma$ constructed from I so that every function or predicate symbol in σ is assigned the same value in both I and $I_{|\sigma}$. We call formula G in (48) a *definition* of $p(\mathbf{x})$.

Theorem 8 (*Lemma on Explicit Definitions*)
Let F be a first-order formula, **q** be a set of predicate constants that do not occur in F, and **p** be an arbitrary set of predicate constants in F. Let D be a formula in Clark normal form relative to **q** so that in every conjunctive term (48) of D there is no occurrence of an element in **q** in G. Then

i $M \mapsto M_{|\sigma(F)}$ is a 1-1 correspondence between the models of $\mathrm{SM}_{\mathbf{pq}}[F \wedge D]$ and the models $\mathrm{SM}_{\mathbf{p}}[F]$, and
ii $\mathrm{SM}_{\mathbf{pq}}[F \wedge D]$ and $\mathrm{SM}_{\mathbf{pq}}[F^{\mathbf{q}} \wedge D]$ 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 $q(\mathbf{x})$.
iii $\mathrm{SM}_{\mathbf{pq}}[F \wedge D]$ and $\mathrm{SM}_{\mathbf{pq}}[F'^{\mathbf{q}} \wedge D]$ are equivalent, where we understand $F'^{\mathbf{q}}$ as a formula obtained from F by replacing occurrences of any subformula of the definitions of $q(\mathbf{x})$ in D with $q(\mathbf{x})$.

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

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

and the program $p(\mathbf{y}) \leftarrow F(\mathbf{x}, \mathbf{y})$ are strongly equivalent. Thus, we can identify rule (23) in the *Plan-disj* encoding with the rule

$$sthHpd(I) \leftarrow 1 \leq \#count\{A : o(A, I)\}. \tag{54}$$

Using this fact and Theorem 8 allows us to support *Claim* 6. Take F to be the FOL representation of *Plan-choice* encoding extended with any *Plan-instance* and D be the FOL representation of (54), **q** be composed of a single predicate *sthHpd* and **p** be composed of all the predicates in *Plan-choice* and *Plan-instance*.

3.4 Projection

Harrison and Lierler (2016) considered a rewriting technique called projection. We start by reviewing their results. We then illustrate how the theory developed here is applicable in their settings. Furthermore, it allows us to generalize their results to more complex programs. In addition, our results give us a proof of correctness for system PROJECTOR (Hippen and Lierler 2019) that implements so called α- and β-projections.

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

Harrison and Lierler (2016) considered programs to be first-order sentence formed as a conjunction of formulas of the form

$$\widetilde{\forall}(a_{k+1} \wedge \cdots \wedge a_l \wedge \neg a_{l+1} \wedge \cdots \wedge \neg a_m \wedge \neg\neg a_{m+1} \wedge \cdots \wedge \neg\neg a_n \rightarrow a_1 \vee \cdots \vee a_k).$$

It is easy to see that the FOL-representation of RASPL-1 rule without aggregate expressions comply with this form. We will now generalize the main result by Harrison and Lierler (2016) to the case of RASPL-1 programs.

Recall how in Section 3.2.2 we identify the logical connective $\neg$ with its counterpart used in logic programs, namely, *not*. This allows us to call an expression *not* a, where a is an atom, a literal. To simplify the presentation of rewriting in this section we will treat $L_1, \ldots, L_k$ in (28) as a set of literals. We will also identify body (29) with the set $\{e_1, \ldots, e_m, not\ e_{m+1}, not\ e_n\}$ of its elements.

Let R be a RASPL-1 rule in a program Π, and let $\mathbf{x}$ be a nonempty tuple of variables occurring only in body of R outside of any aggregate expression. By $\alpha(\mathbf{x}, \mathbf{y})$ we denote a set of literals in the body of R so that it includes all literals in the body of R that contain at least one variable from $\mathbf{x}$. Tuple $\mathbf{y}$ denotes all the variables occurring in the literals of $\alpha(\mathbf{x}, \mathbf{y})$ different from $\mathbf{x}$. By α' we denote any subset of $\alpha(\mathbf{x}, \mathbf{y})$ whose literals do not contain any variables occurring in $\mathbf{x}$. By *Body* and H we denote the body and the head of R, respectively. Let u be a predicate symbol that does not occur in Π. Then a *result of projecting variables* $\mathbf{x}$ *out of* R *using predicate symbol* u consists of the following two rules

$$H \leftarrow (Body \setminus \alpha(\mathbf{x}, \mathbf{y})) \cup \alpha' \cup \{u(\mathbf{y})\}$$
$$u(\mathbf{y}) \leftarrow \alpha(\mathbf{x}, \mathbf{y}).$$

For example, one possible result of projecting Y out of

$$s(X, Z) \leftarrow p(Z), q(X, Y), r(X, Y), t(X) \tag{55}$$

using predicate symbol u is

$$s(X, Z) \leftarrow u(X), p(Z), t(X) \tag{56}$$
$$u(X) \leftarrow q(X, Y), r(X, Y). \tag{57}$$

Another possible result of projecting Y out of rule (55) using predicate symbol u consists of rule (56) and rule

$$u(X) \leftarrow q(X, Y), r(X, Y), t(X). \tag{58}$$

Yet, another possible result of projecting Y out of rule (55) using predicate symbol u consists of rule

$$s(X, Z) \leftarrow u(X), p(Z) \tag{59}$$

and rule (58).

We are now ready to state a formal result about projecting that is a generalization of Theorem 6 in Harrison and Lierler (2016).

Theorem 9

Let R be a RASPL-1 rule in a program Π, and let $\mathbf{x}$ be a nonempty tuple of variables occurring only in body of R outside of any aggregate expression and not in the head. If Π' is constructed from Π by replacing R in Π with a result of projecting variables $\mathbf{x}$

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

out of R using a predicate symbol u that is not in the signature of Π, then $M \mapsto M_{|\sigma(\widehat{\Pi})}$ is a 1-1 correspondence between the models of $\mathrm{SM}_{\mathbf{p},u}[\widehat{\Pi'}]$ and the models of $\mathrm{SM}_{\mathbf{p}}[\widehat{\Pi}]$.

This result on correctness of projection is immediate consequences of Lemma on Explicit Definitions presented here. We note that the proof of a more restricted statement by Harrison and Lierler (2016) is rather complex relying directly on the definition of SM operator. This illustrates the utility of presented theory, for example, Lemma on Explicit Definitions, as it equips ASP practitioners with a formal result that eases a construction of proofs of correctness of their rewritings.

Hippen and Lierler (2019) considered rewritings that they call α- and β-projections. They also implement these rewritings in system PROJECTOR. Both α- and β-projections are instances of the projection defined here. As a result, Theorem 9 provides a proof of correctness for the α- and β-projections.

Here we reproduce the definition of α-projection by Hippen and Lierler (2019, Section 2) for the case of *positive* rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m,$$

where a_0 is an atom or $\bot$ and $a_1, \ldots, a_m$ are atoms (we use the notation of this paper to reproduce the definition). Expression (55) exemplifies a positive rule. For a positive rule ρ and a set $\mathbf{x}$ of variables, by $alpha(\rho, \mathbf{x})$ we denote the set of all atoms in the body of ρ such that they contain *some* variable in $\mathbf{x}$. For instance, let ρ_1 be rule (55). Then,

$$alpha(\rho_1, \{Y\}) = \{q(X,Y), r(X,Y)\}$$
$$alpha(\rho_1, \{X,Y\}) = \{q(X,Y), r(X,Y), t(X)\}.$$

For a set $\mathbf{x}$ of variables and a positive rule ρ of the form $H \leftarrow Body$, where no variable in $\mathbf{x}$ occurs in H, the process of *α-projecting* $\mathbf{x}$ out of this rule will result in replacing it by two rules:

1. a rule
$$u(\mathbf{y}) \leftarrow alpha(\rho, \mathbf{x}).$$
so that
   - u is a fresh predicate symbol with respect to original program, and
   - $\mathbf{y}$ is composed of the variables that occur in $alpha(\rho, \mathbf{x})$, but not in $\mathbf{x}$;
2. a rule
$$H \leftarrow (Body \setminus alpha(\rho, \mathbf{x})) \cup \{u(\mathbf{y})\}.$$

For instance, replacing rule (55) with rules (56) and (57) exemplifies α-projection of Y. It is easy to see that α-projection on positive programs is an instance of projection rewriting studied here. The definitions of α and β-projections for general programs require substantially more notation. Thus, we refer an interested reader to the paper by Hippen and Lierler (2019, Section 2) for the details. Yet, it is still easy to see that these rewritings are instances of projection as defined here. For example, replacing rule (55) with rules (59) and (58) exemplifies β-projection.

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

4 Conclusions

We illustrated how the concepts of strong equivalence and conservative extensions can be used jointly to argue the correctness of a newly designed program or correctness of program rewritings. This work outlines a methodology for such arguments. Also, 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. For example, system PROJECTOR discussed in the last section implements projection rewritings for the sake of performance. In this work, we utilized the presented formal results to argue the correctness of this system.

Supplementary material

To view supplementary material for this article, please visit http://dx.doi.org/10.1017/S1471068421000545.

References

Babb, J. and Lee, J. 2013. Cplus 2asp: Computing action language c+ in answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, P. Cabalar and T. C. Son, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 122–134.

Ben-Eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence 12*, 53–87.

Bichler, M., Morak, M. and Woltran, S. 2016. lpopt: A rule optimization tool for answer set programming. In *Proceedings of International Symposium on Logic-Based Program Synthesis and Transformation*.

Brewka, G., Eiter, T. and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM 54(12)*, 92–103.

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

Eiter, T. and Fink, M. 2003. Uniform equivalence of logic programs under the stable model semantics. In *Logic Programming*, C. Palamidessi, Ed. Springer Berlin Heidelberg, Berlin, Heidelberg, 224–238.

Eiter, T., Fink, M., Tompits, H., Traxler, P. and Woltran, S. 2006a. Replacements in nonground answer-set programming. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

Eiter, T., Tompits, H. and Woltran, S. 2005. On solution correspondences in answer-set programming. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 97–102.

Eiter, T., Traxler, P. and Woltran, S. 2006b. An implementation for recognizing rule replacements in non-ground answer-set programs. In *Proceedings of European Conference On Logics In Artificial Intelligence (JELIA)*.

Erdoğan, S. T. and Lifschitz, V. 2004. Definitions in answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, V. Lifschitz and I. Niemelä, Eds. Springer-Verlag, 114–126.

Ferraris, P. 2005. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 119–131.

Ferraris, P., Lee, J. and Lifschitz, V. 2011. Stable models and circumscription. *Artificial Intelligence 175*, 236–263.

Ferraris, P., Lee, J., Lifschitz, V. and Palla, R. 2009. Symmetric splitting in the general theory of stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI press, 797–803.

Ferraris, P. and Lifschitz, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5*, 45–74.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V. and Schaub, T. 2015. Abstract gringo. *Theory and Practice of Logic Programming 15*, 449–463.

Gelfond, M. and Kahl, Y. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press.

Gelfond, M. and Lifschitz, V. 1998. Action languages (http://www.ep.liu.se/ea/cis/1998/016/). *Electronic Transactions on Artificial Intelligence 3*, 195–210.

Gelfond, M., Lifschitz, V., Przymusińska, H. and Truszczyński, M. 1991. Disjunctive defaults. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, J. Allen, R. Fikes and E. Sandewall, Eds., 230–237.

Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N. and Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence 153(1–2)*, 49–104.

Harrison, A. and Lierler, Y. 2016. First-order modular logic programs and their conservative extensions. In *Theory and Practice of Logic programming, 32nd Int'l. Conference on Logic Programming (ICLP) Special Issue*.

Hippen, N. and Lierler, Y. 2019. Automatic program rewriting in non-ground answer set programs. In *Proceedings of the 21st International Symposium on Practical Aspects of Declarative Languages (PADL)*.

Janhunen, T., Oikarinen, E., Tompits, H. and Woltran, S. 2007. Modularity aspects of disjunctive stable models. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 175–187.

Lee, J., Lifschitz, V. and 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.

Lee, J., Lifschitz, V. and Yang, F. 2013. Action language bc: Preliminary report. In *IJCAI*, 983–989.

Leite, J. 2017. A bird's-eye view of forgetting in answer-set programming. In *Logic Programming and Nonmonotonic Reasoning*, M. Balduccini and T. Janhunen, Eds. Springer International Publishing, Cham, 10–22.

Lierler, Y. 2019. Strong equivalence and program's structure in arguing essential equivalence between first-order logic programs. In *Proceedings of the 21st International Symposium on Practical Aspects of Declarative Languages (PADL)*.

Lifschitz, V. 2016. *Introduction to Mathematical Logic, Handout 4, Natural Deduction*. URL: https://www.cs.utexas.edu/users/vl/teaching/388L/h4.pdf. [Accessed on August 2017].

Lifschitz, V., Morgenstern, L. and Plaisted, D. 2008. Knowledge representation and classical logic. In *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz and B. Porter, Eds. Elsevier, 3–88.

Lifschitz, V., Pearce, D. and Valverde, A. 2001. Strongly equivalent logic programs. *ACM Transactions on Computational Logic 2*, 526–541.

https://doi.org/10.1017/S1471068421000545 Published online by Cambridge University Press

Lifschitz, V., Pearce, D. and Valverde, A. 2007. A characterization of strong equivalence for logic programs with variables. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 188–200.

Lifschitz, V., Tang, L. R. and Turner, H. 1999. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence 25*, 369–389.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *Proceedings of International Conference on Logic Programming (ICLP)*, P. Van Hentenryck, Ed. 23–37.

Lifschitz, V. and Turner, H. 1999. Representing transition systems by logic programs. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 92–106.

Linke, T., Tompits, H. and Woltran, S. 2004. On acyclic and head-cycle free nested logic programs. In *Proceedings of 19th International Conference on Logic Programming (ICLP)*. 225–239.

Lukasiewicz, J. 1941. Die logik und das grundlagenproblem. *Les Entries de Zürich sur les Fondaments et la Méthode des Sciences Mathématiques 12,* 6–9 (1938), 82–100.

Mints, G. 2000. *A Short Introduction to Intuitionistic Logic*. Kluwer.

Mints, G. 2010. Cut-free formulations for a quantified logic of here and there. *Annals of Pure and Applied Logic 162,* 3, 237–242. Special Issue: Dedicated to Nikolai Alexandrovich Shanin on the occasion of his 90th birthday.

Oetsch, J. and Tompits, H. 2008. Program correspondence under the answer-set semantics: The non-ground case. In *Logic Programming*, M. Garcia de la Banda and E. Pontelli, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 591–605.

Pearce, D. and Valverde, A. 2012. Synonymous theories and knowledge representations in answer set programming. *Journal of Computer and System Sciences 78,* 1, 86–104. JCSS Knowledge Representation and Reasoning.

Woltran, S. 2004. Characterizations for relativized notions of equivalence in answer set programming. In *Logics in Artificial Intelligence*, J. J. Alferes and J. Leite, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 161–173.

Woltran, S. 2008. A common view on strong, uniform, and other notions of equivalence in answer-set programming. *Theory and Practice of Logic Programming 8,* 2, 217234.