September, 2018

Strong Equivalence and Conservative Extensions Hand in Hand for Arguing Correctness of New Action Language C Formalization

Yuliya Lierler

Available at: https://works.bepress.com/yuliya_lierler/83/
Strong Equivalence and Conservative Extensions Hand in Hand for Arguing Correctness of New Action Language $C$ Formalization

Yuliya Lierler

Received: December 29, 2017 / Accepted: date

Abstract Answer set programming is a declarative programming paradigm used in formulating combinatorial search problems and implementing distinct knowledge representation and reasoning formalisms. It is common that several related and yet substantially different answer set programs exist for a given problem. Uncovering precise formal links between these programs is often of value. This paper develops a methodology for establishing such links. This methodology relies on the notions of strong equivalence and conservative extensions and a body of earlier theoretical work related to these concepts. We use distinct answer set programming formalizations of an action language $C$ and a syntactically restricted action language $C^+$ to showcase the results. As a by product we make the frequent claim occurring in the literature, namely, the action language $C$ is the immediate predecessor of $C^+$, mathematically precise. Indeed, we illustrate that an action description in $C^+$ limited to the syntactically allowed expressions by $C$ is also an action description in $C$ and the other way around.

Keywords answer set programming · action languages · essential equivalence

1 Introduction

Answer set programming 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 $B$ (Gelfond and Lifschitz, 1998, Section...
5), \( \mathcal{C} \) (Lifschitz and Turner, 1999), \( \mathcal{BC} \) (Lee et al, 2013), and \( \mathcal{C}^{+} \) (Giunchiglia et al, 2004; Babb and Lee, 2013).

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 tackles a problem domain by means of answer set programming it is a common practice to first develop a 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 assumption that the original encoding to a problem is correct how can we argue that a related and yet different encoding is also correct.

Lifschitz et al (2001) introduced the concept of strong equivalence between logic programs and demonstrated how it can be used in analysis of programs and their rewritings. Strong equivalence assumes the same signatures in related programs. Harrison and Lierler (2016) argued the importance of developing formal tools for the analysis of programs that are defined over distinct signatures. They introduced the notion of a logic program being a conservative extension of another one. In this paper we show how these concepts, strong equivalence and conservative extension, are of use in illustrating that two programs over different signatures and with significantly different structure are “essentially the same” in a sense that they capture solutions to the same problem.

We start by presenting the original formalization of action language \( \mathcal{C} \) 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 \( \mathcal{C} \) to a logic program \( l_{pT}(D) \) 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 language of logic programs has incorporated new features such as, for instance, choice rules. At present, these are commonly used by the practitioners of answer set programming. It is easy to imagine that in a modern formalization of action language \( \mathcal{C} \), given a system description \( D \) a resulting program will be different from the original \( l_{pT}(D) \). In fact, Babb and Lee (2013) present a translation of an action language \( \mathcal{C}^{+} \) (note how \( \mathcal{C} \) is said to be the immediate predecessor of \( \mathcal{C}^{+} \) (Giunchiglia et al, 2004, Section 7.3)) that utilizes modern language features such as choice rules. Here, we present this translation for the case of \( \mathcal{C} \). In particular, we restrict the language of \( \mathcal{C}^{+} \) to boolean, or two-valued, fluents (in general, \( \mathcal{C}^{+} \) permits multivalued fluents). We call this translation \( \text{simp}_{pT}(D) \). Although, \( l_{pT}(D) \) and \( \text{simp}_{pT}(D) \) share a lot in common they are substantially different. To begin with, the
signatures of these programs are not identical. Also, \( \text{simp}_T(D) \) utilizes choice rules. The programs \( \text{lp}_T(D) \) and \( \text{simp}_T(D) \) 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 \( \text{simp}_T(D) \) is "essentially the same" as program \( \text{lp}_T(D) \); to illustrate that the answer sets of \( \text{simp}_T(D) \) capture all the "histories" of length \( T \) in the transition system specified by \( D \) by relying

1. on the definitions of action language \( C \);
2. on the properties of programs \( \text{lp}_T(D) \) and \( \text{simp}_T(D) \) 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. 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.

Our work, which illustrates that logic programs \( \text{lp}_T(D) \) and \( \text{simp}_T(D) \) are essentially the same, also uncovers a precise formal link between the action description languages \( C \) and \( C^+ \). Although, the authors of \( C^+ \) claimed that \( C \) is an immediate predecessor of \( C^+ \), the exact formal link between the two languages has not been stated, to the best of our knowledge. Thus, prior one could view \( C^+ \) as a generalization of \( C \) only informally alluding to the fact that \( C^+ \) allows the same intuitive interpretation of syntactic expressions of \( C \), but generalizes 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

\[
\text{caused } f_0 \text{ if } f_1 \land \cdots \land f_m \text{ after } a_{m+1} \land \cdots \land a_n.
\]

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 \( C^+ \) relies on the concepts of causal theories (Giunchiglia et al, 2004). The semantics of \( C \) makes no reference to these theories. Here we recall the translations of \( C \) and \( C^+ \) 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, \( C^+ \)
can be viewed as a true generalization of the language $\mathcal{C}$ to the case of multi-valued fluents.

The paper is structured as follows. We start by reviewing action language $\mathcal{C}$ in Section 2. Section 3 presents the concepts of (i) a logic program, (ii) strong equivalence between logic programs, and (iii) a logic program being a conservative extension of another one. In particular, we review a weak natural deduction system that is later used to formally argue the strong equivalence between groups of logic rules. Section 4 introduces a rewriting technique based on what we call explicit definitions and illustrates its correctness. (This technique is 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.) In Section 5, we present an original, or gold standard, translation of language $\mathcal{C}$ to a logic program. Section 6 states a modern formalization stemming from the translation of a syntactically restricted $\mathcal{C}^+$. At last, in Section 6:relation we showcase how we can argue on the correctness of a modern formalization by illustrating the formal relation between the original and modern translations of language $\mathcal{C}$. We utilize reasoning by weak natural deduction and a formal result on explicit definition rewriting in this illustration. The paper concludes with the discussion of future work directions.

2 Review of Action Language $\mathcal{C}$

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

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

Syntactically, a $\mathcal{C}$ action description is a set of static and dynamic laws. Static laws are of the form
\[
\text{caused } l_0 \text{ if } l_1 \land \cdots \land l_m
\] (1)
and dynamic laws are of the form
\[
\text{caused } l_0 \text{ if } l_1 \land \cdots \land l_m \text{ after } l_{m+1} \land \cdots \land l_n
\] (2)
where
- $l_0$ is either a literal over $\sigma^{fl}$ or the symbol $\bot$,
- $l_i$ ($1 \leq i \leq m$) is a literal in $\sigma^{fl}$,
- $l_i$ ($m+1 \leq i \leq n$) is a literal in $\sigma$, and
- conjunctions $l_1 \land \cdots \land l_m$ and $l_{m+1} \land \cdots \land l_n$ are possibly empty and understood as $\top$.

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 $\sigma^H$ that satisfies implication $l_1 \land \cdots \land l_m \rightarrow l_0$ for every static law (1) 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 (1) from $D$ such that $s'$ satisfies $l_1 \land \cdots \land l_m$, or
-- the head of a dynamic law (2) from $D$ such that $s'$ satisfies $l_1 \land \cdots \land l_m$ and $s \cup a$ satisfies $l_{m+1} \land \cdots \land l_n$.

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 from (Lifschitz and Turner, 1999) that formalizes the effects of putting an object in water. We use this domain as a running example in the rest of the paper. It uses the fluent names inWater and wet and the elementary action name putInWater. In the notation introduced in (Gelfond and Lifschitz, 1998, Section 6), the action description for water domain follows

\[
\text{caused wet if inWater} \\
\text{putInWater causes inWater} \\
\text{inertial inWater, \neg inWater, wet, \neg wet}
\]

Written in full this action description contains six laws:

\[
\text{caused wet if inWater} \\
\text{caused inWater if } \top \text{ after putInWater} \\
\text{caused inWater if inWater after inWater} \\
\text{caused \neg inWater if } \neg inWater after \neg inWater \\
\text{caused wet if wet after wet} \\
\text{caused \neg wet if } \neg wet after \neg wet
\]

The corresponding transition system has 3 states:

\[
\neg inWater \neg wet, \neg inWater wet, inWater wet
\]

and 6 causally explained transitions

\[
\langle \neg inWater \neg wet, \neg putInWater, \neg inWater \neg wet \rangle, \\
\langle \neg inWater \neg wet, putInWater, inWater wet \rangle, \\
\langle \neg inWater wet, \neg putInWater, \neg inWater wet \rangle, \\
\langle inWater wet, putInWater, inWater wet \rangle, \\
\langle inWater wet, \neg putInWater, inWater wet \rangle, \\
\langle inWater wet, putInWater, inWater wet \rangle.
\]

We depict this transition system in Figure 1.

\footnote{\textsuperscript{1} We remark on the key word \textit{inertial} as 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.}
3 Traditional Logic Programs and their Equivalences

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

\[ a_0 \leftarrow a_1, \ldots, a_l, \text{not } a_{l+1}, \ldots, \text{not } a_m, \text{not not } a_{m+1}, \ldots, \text{not not } a_n, \]  

(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.

For the definition of an answer set of a traditional program we refer the reader to (Ferraris, 2005). For the purpose of this paper it is sufficient to say that an answer set is a set of atoms satisfying certain conditions. A program may have no or multiple answer sets.

According to (Ferraris and Lifschitz, 2005) and (Ferraris, 2005), rules of the form (4) 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

\[ p \leftarrow q, \text{not not } p. \]

We use choice rule notation in the sequel.

3.1 Strong Equivalence and "Weak" Natural Deduction

Traditional programs \( \Pi_1 \) and \( \Pi_2 \) are strongly equivalent (Lifschitz et al, 2001) when for every program \( \Pi \), programs \( \Pi_1 \cup \Pi \) and \( \Pi_2 \cup \Pi \) have the same answer sets. In addition to introducing the 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), an intermediate logic between classical and intuitionistic one.

We follow the steps of (Lifschitz et al, 2001) and identify a rule (4) with the
propositional formula
\[ a_1 \land \cdots \land a_l \land \neg a_{l+1} \land \cdots \land \neg a_m \land 
\neg \neg a_{m+1} \land \cdots \land \neg \neg a_n \rightarrow a_0. \]  

(Lifschitz et al, 2001) state that every formula provable in the natural deduction
system, where the axiom of the law of the excluded middle \((F \lor \neg F)\) is replaced by
the weak law of the excluded middle \((\neg F \lor \neg \neg F)\), is a theorem of HT. We call this
system weak natural deduction system. Since we use this observation in providing
formal arguments, we review the weak natural deduction system next. We denote this
system by \(N\). Its review follows the lines of (Lifschitz, 2016) to a large extent. For
another reference to natural deductions system we refer the reader to Lifschitz et al
(2008).

A sequent is an expression of the form
\[ \Gamma \Rightarrow F \]  
(“\(F\) under assumptions \(\Gamma\)”), where \(F\) is a propositional formula that allows connec-
tives \(\bot, \top, \neg, \land, \lor, \rightarrow\) and \(\Gamma\) is a finite set of formulas. If \(\Gamma\) is written as \(\{G_1, \ldots, G_n\}\),
we will drop the braces and write (6) as \(G_1, \ldots, G_n \Rightarrow F\). Intuitively, this sequent is
understood as the formula \((G_1 \land \cdots \land G_n) \rightarrow F\) if \(n > 0\), and as \(F\) if \(n = 0\).

The axioms of \(N\) are sequents of the forms
\[ F \Rightarrow F, \quad \Rightarrow \top, \quad \text{and} \quad \Rightarrow \neg F \lor \neg \neg F. \]

In the list of inference rules presented in Figure 2, \(\Gamma, \Delta, \Sigma\) are finite sets of
formulas, and \(F, G, H\) are formulas. The inference rules of \(N\) except for the two rules
at the last row are classified into introduction rules \((\cdot I)\) and elimination rules \((\cdot E)\); the exceptions are the contradiction rule \((C)\) and the weakening rule \((W)\).

\begin{align*}
(\land I) \quad & \Gamma \Rightarrow F \quad \Delta \Rightarrow G \\
\hline 
\Gamma, \Delta \Rightarrow F \land G \\
(\lor I) \quad & \Gamma \Rightarrow F \\
\Delta \Rightarrow G \\
\hline 
\Gamma \Rightarrow F \lor G \\
(\neg I) \quad & \Gamma \Rightarrow \bot \\
\hline 
\Gamma \Rightarrow \neg \bot \\
(\land E) \quad & \Gamma \Rightarrow F \land G \\
\hline 
\Gamma \Rightarrow F \\
(\lor E) \quad & \Gamma \Rightarrow F \lor G \\
\Delta \Rightarrow H \\
\hline 
\Gamma, \Delta \Rightarrow H \\
(\neg E) \quad & \Gamma \Rightarrow F, \Delta \Rightarrow \neg F \\
\hline 
\Gamma \Rightarrow \neg F \\
(C) \quad & \Gamma \Rightarrow \bot \\
\hline 
\Gamma \Rightarrow \bot \\
(W) \quad & \Gamma \Rightarrow H \\
\hline 
\Gamma, \Delta \Rightarrow H \\
\end{align*}

Fig. 2 Inference rules of system \(N\).
A proof/derivation is a list of sequents \( S_1, \ldots, S_n \) such that each \( S_i \) is either an axiom or can be derived from some of the sequents in \( S_1, \ldots, S_{i-1} \) by one of the inference rules. To prove a sequent \( S \) means to find a proof with the last sequent \( S \). To prove a formula \( F \) means to prove the sequent \( \Rightarrow F \).

The De Morgan’s law

\[ \lnot (F \lor G) \leftrightarrow \lnot F \land \lnot G \]

is provable intuitionistically (where we understand formula \( H \leftrightarrow H' \) as an abbreviation for \( (H \Rightarrow H') \land (H' \Rightarrow H) \)). Thus, formulas \( \lnot (F \lor G) \) and \( \lnot F \land \lnot G \) are intuitionistically equivalent. The other De Morgan’s law

\[ \lnot (F \land G) \leftrightarrow \lnot F \lor \lnot G \]

is such that its one half is provable intuitionistically, while the other one is provable in HT (thus, formulas \( \lnot (F \land G) \) and \( \lnot F \lor \lnot G \) are equivalent in HT-logic). We illustrate the latter fact in Figure 3 using system \( \mathbf{N} \). In other words, we prove sequent \( \Rightarrow \lnot (F \land G) \rightarrow \lnot F \lor \lnot G \) in \( \mathbf{N} \). It is convenient to introduce abbreviations for the assumptions used in the proofs so that \( A_1 \) abbreviates assumption \( \lnot (F \land G) \) in Figure 3.

\begin{align*}
1. & \Rightarrow \lnot F \lor \lnot F & \text{axiom} \\
A_1. & \lnot (F \land G) & \text{axiom} \\
2. & A_1 \Rightarrow \lnot (F \land G) & \text{axiom} \\
3. & G \Rightarrow G & \text{axiom} \\
4. & F \Rightarrow F & \text{axiom} \\
5. & F, G \Rightarrow F \land G & (\land I) 3,4 \\
6. & A_1, F, G \Rightarrow \bot & (\lnot E) 2,5 \\
7. & A_1, G \Rightarrow \lnot F & (\lnot I) 6 \\
8. & \lnot \lnot F \Rightarrow \lnot F & \text{axiom} \\
9. & A_1, G, \lnot F \Rightarrow \bot & (\lnot I) 7,8 \\
10. & A_1, \lnot F \Rightarrow \lnot G & (\lnot I) 9 \\
11. & A_1, \lnot F \Rightarrow \lnot F \lor \lnot G & (\lor I) 10 \\
12. & \lnot F \Rightarrow \lnot F & \text{axiom} \\
13. & \lnot F \Rightarrow \lnot F \lor \lnot G & (\lor I) 12 \\
14. & A_1 \Rightarrow \lnot F \lor \lnot G & (\lor E) 1, 11, 13 \\
15. & \Rightarrow \lnot (F \land G) \rightarrow (\lnot F \lor \lnot G) & (\rightarrow I) 14
\end{align*}

**Fig. 3** Prove of sequent \( \Rightarrow \lnot (F \land G) \rightarrow \lnot F \lor \lnot G \) in system \( \mathbf{N} \).

It is easy to show that the propositional formulas \( F \rightarrow \bot \) and \( \lnot F \) are equivalent using \( \mathbf{N} \), so that in the sequel we often identify rules of the form

\[ a_1 \land \ldots \land a_l \land \lnot a_{l+1} \land \ldots \land \lnot a_m \land \lnot \lnot a_{m+1} \land \ldots \land \lnot \lnot a_n \rightarrow \bot \]

with the propositional formula

\[ \lnot (a_1 \land \ldots \land a_l \land a_{l+1} \land \ldots \land a_m \land \lnot a_{m+1} \land \ldots \land \lnot a_n). \]

### 3.2 Conservative Extensions

Harrison and Lierler (2016) defined the notion of a conservative extension for the case of logic programs. This concept is related to strong equivalence for 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 $\Pi$, by $\text{atoms}(\Pi)$ we denote the set of atoms occurring in $\Pi$. Let $\Pi$ and $\Pi'$ be programs such that $\text{atoms}(\Pi) \subseteq \text{atoms}(\Pi')$. We say that program $\Pi'$ is a conservative extension of $\Pi$ if $X \rightarrow X \cap \text{atoms}(\Pi)$ is a 1-1 correspondence between the answer sets of $\Pi'$ and the answer sets of $\Pi$. For instance, program
\[
\begin{align*}
\neg q & \rightarrow p \\
\neg p & \rightarrow q
\end{align*}
\]
is a conservative extension of program $\{p\}$. Furthermore, given program $\Pi$ such that (i) it contains rule $\{p\}$ and (ii) $q \not\in \text{atoms}(\Pi)$, a program constructed from $\Pi$ by replacing $\{p\}$ with (7) is a conservative extension of $\Pi$.

4 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
\[
a_1 \land \cdots \land a_l \land \neg a_{l+1} \land \cdots \land \neg a_m \land \neg \neg a_{m+1} \land \cdots \land \neg \neg a_n,
\]
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.

Let $\Pi$ be a program, $Q$ be a set of atoms that do not occur in $\Pi$. For an atom $q \in Q$, let $\text{def}(q)$ denote a basic conjunction (8) where $a_i$ ($1 \leq i \leq n$) in $\text{atoms}(\Pi)$. We say that $\text{def}(q)$ is an explicit definition of $q$ in terms of $\Pi$. By $\text{def}(Q)$ we denote a set of formulas $\text{def}(q)$ for each atom $q \in Q$. We assume that all these formulas are distinct. Program $\Pi[Q,\text{def}(Q)]$ is constructed from $\Pi$ as follows:

- all occurrences of all formulas $\text{def}(q)$ from $\text{def}(Q)$ in some body of $\Pi$ are replaced by respective $q$,
- for every atom $q \in Q$ a rule of the form
  \[
  \text{def}(q) \rightarrow q
  \]
  is added to the program.

For instance, let $\Pi$ be a program
\[
\begin{align*}
\neg q & \rightarrow p \\
\neg p & \rightarrow q
\end{align*}
\]
then $\Pi[\{r\}, \{\text{def}(r) : \neg q\}]$ follows
\[
\begin{align*}
r & \rightarrow p \\
\neg q & \rightarrow r
\end{align*}
\]
The proposition below supports the fact that the latter program is a conservative extension of the former.
Proposition 1  Let $\Pi$ be a program, $Q$ be a set of atoms that do not occur in $\Pi$, and $\text{def}(Q)$ be a set composed of explicit definitions for each element in $Q$ in terms of $\Pi$. Program $\Pi[Q, \text{def}(Q)]$ is a conservative extension of $\Pi$.

To prove Proposition 1 the following proposition is of use.

Proposition 2 (Replacement Theorem in (Mints, 2000), Section 2.8)  If $F$ is a formula containing a subformula $G$ and $F'$ is the result of replacing that subformula by $G'$ then $G \leftrightarrow G'$ intuitionistically implies $F \leftrightarrow F'$.

To rely on formal results stated earlier in the literature, we now consider the case of programs that are more general than traditional logic programs. We call such programs definitional. In other words, traditional programs are their special case. A definitional program consists of rules of the form (5) (recall that we identify rule (4) with the propositional formula (5)) and rules of the form $a \rightarrow F$, where $a$ is an atom and $F$ is a basic conjunction. If a program contains two rules $F \rightarrow a$ and $a \rightarrow F$ we abbreviate that by a single expression $F \leftrightarrow a$. A definitional program is a special case of propositional theories presented in (Ferraris, 2005). We understand the notion of answer sets for such programs as presented in that work. We note that (Ferraris, 2005) generalized main results about strong equivalence from (Lifschitz et al, 2001) to the case of propositional theories. Thus, if two definitional programs are equivalent in HT-logic then these programs are strongly equivalent.

Proof (Proposition 1) By $\Pi'$ we denote a program constructed from $\Pi$ by adding a rule $\text{def}(q) \rightarrow q$ for every atom $q \in Q$. By Proposition 4 from (Ferraris, 2005), $\Pi'$ is a conservative extension of $\Pi$. By Proposition 5 from (Ferraris, 2005), traditional program $\Pi'$ has the same answer sets as the definitional program $\Pi''$ constructed from $\Pi'$ by replacing a rule $\text{def}(q) \rightarrow q$ with a rule $\text{def}(q) \leftrightarrow q$. Similarly, traditional program $\Pi[Q, \text{def}(Q)]$ has the same answer sets as the definitional program $\Pi[Q, \text{def}(Q)]'$ constructed from it by replacing a rule $\text{def}(q) \rightarrow q$ with a rule $\text{def}(q) \leftrightarrow q$. By Replacement Theorem, $\Pi''$ and $\Pi[Q, \text{def}(Q)]'$ are strongly equivalent.

5 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 $\text{lpt}(D)$ parametrized with a positive integer $T$ that intuitively represents a time horizon. The remarkable property of logic program $\text{lpt}(D)$ that its answer sets correspond to "histories" – path of length $T$ in the transition system described by $D$.

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

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

and

$$-\sigma^{act} = \{ \neg a \mid a \in \sigma^{act} \}.$$
For a literal \( l \), we define

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

and

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

The language of \( \mathit{lp}_T(D) \) has atoms of four kinds:

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

Program \( \mathit{lp}_T(D) \) consists of the following rules:

1. for every atom \( a \) that is a fluent or action atom of the language of \( \mathit{lp}_T(D) \)

\[
\bot \leftarrow a, \neg a
\]

and

\[
\bot \leftarrow \neg a, \neg \neg a
\]

2. for every static law (1) in \( D \), the rules

\[
\widehat{l}_0(t) \leftarrow \neg \widehat{l}_1(t), \ldots, \neg \widehat{l}_m(t)
\]

for all \( t = 0, \ldots, T \) (we understand \( \widehat{l}_0(t) \) as \( \bot \) if \( l_0 \) is \( \bot \)),

3. for every dynamic law (2) in \( D \), the rules

\[
\widehat{l}_0(t + 1) \leftarrow \neg \widehat{l}_1(t + 1), \ldots, \neg \widehat{l}_m(t + 1), \widehat{l}_{m+1}(t), \ldots, \widehat{l}_n(t),
\]

for all \( t = 0, \ldots, T - 1 \),

4. the rules

\[
\neg a(0) \leftarrow \neg a(0), \quad a(0) \leftarrow \neg \neg a(0),
\]

for all fluent names \( a \) in \( \sigma^f \) and

5. for every atom \( a \) that is an action atom of the language of \( \mathit{lp}_T(D) \) the rules

\[
\neg a \leftarrow \neg a, \quad a \leftarrow \neg \neg a.
\]

**Proposition 3 (Proposition 1 in (Lifschitz and Turner, 1999))** For a set \( X \) of atoms, \( X \) is an answer set for \( \mathit{lp}_T(D) \) 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 \( (s_0, a_0, s_1, \ldots, s_{T-1}, a_{T-1}, s_T) \) 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 $-a(i)$ (where $-a(i)$ intuitively stands for literal $\neg a(i)$), while the additional constraints have the form (9).

The translation of $\mathcal{C}$ action description (3) consists of all rules of the form

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

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

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

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

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

6 Simplified Modern Translation

As in 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 $\text{simpl}_T(D)$ inspired by $lp_T(D)$ and the advances in answer set programming languages. The main property of logic program $\text{simpl}_T(D)$ 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 $\mathcal{C}+$ that is limited to two-valued fluents.

The language of $\text{simpl}_T(D)$ 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

$$\tilde{l} = \begin{cases} 
\overline{a} & \text{if } l \text{ is a literal of the form } \neg a, \text{ where } a \in \sigma^{\text{act}} \\
\overline{l} & \text{otherwise}
\end{cases}$$

Program $\text{simpl}_T(D)$ consists of the following rules:

1. for every fluent atom $a$ the rules of the form (9) and (10),
2. for every static law (1) in $D$, $\text{simp}_T(D)$ contains the rules of the form

\[ \hat{l}_0(t) \leftarrow \text{not not } \hat{l}_1(t), \ldots, \text{not not } \hat{l}_m(t) \]  

(14)

for all $t = 0, \ldots, T^2$.

3. for every dynamic law (2) in $D$, the rules

\[ \hat{l}_0(t+1) \leftarrow \text{not not } \hat{l}_1(t+1), \ldots, \text{not not } \hat{l}_m(t+1), \]

\[ \hat{l}_{m+1}(t), \ldots, \hat{l}_n(t), \]

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

4. the rules

\[
\begin{align*}
\{\neg a(0)\} & \\
\{a(0)\} &
\end{align*}
\]  

(15)

for all fluent names $a$ in $\sigma^f$ and

5. for every atom $a$ that is an action atom of the language of $lpr(T(D)$, the choice rules $\{a\}$.

The $\text{simp}_T$ translation of $C$ action description (3) consists of all rules of the form

\begin{itemize}
  \item 1. $\bot \leftarrow \text{inWater}(t), \neg \text{inWater}(t)$
  \item 2. $\neg \neg \text{wet}(t) \leftarrow \neg \neg \neg \text{wet}(t)$
  \item 3. $\text{inWater}(t+1) \leftarrow \text{putInWater}(t)$
  \item 4. $\{\neg \text{inWater}(t)\}$
  \item 5. $\{\neg \text{wet}(t)\}$
\end{itemize}

$\text{putInWater}(t)$

---

$^2$ Babb and Lee (2013) allow rules with arbitrary formulas in their bodies so that in place of (14) they consider rule $\hat{l}_0(t) \leftarrow \neg \neg \neg (\hat{l}_1(t) \land \cdots \land \hat{l}_m(t))$. 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).

$^3$ Language $C$ assumes every action to be exogenous, whereas this is not the case in $C+$, 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.
7 On the Relation Between Programs $lp_T$ and $simp_T$

Proposition 4 stated in this section is the main result 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 $C$ are essentially the same. The key claim of the proof is that logic program $lp_T(D)$ is a conservative extension of $simp_T(D)$.

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

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

Similarly, we show that

- the rules in group 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 these 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.

It is currently difficult to state precisely how the described analysis can be automated. Yet, some key points are identified. For example, if to follow the outlines method a theorem proving engine for automatic weak natural deduction is of an essence. It is also obvious that in the presented analysis, we at multiple times focused on the relation of parts of programs (i.e., we claimed the relation between the rules in group 1 and 2 of $lp_T(D)$ and the rules in group 1 and 2 of $simp_T(D)$ and so force). Thus, developments in modular formalisms of logic programs may have an important contribution in systematic program analysis. At last, the development of a portfolio of formal results for common program rewritings may prove to be essential. For example, here we presented Proposition 1 that allows us to talk about common explicit definition rewriting. Harrison and Lierler (2016) present a formal result about a projection rewriting. A portfolio of such results will aid in the formal analysis of program’s relations. At present, we believe that the detailed proof of Proposition 4 is the right step in the direction of equipping ASP developers with the portfolio of tools in the formal analysis of the variants of programs they develop for an application at hand.

We believe that the detailed description of all the steps required in establishing the exact relation between the $lp_T(D)$ and $simp_T(D)$ programs is

We now turn our attention to formal statement of the main result of this paper. Recall that the language of $simp_T(D)$ includes the action atoms—the action names of $\sigma^{act}$ followed by $(t)$ where $t = 0, \ldots, T - 1$. We denote the action atoms by $\sigma^{act}_T$.

**Proposition 4** For a set $X$ of atoms, $X$ is an answer set for $simp_T(D)$ if and only if set $X \cup \{-a \mid a \in \sigma^{act}_T \setminus X\}$ has the form

$$\left[\bigcup_{t=0}^{T-1} \{\hat{t}(t) \mid l \in s_t \cup a_t\}\right] \cup \{\hat{t}(T) \mid l \in s_T\}$$
We allow ourselves a freedom to use De Morgan’s Laws as if they were given to us.

Proof
We illustrate the proof in $N$.

Lemma 1 If $F$ is a formula containing a subformula $G$ and $F'$ is the result of replacing that subformula by $G'$ then the sequent

\[ \Gamma \Rightarrow (G \leftrightarrow G') \rightarrow (F \leftrightarrow F') \]

is provable in $N$, where $\Gamma$ is arbitrary set of assumptions.

Proof Trivially follows from the Replacement Theorem stated as Proposition 2 here.

Lemma 2 The sequent

\[ \neg(F \land G), \neg(\neg F \land \neg G) \Rightarrow \neg F \leftrightarrow \neg G \land \neg G \leftrightarrow \neg F \]

is provable in $N$.

Proof We illustrate the proof in $N$ for the sequent

\[ \neg(F \land G) \Rightarrow \neg F \land \neg G \]

We allow ourselves a freedom to use De Morgan’s Laws as if they were given to us as additional inference rules in $N$.

\begin{align*}
A_1. & \quad \neg (F \land G) \\
1. & \quad A_1 \Rightarrow \neg (F \land G) \quad \text{axiom} \\
2. & \quad A_1 \Rightarrow \neg F \lor \neg G \quad \text{De Morgan’s Law 1} \\
3. & \quad \neg F \Rightarrow \neg F \quad \text{axiom} \\
4. & \quad \neg G \Rightarrow \neg G \quad \text{axiom} \\
5. & \quad \neg \neg F \Rightarrow \neg \neg F \quad \text{axiom} \\
6. & \quad \neg \neg F, \neg F \Rightarrow \bot \quad (\neg \text{E}) 3, 5 \\
7. & \quad \neg \neg F, \neg F \Rightarrow \neg G \quad (C) 6 \\
8. & \quad A_1, \neg \neg F \Rightarrow \neg G \quad (\lor \; \text{E}) 2, 4, 7 \\
9. & \quad A_1 \Rightarrow \neg \neg F \rightarrow \neg G (\rightarrow \; \text{I}) 8
\end{align*}
Similar proofs in structure are available for the sequents
\[ \neg(F \land G) \Rightarrow \neg\neg G \Rightarrow \neg F, \]
\[ \neg(\neg F \land \neg G) \Rightarrow F \Rightarrow \neg\neg G, \text{ and} \]
\[ \neg(\neg F \land \neg G) \Rightarrow G \Rightarrow \neg\neg F. \]

Several applications of \((\land I)\) will allow us to conclude the proof in \(N\) for the sequent in the statement of this lemma.

**Proof (Proposition 4)** It is easy to see that the signatures of \(\text{simp}_T(D)\) and \(\text{lp}_T(D)\) differ by complement action atoms present in \(\text{lp}_T(D)\). What we show next is the fact that \(\text{lp}_T(D)\) is a conservative extension of \(\text{simp}_T(D)\). Then the claim of this proposition follows from Proposition 3.

**Claim 1:** The set of rules from groups 1 and 3 of \(\text{lp}_T(D)\) are strongly equivalent to the set of rules from group 1 of \(\text{lp}_T(D)\) and the rules
\[ \hat{l}_0(t+1) \leftarrow \text{not not} \hat{l}_i(t+1), \ldots, \text{not not} \hat{l}_m(t+1), \]
\[ l_{m+1}(t), \ldots, l_n(t), \]
(16)
for all \(t = 0, \ldots, T - 1\), for every dynamic law (2) in \(D\).

It is easy to see that these sets of rules only differ in structure of rules (12) and (16) so that the atoms of the form \(\hat{l}_i(t + 1)\) \((1 \leq i \leq m)\) in (12) are replaced by the expressions of the form \(\text{not not} \hat{l}_i(t + 1)\) in (16).

Let \(\Gamma\) denote the set of rules from group 1 of \(\text{lp}_T(D)\). Using Lemmas 1 and 2 it is easy to see that the sequent 1 presented in Figure 4 is provable in \(N\). It is easy to construct a proof in \(N\) from this sequent 1 to the sequent 2 in the same figure. This immediately concludes the proof of Claim 1.

**Claim 2:** The set of rules from groups 1 and 2 of \(\text{lp}_T(D)\) are strongly equivalent to the set of rules from group 1 and 2 of \(\text{simp}_T(D)\).

The proof for this claim follows the lines of a proof for Claim 1.

**Claim 3:** The set of rules from groups 1 and 4 of \(\text{lp}_T(D)\) are strongly equivalent to the set of rules from group 1 and 4 of \(\text{simp}_T(D)\).

The proof for this claim is similar to that of a proof of Claim 1.

Due to Claims 1, 2, and 3, it follows that \(\text{lp}_T(D)\) has the same answer sets as the program \(\text{lp}'_T(D)\) constructed from \(\text{lp}_T(D)\) by replacing (i) the rules from group 3 with rules (16) for all \(t = 0, \ldots, T - 1\), for every dynamic law (2) in \(D\), and (ii) the rules from groups 2 and 4 in \(\text{lp}_T(D)\) by the rules from groups 2 and 4 in \(\text{simp}_T(D)\).

It is easy to see that \(\text{lp}'_T(D)\) coincides with the program
\[
\text{simp}_T(D)[\{-a \mid a \in \sigma_T^{\text{act}}\},
\{\text{def}(\neg a) : \neg a \in \sigma_T^{\text{act}}\}].
\]

By Proposition 1, program \(\text{lp}'_T(D)\) is a conservative extension of \(\text{simp}_T(D)\). Consequently, \(\text{lp}_T(D)\) is a conservative extension of \(\text{simp}_T(D)\).
8 Conclusions and Future Work

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’s rewritings. This work outlines a methodology for such arguments. In the future we will generalize this methodology beyond propositional programs as it will be a substantial milestone for many answer set programming applications. We will also look into incorporating more sophisticated features of answer set programs such as rules with aggregates.

Acknowledgements We are grateful to Vladimir Lifschitz for valuable discussions related to this work and his comments on the original draft. This work was partially supported by the University of Nebraska Omaha under ORCA Summer 2016 grant and by National Science Foundation under grant IIS-1707371.

References


