Yet Another Proof of the Strong Equivalence Between Propositional Theories and Logic Programs

Joohyung Lee and Ravi Palla

School of Computing and Informatics
Arizona State University, Tempe, AZ, USA
{joolee, Ravi.Palla}@asu.edu

Abstract. Recently, the stable model semantics was extended to the syntax of arbitrary propositional formulas, which are beyond the traditional rule form. Cabalar and Ferraris, as well as Cabalar, Pearce, and Valverde, showed that any propositional theory under the stable model semantics can be turned into a logic program. In this note, we present yet another proof of this result. Unlike the other approaches that are based on the logic of here-and-there, our proof uses familiar properties of classical logic. Based on this idea, we present a prototype implementation for computing stable models of propositional theories using the answer set solver DLV. We also note that every first-order formula under the stable model semantics is strongly equivalent to a prenex normal form whose matrix has the form of a logic program.

1 Introduction

Recently, the stable model semantics was extended to the syntax of arbitrary propositional formulas, which are beyond the traditional rule form [1, 2]. Ferraris [2] showed that nonmonotone aggregates can be naturally expressed in the extended syntax. On the other hand, Cabalar and Ferraris [3] showed that every propositional theory under the stable model semantics is strongly equivalent [4] to a logic program. They provided two proofs based on the logic of here-and-there, one by syntactic transformation, and the other by constructing a logic program using countermodels of the theory. An approach similar to the first proof was taken in [5], where the authors presented a set of rules for rewriting a propositional theory into a disjunctive logic program. These rules are an extension of the rules for turning a program with nested expressions into a logic program [6], which led to an implementation NLP [7]. The system is essentially a preprocessor to the answer set solver DLV [1] for handling programs with nested expressions.

In this note, we present yet another proof of the theorem on strong equivalence between propositional theories and logic programs. Unlike the other approaches that are based on the logic of here-and-there, our proof is based on an operator that characterizes strong equivalence in terms of classical logic, using an extended signature with two groups of atoms, the original one corresponding to the "there" world, and a group of newly introduced atoms referring to the "here" world. This not only shows that the reduction is possible, but also tells us how to *generate* strongly equivalent logic programs based on equivalence in classical logic.

[1] http://www.dbai.tuwien.ac.at/proj/dlv/ .

The reduction idea has led us to develop a prototype implementation, which we call F2LP,[2] that computes the stable models of an arbitrary propositional theory. Similar to NLP, the system turns a propositional theory into a disjunctive logic program and calls DLV.

We also apply the reduction idea to first-order formulas under the new definition of stable model semantics, recently proposed in [8]. We show that any first-order theory under the stable model semantics is strongly equivalent to a prenex normal form whose matrix has the form of a logic program. Thus the syntactic difference of arbitrarily nested connectives and quantifiers is not essential between the language proposed in [8] and logic programs. On the other hand, since the prenex normal form may contain existential quantifiers, it is different from a logic program, where all variables are assumed to be universally quantified.

In the next section, we review the definition of stable models for arbitrary propositional formulas as well as the definition of strong equivalence between propositional formulas, and present how to find a logic program that is strongly equivalent to a given formula. In Section 3, we present a simpler transformation, and in Section 4, we extend the reduction idea to arbitrary first-order formulas and note that every first-order theory is strongly equivalent to a prenex normal form. In Section 5, we present a prototype implementation of computing the stable models of propositional theories.

2 Reducing propositional formulas to logic programs

We first review the definition of a stable model proposed in [8], by restricting attention to the propositional case. This definition is essentially the same as the encoding of formulas of equilibrium logic by quantified Boolean formulas given in [9], and is equivalent to the fixpoint definition of a stable model proposed in [2].

Let F be a propositional formula and σ a signature consisting of all atoms $p_1, \dots, p_n$ occurring in F. By SM$[F]$ we denote the second-order propositional sentence

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

where $\mathbf{p}$ stands for the tuple $p_1, \dots, p_n$, $\mathbf{u}$ is a tuple of n distinct propositional variables $u_1, \dots, u_n$, equation $\mathbf{u} < \mathbf{p}$ stands for

$$(u_1 \rightarrow p_1) \wedge \cdots \wedge (u_n \rightarrow p_n) \wedge \neg((p_1 \rightarrow u_1) \wedge \cdots \wedge (p_n \rightarrow u_n))$$

as in the definition of circumscription, and $F^*(\mathbf{u})$ is defined recursively, as follows:

- $p_i^* = u_i$;
- $\bot^* = \bot$;
- $(F \odot G)^* = F^* \odot G^*$, where $\odot \in \{\wedge, \vee\}$;
- $(F \rightarrow G)^* = (F^* \rightarrow G^*) \wedge (F \rightarrow G)$.

We regard $\neg F$ as shorthand for $F \rightarrow \bot$. Note that $\neg$ corresponds to *not* in the logic program syntax. For instance, the rule

$$p \leftarrow \textit{not}\ q$$

[2] `http://peace.eas.asu.edu/f2lp .`

is identified with the formula

$$\neg q \rightarrow p \ .$$

The operator $F \mapsto F^*(\mathbf{u})$ replaces each atom with the corresponding propositional variable, and commutes with all propositional connectives except implication. If, in the definition of this operator, we drop the second conjunctive term in the clause for implication, then $F^*(\mathbf{u})$ will turn into the formula $F(\mathbf{u})$ referred to in the definition of circumscription [10, 11]. A model of F is *stable* if it satisfies $\mathrm{SM}[F]$.

According to [12, Section 2.6], a (propositional) formula F is said to be strongly equivalent to a formula G if any formula F' that contains an occurrence of F has the same stable models as the formula G' obtained from F' by replacing that occurrence with G. This condition is more general than the original definition from [4] not only because it is applicable to arbitrary formulas, but also because F is allowed here to be any subformula of F', not necessarily a "subconjunction."

Our reduction idea is based on the following proposition from [8], which generalizes the main theorem from [13], stating that the strong equivalence between two formulas F and G can be characterized in terms of equivalence (in classical logic) between F^* and G^*. Let σ' be a signature consisting of distinct atoms $\{p'_1, \dots, p'_n\}$ that are disjoint from σ, and let $\mathbf{p}'$ stand for the tuple $p'_1, \dots, p'_n$. Formula $F^*(\mathbf{p}')$ is obtained from $F^*(\mathbf{u})$ by substituting the atoms $\mathbf{p}'$ for propositional variables $\mathbf{u}$. Thus $F^*(\mathbf{p}')$ is a transformation of F in signature $\sigma \cup \sigma'$. Equation $\mathbf{p}' \leq \mathbf{p}$ stands for

$$(p'_1 \rightarrow p_1) \wedge \dots \wedge (p'_n \rightarrow p_n)$$

as in the definition of circumscription.

Proposition 1 *[8, Proposition 5] Formulas F and G of signature σ are strongly equivalent iff*

$$\mathbf{p}' \leq \mathbf{p} \rightarrow (F^*(\mathbf{p}') \leftrightarrow G^*(\mathbf{p}')) \tag{1}$$

is a tautology.

As usual, a formula F is in *negation normal form* if, for every subformula $G \rightarrow H$ of F, formula G is an atom, and H is $\bot$. An occurrence of a formula G in a formula F is *positive* if the number of implications in F containing the occurrence of G in the antecedent is even, and *negative* otherwise.

Definition 1. *An implication $F \rightarrow G$ of signature $\sigma \cup \sigma'$ is called a* canonical implication *if F and G are formulas in negation normal form such that every occurrence of atoms from σ' is positive, and every occurrence of atoms from σ is negative.*

For example,

$$p' \wedge q \rightarrow r$$

is not canonical, while

$$(p' \vee (\neg q \wedge r')) \rightarrow (s' \wedge \neg p) \tag{2}$$

is canonical.

Given a formula F of signature $\sigma \cup \sigma'$, by $R(F)$ we denote the formula of signature σ that is obtained from F by dropping all occurrences of $'$ in F. Note that $R(F)$, where F is a canonical implication, can be identified with a logic program with nested expressions [6], by identifying '$\neg$' with *not*, '$\wedge$' with ',', and '$\vee$' with ';'. For instance, in logic programming notation, when F is (2), $R(F)$ can be written as

$$s,\ \textit{not}\, p\ \leftarrow\ p\ ;\ (\textit{not}\, q,\ r)\ .$$

The following proposition tells us how to obtain a logic program that is strongly equivalent to a given formula.

Proposition 2 *Given a formula F, if G is a conjunction of canonical implications that is equivalent to F^*, then F and $R(G)$ are strongly equivalent.*[3]

The proof of Proposition 2 uses the observation that

$$\mathbf{p}' \leq \mathbf{p} \rightarrow (F^* \leftrightarrow (R(G))^*) \tag{3}$$

is a tautology. In view of Proposition 1, it follows that F and $R(G)$ are strongly equivalent. The fact that every propositional theory is strongly equivalent to a logic program follows from the fact that every formula F^* can be equivalently rewritten as a conjunction of canonical implications. One way to do this is by forming a conjunctive normal form (CNF) of $F^*(\mathbf{p}')$, and then converting each of its clauses into a canonical implication as follows. Given a clause C of signature $\sigma \cup \sigma'$, by $\mathit{Tr}(C)$ we denote an implication whose antecedent is the conjunction of

- all p' where $\neg p' \in C$, and
- all $\neg p$ where $p \in C$,

and whose consequent is the disjunction of

- all p' where $p' \in C$, and
- all $\neg p$ where $\neg p \in C$.

For instance, if C is $(p' \vee \neg q' \vee r \vee \neg s)$, then $\mathit{Tr}(C)$ is $(q' \wedge \neg r \rightarrow p' \vee \neg s)$. We can take G in the statement of Proposition 2 to be the conjunction of $\mathit{Tr}(C)$ for all clauses C in a conjunctive normal form of F^*. In view of Proposition 1, it follows that every formula is strongly equivalent to a logic program whose rules have the form

$$a_1; \dots; a_k; \textit{not}\ a_{k+1}; \dots; \textit{not}\ a_l \leftarrow a_{l+1}, \dots, a_m, \textit{not}\ a_{m+1}, \dots, \textit{not}\ a_n$$

($0 \leq k \leq l \leq m \leq n$) where all a_i are atoms.

Example 1 $F = (p \rightarrow q) \rightarrow r$.

$$\begin{aligned}((p \rightarrow q) \rightarrow r)^* &= (((p' \rightarrow q') \wedge (p \rightarrow q)) \rightarrow r') \wedge ((p \rightarrow q) \rightarrow r) \\ &\leftrightarrow (p' \vee p \vee r') \wedge (\neg q' \vee p \vee r') \wedge (p' \vee \neg q \vee r') \wedge (\neg q' \vee \neg q \vee r') \\ &\quad\ \wedge (p \vee r) \wedge (\neg q \vee r)\ .\end{aligned}$$

[3] For convenience, we will often drop "$(\mathbf{p}')$" from $F^*(\mathbf{p}')$ when there is no confusion.

Under the assumption that $(p', q', r') \leq (p, q, r)$, the formula can be simplified to

$$(p \vee r') \wedge (p' \vee \neg q \vee r') \wedge (\neg q' \vee r') \wedge (\neg q \vee r) \ .$$

Applying *Tr* to each clause yields the following formula G:

$$(\neg p \rightarrow r') \wedge (p' \vee \neg q \vee r') \wedge (q' \rightarrow r') \wedge (\neg r \rightarrow \neg q). \tag{4}$$

Thus $R(G)$ is

$$(\neg p \rightarrow r) \wedge (p \vee \neg q \vee r) \wedge (q \rightarrow r) \wedge (\neg r \rightarrow \neg q) \ . \tag{5}$$

In logic programming notation, (5) can be written as follows:

$$\begin{array}{rcl} r & \leftarrow & \textit{not}\, p \\ p \ ; \ \textit{not}\, q \ ; \ r & & \\ r & \leftarrow & q \\ \textit{not}\, q & \leftarrow & \textit{not}\, r \ . \end{array} \tag{6}$$

Proposition 2 tells us that logic program (6) is strongly equivalent to $(p \rightarrow q) \rightarrow r$.

Example 2 $F = p \rightarrow ((q \rightarrow r) \vee s)$.

$$\begin{aligned} (p \rightarrow ((q \rightarrow r) \vee s))^* &= (p' \rightarrow (((q' \rightarrow r') \wedge (q \rightarrow r)) \vee s')) \wedge (p \rightarrow (q \rightarrow r) \vee s) \\ &\leftrightarrow (\neg p' \vee (((\neg q' \vee r') \wedge (\neg q \vee r)) \vee s')) \wedge (\neg p \vee (\neg q \vee r) \vee s) \\ &\leftrightarrow (\neg p' \vee \neg q' \vee r' \vee s') \wedge (\neg p' \vee \neg q \vee r \vee s') \wedge (\neg p \vee \neg q \vee r \vee s) \ . \end{aligned}$$

Applying *Tr* to each clause yields the following formula G:

$$(p' \wedge q' \rightarrow r' \vee s') \wedge (p' \wedge \neg r \rightarrow \neg q \vee s') \wedge (\neg r \wedge \neg s \rightarrow \neg p \vee \neg q) \ . \tag{7}$$

Thus $R(G)$ is

$$(p \wedge q \rightarrow r \vee s) \wedge (p \wedge \neg r \rightarrow \neg q \vee s) \wedge (\neg r \wedge \neg s \rightarrow \neg p \vee \neg q) \ . \tag{8}$$

In logic programming notation, (8) can be written as follows:

$$\begin{array}{rcl} r \ ; \ s & \leftarrow & p, \ q \\ \textit{not}\, q \ ; \ s & \leftarrow & p, \ \textit{not}\, r \\ \textit{not}\, p \ ; \ \textit{not}\, q & \leftarrow & \textit{not}\, r, \ \textit{not}\, s \ . \end{array} \tag{9}$$

Proposition 2 tells us that logic program (9) is strongly equivalent to formula $p \rightarrow ((q \rightarrow r) \vee s)$.

3 Simpler Transformation

The following observation shows how to disregard some redundancies with the translation introduced in the previous section.

Proposition 3 *Let F be a propositional formula of signature σ. Under the assumption $\mathbf{p}' \leq \mathbf{p}$, if F^* is equivalent to $G \wedge H$ where G is a conjunction of canonical implications and H is a formula of signature σ that is entailed by $R(G)$, then F^* is equivalent to $(R(G))^*$.*

Example 1′. $F = (p \rightarrow q) \rightarrow r$ as in Example 1. Note that in (4), the last implication $(\neg r \rightarrow \neg q)$ is entailed by

$$R((\neg p \rightarrow r') \wedge (p' \vee \neg q \vee r') \wedge (q' \rightarrow r')).$$

Therefore, by Proposition 3, F^* is equivalent to

$$((\neg p \rightarrow r) \wedge (p \vee \neg q \vee r) \wedge (q \rightarrow r))^*.$$

In other words, in view of Proposition 1, F is strongly equivalent to the first three rules of (6).

Example 2′. $F = p \rightarrow ((q \rightarrow r) \vee s)$ as in Example 2. Note that in (7), the last implication is entailed by

$$R((p' \wedge q' \rightarrow r' \vee s') \wedge (p' \wedge \neg r \rightarrow \neg q \vee s'))\ .$$

Therefore in view of Proposition 3, F^* is equivalent to

$$((p \wedge q \rightarrow r \vee s) \wedge (p \wedge \neg r \rightarrow \neg q \vee s))^*.$$

In other words, in view of Proposition 1, F is strongly equivalent to the first two rules of (9).

Based on Proposition 3, we consider the following definition which leads to a simpler transformation than the one given in Proposition 2.

Definition 2. *For any formula F of signature σ, $F^\diamond(\mathbf{u})$ is defined as follows:*

– $p_i^\diamond = u_i$;
– $\bot^\diamond = \bot$;
– $(F \vee G)^\diamond = F^* \vee G^*$;
– $(F \wedge G)^\diamond = F^\diamond \wedge G^\diamond$;
– $(F \rightarrow G)^\diamond = (F^* \rightarrow G^*)$.

Note that $F^\diamond$ is different from F^* when we identify F with a conjunction $F_1 \wedge \cdots \wedge F_n$ $(n \geq 1)$, $F^\diamond$ is

$$F_1^\diamond \wedge \cdots \wedge F_n^\diamond$$

where

$$F_i^\diamond = \begin{cases} G^* \rightarrow H^* & \text{if } F_i \text{ is } G \rightarrow H, \\ F_i^* & \text{otherwise.} \end{cases}$$

The following proposition tells us that, in Proposition 2, $F^\diamond$ can be considered in place of F^*.

Proposition 4 *Given a formula F, if G is a conjunction of canonical implications that is equivalent to $F^\diamond$, then F and $R(G)$ are strongly equivalent.*

Example 1$''$ $F = (p \to q) \to r$ as in Example 1. Under the assumption that $(p', q', r') \leq (p, q, r)$,

$$\begin{aligned}F^\diamond(p', q', r') &= ((p' \to q') \wedge (p \to q)) \to r' \\ &\leftrightarrow (p \vee r') \wedge (p' \vee \neg q \vee r') \wedge (\neg q' \vee r') \\ &\leftrightarrow (\neg p \to r') \wedge (p' \vee \neg q \vee r') \wedge (q' \to r') \ .\end{aligned}$$

Thus F is strongly equivalent to

$$(\neg p \to r) \wedge (p \vee \neg q \vee r) \wedge (q \to r) \ ,$$

which is the same as in Example 1$'$.

Example 2$''$ $F = p \to ((q \to r) \vee s)$ as in Example 2. Under the assumption that $(p', q', r', s') \leq (p, q, r, s)$,

$$\begin{aligned}F^\diamond(p', q', r', s) &= p' \to (((q' \to r') \wedge (q \to r)) \vee s') \\ &\leftrightarrow (\neg p' \vee \neg q' \vee r' \vee s') \wedge (\neg p' \vee \neg q \vee r \vee s') \\ &\leftrightarrow (p' \wedge q' \to r' \vee s') \wedge (p' \wedge \neg r \to \neg q \vee s') \ .\end{aligned}$$

Thus F is strongly equivalent to

$$(p \wedge q \to r \vee s) \wedge (p \wedge \neg r \to \neg q \vee s) \ ,$$

which is the same as in Example 2$'$.

Due to lack of space, we do not provide a detailed comparison between our translation method and the others. However, we note that Proposition 2 not only shows that the reduction is possible, but also tells us how to generate strongly equivalent logic programs of preferably smaller size, based on the notion of equivalence in classical logic. This is in contrast with the other approaches that are based on syntactic rewriting rules under the logic of here-and-there. For instance, given a formula

$$((p \to q) \to r) \to r$$

our translation yields the following program:

$$\begin{aligned}q \ ; \ r \ ; \ \textit{not}\, r &\leftarrow p \\ \textit{not}\, p &\leftarrow \textit{not}\, q \ .\end{aligned}$$

On the other hand, the following program is obtained according to Section 3 of [5].

$$\begin{aligned}\textit{not}\, p \ ; \ r &\leftarrow \textit{not}\, q \\ r &\leftarrow r \\ q \ ; \ r \ ; \ \textit{not}\, r &\leftarrow p \\ \textit{not}\, p \ ; \ r \ ; \ \textit{not}\, r &\leftarrow \textit{not}\, q \ .\end{aligned}$$

However, clearly, any translation according to Proposition 4 (or Proposition 2) involves an exponential blowup in size in the worst case. Indeed, it is shown in [5] that there is no polynomial translation from propositional theories to logic programs if we do not introduce new atoms, and that there is one if we allow them.

4 Prenex Normal Form of First-Order Formulas

The translation from an arbitrary propositional theory into a logic program shows that their syntactic difference is not essential, which allows existing answer set solvers to compute the stable models of arbitrary propositional formulas. Can the result be extended to first-order formulas, of which the stable model semantics is presented in [8]?

We begin with a review of the stable model semantics presented in [8], which extends the definition of a stable model reviewed in Section 2 to first-order sentences. Given a first-order sentence F, by SM$[F]$ we denote the second-order sentence

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

where **p** stands for the tuple of all predicate constants $p_1, \dots, p_n$ occurring in F, **u** is a tuple of n distinct predicate variables $u_1, \dots, u_n$, equation $\mathbf{u} < \mathbf{p}$ is defined as in circumscription [11], and $F^*(\mathbf{u})$ is defined recursively, as follows:

– $p_i(t_1, \dots, t_m)^* = u_i(t_1, \dots, t_m)$;
– $(t_1 = t_2)^* = (t_1 = t_2)$;
– $\bot^* = \bot$;
– $(F \odot G)^* = F^* \odot G^*$, where $\odot \in \{\wedge, \vee\}$;
– $(F \rightarrow G)^* = (F^* \rightarrow G^*) \wedge (F \rightarrow G)$;
– $(QxF)^* = QxF^*$, where $Q \in \{\forall, \exists\}$.

A model of F is *stable* if it satisfies SM$[F]$. For the definition of strong equivalence extended to first-order formulas, we refer the reader to Section 4 of [8].

Proposition 1 can be extended to the case where F and G are first-order formulas [8, Proposition 5]. Using the proposition, one can prove that every first-order formula is strongly equivalent to a prenex normal form. The following proposition is essentially Theorem 6.4 of [14].

Proposition 5 *Every first-order formula is strongly equivalent to a prenex normal form.*

The proposition follows from the fact that usual prenex normal form conversion rules for first-order logic (e.g., [15, Lemma 2.29]) preserves strong equivalence. Alternative to the proof in [14], this fact can be proved using [8, Proposition 5]. For instance, $\forall x F(x) \rightarrow G$ is strongly equivalent to $\exists x(F(x) \rightarrow G)$, where x is not free in G. Consider

$$\begin{aligned} (\forall x F(x) \rightarrow G)^* &= (\forall x F(x) \rightarrow G) \wedge (\forall x F^*(x) \rightarrow G^*) \\ &\Leftrightarrow \exists x(F(x) \rightarrow G) \wedge \exists x(F^*(x) \rightarrow G^*) \end{aligned} \tag{10}$$

and

$$(\exists x(F(x) \rightarrow G))^* = \exists x((F(x) \rightarrow G) \wedge (F^*(x) \rightarrow G^*)) \; . \tag{11}$$

Note that (10) and (11) are not (classically) equivalent in general, but they are equivalent under the assumption $\mathbf{p}' \leq \mathbf{p}$, where $\mathbf{p}$ is the tuple of all predicate constants occurring in $F(x)$ and G, and $\mathbf{p}'$ is the tuple of new, pairwise distinct predicate constants of the same length as $\mathbf{p}$. Therefore, by [8, Proposition 5], we conclude that $\forall x F(x) \rightarrow G$ is strongly equivalent to $\exists x(F(x) \rightarrow G)$.

Also, Proposition 2 can be straightforwardly extended to quantifier-free first-order formulas as follows. A first-order formula F is in *negation normal form* if, for every subformula $G \rightarrow H$ of F,

– formula G is an atomic formula, and
– formula H is $\bot$.

For any clause C in a CNF of a quantifier-free first order formula, $Tr(C)$ from Section 2 can be extended in a straightforward way. The equality can be placed either in the consequent or the antecedent (properly negated).

Corollary 1 *Any first-order formula is strongly equivalent to a prenex normal form whose matrix is a conjunction of implications $F \rightarrow G$ where F and G are formulas in negation normal form.*

The matrix of a prenex normal form indicated in Corollary 1 is in the form of a logic program. Thus, similar to the propositional case, the syntactic difference of arbitrarily nested connectives and quantifiers is not essential between the new language proposed in [8] and logic programs. On the other hand, since the prenex normal form may contain existential quantifiers, it is different from a logic program, where all variables are assumed to be universally quantified. For instance, according to [8], the stable models of formula $\exists x\ p(x)$ represent that p is a singleton, as in circumscription. This has no counterpart in logic programs, since their stable models are limited to Herbrand interpretations. For a related discussion, see [16].

5 Implementation

Our implementation, which we call F2LP, turns an arbitrary propositional theory into a logic program and calls DLV to compute its stable models. When the input is already in the syntax of DLV input language, its operation is just as what DLV does. The system is available at

```
http://peace.eas.asu.edu/f2lp .
```

The ASCII representations of propositional connectives used in the syntax of F2LP are summarized in the following chart:

Symbol	$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\bot$	$\top$
ASCII representation	`not`	`&`	`|`	`->`	`false`	`true`

Example 1 is written in the syntax of F2LP as follows:

```
(p->q)->r.
```

F2LP turns this formula into the following DLV input:

```
r :- not p.
p | r | q_bar :-.
r :- q.
q_bar :- not q.
:- q, q_bar.
```

Note that this program is slightly different from the logic program shown in Example 1$'$ (the first three rules of (6)). This is because DLV, like most other answer set solvers, does not allow negation as failure in the head of a rule. However, it can be simulated by introducing new atoms (Section 4 of [17]). The method replaces the occurrence of *not* p in the head of a rule with a new atom $\overline{p}$, and adds rules $\overline{p} \leftarrow$ *not* p and $\leftarrow p, \overline{p}$. The stable models of the program correspond to the stable models of the original program by disregarding the presence of the new atoms. In the example above, `q_bar` is a new atom, and the last two rules are added. After F2LP calls DLV to compute the stable models, it removes all occurrences of the new atoms ("`_bar`") from the stable models returned by DLV.

Example 2 is written in our syntax as follows:

```
p -> ((q->r) | s).
```

This is turned into the following DLV input by F2LP:

```
r | s :- p, q.
q_bar | s :- p, not r.
q_bar :- not q.
:- q, q_bar.
```

6 Conclusion

Our contributions in this note are as follows. First, we presented a new proof of the theorem on strong equivalence between propositional theories and logic programs. Unlike the other approaches that are based on the logic of here-and-there, our proof relies on familiar properties of classical logic. Due to this fact, our proof indicates how corresponding logic programs can be generated using equivalent transformations in classical logic. Second, using the same reduction idea, we showed that arbitrary first-order formulas under the stable model semantics, recently proposed in [8], can be turned into a prenex normal form whose matrix has the form of a logic program. Third, we presented a prototype implementation for computing the stable models of arbitrary propositional formulas based on the reduction method.

For future work, we plan to investigate how the methods of obtaining minimally equivalent theories in classical logic can be applied to finding minimally equivalent logic programs. Recently, Cabalar *et al.* [18] proposed two notions of minimal logic programs. It would be interesting to see how these approaches are related.

Acknowledgements

We are grateful to Paolo Ferraris, Vladimir Lifschitz and the anonymous referees for useful comments on this paper. The authors were partially supported by DTO AQUAINT.

A Appendix: Proof of Proposition 2

Due to lack of space, we present the proof of Proposition 2 only, which follows immediately from Proposition 1 and the following proposition.

Proposition 6 *Let F be a formula of signature σ and G a conjunction of canonical implications that is equivalent to F^*. Then*

$$\mathbf{p}' \leq \mathbf{p} \rightarrow (F^* \leftrightarrow (R(G))^*)$$

is a tautology.

The proof of Proposition 6 uses the following lemmas, most of which can be proven by induction.

Lemma 1. *For any formula F of signature σ, the formula*

$$\mathbf{p}' \leq \mathbf{p} \rightarrow (F^*(\mathbf{p}') \rightarrow F)$$

is logically valid.

Lemma 2. *Every formula F is equivalent to $R(F^*)$.*

Lemma 3. *For any two formulas F and G of signature $\sigma \cup \sigma'$,*

$$(F \leftrightarrow G) \rightarrow (R(F) \leftrightarrow R(G))$$

is a tautology.

Proof. Assume that $F \leftrightarrow G$ holds for all interpretations of $\sigma \cup \sigma'$, which includes the interpretations I such that $p^I = (p')^I$ for all $p \in \mathbf{p}$. It is clear that $F^I = R(F)^I$ and $G^I = R(G)^I$, from which $R(F)^I = R(G)^I$ follows. Since I range over all interpretations of σ, it follows that $R(F) \leftrightarrow R(G)$. ∎

Lemma 4. *For any canonical implication F of signature $\sigma \cup \sigma'$,*

$$(\mathbf{p}' \leq \mathbf{p}) \rightarrow ((F \wedge R(F)) \leftrightarrow (R(F))^*)$$

is a tautology.

Proof of Proposition 6. Assume $\mathbf{p}' \leq \mathbf{p}$ and $F^* \leftrightarrow G$. By Lemma 1, $F^* \rightarrow F$ holds, so that F^* is equivalent to $G \wedge F$. Since F is equivalent to $R(F^*)$ according to Lemma 2, $G \wedge F$ is equivalent to $G \wedge R(F^*)$, which, in turn, is equivalent to $G \wedge R(G)$ according to Lemma 3. By Lemma 4, it follows that $G \wedge R(G)$ is equivalent to $(R(G))^*$. ∎

References

1. Pearce, D.: A new logical characterization of stable models and answer sets. In Dix, J., Pereira, L., Przymusinski, T., eds.: Non-Monotonic Extensions of Logic Programming (Lecture Notes in Artificial Intelligence 1216), Springer-Verlag (1997) 57–70
2. Ferraris, P.: Answer sets for propositional theories. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). (2005) 119–131
3. Cabalar, P., Ferraris, P.: Propositional theories are strongly equivalent to logic programs. Submitted for publication (2005)
4. Lifschitz, V., Pearce, D., Valverde, A.n.: Strongly equivalent logic programs. ACM Transactions on Computational Logic **2** (2001) 526–541
5. Cabalar, P., Pearce, D., Valverde, A.n.: Reducing propositional theoreis in equilibrium logic to logic programs. In: Proceedings of 12th Portuguese Conference on Artificial Intelligence (EPIA 2005). (2005) 4–17
6. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence **25** (1999) 369–389
7. Sarsakov, V., Schaub, T., Tompits, H., Woltran, S.: nlp: A compiler for nested logic programming. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04). Volume 2923 of Lecture Notes in Computer Science., Springer-Verlag Heidelberg (2003) 361 – 364
8. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI). (2007)
9. Pearce, D., Tompits, H., Woltran, S.: Encodings for equilibrium logic and logic programs with nested expressions. In: Proceedings of Portuguese Conference on Artificial Intelligence (EPIA). (2001) 306–320
10. McCarthy, J.: Circumscription—a form of non-monotonic reasoning. Artificial Intelligence **13** (1980) 27–39,171–172
11. Lifschitz, V.: Circumscription. In Gabbay, D., Hogger, C., Robinson, J., eds.: The Handbook of Logic in AI and Logic Programming. Volume 3. Oxford University Press (1994) 298–352
12. Ferraris, P., Lifschitz, V.: Mathematical foundations of answer set programming. In: We Will Show Them! Essays in Honour of Dov Gabbay. King's College Publications (2005) 615–664
13. Lin, F.: Reducing strong equivalence of logic programs to entailment in classical propositional logic. In: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR). (2002) 170–176
14. Pearce, D., n Valverde, A.: A first order nonmonotonic extension of constructive logic. Studia Logica **80** (2005) 323–348
15. Mendelson, E.: Introduction to Mathematical Logic. Wadsworth & Brooks (1987) Third edition.
16. Texas Action Group: Technical discussions: Do we need existential quantifiers in logic programming? (2007) `http://www.cs.utexas.edu/users/vl/tag/discussions.html` .
17. Janhunen, T.: On the effect of default negation on the expressiveness of disjunctive rules. In: Proc. LPNMR 2001. (2001) 93–106
18. Cabalar, P., Pearce, D., Valverde, A.n.: Minimal logic programs. Unpublished draft (2007)