Reducing Strong Equivalence of Logic Programs to Entailment in Classical Propositional Logic

Fangzhen Lin (flin@cs.ust.hk)
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

Recently Lifschitz, Pearce, and Valverde [2001] introduced a notion of strong equivalence between two logic programs, and showed that it can be captured in a 3-valued logic. In this paper, first for propositional logic programs with default negation, constraints, and disjunctions, we show that there is a simple mapping from these programs to propositional theories that reduces this notion of strong equivalence to entailment in classical propositional logic. Furthermore, we also provide a mapping in the other direction thus show that the problem of checking strong equivalence is co-NP-complete. We then consider logic programs with variables. One surprising result is that while the problem of deciding whether two logic programs are equivalent goes from decidable to undecidable when we move from logic programs without variables to ones with, the problem of deciding whether two logic programs are strongly equivalent remains to be co-NP-complete for logic programs with variables and constants.

1 Introduction

In this paper we shall consider logic programs with rules of the following form:

$$h_1; \cdots; h_k \leftarrow l_1, \cdots, l_n$$

where h_i's are atoms, and l_i's are either atoms or literals of the form $\mathsf{not}\, p$ for some atom p. So a logic program here can have default negation (not), constraints (when $k = 0$), and disjunctions in the head of its rules. The semantics of these programs are given by answer sets as defined in [Gelfond and Lifschitz, 1991].

Two such logic programs P_1 and P_2 are said to be *equivalent* if they have the same answer sets, and *strongly equivalent* [Lifschitz *et al.*, 2001] if for any logic program P, $P \cup P_1$ and $P \cup P_2$ are equivalent.

Questions regarding whether two logic programs are strongly equivalent are interesting for a variety of reasons. For instance, Lifschitz, Pearce, and Valverde [2001] argued that in order to see whether a set of rules can always be replaced by another one regardless of the context, one should check whether the two sets of rules are strongly equivalent. For instance, the set of single rule $p \leftarrow p$ is strongly equivalent to the empty set, so this rule can always be eliminated from any logic program. However, the set $\{p \leftarrow q.\quad q \leftarrow p\}$ is equivalent, but not strongly equivalent to the empty set, so the pair of rules cannot be eliminated regardless of the context: in the presence of q the first rule can be used to derive p. As another example, while the rule $p \leftarrow \mathsf{not}\, p$ cannot in general be eliminated from a logic program, it can if the program contains the two rules $p \leftarrow \mathsf{not}\, q$ and $p \leftarrow\ q$: $\{p \leftarrow \mathsf{not}\, q.\ p \leftarrow\ q.\}$ and $\{p \leftarrow \mathsf{not}\, q.\ p \leftarrow\ q.\ p \leftarrow \mathsf{not}\, p.\}$ are strongly equivalent.

It is clear from the definition that strong equivalence implies equivalence. As it turns out, strong equivalence happens to have much better computational properties. So one can use the former to approximate the latter in applications such as program and query optimizations where ideally the latter should be used. In fact, that was exactly what Sagiv [1988] did for datalog programs: he proposed a notion of uniform equivalence and showed that in contrast to equivalence which is not decidable, uniform equivalence, which is the same as strong equivalence as we shall see later, is tractable. He then proposed an optimization algorithm based on uniform equivalence for eliminating redundant rules and redundant atoms in the body of a rule. For instance, the atom $f(w, y)$ in the body of the rule $g(x, y, z) \leftarrow g(x, w, z), f(w, y), f(w, z), f(z, z), f(z, y)$ can be eliminated since the rule is uniformly equivalent to $g(x, y, z) \leftarrow g(x, w, z), f(w, z), f(z, z), f(z, y)$.

Lifschitz, Pearce, and Valverde [2001] showed that checking for strong equivalence between two logic programs can be done in the logic of here-and-there, a three-valued non-classical logic somewhere between classical logic and intuitionistic logic. More recently, Turner [2001a] gave a model-theoretic characterization of strong equivalence in terms of pairs of sets of atoms. In this paper we shall provide a simple mapping from logic programs to propositional theories that reduces strong equivalence to entailment in classical propositional logic. Our mapping is motivated by both Turner's model-theoretic characterization and the translation of logic programs to circumscriptive propositional theories in [Lin, 1991], which was in turn derived from the mapping from default logic to Lin and Shoham's logic of knowledge and justified assumption in [Lin and Shoham, 1992], a nonmonotonic bi-modal logic.

This paper is organized as follows. We shall first consider propositional logic programs. In section 2, we shall give a translation from logic programs to propositional theories that reduces strong equivalence of logic programs to entailment in propositional logic. In section 3, we shall give a converse translation from clauses to rules, thus showing that strong equivalence is co-NP-complete. In section 4, we consider logic programs that may contain variables and constants but not proper functions, i.e. those with arities greater than 0, and show that the problem of checking the strong equivalence of two logic programs with variables remains to be co-NP-complete. In section 5 we consider a more general notion of equivalence, and discuss some related work. Finally in section 6 we conclude this paper.

2 From strong equivalence to propositional entailment

In this section, we assume a fixed propositional language L. We assume that L has two special members: $true$, a tautology, and $false$, negation of a tautology. All logic programs in this section are supposed to be written in this language, i.e. all atoms in the logic programs come from L. For each atom $p \in L$, we assume that p' is a new atom not in L.

As mentioned above, a logic program P is a finite set of rules of the following form:

$$h_1; \cdots; h_k \leftarrow p_1, \cdots, p_m, \mathsf{not}\, p_{m+1}, \cdots, \mathsf{not}\, p_n. \qquad (1)$$

where $h_1, ..., h_k, p_1, ..., p_n$ are atoms in L. Now for each rule (1) in P, let $\Gamma(P)$ contain the following two sentences:

$$p_1 \wedge \cdots \wedge p_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h_1 \vee \cdots \vee h_k, \quad (2)$$

$$p'_1 \wedge \cdots \wedge p'_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h'_1 \vee \cdots \vee h'_k. \quad (3)$$

Notice that if $m = n = 0$, then the left sides of the implications in (2) and (3) are considered to be $true$, and if $k = 0$, then the right sides of the implications in (2) and (3) are considered to be $false$.

As it turns out, this translation captures strong equivalence in propositional logic under the assumption that $p \supset p'$ for each p in the language:[1]

Theorem 1 *For any two logic programs P_1 and P_2, they are strongly equivalent iff the following two assertions hold:*

$$\{p \supset p' \mid p \in L\} \cup \Gamma(P_1) \models \Gamma(P_2),$$
$$\{p \supset p' \mid p \in L\} \cup \Gamma(P_2) \models \Gamma(P_1),$$

where "$\models$" is logical entailment in propositional logic.

[1]Independently, [Pearce *et al.*, 2001] proposed a similar translation for propositional logic programs with nested expressions but without disjunctions.

Proof: The proof of this theorem is based on Turner's result in [Turner, 2001a]. Let $X \subseteq Y$ be two sets of atoms. According to Turner, the pair (X, Y) is called an HT-model if both X and Y are closed under P^Y, where

- the program P^Y, the reduct of P on Y, is obtained from P as follows: for any rule $p_1; \cdots; p_k \leftarrow G$ in P, if G has a literal $\mathsf{not}\, q$ such that $q \in Y$, then delete this rule; otherwise, delete all the literals of the form $\mathsf{not}\, q$ in G; and
- a set of literals S is closed under a program Q without negation if for any rule $p_1; \cdots; p_k \leftarrow G$ in Q, if all the atoms in G are in S, then for some $1 \leq i \leq n$, p_i is in S.

Turner [2001a] showed that for any two programs P_1 and P_2, they are strongly equivalent iff they have the same HT-models. So to prove the theorem, we show that there is a one-to-one correspondence between the HT-models of P and the pairs: $(M_L, M_{L'})$, where M is a model of $\Sigma \cup \Gamma(P)$, and

$$\Sigma = \{p \supset p' \mid p \in L\},$$
$$M_L = \{p \mid p \in L \text{ and } M \models p\},$$
$$M_{L'} = \{p \mid p \in L \text{ and } M \models p'\}.$$

If M is a model of $\Sigma \cup \Gamma(P)$, then it can be shown that $(M_L, M_{L'})$ is an HT-model of P:

- $M_L \subseteq M_{L'}$ because $M \models \Sigma$.
- M_L is closed under $P^{M_{L'}}$ because for each rule (1) in P, M satisfies (2).
- $M_{L'}$ is closed under $P^{M_{L'}}$ because for each rule (1) in P, M satisfies (3).

Conversely, if (X, Y) is an HT-model of P, then we can construct the following truth assignment M: $M \models p$ iff $p \in X$, and $M \models p'$ iff $p \in Y$. Then it can be shown that M is a model of $\Sigma \cup \Gamma(P)$. ∎

Examples

1. Let $P_1 = \{p \leftarrow p\}$. $\Gamma(P_1)$ is $\{p \supset p, p' \supset p'\}$, which is equivalent to tautology. So P_1 is strongly equivalent to the empty set, and can always be deleted from a logic program.
2. Let $P_2 = \{p \leftarrow \mathsf{not}\, p\}$. $\Gamma(P_2) = \{\neg p' \supset p, \neg p' \supset p'\}$, which is equivalent to $\{p'\}$. So P_2 is not strongly equivalent to the empty set, thus cannot in general be deleted.
3. Consider $P_3 = \{p \leftarrow q.\ p \leftarrow \mathsf{not}\, q\}$. $P_2 \cup P_3$ is strongly equivalent to P_3, thus in the presence of the rules in P_3, the rule in P_2 can always be deleted: $\Gamma(P_3)$ is $\{q \supset p, q' \supset p', \neg q' \supset p, \neg q' \supset p'\}$, which is equivalent to $\{q \supset p, \neg q' \supset p, p'\}$. Thus $\Gamma(P_3)$ entails $\Gamma(P_2)$.
4. P_3 and $P_4 = \{p\}$ are equivalent but not strongly equivalent. However, $P_3 \cup \{\leftarrow q\}$ and $P_4 \cup \{\leftarrow q\}$ are strongly equivalent: this is easy to check as $\Gamma(\{\leftarrow q\})$ is equivalent to $\neg q \wedge \neg q'$. This means the two rules in P_3 can always be replaced by the single rule in P_4 under the constraint $\leftarrow q$.

5. $P_5 = \{q \leftarrow p\}$ and $P_6 = \{q \leftarrow p. \quad \leftarrow p, \mathsf{not}\, q\}$ are strongly equivalent: $\Gamma(P_5)$ is equivalent to $(p \supset q) \wedge (p' \supset q')$, and $\Gamma(P_6)$ to $(p \supset q) \wedge (p' \supset q') \wedge (p \supset q') \wedge (p' \supset q')$; while these two sentences are not logically equivalent, they are under the axiom $q \supset q'$. This example shows that the set of constraints $\{p \supset p' \mid p \in L\}$ is necessary in Theorem 1.
6. For an example of disjunctive logic programs, consider the following two logic programs which have shown to be strongly equivalent in [Lifschitz *et al.*, 2001]: $P_7 = \{p; q. \quad \leftarrow p, q\}$ and $P_8 = \{p \leftarrow \mathsf{not}\, q.\ q \leftarrow \mathsf{not}\, p. \quad \leftarrow p, q\}$. $\Gamma(P_7)$ is $\{p \vee q, p' \vee q', \neg(p \wedge q), \neg(p' \wedge q')\}$, and $\Gamma(P_8)$ is
$$\{\neg q' \supset p, \neg q' \supset p', \neg p' \supset q, \neg p' \supset q', \neg(p \wedge q), \neg(p' \wedge q')\},$$
which is equivalent to $\{q' \vee p, q' \vee p', p' \vee q, \neg p \vee \neg q, \neg p' \vee \neg q'\}$. It can be easily verified by resolution that $\Gamma(P_7)$ and $\Gamma(P_8)$ are equivalent under $\{p \supset p', q \supset q'\}$.

3 From propositional entailment to strong equivalence

We now show that given two sets of clauses, the problem of checking whether one of them entails the other can be reduced to that of checking whether two logic programs are strongly equivalent. Together with the result in the previous section, this then shows that the problem of checking whether two logic programs are strongly equivalent is co-NP-complete.

Let L be our language. For each $p \in L$, let $\hat{p}$ be a new proposition. In the following, we let $\hat{L} = L \cup \{\hat{p} \mid p \in L\}$, and denote by Π the set of following rules:[2] for each $p \in L$,
$$p \leftarrow \mathsf{not}\, \hat{p},$$
$$\hat{p} \leftarrow \mathsf{not}\, p,$$
$$\leftarrow p, \hat{p}.$$
This is by now a well-known technique of making atom p an assumption, i.e. one can assume either p or its negation $\hat{p}$, but not both.

For any clause $c = p_1 \vee \cdots \vee p_m \vee \neg p_{m+1} \vee \cdots \vee \neg p_n$ in L, let $\Delta(c)$ be the following constraint in $\hat{L}$:
$$\leftarrow \hat{p}_1, ..., \hat{p}_m, p_{m+1}, ..., p_n.$$
For any set S of clauses in L, let $\Delta(S) = \{\Delta(c) \mid c \in S\}$.

Theorem 2 *Let S_1 and S_2 be two sets of propositional clauses in L. $S_1 \models S_2$ iff $\Pi \cup \Delta(S_1)$ and $\Pi \cup \Delta(S_1) \cup \Delta(S_2)$ are strongly equivalent in $\hat{L}$.*

Proof: It is easy to see that for any set S of clauses in L, and any program P in $\hat{L}$, a set of atoms $M \subseteq \hat{L}$ is a stable model of $\Pi \cup \Delta(S) \cup P$ iff M contains exactly one of p and $\hat{p}$ for any $p \in L$, and satisfies S and P (as a set of clauses). From this and the definition of strong equivalence, the theorem follows. ■

From Theorem 1 and Theorem 2, we have the following corollary:[3]

Corollary 2.1 *The problem of checking the strong equivalence of two logic programs, with or without disjunctions, is co-NP-complete.*

Notice that we made use of constraints in our translation above from clauses to logic programs. In general, constraints cannot be replaced by rules. For instance, we can show that the constraint $\leftarrow p$ is not strongly equivalent to any set of rules with non-empty heads.

Proposition 3.1 *There does not exist any set P of rules with non-empty heads such that P and $\{\leftarrow p\}$ is strongly equivalent.*

Proof: Suppose P is such a program. Then $\Gamma(P)$ is a set of sentences of the form $A \supset \alpha$, where α is a disjunction of atoms. Now let M be an interpretation that assigns every proposition true, then M satisfies $\Gamma(P)$ and $\{q \supset q' \mid q \in L\}$, but does not satisfy $\Gamma(\{\leftarrow p\}) = \{\neg p, \neg p'\}$. So by Theorem 1, P and $\{\leftarrow p\}$ cannot be strongly equivalent. ■

This result may seem counter-intuitive as there is a well-known translation from constraints to rules using a special "fail" atom: let P be any program, and P' the result of replacing each constraint $\leftarrow Body$ in P by $fail \leftarrow Body$ and adding the new rule:
$$dummy \leftarrow \mathsf{not}\, dummy, fail$$
where $fail$ and $dummy$ are two new atoms. Clearly, P and P' are equivalent. In fact, for any program Q that does not contain $fail$ and $dummy$, $P \cup Q$ and $P' \cup Q$ are equivalent. However, it is obvious that they are not strongly equivalent.

Turner[4] proposed the following translation from clauses to normal logic programs without using constraints. Again for each $p \in L$, let $\hat{p}$ be a new proposition. Let $\hat{L} = L \cup \{\hat{p} \mid p \in L\}$. Now denote by Π' the set of following rules: for each $p \in L$,
$$p \leftarrow \mathsf{not}\, \hat{p}, \mathsf{not}\, fail$$
$$\hat{p} \leftarrow \mathsf{not}\, p, \mathsf{not}\, fail$$
$$fail \leftarrow p, \hat{p}, \mathsf{not}\, fail,$$
where $fail$ is a new proposition.

Now let h^+ and h^- be the following mappings: for each $p \in L$, $h^+(p) = p$ and $h^+(\neg p) = \hat{p}$, and $h^-(p) = \hat{p}$ and $h^-(\neg p) = p$. For any clause $c = l_1 \vee \cdots \vee l_m$, $m \geq 1$, in L, let $\Delta'(c)$ be the following rule in $\hat{L}$:
$$h^+(l_1) \leftarrow h^-(l_2), ..., h^-(l_m).$$

[2] As we have seen in the last section, we could replace the first two rules by the disjunction $p; \hat{p}$.

[3] Turner [2001b] independently showed that the problem of checking whether two logic programs are strongly equivalent is in co-NP.

[4] Personal communication, 2001.

For a set S of clauses in L, let $\Delta'(S) = \{\Delta'(c) \mid c \in S\}$. Turner showed that Theorem 2 holds for this translation: for any two sets of clauses S_1 and S_2, $S_1 \models S_2$ iff $\Pi' \cup \Delta'(S_1)$ and $\Pi' \cup \Delta'(S_1) \cup \Delta'(S_2)$ are strongly equivalent. Thus, even for normal logic programs, checking whether two programs are strongly equivalent is co-NP-complete.

4 Logic programs with variables but without functions

For most applications of logic programming, rules normally have variables. In this section we try to extend our results to these rules.

In the following, we shall assume a first-order language L that has a finite set of predicates, a finite set of constants, and no other functions. Unless stated otherwise, in the following, a logic program is a finite set of rules in L. Let P be such a program. A *domain* for P is then a finite set of elements that include all constants in P. Given a domain D for P, the instantiation of P on D, written P_D, is the set of rules resulted from substituting elements of D for the variables in the rules of P.

We say that two logic programs P and Q are (resp. strongly) equivalent if for any domain D of P and Q, P_D and Q_D are (resp. strongly) equivalent as two propositional logic programs. Notice that a domain for both P and Q must contain all constants in $P \cup Q$.

Unfortunately, the problem of checking whether two programs with variables are equivalent is undecidable. This can be seen from a similar result about *datalog programs* in deductive databases.

A datalog program is a set of *definite rules* that does not have any constants and function symbols, where a rule is definite if it has no negation and disjunction. A relation in a datalog program is called *extensional* if there is no rule about it in the program; otherwise, it is called *intensional*. In deductive database literature, two datalog programs are said to be equivalent if for any given finite extensional relations, the two programs yield the same intensional relations (cf. [Sagiv, 1988]). It is well-known in deductive databases that the problem of checking if two datalog programs are equivalent is undecidable [Shmueli, 1986]. From this, we conclude that the problem of checking whether two logic programs with variables are equivalent is also undecidable because of the following simple observation. Let P and Q be two datalog programs such that $E_1, ..., E_k$ are their extensional relations. Let $\hat{E}_1, ..., \hat{E}_k$ be new relations of the same arities with $E_1, ..., E_k$, respectively. Let Σ be the set of following rules:

$$E_i(\vec{x}_i) \leftarrow \mathsf{not}\, \hat{E}_i(\vec{x}_i),$$
$$\hat{E}_i(\vec{x}_i) \leftarrow \mathsf{not}\, E_i(\vec{x}_i),$$

for each $1 \leq i \leq k$. Clearly, P and Q are equivalent as two datalog programs iff $P \cup \Sigma$ and $Q \cup \Sigma$ are equivalent according to our definition.

In deductive databases, Sagiv [1988] and Maher [1988] also introduced a stronger notion of equivalence between two datalog programs: two datalog programs are *uniformly equivalent* if given any finite extensional relations, and any initial finite intensional relations, the two programs always yield the same intensional relation. It can be shown that two datalog programs are uniformly equivalent iff they are strongly equivalent according to our definition above. In contrast to equivalence, Sagiv [1988] showed that checking whether two datalog programs are uniformly equivalent is a tractable problem and provided an algorithm for doing so. He also showed that using the algorithm one can sometimes improve the efficiency of a datalog program by eliminating redundant subgoals in the body of a rule.

The class of logic programs considered in this paper is much more expressive than that of datalog programs. Even in the propositional case, checking strong equivalence is not a tractable problem. Still, unlike equivalence, for which there is no algorithm to verify it when there are variables in the rules, we shall show that the problem of verifying strong equivalence for programs with variables continues to be decidable, and in fact belongs to the same complexity class as that for propositional logic programs.

To this end, we first generalize Theorem 1 to programs with variables. To do this, we need to introduce some notations. In the following, we use $\models_{FM}$ to denote entailment under finite models. That is, for any theory T and sentence φ, $T \models_{FM} \varphi$ if for every first-order structure M with a finite domain, M satisfies φ whenever it satisfies every sentence in T. The meaning of $T \models_{FM} T'$ is defined similarly when both T and T' are sets of sentences.

As in the propositional case, for each predicate p in the language L, let p' be a new predicate of the same arity as p. We also need an axiom for each predicate p saying that p implies p'. In the following, we let

$$\Sigma = \{(\forall \xi).p(\xi) \supset p'(\xi) \mid p \text{ is a predicate in } L\}.$$

Now for each rule

$$h_1(u_1); \cdots; h_k(u_k) \leftarrow p_1(v_1), \cdots, p_m(v_m), \mathsf{not}\, p_{m+1}(v_{m+1}), \cdots, \mathsf{not}\, p_n(v_n)$$

in P, let $\Gamma(P)$ contain the following two sentences:

$$(\forall \xi).[p_1(v_1) \land \cdots \land p_m(v_m) \land \neg p'_{m+1}(v_{m+1}) \land \cdots \land \neg p'_n(v_n)] \supset h_1(u_1) \lor \cdots \lor h_k(u_k),$$
$$(\forall \xi).[p'_1(v_1) \land \cdots \land p'_m(v_m) \land \neg p'_{m+1}(v_{m+1}) \land \cdots \land \neg p'_n(v_n)] \supset h'_1(u_1) \lor \cdots \lor h'_k(u_k),$$

where ξ is the tuple of variables in the rule.

From Theorem 1, we immediately have the following results:

Proposition 4.1 *For any domain D of P and Q, P_D and Q_D are strongly equivalent iff for any interpretation I with domain D, if I satisfies $Uni \cup \Sigma$, then I satisfies*

$\Gamma(P)$ *iff it satisfies* $\Gamma(Q)$*, where* Uni *is the set of unique names axioms about the constants in* P *and* Q*:*

$$Uni = \{c \neq d \mid \text{ for any pair of distinct constants } c \text{ and } d \text{ in } P \text{ and } Q\}.$$

Notice that the unique names axioms are implicitly assumed in logic programs.

From this proposition, we have:

Theorem 3 *Two logic programs* P *and* Q *are strongly equivalent iff*

$$Uni \cup \Sigma \cup \Gamma(P) \models_{FM} \Gamma(Q),$$
$$Uni \cup \Sigma \cup \Gamma(Q) \models_{FM} \Gamma(P),$$

where Uni *is as given in Proposition 4.1 above.*

In general, entailment under finite models is not decidable - it follows from Trahtenbrot's theorem (cf. [Ebbinghaus and Flum, 1999]) that the class of the sentences true in all finite models is not even recursively enumerable. However, for the class of sentences in Theorem 3, it is decidable.

To show this, we need the following lemma from first-order logic.

Lemma 4.1 *Let* φ *be a formula of form*

$$\forall x_1 \cdots \forall x_n \exists y_1 \cdots \exists y_m . \phi,$$

where $n \geq 1$*,* $m \geq 0$*, and* ϕ *is a formula that contains no quantifiers, constants and function symbols. Then for any* $k \geq n$*,* φ *is valid iff it is true for all interpretations with a domain of* k *objects.*

Proof: Exercise 2.58, page 74, of [Mendelson, 1987]. ■

For each rule, we call the number of variables in it its *variable rank*. The variable rank of a program is then the maximum of the variable ranks of rules in the program.

Theorem 4 *Let* P *and* Q *be two logic programs. Let* n *be the maximum of the variable ranks of* P *and* Q*, and* m *the number of constants in* $P \cup Q$*. Then* P *and* Q *are strongly equivalent iff* P_D *and* Q_D *are strongly equivalent for some domain* D *of* $m+n$ *elements for* P *and* Q*.*

Proof: Notice that for any program R, $\Gamma(R)$ is a finite set of sentences of the form: $\forall x_1 \cdots \forall x_k . W$, where W is a formula that contains no function symbols of arities greater than 0. So $\Gamma(R)$ is equivalent to a sentence also of such a form such that k is the variable rank of R. Without loss of generality, let the variable ranks of P and Q be n and t, respectively, and $n \geq t$, and that $\Gamma(P)$ and $\Gamma(Q)$ are equivalent to $\forall x_1 \cdots \forall x_n . W$ and $\forall y_1 \cdots \forall y_t . W'$, respectively.

It is easy to see that Σ is equivalent to a sentence of the form $(\forall \vec{z}).A(\vec{z})$ such that A does not contain any quantifiers, constants, and functions.

The unique names axioms Uni about constants in these two programs can be simulated by introducing m new unary predicates: for instance, the unique names axioms about three constants c_1, c_2, and c_3 can be axiomatized by the following axiom:

$$U_1(c_1) \wedge \neg U_1(c_2) \wedge \neg U_1(c_3) \wedge$$
$$U_2(c_2) \wedge \neg U_2(c_1) \wedge \neg U_2(c_3) \wedge$$
$$U_3(c_3) \wedge \neg U_3(c_1) \wedge \neg U_3(c_2).$$

In the following, we shall understand Uni to be an axiom of such nature.

Now assuming that $\vec{x}$, $\vec{y}$, and $\vec{z}$ have no common elements, then by Theorem 3, P and Q are strongly equivalent iff

$$\models_{FM} [Uni \wedge (\forall \vec{z})A \wedge (\forall \vec{x})W] \supset (\forall \vec{y})W',$$
$$\models_{FM} [Uni \wedge (\forall \vec{z})A \wedge (\forall \vec{y})W'] \supset (\forall \vec{x})W,$$

iff

$$\models_{FM} (\forall \vec{y})(\exists \vec{x})(\exists \vec{z}).\neg Uni \vee \neg A \vee \neg W \vee W',$$
$$\models_{FM} (\forall \vec{x})(\exists \vec{y})(\exists \vec{z}).\neg Uni \vee \neg A \vee \neg W' \vee W,$$

iff

$$\models_{FM} (\forall \vec{u})(\forall \vec{y})(\exists \vec{x})(\exists \vec{z}).\neg Uni(\vec{c}/\vec{u}) \vee \neg A \vee \neg W(\vec{c}/\vec{u}) \vee W'(\vec{c}/\vec{u}),$$
$$\models_{FM} (\forall \vec{u})(\forall \vec{x})(\exists \vec{y})(\exists \vec{z}).\neg Uni(\vec{c}/\vec{u}) \vee \neg A \vee \neg W'(\vec{c}/\vec{u}) \vee W(\vec{c}/\vec{u}),$$

where $\vec{c}$ is the tuple of m constants in Uni, $\vec{u}$ a tuple of m new variables, and for any formula φ, $\varphi(\vec{c}/\vec{u})$ is the result obtained from φ by replacing each constant in $\vec{c}$ by its corresponding variable in $\vec{u}$.

Thus by Lemma 4.1, P and Q are strongly equivalent iff the following two conditions hold:

1. $(\forall \vec{u})(\forall \vec{y})(\exists \vec{x})(\exists \vec{z}).\neg Uni(\vec{c}/\vec{u}) \vee \neg A \vee \neg W(\vec{c}/\vec{u}) \vee W'(\vec{c}/\vec{u})$ is true in every interpretation of size $m+n$.
2. $(\forall \vec{u})(\forall \vec{x})(\exists \vec{y})(\exists \vec{z}).\neg Uni(\vec{c}/\vec{u}) \vee \neg A \vee \neg W'(\vec{c}/\vec{u}) \vee W(\vec{c}/\vec{u})$ is true in every interpretation of size $m+n$.

The above two conditions hold iff for some domain D of $m+n$ objects, P_D and Q_D are equivalent as two propositional logic programs. ■

Corollary 4.1 *The question of whether any two logic programs that may contain variables and constants but not proper functions are strongly equivalent is co-NP-complete.*

Proof: From Theorem 5 and the fact that the size of P_D is polynomial in terms of the size of P and D. ■

It is unlikely that this result will hold for logic programs with function symbols as these programs cannot be instantiated into finite propositional ones.

5 Remarks

5.1 Relative equivalence

In the most general case, we can define equivalence between two logic programs relative to a set of atoms: two logic programs P_1 and P_2 are said to be *equivalent with respect to a set* A of atoms if for any program P that mention only atoms in A, $P_1 \cup P$ and $P_2 \cup P$ are equivalent. Clearly, two programs are equivalent if they are equivalent w.r.t. the empty set, and strongly equivalent if they are equivalent w.r.t. the set of all atoms in the language.

It is possible that different applications may call for different notions of equivalence between logic programs. For the purpose of logic program optimization, perhaps the most appropriate is the one like that for datalog programs. For instance, the following program [Niemelä, 1999] for solving 3-color graph coloring problem is typical of those in answer set programming applications:

```
color(red).
color(blue).
color(yellow).
col(X,red) :- node(X), not col(X, blue),
              not col(X,yellow).
col(X,blue) :- node(X), not col(X, red),
               not col(X,yellow).
col(X,yellow) :- node(X), not col(X, blue),
                 not col(X,red).
fail :- edge(X,Y), color(C), col(X,C), col(Y,C).
```

Notice here that there is no rules about `edge` and `node` relations - these are supposed to be extensional predicates whose definitions will be provided once a specific graph is given. If one wants to optimize this program by finding an equivalent one that can be more efficiently computed using, say *smodels*, then the equivalence should be proved w.r.t. extensional predicates. However, as we know that this notion of equivalence is not in general decidable, strong equivalence may well turn out to be a useful approximation.

Another possible use of this notion of relative equivalence is to study some program transformation rules that are not, but almost, strongly equivalent. For instance, the transformation given in Section 3 for eliminating constraints are equivalent w.r.t. any language that does not contain $fail$ and $dummy$.

5.2 Related work

This work is closely related to and strongly influenced by [Lifschitz *et al.*, 2001; Turner, 2001a; 2001b]. Semantically, as can be seen from the proof of Theorem 1, the ordinary propositions correspond to the first element, and the primed the second element in Turner's HT-model. Similarly, the ordinary propositions are "here" and primed "there" in the logic of here-and-there. Indeed [Pearce *et al.*, 2001] proposed a translation of logic of here-and-there to propositonal logic using similar techniques.

There are some differences in the classes of logic programs considered here and in [Lifschitz *et al.*, 2001; Turner, 2001a; 2001b]. On the one hand, in the propositional case, the class of logic programs considered in [Lifschitz *et al.*, 2001; Turner, 2001a; 2001b] is much more expressive than that considered in this paper. Their logic programs, called nested formulas,[5] allow rules of the form $\varphi \leftarrow \phi$, where φ and ϕ are formulas constructed from atoms using "not", ";", and ",". For instance, $(\mathsf{not}\, a); (\mathsf{not}\, b, a) \leftarrow (\mathsf{not}\, c), ((\mathsf{not}\, a); d)$ is a rule. While we have not proved it, we believe our translation can be extended to rules of this form as well: again each rule will be translated to two formulas: in one of them each atom is replaced by a new one formed by appending prime to its end, and in the other only atoms under the scope of not are so changed; and in both formulas, "not" will be replaced by $\neg$, ";" by $\vee$, and "," by $\wedge$. For instance, the above rule will be translated into the following two formulas:

$$\neg c' \wedge (\neg a' \vee d) \supset (\neg a' \vee (\neg b' \wedge a)),$$
$$\neg c' \wedge (\neg a' \vee d') \supset (\neg a' \vee (\neg b' \wedge a')).$$

On the other hand, we considered rules with variables and constants. We believe this extension is important as in applications, most programs have variables and many of them make use of constants. As we have seen above, this is a non-trivial extension even though the variables are eventually instantiated. It happens that for the problem of verifying strong equivalence, it remains to be co-NP-complete, but for normal equivalence, it becomes undecidable when rules have variables.

Historically, the translation that we gave above from logic programs to propositional theories has its root in a translation from logic programs to circumscriptive theories in [Lin, 1991]. Specifically, each rule of the form:

$$h \leftarrow p_1, \cdots, p_m, \mathsf{not}\, p_{m+1}, \cdots, \mathsf{not}\, p_n$$

is translated into the following sentence:

$$p_1 \wedge \cdots \wedge p_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h.$$

For each program P, we will then get a propositional theory $T(P)$, and the original propositions in $T(P)$ are then circumscribed with the new "primed propositions" fixed. Finally, the equivalence $p \equiv p'$ for each proposition p is added to the result of the circumscription. It is shown [Lin, 1991] that each model of the final theory corresponds to a stable model of P, and vise versa.

As one can see, this translation is part of the one given in section 2 for programs without disjunctions and constraints. As it turns out, for capturing stable models, we could use the translation in section 2 as well:

$$Circum(\Gamma(P) \cup \Sigma; L; L') \wedge \bigwedge_{p \in L} (p \equiv p')$$

is equivalent to

$$Circum(T(P); L; L') \wedge \bigwedge_{p \in L} (p \equiv p'), \qquad (4)$$

where $L' = \{p' \mid p \in L\}$, $\Sigma = \{p \supset p' \mid p \in L\}$, and $Circum(W; A; B)$ denotes the circumscription of propositions in A in the theory W with propositions in B fixed.

[5] Turner [2001b] even allowed weight constraints.

This translation from normal logic programs to circumscription in [Lin, 1991] was originally derived from a translation from default logic to Lin and Shoham's logic of GK [Lin and Shoham, 1992], which is a preferential logic based on a bi-modal logic. A result by Lifschitz [1994] shows that disjunctive logic programs can be similarly captured in logic of GK, which implies that the aforementioned translation from logic programs to circumscriptive theories can be extended to disjunctive logic programs as well. In fact, it can be shown that if we let $T(P)$ to be the theory that contains formulas (2) and (3) for each rule (1) in P, then there will be a one-to-one correspondence between answer sets of P and models of (4).

6 Conclusions

We have investigated strong equivalence in logic programming using classical logic, and proved that the problem is co-NP-complete. There are several directions for future work. An important one is to collect a set of pairs of strongly equivalent programs that can be effectively used to optimize logic programs and queries.

7 Acknowledgements

I have benefited from discussing this work with Jicheng Zhao. I would also like to thank Vladimir Lifschitz and Hudson Turner for their useful comments on an earlier version of this paper. My special thanks to Hudson for giving me permission to include in this paper his translation from clauses to rules that does not use any constraints. Part of the work was done when the author was visiting the University of New South Wales and University of Western Sydney under the sponsorship of Norman Foo and Yan Zhang, respectively. This work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grants HKUST6145/98E and HKUST6061/00E.

References

[Ebbinghaus and Flum, 1999] H. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, second edition, 1999.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Lifschitz *et al.*, 2001] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, To appear, 2001.

[Lifschitz, 1994] V. Lifschitz. Minimal belief and negation as failure. *Artificial Intelligence*, 70:53–72, 1994.

[Lin and Shoham, 1992] F. Lin and Y. Shoham. A logic of knowledge and justified assumptions. *Artificial Intelligence*, 57:271–289, 1992.

[Lin, 1991] F. Lin. *A Study of Nonmonotonic Reasoning*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, 1991.

[Maher, 1988] M. J. Maher. Equivalences of logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann Publishers, San Mateo, CA., 1988.

[Mendelson, 1987] E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth & Brooks/Cole Advanced Books & Software, third edition, 1987.

[Niemelä, 1999] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. and AI*, 25(3-4):241–273, 1999.

[Pearce *et al.*, 2001] D. Pearce, H. Tompits, and S. Woltran. Encodings for Equilibrium Logic and Logic Programs with Nested Expressions. In *Proc. EPIA-01*, pages 306–320, 2001.

[Sagiv, 1988] Y. Sagiv. Optimizing datalog programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann Publishers, San Mateo, CA., 1988.

[Shmueli, 1986] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Sixth ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of database Systems*, pages 237–249, 1986.

[Turner, 2001a] H. Turner. Strong equivalence for logic programs and default theories (made easy). In *Proceedings of LPNMR'2001*, pages 81–92, 2001.

[Turner, 2001b] H. Turner. Strong equivalence made easy: nested expressions and weight constraints. Draft available on the author's web page, 2001.