Discovering Classes of Strongly Equivalent Logic Programs*

Fangzhen Lin
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Yin Chen
Software Institute
Sun Yat-Sen University
Guangzhou, China

Abstract

We report on a successful experiment of computer-aided theorem discovery in the area of logic programming with answer set semantics. Specifically, with the help of computers, we discovered exact conditions that capture the strong equivalence between a set of a rule and the empty set, a set of a rule and another set of a rule, a set S of two rules and a subset of S with one rule, a set of two rules and a set of another rule, and a set S of three rules and a subset of S with two rules. We prove some general theorems that can help us verify the correctness of these conditions, and discuss the usefulness of our results in program simplification.

1 Introduction

This paper makes two contributions. First, it reports on a successful experiment of computer-aided theorem discovery in logic programming with answer set semantics. Second, it contributes to the theory and practice of logic programming in that the discovered theorems that capture certain classes of strongly equivalent logic programs are new, non-trivial, and lead to new program simplification rules that preserve strong equivalence.

Theorem discovery is a highly creative human process. Generally speaking, we can divide it into two steps: (i) conjecture formulation, and (ii) conjecture verification, and computers can help in both of these two steps. For instance, machine learning tools can be used in the first step, i.e. in coming up with reasonable conjectures, and automated deduction tools can be used in the second step, i.e. in verifying the correctness of these conjectures.

While theorem discovery may make use of learning, these two tasks are fundamentally different. Theorem discovery starts with a theory, and aims at finding *interesting* consequences of the theory, while learning is mostly about induction, i.e. it starts with examples/consequences, and aims at finding a theory that would explain the given examples/consequences.

*Our thanks to Yan Zhang for his comments on an earlier version of this paper. This work was supported in part by HK RGC CERG HKUST6170/04E.

Using computers to discover theorems is an old aspiration. There have been many success stories. For instance, AM [Lenat, 1979] was reported to be able to come up with some interesting concepts and theorems in number theory, and the remarkable system described in [Petkovsek *et al.*, 1996] automates the discovery and proofs of identities, especially hypergeometric identities involving sums of binomial coefficients that are important for the analyses of algorithms. Yet another example where "interesting" theorems can be discovered almost fully automatically is a recent work by Lin [2004] on discovering state invariants in planning domains. Lin showed that there are ways to classify potentially interesting constraints according to their syntactic properties, and these constraints can be easily enumerated for most domains. Furthermore, for many of these constraints whether they are invariants can be checked automatically. As a result, the system described in [Lin, 2004] could discover many common invariants in planning domains, and for the logistics domain, it could even discover a set of "complete" state invariants.

In this paper, we consider the problem of discovering theorems about strongly equivalent logic programs under answer set semantics.

The notion of strongly equivalent logic programs is interesting for a variety of reasons. For instance, as Lifschitz *et al.* [2001] noted, if two sets of rules are strongly equivalent, then one can be replaced by the other in any logic program regardless of the context. Thus identifying strongly equivalent sets of rules is a useful exercise that may have applications in program simplification, and the purpose of this paper is to discover conditions under which a set of rules is strongly equivalent to another. It is important that these conditions need to be computationally effective as in general checking if two sets of rules are strongly equivalent is coNP-complete (c.f.[Lin, 2002]).

To discover these conditions, we follow the methodology of [Lin, 2004] by looking at domains of small sizes first. For instance, to discover for what kinds of rules r_1 and r_2 we have that $\{r_1\}$ is strongly equivalent to $\{r_2\}$, we consider a language with, say three atoms, and enumerate all possible pairs of rules in this language that are strongly equivalent. We then conjecture a condition that would capture exactly this set of pairs of rules, and then try to verify if this conjecture is true in the general case. In [Lin, 2004], a general theorem is proved to automate the verification part. We try to do the same here by proving some general theorems that will make the verification part easier, almost semi-automatic.

This paper is organized as follows. In the next section, we briefly review the basic concepts of logic programming under answer set semantics. Then in section 3 we state in more precise terms the type of theorems that we want to discover. In section 4 we prove some general theorems that will help us prove these theorems, and in section 5, we describe some of the theorems that we discovered. We then discuss an application to logic program simplification in section 6, and finally we conclude this paper in section 7.

2 Logic programming with answer set semantics

Let L be a propositional language, i.e. a set of atoms. In this paper we shall consider logic programs with rules of the following form:

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

where h_i's and p_i's are atoms in L. So a logic program here can have default negation (not), constraints (when $k = 0$), and disjunctions in the head of its rules. In the following, if r is a rule of the above form, we write Hd_r to denote the set $\{h_1, ..., h_k\}$, Ps_r the set $\{p_1, ..., p_m\}$, and Ng_r the set $\{p_{m+1}, ..., p_n\}$. Thus a rule r can also be written as $Hd_r \leftarrow Ps_r, not\, Ng_r$. The semantics of these programs are given by answer sets as defined in [Gelfond and Lifschitz, 1991]. To save space, we do not give the definition here.

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] (under the language L) if for any logic program P in L, $P \cup P_1$ and $P \cup P_2$ are equivalent. For example, $\{a \leftarrow b\}$ and $\{a \leftarrow c\}$ are equivalent, but not strongly equivalent. It can be shown that $\{a \leftarrow not\, a\}$ and $\{\leftarrow not\, a\}$ are strongly equivalent.

Lifschitz *et al.* [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. Lin [2002] provided a mapping from logic programs to propositional theories and showed that two logic programs are strongly equivalent iff their corresponding theories in propositional logic are equivalent. This result will be used here both for generating example pairs of strongly equivalent logic programs, and for verifying a conjecture. We repeat it here.

Let P_1 and P_2 be two finite logic programs, and L the set of atoms in them.

Theorem 1 *[Lin, 2002] P_1 and P_2 are strongly equivalent iff in propositional logic, the following sentence is valid:*

$$(\bigwedge_{p \in L} p \supset p') \supset [\bigwedge_{r \in P_1} \delta(r) \equiv \bigwedge_{r \in P_2} \delta(r)], \qquad (2)$$

where for each $p \in L$, p' is a new atom, and for each rule r of the form (1), $\delta(r)$ is the conjunction of the following two sentences:

$$p_1 \wedge \cdots \wedge p_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h_1 \vee \cdots \vee h_k, \quad (3)$$

$$p'_1 \wedge \cdots \wedge p'_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h'_1 \vee \cdots \vee h'_k. \quad (4)$$

Notice that if $m = n = 0$, then the left sides of the implications in (3) and (4) are considered to be $true$, and if $k = 0$, then the right sides of the implications in (3) and (4) are considered to be $false$.

3 The problem

As we mentioned above, one possible use of the notion of strongly equivalent logic programs is in program simplification. For instance, given a logic program, for each rule r in it, we may ask whether it can be deleted without knowing what other rules are in P, i.e. whether $\{r\}$ is strongly equivalent to the empty set. Or we may ask whether a rule r in P can be deleted if one knows that another rule r' is already in P, i.e. whether $\{r, r'\}$ is strongly equivalent to $\{r'\}$. In general, we may ask the following k-m-n question: Is $\{r_1, ..., r_k, u_1, ..., u_m\}$ strongly equivalent to $\{r_1, ..., r_k, v_1, ..., v_n\}$? Thus our theorem discovery task is to come up, for a given k-m-n problem, a computationally effective condition that holds if and only if the answer to the k-m-n question is positive.

Now suppose we have such a condition C, and suppose that when $\{r_1, ..., r_k, u_1, ..., u_m\}$ is strongly equivalent to $\{r_1, ..., r_k, v_1, ..., v_n\}$, it is better to replace $\{u_1, ..., u_m\}$ by $\{v_1, ..., v_n\}$ in the presence of $r_1, ..., r_k$ for the purpose of, say computing the answer sets of a program. One way to use this result to simplify a given program P is to first choose k rules in P, and for any other m rules in it, try to find n rules so that the condition C holds, and then replace the m rules in P by the simpler n rules.

However, even if checking whether C holds would take a negligible constant time, using the above procedure to simplify a given logic program will be practical only when k, m, n are all very small or when k is almost the same as the number of the rules in the given program, and m and n are very small. Thus it seems to us that it is worthwhile to solve the k-m-n problem only when k, m, n are small. In particular, in this paper, we shall concentrate on the 0-1-0 problem (whether a rule can always be deleted), the 0-1-1 problem (whether a rule can always be replaced by another one), the 1-1-0 problem (in the presence of a rule, whether another rule can be deleted), the 2-1-0 problem (in the presence of two rules, whether a rule can always be deleted), and the 0-2-1 problem (if a pair of rules can be replaced by a single rule).

An example of theorems that we want to discover about these problems is as follows:

> For any rule r, $\{r\}$ is strongly equivalent to the empty set $\emptyset$ iff $(Hd_r \cup Ng_r) \cap Ps_r \neq \emptyset$. $\qquad (*)$

4 Some General Theorems

In this section, we prove some general theorems that will help us verify whether an assertion like $(*)$ above is true.

Let L be a propositional language, i.e. a set of atoms. From L, construct a first-order language F_L with equality, two unary predicates H_1 and H_2, three unary predicates Hd_r, Ps_r, and Ng_r for each logic program rule r in L (we assume that each rule in L has a unique name), and three unary predicates X_i, Y_i, and Z_i for each positive number i.

Notice that we have used Hd_r, Ps_r, and Ng_r to denote sets of atoms previously, but now we overload them as unary predicates. Naturally, the intended interpretations of these unary predicates are their respective sets.

Definition 1 *Given a set L of atoms, an* intended model *of F_L is one whose domain is L, and for each rule r in L, the unary predicates Ps_r, Hd_r, and Ng_r are interpreted by their corresponding sets of atoms, Ps_r, Hd_r, and Ng_r, respectively.*

Conditions on rules in L, such as $Ps_r \cap Ng_r \neq \emptyset$, will be expressed by special sentences called *properties* in F_L.

Definition 2 *A sentence of F_L is a property about n rules if it is constructed from equality and predicates X_i, Y_i, and Z_i, $1 \leq i \leq n$. A property Φ about n rules is true (holds) on a sequence $P = [r_1, ..., r_n]$ of n rules if $\Phi[P]$ is true in an intended model of F_L, where $\Phi[P]$ is obtained from Φ by replacing each X_i by Hd_{r_i}, Y_i by Ps_{r_i}, and Z_i by Ng_{r_i}.*

Notice that since $\Phi[P]$ does not mention predicates X_i, Y_i, Z_i, H_1, and H_2, if it is true in one intended model, then it is true in all intended models.

As we have mentioned above, we are interested in capturing the strong equivalence between two programs by a computationally effective condition. More specifically, for some small k, m, and n, we are interested in finding a property Φ about $k+m+n$ rules such that for any sequence of $k+m+n$ rules, $P = [r_1, ..., r_k, u_1, ..., u_m, v_1, ..., v_n]$,

$\{r_1, ..., r_k, u_1, ..., u_m\}$ and $\{r_1, ..., r_k, v_1, ..., v_n\}$ are strongly equivalent if and only if Φ is true on P. (5)

We shall now prove some general theorems that can help us verify the above assertion for a class of formulas Φ.

First of all, Theorem 1 can be reformulated in F_L as follows by reading $H_1(p)$ as "p holds", and $H_2(p)$ as "p' holds":

Theorem 2 *P_1 and P_2 are strongly equivalent in L iff the following sentence*

$$\forall x(H_1(x) \supset H_2(x)) \supset [\bigwedge_{r\in P_1} \gamma(r) \equiv \bigwedge_{r\in P_2} \gamma(r)] \qquad (6)$$

is true in all intended models of F_L, where $\gamma(r)$ is the conjunction of the following two sentences:

$$[\forall x(Ps_r(x) \supset H_1(x)) \wedge \forall x(Ng_r(x) \supset \neg H_2(x))] \supset \exists x(Hd_r(x) \wedge H_1(x)), \qquad (7)$$

$$[\forall x(Ps_r(x) \supset H_2(x)) \wedge \forall x(Ng_r(x) \supset \neg H_2(x))] \supset \exists x(Hd_r(x) \wedge H_2(x)). \qquad (8)$$

In first order logic, if a prenex formula of the form $\exists\vec{x}\forall\vec{y}B$ is satisfiable, then it is satisfiable in a structure with n elements, where n is the length of $\vec{x}$ if it is non-empty, and 1 when $\vec{x}$ is empty. We can prove a similar result for our first-order languages and their intended models here.

From this and Theorem 2, we can show the following theorem which will enable us to automate the verification of the "if" part of (5) when the property Φ is in the prenex format.

Theorem 3 *Without loss of generality, suppose $m \geq n$. If Φ is a property about k+m+n rules of the form $\exists\vec{x}\forall\vec{y}Q$, where $\vec{x}$ is a tuple of w variables, and Q a formula that does not have any quantifiers, then the following two assertions are equivalent:*

(a) For any sequence of k+m+n rules, $P = [r_1, ..., r_k, u_1, ..., u_m, v_1, ..., v_n]$, if Φ is true on P, then $\{r_1, ..., r_k, u_1, ..., u_m\}$ is strongly equivalent to $\{r_1, ..., r_k, v_1, ..., v_n\}$.

(b)(b.1) If $n > 0$, then for any sequence $P = [r_1, ..., r_k, u_1, ..., u_m, v_1, ..., v_n]$ of rules with at most $w + 2(k + m)$ atoms, if Φ is true on P, then $\{r_1, ..., r_k, u_1, ..., u_m\}$ is strongly equivalent to $\{r_1, ..., r_k, v_1, ..., v_n\}$.

(b.2) If $n = 0$, then for any sequence $P = [r_1, ..., r_k, u_1, ..., u_m]$ of rules with at most K atoms, if Φ is true on P, then $\{r_1, ..., r_k, u_1, ..., u_m\}$ is strongly equivalent to $\{r_1, ..., r_k\}$, where K is $w + 2k$ if $w + 2k > 0$, and $K = 1$ otherwise.

The "only if" part of (5) can often be proved with the help of the following theorem.

Theorem 4 *Let L_1 and L_2 be two languages, and f a function from L_1 to L_2. If P_1 and P_2 are two programs in L_1 that are strongly equivalent, then $f(P_1)$ and $f(P_2)$ are two programs in L_2 that are also strongly equivalent. Here $f(P)$ is obtained from P by replacing each atom p in it by $f(p)$.*

Proof: By Theorem 1 and the fact that in propositional logic, if φ is a tautology, and f a function from L_1 to L_2, then $f(\varphi)$ is also a tautology, where $f(\varphi)$ is the formula obtained from φ by replacing each atom p in it by $f(p)$. ■

For an example of using the theorems in this section for proving assertions of the form (5), see Section 5.1.

5 Computer aided theorem discovery

Given a *k-m-n* problem, our strategy for discovering theorems about it is as follows:

1. Choose a small language L;
2. Generate all possible triples
$$(\{r_1, ..., r_k\}, \{u_1, ..., u_m\}, \{v_1, ..., v_n\}) \qquad (9)$$
of sets of rules in L such that $\{r_1, ..., r_k, u_1, ..., u_m\}$ is strongly equivalent to $\{r_1, ..., r_k, v_1, ..., v_n\}$ in L;
3. Formulate a conjecture on the *k-m-n* problem that holds in the language L, i.e. a condition that is true for a triple of the form (9) iff it is generated in Step 2;
4. Verify the correctness of this conjecture in the general case.

This process may have to be iterated. For instance, a conjecture came up at Step 3 may fail to generalize in Step 4, so the whole process has to be repeated. Or we may start with a language L with, say three atoms but have to increase it to five or six atoms later on.

Ideally, we would like this process to be automatic. However, it is difficult to automate Steps 3 and 4 - the number of possible patterns that we need to examine in order to come up with a good conjecture in Step 3 is huge, and we do not have a general theorem that enables us to automate the verification part in Step 4: while Theorem 3 enables us to automate the proof of the sufficient part of the assertion (5) for a class of formulas Φ, we do not have a similar result for the necessary part. Theorem 4 helps, but it is not fully automatic yet. Nonetheless, computers play a crucial role in all steps, and in the following we report some of the theorems discovered using the above procedure.

5.1 The 0-1-0 problem

This problem asks if a given rule is strongly equivalent to the empty set, thus can always be deleted from any program. We have the following experimental result:

Lemma 1 *If a rule r mentions at most three distinct atoms, then $\{r\}$ is strongly equivalent to $\emptyset$ iff $(Hd_r \cup Ng_r) \cap Ps_r \neq \emptyset$.*

Using Theorem 4, we can show the following result:

Lemma 2 *If there is a rule r of the form (1) such that $\{r\}$ is strongly equivalent to $\emptyset$ and $(Hd_r \cup Ng_r) \cap Ps_r \neq \emptyset$ is not true, then there is such a rule that mentions at most three atoms.*

Proof: Suppose $\{r\}$ is strongly equivalent to $\emptyset$, $Hd_r \cap Ps_r = \emptyset$, and $Ps_r \cap Ng_r = \emptyset$. Suppose L is the set of atoms in r, and a, b, c are three new atoms. Let

$$f(p) = \begin{cases} a & p \in Hd_r \\ b & p \in Ps_r \\ c & otherwise \end{cases}$$

By Theorem 4, $\{f(r)\}$ is also strongly equivalent to $\emptyset$. By the construction of f, we also have $Hd_{f(r)} \cap Ps_{f(r)} = \emptyset$, and $Ps_{f(r)} \cap Ng_{f(r)} = \emptyset$, and that $f(r)$ mentions at most three distinct atoms. ■

Theorem 5 (The 0-1-0 problem) *Lemma 1 holds in the general case, i.e. without any restriction on the number of atoms in r.*

Proof: We notice that the condition in Lemma 1, $(Hd_r \cup Ng_r) \cap Ps_r \neq \emptyset$, is equivalent to the following property

$$\exists x.(X_1(x) \vee Z_1(x)) \wedge Y_1(x)$$

being true on $[r]$. Thus the "if" part follows from Theorem 3 and Lemma 1. The "only if" part follows from Lemma 1 and Lemma 2. ■

The "if" part of the theorem is already well-known, first proved by Osorio *et. al.* [2001]. To the best of our knowledge, the "only if" part is new.

We notice here that there is no need to consider the 0-n-0 problem for $n > 1$, because for any n, $\{r_1, ..., r_n\}$ is strongly equivalent to $\emptyset$ iff for each $1 \leq i \leq n$, $\{r_i\}$ is strongly equivalent to $\emptyset$.

5.2 The 1-1-0 and the 0-1-1 problems

The 1-1-0 problem asks if a rule can always be deleted in the presence of another rule, and the 0-1-1 problem asks if a rule can always be replaced by another one. We first solve the 1-1-0 problem, and the solution to the 0-1-1 problem will come as a corollary.

We have the following experimental result for the 1-1-0 problem:

Lemma 3 *For any two rules r_1 and r_2 that mentions at most three atoms, $\{r_1, r_2\}$ and $\{r_1\}$ are strongly equivalent iff one of the following two conditions is true:*

1. *$\{r_2\}$ is strongly equivalent to $\emptyset$.*
2. *$Ps_{r_1} \subseteq Ps_{r_2}$, $Ng_{r_1} \subseteq Ng_{r_2}$, and $Hd_{r_1} \subseteq Hd_{r_2} \cup Ng_{r_2}$.*

Lemma 4 *If there are two rules r_1 and r_2 such that $\{r_1, r_2\}$ and $\{r_2\}$ are strongly equivalent, but none of the two conditions in Lemma 3 hold, then there are two such rules that mention at most three atoms.*

Theorem 6 (The 1-1-0 problem) *Lemma 3 holds in the general case, without any restriction on the number of atoms in r_1 and r_2.*

Proof: The condition in Lemma 3 is equivalent to the following property

$$[\exists x.(X_2(x) \vee Z_2(x)) \wedge Y_2(x)] \vee$$
$$\{[\forall x.Y_1(x) \supset Y_2(x)] \wedge [\forall x.Z_1(x) \supset Z_2(x)] \wedge$$
$$[\forall x.X_1(x) \supset (X_2(x) \vee Z_2(x))]\}$$

being true on $[r_1, r_2]$. Thus the "if" part follows from Theorem 3 and Lemma 3, by noticing that the above property can be written as $\exists x \forall \vec{y}.Q$ as required by Theorem 3. The "only if" part follows from Lemma 3 and Lemma 4. ■

Thus if a rule r_2 cannot be deleted on its own but can be deleted in the presence of another rule r_1, then it must be the case that r_2 is redundant given r_1: if the body of r_2 is satisfied, then the body of r_1 is satisfied as well; furthermore, r_2 can entail no more than what can be entailed by r_1 ($Hd_{r_1} \subseteq Hd_{r_2} \cup Ng_{r_2}$).

Osorio *et al.* [2001] proved that $\{r_1, r_2\}$ and $\{r_1\}$ are strongly equivalent if either $Ps_{r_1} \cup Ng_{r_1} = \emptyset$ and $Hd_{r_1} \subseteq Ng_{r_2}$ or $Ps_{r_1} \subseteq Ps_{r_2}$, $Ng_{r_1} \subseteq Ng_{r_2}$, and $Hd_{r_1} \subseteq Hd_{r_2}$. More recently, Eiter *et al.* [2004] showed that $\{r_1, r_2\}$ and $\{r_1\}$ are strongly equivalent if r_1 *s-implies* r_2 [Wang and Zhou, 2005], i.e. if there exists a set $A \subseteq Ng_{r_2}$ such that $Hd_{r_1} \subseteq Hd_{r_2} \cup A$, $Ng_{r_1} \subseteq Ng_{r_2} \setminus A$, and $Ps_{r_1} \subseteq Ps_{r_2}$.

As one can see, these are all special cases of the "if" part of Theorem 6. Our result is actually more general. For instance, these special cases do not apply to $\{(c \leftarrow b, not\, c), (\leftarrow b, not\, c)\}$ and $\{c \leftarrow b, not\, c\}$, but one can easily show that these two sets are strongly equivalent using our theorem.

From our solution to the 1-1-0 problem, we can derive a solution to the 0-1-1 problem.

Theorem 7 (The 0-1-1 problem) *For any two rules r_1 and r_2, $\{r_1\}$ and $\{r_2\}$ are strongly equivalent iff one of the following two conditions is true:*

1. *$\{r_1\}$ and $\{r_2\}$ are both strongly equivalent to $\emptyset$.*
2. *$Ps_{r_1} = Ps_{r_2}$, $Ng_{r_1} = Ng_{r_2}$, and $Hd_{r_1} \cup Ng_{r_1} = Hd_{r_2} \cup Ng_{r_2}$.*

Proof: By Theorem 1, it is easy to see that $\{r_1\}$ and $\{r_2\}$ are strongly equivalent iff $\{r_1, r_2\}$ and $\{r_1\}$ are strongly equivalent and $\{r_1, r_2\}$ and $\{r_2\}$ are strongly equivalent. ■

Thus two rules r_1 and r_2 can always be interchanged if either both of them can be deleted (strongly equivalent to the empty set) or they have the same body, and the same consequences when the body is true. For instance, $\{a \leftarrow B, not\, a\}$ and $\{\leftarrow B, not\, a\}$ are strongly equivalent no matter what B is, because the two rules have the same body, and when the body is true, the same consequence - a contradiction. As another example, $\{a; b \leftarrow not\, a\}$ and $\{b \leftarrow not\, a\}$ are strongly equivalent because the two rules have the same body, and, when the body is true, the same consequence, b.

5.3 The 2-1-0, 0-2-1, and 0-2-2 problems

The 2-1-0 problem asks if a rule can be deleted in the presence of another two rules, the 0-2-1 problem asks if two rules can be replaced by a single rule, and the 0-2-2 problem asks if two rules can be replaced by another two rules. Similar to the previous subsection, the solution to the 0-2-1 and 0-2-2 problems will follow from a solution to the 2-1-0 problem.

We have the following experimental result for the 2-1-0 problem:

Lemma 5 *For any three rules r_1, r_2 and r_3 that mentions at most six atoms, $\{r_1, r_2, r_3\}$ and $\{r_1, r_2\}$ are strongly equivalent iff one of the following four conditions is true:*

1. *$\{r_3\}$ is strongly equivalent to $\emptyset$.*
2. *$\{r_1, r_3\}$ is strongly equivalent to $\{r_1\}$.*
3. *$\{r_2, r_3\}$ is strongly equivalent to $\{r_2\}$.*
4. *There is an atom p such that:*
   - *4.1 $p \in (Ps_{r_1} \cup Ps_{r_2}) \cap (Hd_{r_1} \cup Hd_{r_2} \cup Ng_{r_1} \cup Ng_{r_2})$*
   - *4.2 $Hd_{r_i} \setminus \{p\} \subseteq Hd_{r_3} \cup Ng_{r_3}$ and $Ps_{r_i} \setminus \{p\} \subseteq Ps_{r_3}$ and $Ng_{r_i} \setminus \{p\} \subseteq Ng_{r_3}$, where $i = 1, 2$*
   - *4.3 If $p \in Ps_{r_1} \cap Ng_{r_2}$, then $Hd_{r_1} \cap Hd_{r_3} = \emptyset$*
   - *4.4 If $p \in Ps_{r_2} \cap Ng_{r_1}$, then $Hd_{r_2} \cap Hd_{r_3} = \emptyset$*

Notice that in principle, given a language L, every subset of L can be the *Hd*, *Ps*, or *Ng* of a rule. Thus when the size of L is six, there are in principle $(2^6)^3 - 1 = 262,143$ possible rules, and $262,143^3$ triples of them. Thus at first glance, it seems that verifying Lemma 5 experimentally using the currently available computers would be impossible. However, we can cut the numbers down significantly with the results that we already have proved. First, we only have to consider rules that do not have common elements in *Hd*, *Ps*, and *Ng*: if either *Hd* and *Ps* or *Ps* and *Ng* have a common element, then by Theorem 5, this rule can be deleted; if *Hd* and *Ng* have common elements, then according to Theorem 7, we obtain a strongly equivalent rule by deleting the common elements in *Hd*. Second, we do not have to consider isomorphic rules: if there is a one-to-one onto function from L to L that maps $\{r_1, r_2, r_3\}$ to $\{r'_1, r'_2, r'_3\}$, then these two sets of rules are essentially the same except for the names of atoms in them. By using a certain normal form for triples of rules that avoids these redundant cases, we ended up with roughly 120 million triples of rules to consider for testing Lemma 5, which took about 10 hours on a Solaris server consisting of 8 Sun Ultra-SPARC III 900Mhz CPUs with 8GB RAM.

The following lemma is the reason why we need to consider a language with six atoms for this problem.

Lemma 6 *If there are three rules r_1,r_2 and r_3 such that $\{r_1, r_2, r_3\}$ and $\{r_1, r_2\}$ are strongly equivalent, but none of the four conditions in Lemma 5 hold, then there are three such rules that mention at most six atoms.*

Theorem 8 (The 2-1-0 problem) *Lemma 5 holds in the general case, without any restriction on the number of atoms in r_1, r_2, r_3.*

The conditions in Lemma 5 (Theorem 8) are rather complex, and the reason why it is difficult to automate Step 3 of the procedure at the beginning of the section. These conditions capture all possible cases when r_3 is "subsumed" by r_1 and r_2, and are difficult to describe concisely by words. We give some examples.

Consider the following three rules: r_1: $(a_2 \leftarrow a_1)$, r_2: $(a_3 \leftarrow not\, a_1)$, and r_3: $(a_3 \leftarrow not\, a_2)$. We have that $\{r_1, r_2, r_3\}$ and $\{r_1, r_2\}$ are strongly equivalent because the condition (4) in Lemma 5 holds.

However, if we change r_3 into r'_3 : $a_2 \leftarrow not\, a_3$, then $P_1 = \{r_1, r_2, r'_3\}$ and $P_2 = \{r_1, r_2\}$ are not strongly equivalent: one could check that condition (4.3) in Lemma 5 does not hold, and indeed, while $P_2 \cup \{a_1 \leftarrow a_2\}$ has a unique answer set $\{a_3\}$, $P_1 \cup \{a_1 \leftarrow a_2\}$ has two answer sets $\{a_3\}$ and $\{a_1, a_2\}$.

It is also easy to show by Theorem 8 that $a_3 \leftarrow not\, a_2$ is "subsumed" by $\{(a_1; a_2; a_3 \leftarrow), (a_2; a_3 \leftarrow a_1)\}$, and $a_2; a_3 \leftarrow$ is "subsumed" by $\{(a_2 \leftarrow a_1), (a_3 \leftarrow not\, a_1)\}$.

With the results that we have, the following theorem will yield a solution to the 0-2-1 problem.

Theorem 9 (the 0-2-1 problem) *For any three rules r_1, r_2 and r_3, $\{r_1, r_2\}$ and $\{r_3\}$ are strongly equivalent iff the following three conditions are true:*

1. *$\{r_1, r_2, r_3\}$ and $\{r_1, r_2\}$ are strongly equivalent.*
2. *$\{r_1, r_3\}$ and $\{r_3\}$ are strongly equivalent.*
3. *$\{r_2, r_3\}$ and $\{r_3\}$ are strongly equivalent.*

For example, $\{(a_2 \leftarrow a_1, not\, a_3), (a_1; a_2 \leftarrow not\, a_3)\}$ is strongly equivalent to $\{a_2 \leftarrow not\, a_3\}$. While $\{(\leftarrow a_2, a_3), (\leftarrow a_3, not\, a_2)\}$ is strongly equivalent to $\{\leftarrow a_3\}$, $\{(a_1 \leftarrow a_2, a_3), (a_1 \leftarrow a_3, not\, a_2)\}$ is not strongly equivalent to $\{a_1 \leftarrow a_3\}$. Similarly, we have the following theorem

Theorem 10 (the 0-2-2 problem) *For any four rules r_1, r_2, r_3, r_4, $\{r_1, r_2\}$ and $\{r_3, r_4\}$ are strongly equivalent iff the following four conditions are true:*

1. *$\{r_1, r_2, r_3\}$ and $\{r_1, r_2\}$ are strongly equivalent.*
2. *$\{r_1, r_2, r_4\}$ and $\{r_1, r_2\}$ are strongly equivalent.*
3. *$\{r_3, r_4, r_1\}$ and $\{r_3, r_4\}$ are strongly equivalent.*
4. *$\{r_3, r_4, r_2\}$ and $\{r_3, r_4\}$ are strongly equivalent.*

6 Program simplification

We have mentioned that one possible use of the notion of strongly equivalent logic programs is in simplifying logic programs: if P and Q are strongly equivalent, and that Q is "simpler" than P, we can then replace P in any program that contains it by Q.

Most answer set programming systems perform some program simplifications. However, only Smodels [Niemelä *et al.*, 2000] has a stand-alone front-end called lparse that can be used to ground and simplify a given logic program. It seems that lparse simplifies a grounded logic program by computing first its well-founded model. It does not, however, perform any program simplification using the notion of strong equivalence. For instance, lparse-1.0.13, the current version of lparse, did nothing to the following set of rules: $\{(a \leftarrow \mathit{not}\, b), (b \leftarrow \mathit{not}\, a), (a \leftarrow a)\}$. Nor does it replace the first rule in the following program $\{(a \leftarrow \mathit{not}\, a), (a \leftarrow \mathit{not}\, b), (b \leftarrow \mathit{not}\, a)\}$ by the constraint $\leftarrow \mathit{not}\, a$.

It is unlikely that anyone would be intentionally writing rules like $a \leftarrow a$ or $b \leftarrow a, \mathit{not}\, a$. But these type of rules can arise as a result of grounding some rules with variables. For instance, the following is a typical recursive rule used in logic programming encoding of the Hamiltonian Circuit problem [Niemelä, 1999; Marek and Truszczynski, 1999]:

$$reached(X) \leftarrow arc(Y, X), hc(Y, X), reached(Y).$$

When instantiated on a graph with cyclic arcs like $arc(a, a)$, this rule generates cyclic rules of the form $reached(X) \leftarrow hc(X, X), reached(X)$. Unless deleted explicitly, these rules will slow down many systems, especially those based on SAT.

It is thus useful to consider using the results that we have here for program simplification. Indeed, transformation rules such as deleting those that contain common elements in their heads and positive bodies have been proposed [Brass and Dix, 1999], and studied from the perspective of strong equivalence [Osorio *et al.*, 2001; Eiter *et al.*, 2004]. Our results add new such transformation rules. For instance, by Theorem 7, we can delete those elements in the head of a rule that also appear in the negation-as-failure part of the rule. Theorems 6, 8, and 9 can also be used to define some new transformation rules.

7 Concluding remarks and future work

Donald Knuth, in his Forward to [Petkovsek *et al.*, 1996], said

> "Science is what we understand well enough to explain to a computer. Art is everything else we do. ...Science advances whenever an Art becomes a Science. And the state of the Art advances too, because people always leap into new territory once they have understood more about the old."

We hope that with this work, we are one step closer to making discovering classes of strongly equivalent logic programs a Science.

We have mentioned that the methodology used in this paper is similar to that in [Lin, 2004]. In both cases, plausible conjectures are generated by testing them in domains of small sizes, and general theorems are proved to aid the verification of these conjectures in the general case. However, while plausible conjectures are generated automatically in [Lin, 2004], they are done manually here. While the verifications of most conjectures in [Lin, 2004] are done automatically as well, they are done only semi-automatically here. Overcoming these two weaknesses is the focus of our future work. Specifically, we would like to make Step 3 of the procedure in Section 5 automatic, and prove a theorem similar to Theorem 3 to automate the proofs of the "only if" parts of theorems like Theorems 5 - 8, in the same way that Theorem 3 makes the proofs of the "if" parts of these theorems automatic. This way, we would be able to discover more interesting theorems in this area, and more easily!

References

[Brass and Dix, 1999] S. Brass and J. Dix. Semantics of (disjunctive) logic programs based on partial evaluation. *J. Log. Program. 40(1)*, pages 1–46, 1999.

[Eiter *et al.*, 2004] T. Eiter, M. Fink, H. Tompits, and S. Woltran. Simplifying logic programs under uniform and strong equivalence. In *LPNMR*, pages 87–99, 2004.

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

[Lenat, 1979] D. B. Lenat. On automated scientific theory formation: A case study using the AM program. *Machine Intelligence 9*, pages 251–283, 1979. Jean Hayes, Donald Michie, and L. I. Mikulich, eds.

[Lifschitz *et al.*, 2001] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[Lin, 2002] F. Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *Proc. KR'02*, pages 170–176, 2002.

[Lin, 2004] F. Lin. Discovering state invariants. In *Proc. KR'04*, pages 536–544, 2004.

[Marek and Truszczynski, 1999] V. W. Marek and M. Truszczynski. Stable logic programming - an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*. K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren, eds, Springer-Verlag, 1999.

[Niemelä *et al.*, 2000] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: a system for answer set programming. In *Proc NMR-2000*. (CoRR: arXiv:cs.AI/0003033) http://www.tcs.hut.fi/Software/smodels/.

[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.

[Osorio *et al.*, 2001] M. Osorio, J. A. Navarro, and J. Arrazola. Equivalence in answer set programming. In *LOPSTR 2001*, pages 57–75, 2001.

[Petkovsek *et al.*, 1996] M. Petkovsek, H. S. Wilf, and D. Zeilberger. *A = B*. Wellesley, Mass. : A K Peters, 1996.

[Wang and Zhou, 2005] K. Wang and L. Zhou. Comparisions and computation of well-founded semantics for disjunctive logic programs. *ACM Transactions on Computational Logic*, 2005. To appear.