First-Order Loop Formulas for Normal Logic Programs [*]

Yin Chen
South China Normal University
Guangzhou, P.R.China

Fangzhen Lin
Hong Kong University of
Science and Technology
Hong Kong

Yisong Wang
Guizhou University
Guiyang, P.R.China

Mingyi Zhang
Guizhou Academy of Sciences
Guiyang, P.R.China

Abstract

In this paper we extend Lin and Zhao's notions of loops and loop formulas to normal logic programs that may contain variables. Under our definition, a loop formula of such a logic program is a first-order sentence. We show that together with Clark's completion, our notion of first-order loop formulas captures the answer set semantics on the instantiation-basis: for any finite set F of ground facts about the extensional relations of a program P, the answer sets of the ground program obtained by instantiating P using F are exactly the models of the propositional theory obtained by instantiating using F the first order theory consisting of the loop formulas of P and Clark's completion of the union of P and F. We also prove a theorem about how to check whether a normal logic program with variables has only a finite number of non-equivalent first-order loops.

Introduction

For a propositional normal logic program, Lin and Zhao (2004) showed that by adding what they called loop formulas to Clark's completion (Clark 1978), one obtains a one to one correspondence between the models of the resulting propositional theory and the answer sets of the logic program. There are several reasons why one wants to extend this result to the first-order case. For one, Clark's completion was originally defined to be a first-order theory on a set of rules with variables. More importantly, in answer set programming (ASP) applications (c.f. (Niemelä 1999; Marek & Truszczynski 1999; Lifschitz 1999; Nogueira *et al.* 2001; Baral 2003; Erdem *et al.* 2003; Eiter *et al.* 2004)), a logic program typically has two parts: a set of general rules with variables that encodes general knowledge about the application domain, and a set of facts that encodes the specific problem instance of the application domain and is used to ground the variables in the set of general rules. This means that when applying a logic program with variables to two problem instances, we have to compute the loops and loop formulas of two different propositional logic programs separately although these two programs, being obtained from grounding the same logic program on different domains, have basically the same structure thus should have the same kinds of loops and loop formulas. By defining loops and loop formulas directly on logic programs with variables, we can hopefully avoid this problem of having to compute similar loops and loop formulas every time a program is grounded on a domain. Thus extending loop formulas in logic programming to first-order case is not only theoretically interesting, but may also be of practical relevance.

Specifically, in this paper, we propose notions of first-order loops and loop formulas so that for any set P of rules with variables and any set F of ground facts about the extensional relations of P, the answer sets of the logic program obtained from grounding P using F will correspond to the propositional models of the theory obtained by instantiating $COMP(P \cup F) \cup LF(P)$ on D, where $COMP(P \cup F)$ is the completion of $P \cup F$, $LF(P)$ the set of loop formulas of P, D the set of constants in P and F, and an extensional relation is one that does not occur in the heads of any rules.

This paper is organized as follows. In the next section, we first define some basic notions such as the instantiations (groundings) of a first-order theory and a set of rules on a given finite domain. We then define the notions of positive dependency graph, loops, and loop formulas of a finite logic program with variables. We then prove some properties of our notions. Besides the correctness result as stated above, we also show some results about when a program has only a finite number of non-equivalent loops. Perhaps not surprisingly, given a finite set P of first-order rules, while the completion of P is a finite first-order theory, the set of loop formulas of P may be an infinite set of first-order formulas because there may be an infinite number of non-equivalent loops. Thus it is important to know whether a given program has only a finite number of non-equivalent loops. As we shall see, surprisingly perhaps, there is an algorithm for checking this.

Logical preliminaries

We assume a first-order language with equality but without proper functions. In the following, we shall call a formula of the form $t = t'$ an *equality atom*, and by an *atom* we mean an atomic formula in the language that is not an equality atom.

[*]Authors' names are listed according to the alphabetical order.

Instantiating sentences on a finite domain

Given a first-order sentence φ, and a finite set D of constants, we define the *instantiation* of φ on D, written $\varphi|D$, to be a propositional formula defined inductively as follows:

- if φ does not have quantifications, then $\varphi|D$ is the result of replacing $d = d$ by $true$ and $d_1 = d_2$ by $false$ in φ, where d is any constant, and d_1 and d_2 are any two distinct constants.
- $\exists x.\varphi|D$ is $(\bigvee_{d\in D} \varphi(x/d))|D$, where $\varphi(x/d)$ is obtained from φ by replacing in it every free occurrence of x by d;
- $(\varphi_1 \vee \varphi_2)|D = \varphi_1|D \vee \varphi_2|D$;
- $(\neg\varphi)|D = \neg(\varphi|D)$.

Other connectives such as $\wedge$ and the universal quantification $\forall$ are treated as shorthands as usual. Notice that when instantiating a first-order sentence on a finite domain, we make unique names assumptions. So the instantiation of $a = a$ is $true$, the instantiation of $a = b$ is $false$, and the instantiation of $\forall x.p(x) \equiv x = a$ on $\{a, b\}$ is equivalent to $p(a) \wedge \neg p(b)$.

Now if T is a first-order theory (i.e. a set of first-order sentences) and D a finite set of constants, then we define the instantiation of T on D, written $T|D$, to be the set of the instantiations of the sentences in T on D.

In the following, we identify a propositional model of a propositional theory with the set of atoms true in the model. Given a first-order structure M with domain D, we define I_M to be following set:

$$\{p(\vec{t}) \mid \vec{t} \in p^M\},$$

where p^M is the interpretation of p in M. Symmetrically, given a set I of atoms, and a set D of constants, we define a first-order structure M_I^D as follows:

- The domain of M_I^D is D, and each constant in D is mapped to itself.
- For each predicate p, $\vec{t} \in p^{M_I^D}$ iff $p(\vec{t}) \in I$.

Clearly, if I is a set of ground atoms that mention only constants from D, then $I_{M_I^D} = I$.

Proposition 1 *Suppose φ is a first-order sentence, and D a finite domain that include all constants in φ. If I is a propositional model of $\varphi|D$, then M_I^D is a model of φ. Conversely, if M is a model of φ whose domain is D and it maps each constant in D to itself, then I_M is a propositional model of $\varphi|D$.*

Proof: We show that more generally, if D is a finite set that includes all of the constants in φ, and I a set of atoms such that

$$I \subseteq \{p(d_1, ..., d_k) \mid p \text{ a predicate and } d_1, ..., d_k \in D\},$$

then for any variable assignment σ, any formula ψ that mentions constants only in D, $M_I^D, \sigma \models \psi$ iff $I \models (\psi\sigma)|D$, where $\psi\sigma$ is the result of replacing every free variable in ψ by its value in σ. We prove this by induction.

- $M_I^D, \sigma \models t_1 = t_2$ iff $t_1\sigma = t_2\sigma$ (since all constants in t_1 and t_2 are in D, thus mapped to themselves) iff $[(t_1 = t_2)\sigma]|D$ is $true$.
- $M_I^D, \sigma \models p(\vec{t})$ iff $\vec{t}\sigma \in p^{M_I^D}$ iff $p(\vec{t})\sigma \in I$ iff $[p(\vec{t})\sigma]|D \in I$.
- $M_I^D, \sigma \models \exists x.\psi$ iff for some σ' that differs from σ only on x, we have that $M_I^D, \sigma' \models \psi$
  iff for some σ' that differs from σ only on x, we have that $(\psi\sigma')|D$ is true in I (by the inductive assumption)
  iff for some $d \in D$, $((\psi(x/d))\sigma)|D$ is true in I
  iff $\bigvee_{d\in D}((\psi(x/d))\sigma)|D$ is true in I
  iff $[(\bigvee_{d\in D} \psi(x/d))\sigma]|D$ is true in I
  iff $[(\exists x\psi)\sigma]|D$ is true in I.
- The cases for $\forall x\psi$, $\neg\psi$ and $\psi_1 \vee \psi_2$ are similar.

Similarly, we can show that if D is a finite set that includes all constants in φ, and M a first-order structure whose domain is D and such that it maps a constant in D to itself, then for any variable assignment σ, any formula ψ, $M, \sigma \models \psi$ iff $I_M \models (\psi\sigma)|D$. ■

The following is the key result about our notion of instantiation.

Proposition 2 *For any sentences φ and ψ, and any domain D that includes the constants in φ and ψ, if φ and ψ are logically equivalent in first-order logic (with equality), then the instantiations of φ and ψ on D are logically equivalent in propositional logic.*

Proof: Suppose I satisfies $\varphi|D$, then M_I^D is a model of φ. So M_I^D is a model of ψ. Thus $I_{M_I^D}$ satisfies $\psi|D$. But $I_{M_I^D} = I$. So I satisfies $\psi|D$ as well. ■

Notice that it is crucial for D to include all constants in φ and ψ. For instance, $\exists x(x = a)$ is logically equivalent to $true$, but $\exists x(x = a)|\{b\}$ is $false$.

Instantiating logic programs on a finite domain

A (first-order normal) logic program P is a set of (first-order normal) rules of the form:

$$h \leftarrow Body \qquad (1)$$

where h is an atom, and $Body$ is a set of atoms, equality atoms, and expressions of the form $not\ A$, where A is either an atom or an equality atom. In the following, we use $t \neq t'$ as a shorthand for $not\, t = t'$. (However, as usual, when $t \neq t'$ occurs in a formula, it stands for $\neg t = t'$.)

According to this definition, a logic program can contain an infinite number of rules. However, in this paper, we consider only finite logic programs because Clark's completion may not be well-defined for an infinite set of rules. So in the following, unless otherwise specified, a logic program will be assumed to be finite.

In this paper, the semantics of a logic program with variables will be defined according to the answer set semantics (Gelfond & Lifschitz 1991) of propositional logic programs by grounding the first-order program on a domain.

Let r be a rule, we shall denote by $Var(r)$ the set of variables occurring in r, and $Const(r)$ the set of constants occurring in r. We call a variable $x \in Var(r)$ a *local* variable of r if it does not occur in the head of the rule. Similarly, given a set P of rules, we shall denote by $Var(P)$ the set of variables in P, and $Const(P)$ the set of constants in P.

Given a set V of variables, and a set D of constants, a variable assignment of V on D is a mapping from V to D. If r is a rule, and σ a variable assignment, then $r\sigma$ is the result of replacing the variables in r by their values in σ.

Let r be a rule and D a set of constants. We define *the instantiation of* r *on* D, written $r|D$, to be the set of ground rules obtained from

$$R = \{r\sigma \mid \sigma \text{ is a variable assignment of } Var(r) \text{ on } D\}$$

by the following two transformations:

- if a rule in R contains either $a = b$ for some distinct constants a and b or $a \neq a$ for some constant a, then delete this rule;
- delete $a = a$ and $a \neq b$ for all constants a and b in the bodies of the remaining rules.

For instance, if r is the rule

$$p(x) \leftarrow q(x, y), x = a, x \neq y,$$

then its instantiation on $\{a, b\}$, $r|\{a, b\}$, is

$$\{p(a) \leftarrow q(a, b)\}.$$

The instantiation of a program P on D, written $P|D$, is then the union of the instantiations of all the rules in P on D.

Normal forms and program completions

Let r be a rule of the form (1), and suppose that h is $p(\vec{t})$ for some predicate p and tuple $\vec{t}$ of terms. If $\vec{x}$ is a tuple of variables not in r, and matches p's arity (so that $p(\vec{x})$ is well-formed), then the *normal form* of r on $\vec{x}$ is the following rule:

$$p(\vec{x}) \leftarrow \vec{x} = \vec{t} \cup Body,$$

where $(x_1, ..., x_k) = (t_1, ..., t_k)$ is the set:

$$\{x_1 = t_1, ..., x_k = t_k\}.$$

For instance, the normal form of the following rule

$$p(x, z) \leftarrow q(x, y), x \neq y$$

on (u, v) is

$$p(u, v) \leftarrow x = u, z = v, q(x, y), x \neq y$$

Notice that if none of the variables in $\vec{x}$ occur in a rule r, then all variables in r become local variables in the normal form of r on $\vec{x}$.

Using this notion of normal forms, we can define the *completion* of a predicate p under a program P, written $COMP(p, P)$, to be the following formula:

$$\forall \vec{x}.p(\vec{x}) \equiv \bigvee_{1 \leq i \leq k} \exists \vec{y_i} \widehat{Body_i},$$

where

- $\vec{x}$ is a tuple of distinct variables that are not in P, and matches p's arity;
- $(p(\vec{x}) \leftarrow Body_1), \cdots, (p(\vec{x}) \leftarrow Body_k)$ are the normal forms on $\vec{x}$ of all the rules in P whose heads mention the predicate p;
- for each $1 \leq i \leq k$, $\vec{y_i}$ is the tuple of local variables in the rule $(p(\vec{x}) \leftarrow Body_i)$, i.e., they occur in $Body_i$ but not in $\vec{x}$;
- $\widehat{Body_i}$ in the formula stands for the conjunction of all elements in $Body_i$ with "not" replaced by logical negation "$\neg$".

In particular, if a predicate $p(\vec{x})$ does not occur in the head of any rule in P, then its completion under P is equivalent to $\forall \vec{x} \neg p(\vec{x})$.

Now given a program P, we define the completion of P to be the set of the completions of all predicates in P.

Positive dependency graphs, loops, and loop formulas

As usual, a *binding* is an expression of the form x/t, where x is a variable, and t a term. A *substitution* is a set of bindings containing at most one binding for each variable. In the following, if φ is a first-order formula, and θ a substitution, we denote by $\varphi\theta$ the result of replacing every free variable in φ according to θ. In particular, a variable assignment can be considered a substitution, and if $\exists\vec{x}\varphi$ is a sentence, then $(\exists\vec{x}\varphi)|D$ is equivalent to the disjunction of the sentences in the set

$$\{(\varphi\sigma)|D \mid \sigma \text{ is a variable assignment of } \vec{x} \text{ on } D\},$$

and $(\forall\vec{x}\varphi)|D$ is equivalent to the conjunction of the sentences in the set

$$\{(\varphi\sigma)|D \mid \sigma \text{ is a variable assignment of } \vec{x} \text{ on } D\}.$$

Let P be a program. The *(first-order) positive dependency graph* of P, written G_P, is the infinite graph (V, E), where V is the set of atoms that do not mention any constants other than those in P, and for any $A, A' \in V$, $(A, A') \in E$ if there is a rule (1) in P and a substitution θ such that $h\theta = A$ and $b\theta = A'$ for some $b \in Body$.

A finite non-empty subset L of V is called a *(first-order) loop* of P if there is a non-zero length cycle that goes through only and all the nodes in L. This is the same as saying that for any A and A' in L, there is a non-zero length path from A to A' in the subgraph of G_P induced by L.

It is easy to see that if the given logic program P does not contain any variables, then the above definitions of positive dependency graph and loops are the same as those in (Lin & Zhao 2004).

Example 1 The following are some examples that illustrate our notion of loops.

- Let $P_1 = \{p(x) \leftarrow p(x)\}$. The vertices in P_1's positive dependency graph are atoms of the form $p(\xi)$, where ξ is a variable, and there is an arc from $p(\xi)$ to $p(\xi)$. Thus the loops of the program are singletons of the form $\{p(\xi)\}$, where ξ is a variable.

- Let P_2 be the following program:

$$\{(p(x) \leftarrow q(x)), (q(y) \leftarrow p(y)),$$
$$(p(x) \leftarrow r(x)), (q(y) \leftarrow not\, s(y))\}.$$

  The vertices in P_2's positive dependency graph are atoms of the forms $p(\xi)$, $q(\xi)$, $s(\xi)$ or $r(\xi)$, where ξ is a variable, and there is an arc from $p(\xi)$ to $q(\xi)$ and vice versa, and there is an arc from $p(\xi)$ to $r(\xi)$. Thus the loops of the program are sets of atoms of the form $\{p(\xi), q(\xi)\}$, where ξ is a variable.
- Let $P_3 = \{p(x) \leftarrow p(y)\}$. The vertices in P_3's positive dependency graph are atoms of the form $p(\xi)$, where ξ is a variable, and for any variables ξ and ζ, there is an arc from $p(\xi)$ to $p(\zeta)$. Thus any non-empty finite set of vertices is a loop. In terms of loops, this program has the same structure as Niemelä's program for Hamiltonian Circuit problem (Niemelä 1999).
- Let $P_4 = \{p(x, y) \leftarrow p(a, b)\}$, where a and b are constants. The vertices in P_4's positive dependency graph are atoms of the form $p(\xi, \zeta)$, where ξ and ζ are either a, b, or variables. There is an arc from $p(\xi_1, \zeta_1)$ to $p(\xi_2, \zeta_2)$ iff $\xi_2 = a$ and $\zeta_2 = b$. So the only loop of P_4 is $\{p(a, b)\}$.

To define loop formulas, we follow Lee (2005), define the *external support formula* of an atom in a loop first. Given a loop L of a program P, an atom $p(\vec{t})$, the external support formula of of $p(\vec{t})$ w.r.t. L, written $ES(p(\vec{t}), L, P)$, is the following formula:

$$\bigvee_{1 \le i \le k} \exists \vec{y_i} \left[\widehat{Body_i}\theta \wedge \bigwedge_{\substack{q(\vec{u}) \in Body_i\theta \\ q(\vec{v}) \in L}} \vec{u} \neq \vec{v} \right],$$

where

- $(p(\vec{x}) \leftarrow Body_1), \cdots, (p(\vec{x}) \leftarrow Body_k)$ are the normal forms on $\vec{x}$ of the rules in P whose heads mention the predicate p;
- $\vec{x}$ is a tuple of distinct variables that are not in P, and if $\vec{t} = (t_1, ..., t_n)$ and $\vec{x} = (x_1, ..., x_n)$, then

$$\theta = \{x_1/t_1, ..., x_n/t_n\}$$

  (so that $\vec{x}\theta = \vec{t}$);
- for each $1 \le i \le k$, $\vec{y_i}$ is the tuple of local variables in $p(\vec{x}) \leftarrow Body_i$. We assume that $\vec{y_i} \cap Var(L) = \emptyset$, by renaming local variables in $Body_i$ if necessary, where $Var(L)$ is the set of variables in L. In other words, if a rule has a local variable that is also in the loop, then we have to rename the local variable in the rule.

Notice that the only free variables in the formula $ES(p(\vec{t}), L, P)$ are those in $\vec{t}$. The choice of the variables in $\vec{x}$ is not material.

The (first-order) loop formula of L in P, written $LF(L, P)$, is then the following formula:

$$\forall \vec{x} \left[\bigvee_{A \in L} A \supset \bigvee_{A \in L} ES(A, L, P) \right], \qquad (2)$$

where $\vec{x}$ is the tuple of variables in L.

Example 2 We continue with the logic programs in Example 1.

- Consider $P_1 = \{p(x) \leftarrow p(x)\}$, and the loop $\{p(z)\}$. The normal form of the rule on y is $p(y) \leftarrow y = x, p(x)$. So the external support formula of $p(z)$ w.r.t. this loop is

$$\exists x.z = x \wedge p(x) \wedge x \neq z$$

  which is equivalent to $false$. Thus the loop formula for this loop is equivalent to $\forall x \neg p(x)$. The loop formulas of the other loops like $\{p(x)\}$ is equivalent to this formula as well.
- Consider

$$P_2 = \{(p(x) \leftarrow q(x)), (q(y) \leftarrow p(y)),$$
$$(p(x) \leftarrow r(x)), (q(y) \leftarrow not\, s(y))\}.$$

  and the loop $\{p(x), q(x)\}$. To compute the external support formula of $p(x)$, we first normalize all the rules about p on a new variable, say z:

$$p(z) \leftarrow z = x, q(x),$$
$$p(z) \leftarrow z = x, r(x).$$

  Since x is a local variable in the above rules, and it occurs in the loop, we replace it by another variable, say x_1:

$$p(z) \leftarrow z = x_1, q(x_1),$$
$$p(z) \leftarrow z = x_1, r(x_1).$$

  Thus the external support formula of $p(x)$ is

$$\exists x_1(x = x_1 \wedge q(x_1) \wedge x \neq x_1) \vee \exists x_1(x = x_1 \wedge r(x)),$$

  which is equivalent to $r(x)$. Similarly, the external support formula of $q(x)$ is equivalent to $\neg s(x)$. Thus the loop formula of this loop is equivalent to

$$\forall x.(p(x) \vee q(x)) \supset (r(x) \vee \neg s(x)).$$

  It can be seen that the loop formulas of all loops in P_2 are equivalent to this sentence.
- Consider $P_3 = \{p(x) \leftarrow p(y)\}$, and the loop $\{p(x_1), \cdots, p(x_k)\}$. The external support formula for each $p(x_i)$ is equivalent to

$$\exists y.p(y) \wedge \bigwedge_{1 \le i \le k} x_i \neq y.$$

  Thus the loop formula of this loop is equivalent to

$$\forall x_1, ..., x_k. \bigvee_{1 \le i \le k} p(x_i) \supset \exists y.p(y) \wedge \bigwedge_{1 \le i \le k} x_i \neq y.$$

Main theorem

In the following, we call a predicate in a program *extensional* if it does not occur in the head of any rule in the program, and the other predicates in the program are called *intensional*.

Theorem 1 *Let P be a finite set of rules, and F a finite set of ground facts about the extensional relations. Let D be the set of constants in $P \cup F$. A set M of ground facts is an answer set of $(P|D) \cup F$ iff M is a propositional model of $(COMP(P \cup F) \cup LF(P))|D$, where $COMP(P \cup F)$ is the completion of $P \cup F$ and $LF(P)$ the set of the loop formulas of P.*

Proof: See the next section. ■

Notice that *COMP*$(P \cup F)$ is actually the union of two sets: the set of the completions of the intensional predicates of P under P and the set of the completions of the extensional predicates of P under F. The first set can be computed before we are given the actual problem instance F.

We have defined a loop to be a set of vertices such that there is a *non-zero* length cycle that goes through all and only the vertices in the set. We can generalize this notion of loops by allowing zero length cycle, like what is done in (Lee & Lin 2006) and (Lee 2005). This will make every singleton set of nodes to be a loop, and the loop formulas of these singleton sets will correspond to Clark's completion. Under this more general notion of loops, Theorem 1 can be restated as follows:

> Let P be a finite set of rules, and F a finite set of ground facts about the extensional relations. Let D be the set of constants in $P \cup F$. A set M of ground facts is an answer set of $(P|D) \cup F$ iff M is a propositional model of $(\widehat{P} \cup F \cup LF'(P))|D$, where $LF'(P)$ is the set of the loop formulas for the (generalized) loops in P, and $\widehat{P}$ is the set of sentences obtained from P by replacing each rule $(F \leftarrow B)$ in it by the sentence $\forall \vec{x}.\widehat{B} \supset F$, where $\vec{x}$ is the tuple of variables in the rule.

Example 3 The following are some examples that illustrate the theorem.

- Consider P_2 from Examples 1 and 2.
  Suppose $F = \{r(a), s(b)\}$. Then $D = \{a, b\}$. By normalizing all rules on z, it is easy to see that the completion of $P_2 \cup F$ is equivalent to
  $$\forall z.p(z) \equiv \exists x(z = x \wedge q(x)) \vee \exists x(z = x \wedge r(x)),$$
  $$\forall z.q(z) \equiv \exists y(z = y \wedge p(y)) \vee \exists y(z = y \wedge \neg s(y)),$$
  $$\forall z.r(z) \equiv z = a, \qquad \forall z.s(z) \equiv z = b,$$
  which is equivalent to
  $$\forall z.p(z) \equiv q(z) \vee r(z), \qquad \forall z.q(z) \equiv p(z) \vee \neg s(z)),$$
  $$\forall z.r(z) \equiv z = a, \qquad \forall z.s(z) \equiv z = b.$$
  Thus by Proposition 2, *COMP*$(P_2 \cup F)|D$ is equivalent to the set of the following set of sentences:
  $$p(a) \wedge q(a) \wedge p(b) \equiv q(b),$$
  $$r(a) \wedge \neg r(b) \wedge \neg s(a) \wedge s(b).$$
  As we have shown in Example 2, the loop formulas of all the loops are equivalent to
  $$\forall x.(p(x) \vee q(x)) \supset (r(x) \vee \neg s(x)).$$
  Thus $LF(P_2)|D$ is equivalent to $\neg p(b) \wedge \neg q(b)$. Thus the only propositional model of *COMP*$(P_2 \cup F) \cup LF(P_2)|D$ is $\{p(a), q(a), r(a), s(b)\}$.
  Now the instantiation of the program $P_2 \cup F$ on $\{a, b\}$ is:
  $$p(a) \leftarrow q(a).\ p(b) \leftarrow q(b).\ p(a) \leftarrow r(a).$$
  $$p(b) \leftarrow r(b).\ q(a) \leftarrow p(a).\ q(b) \leftarrow p(b).$$
  $$q(a) \leftarrow not\, s(a).\ q(b) \leftarrow not\, s(b).$$
  There is a unique answer set of this program, which is the same as the propositional model given above.
- Consider the program P_5 that contains the following rules:
  $$p(x) \leftarrow p(y)$$
  $$p(x) \leftarrow q(x, y), x \neq y.$$
  This program adds one more rule about p to the program P_3 in Examples 1 and 2. The loops of the program are the same as those of P_3, and of the form $\{p(\xi_1), ..., p(\xi_k)\}$. The loop formulas are equivalent to the following sentence:
  $$\forall x_1, ..., x_k. \bigvee_{1 \le i \le k} p(x_i) \supset$$
  $$[\exists y.p(y) \wedge \bigwedge_{1 \le i \le k} x_i \neq y] \vee [\bigvee_{1 \le i \le k} \exists y.q(x_i, y) \wedge x_i \neq y].$$
  Now let $F = \{q(a, b)\}$. Then $D = \{a, b\}$. If $k = 1$, then the instantiation of the above sentence is equivalent to
  $$(p(a) \supset p(b) \vee q(a, b)) \wedge (p(b) \supset p(a) \vee q(b, a)). \quad (3)$$
  For $k \geq 2$, the instantiations of the above sentences are all equivalent to the conjunction of (3) and the following sentence:
  $$(p(a) \vee p(b)) \supset (q(a, b) \vee q(b, a)).$$
  Now for the completion of $P_5 \cup F$, by normalizing all rules on u, v, we see that it is equivalent to
  $$\forall u.p(u) \equiv \exists x, y.(u = x \wedge p(y)) \vee$$
  $$\exists x, y.(u = x \wedge q(x, y) \wedge x \neq y),$$
  $$\forall u, v.q(u, v) \equiv (u = a \wedge v = b).$$
  Thus *COMP*$(P_5 \cup F)|D$ is equivalent to
  $$q(a, b) \wedge \neg q(a, a) \wedge \neg q(b, a) \wedge \neg q(b, b) \wedge p(a) \wedge p(b).$$
  Notice that this formula implies $LF(P_5)|D$, so the latter is not needed for this example. Now the instantiation of the program $P_5 \cup F$ on $\{a, b\}$ is:
  $$p(a) \leftarrow p(a).\ p(a) \leftarrow p(b).\ p(b) \leftarrow p(a).\ p(b) \leftarrow p(b).$$
  $$p(a) \leftarrow q(a, b).\ p(b) \leftarrow q(b, a).\ q(a, b).$$
  The unique answer set of this program is the same as the model of *COMP*$(P_5 \cup F)|D$.

Proof of the main theorem

The proof of Theorem 1 is tedious, but the basic idea is simple. Instead of $P|D$, we use an equivalent logic program $P_{eq}|D$ as defined below. The main task is then to show that there is a one-to-one correspondence between the loops of $P_{eq}|D$ and the instantiations of the first-order loops of P, and that this correspondence works between the loop formulas of $P_{eq}|D$ and the instantiations of the first-order loop formulas of P.

Notice that the instantiation of a program on a finite domain, $P|D$, eliminates all equality atoms. While doing this, it also delete rules whose bodies have an equality literal that is false. Thus the ground loops of $P|D$ and the instantiations of the first-order loops of P on D may not be in one-to-one correspondence. So to faithfully preserve the loops of P, we introduce P_{eq} as defined below.

Let P be a program. Without loss of generality, suppose that P does not mention eq as a binary predicate. Let P_{eq} be the program obtained from P by replacing $t = t'$ in it by $eq(t, t')$. Notice that $t \neq t'$ will be replaced by $not\, eq(t, t')$ as the former is a shorthand for $not\, t = t'$.

Proposition 3 *For any finite domain D such that $Const(P) \subseteq D$, a set M of atoms occurring in $P|D$ is an answer set of $P|D$ iff $M \cup EQ_D$ is an answer set of $P_{eq}|D \cup EQ_D$, where $EQ_D = \{eq(d, d) \mid d \in D\}$.*

Proof: Since eq does not occur in the head of any rule in $P_{eq}|D$, $(P_{eq}|D) \cup EQ_D$ is equivalent to $(P|D) \cup EQ_D$. ■

So $P|D$ and $P_{eq} \cup EQ_D$ are essentially the same. But the latter is a more faithful way to ground P on D in the sense that it will not lose any loops of P. For instance, $\{p(x), p(y)\}$ is a loop of program

$$P = \{p(x) \leftarrow p(y), x \neq x\},$$

but for any D, $P|D$ is the empty program, and thus has no loop. But $P_{eq}|D$ has the same loops as those of P. Our next proposition shows that this is true in general.

Proposition 4 *Let P be a finite set of rules and D a finite set of constants containing $Const(P)$. If GL is a loop of $P_{eq}|D$ then for some loop L of P, and some variable assignment θ on D, $GL = L\theta$. Conversely, if L is a loop of P, and θ is a variable assignment on D, then $L\theta$ is a loop of $P_{eq}|D$.*

Proof: Let G_0 be the positive dependency graph of the propositional program $P_{eq}|D$, and G_1 the positive dependency graph of P.

Let A and B be two atoms in $P_{eq}|D$ that do not mention eq. Let $d_1, ..., d_k$ be the constants in A and B that are not in $Const(P)$, and A' and B' the results of replacing each d_i by x_i in A and B, respectively. Then we have that

(A, B) is an arc in G_0
iff
There is a rule $A \leftarrow B, Body$ in $P_{eq}|D$
iff
There is a rule $\alpha \leftarrow \beta, body$ in P_{eq} and a variable assignment σ such that $A = \alpha\sigma$ and $B = \beta\sigma$
iff
There is a rule $\alpha \leftarrow \beta, body'$ in P (since α and β do not mention eq) and a variable assignment σ such that $A = \alpha\sigma$ and $B = \beta\sigma$
iff
There is a rule $\alpha \leftarrow \beta, body'$ in P and a variable assignment σ' such that $A' = \alpha\sigma'$ and $B' = \beta\sigma'$. (σ' can be obtained from σ by replacing every $x/d_i (1 \leq i \leq k)$ in σ with x/x_i.)
iff
(A', B') is an arc in G_1.

Now if GL is a loop of $P_{eq}|D$, then there is a cycle $(A_1, ..., A_n, A_{n+1})$, where $A_{n+1} = A_1$, in G_0 such that $GL = \{A_1, ..., A_n\}$. Thus for each $1 \leq i \leq n$, A_i does not mention eq and (A_i, A_{i+1}) is an arc in G_0. Thus for each $1 \leq i \leq n$, (A'_i, A'_{i+1}) is an arc in G_1, where A'_i's are defined as above. Thus $\{A'_1, ..., A'_n\}$ is a loop of P.

Conversely, suppose L is a loop of P, and θ a variable assignment of $Var(L)$ on D. Then there is a cycle $(\alpha_1, ..., \alpha_n, \alpha_{n+1} = \alpha_1)$ in G_1 such that $L = \{\alpha_1, ..., \alpha_n\}$. Thus for each $1 \leq i \leq n$, (α_i, α_{i+1}) is an arc in G_1. Thus for each $1 \leq i \leq n$, $(\alpha_i\theta, \alpha_{i+1}\theta)$ is an arc in G_0 (the fact that each $\alpha_i\theta$ is a node in G_0 follows from the assumption that $Const(P) \subseteq D$). Thus $L\theta = \{\alpha_1\theta, ..., \alpha_n\theta\}$ is a loop of $P_{eq}|D$. ■

In the following, if P is a ground logic program, L a set of atoms in P, and A an atom in L, then we define the *ground support formula* of A w.r.t. L, written $GES(A, L, P)$, to be the following formula

$$\bigvee_{1 \leq i \leq k, B_i \cap L = \emptyset} \widehat{B_i}$$

where $(A \leftarrow B_1), ..., (A \leftarrow B_k)$ are the rules in P whose heads are A. Now if L is a loop of the ground program P, then the following formula, written $GLF(L, P)$,

$$\bigvee_{A \in L} A \supset \bigvee_{A \in L} GES(A, L, P)$$

is equivalent to the loop formula of L as defined in (Lin & Zhao 2004). In the following, we call $GLF(L, P)$ the *ground loop formula* of L in P.

Lemma 1 *Let P be a program, L be a loop of P_{eq} and D be a finite set of constants containing $Const(P)$. Then, for any an atom A in L and any substitution θ of $Var(L)$ on D, we have*

$$GES(A\theta, L\theta, P_{eq}|D) \equiv ES(A, L, P_{eq})\theta|D.$$

Proof: Let $A = p(\vec{t})$. Without loss of generality, we assume that there is only one rule in P whose head mentions p: $p(\vec{w}) \leftarrow B$, and does not contain variables in L (by renaming variables in the rule if necessary). Suppose that $\vec{x}$ is a tuple of variables not in P and L, and we normalize this rule on $\vec{x}$: $p(\vec{x}) \leftarrow B \cup \{\vec{x} = \vec{w}\}$. $GES(A\theta, L\theta, P_{eq}|D)$ is

$$\bigvee_{\vec{w}\sigma = \vec{t}\theta, B_{eq}\sigma \cap L\theta = \emptyset} \widehat{B_{eq}}\sigma, \tag{4}$$

where σ ranges over all variable assignments on D, and B_{eq} is the result of replacing equality in B by eq.

$ES(A, L, P_{eq})$ is

$$(\exists \vec{y})(\widehat{B_{eq}} \wedge \vec{t} = \vec{w} \wedge \bigwedge_{\substack{q(\vec{u}) \in B_{eq} \\ q(\vec{v}) \in L}} \vec{u} \neq \vec{v}),$$

where $\vec{y}$ is the tuple of variables in B and $\vec{w}$. Since none of the variables in B and $\vec{w}$ occur in L, and θ is a substitution of variables in L, so $ES(A, L, P_{eq})\theta$ is

$$(\exists \vec{y})(\widehat{B_{eq}} \wedge \vec{t}\theta = \vec{w} \wedge \bigwedge_{\substack{q(\vec{u}) \in B_{eq} \\ q(\vec{v}) \in L}} \vec{u} \neq \vec{v}\theta).$$

Thus $ES(A, L, P_{eq})\theta|D$ is equivalent to

$$\bigvee_{\sigma} \widehat{B_{eq}}\sigma, \tag{5}$$

where σ ranges over variable assignments on D such that

- $\vec{t}\theta$ and $\vec{w}\sigma$ are the same
- for every atom $q(\vec{v}) \in L$ and every atom $q(\vec{u}) \in B_{eq}$, $\vec{u}\sigma$ and $\vec{v}\theta$ are not the same, that is $L\theta \cap B_{eq}\sigma = \emptyset$.

Thus (4) and (5) are the same, and this proves the proposition. ■

Proposition 5 *Let P be a program and D be a finite set of constants containing $Const(P)$. Then $LF(P_{eq})|D$ and $GLF(P_{eq}|D)$ are logically equivalent, where for any program Q, $GLF(Q)$ is the set of ground loop formulas of Q.*

Proof: $LF(P_{eq})|D$ is

$$\{LF(L, P_{eq})|D \mid L \text{ is a loop of } P_{eq}\}.$$

For each loop L of P_{eq}, $LF(L, P_{eq})$ is

$$\forall \vec{x} \left[\bigvee_{A \in L} A \supset \bigvee_{A \in L} ES(A, L, P_{eq}) \right].$$

Thus $LF(L, P_{eq})|D$ is equivalent to the set of following sentences:

$$\left[\bigvee_{A \in L} A \supset \bigvee_{A \in L} ES(A, L, P_{eq}) \right] \sigma|D,$$

where σ is a variable assignment of $\vec{x}$ on D. The above sentence is equivalent to

$$\left[\bigvee_{A \in L} A\sigma \supset \bigvee_{A \in L} ES(A, L, P_{eq})\sigma \right] |D,$$

which is equivalent to the following sentence as for each $A \in L$, $A\sigma$ is a ground atom,

$$\bigvee_{A \in L} A\sigma \supset \bigvee_{A \in L} (ES(A, L, P_{eq})\sigma|D),$$

which, by Lemma 1, is equivalent to

$$\bigvee_{A \in L} A\sigma \supset \bigvee_{A \in L} GES(A\sigma, L\sigma, P_{eq}|D),$$

which is equivalent to

$$[\bigvee_{B \in L\sigma} B \supset \bigvee_{B \in L\sigma} GES(B, L\sigma, P_{eq}|D),$$

which is $GLP(L\sigma, P_{eq}|D)$. Thus by Proposition 4, $LF(P_{eq})|D$ is equivalent to $GLF(P_{eq}|D)$. ■

In the following, if P is a ground logic program, then we denote by $GCOMP(a, P)$ the ground completion of an atom a in P.

Proposition 6 *Let P be a program, and D a finite set of constants containing $Const(P)$. For each predicate p in P, $COMP(p, P_{eq})|D$ is equivalent to*

$$\{GCOMP(p(\vec{x})\sigma, P_{eq}|D) \mid \sigma \text{ is a variable assignment of } \vec{x} \text{ on } D\}.$$

Proof: Again without loss of generality, we assume that $p(\vec{w}) \leftarrow B$ is the only rule in P with p in its head, and $\vec{x}$ is a tuple of variables not in P. Then $COMP(p, P_{eq})$ is equivalent to

$$\forall \vec{x} p(\vec{x}) \equiv \exists \vec{y} \widehat{B_{eq}} \wedge \vec{x} = \vec{w},$$

where $\vec{y}$ is the tuple of variables in B and $\vec{w}$. So $COMP(p, P_{eq})|D$ is equivalent to the conjunction of the following sentences:

$$p(\vec{x})\sigma \equiv (\exists \vec{y} \widehat{B_{eq}} \wedge \vec{x}\sigma = \vec{w})|D,$$

where σ is a variable assignment on D, which is equivalent to

$$p(\vec{x})\sigma \equiv \bigvee_{\vec{x}\sigma = \vec{w}\tau} \widehat{B_{eq}}\tau,$$

where τ ranges over all variable assignments of $\vec{x}$ on D, which is exactly $GCOMP(p(\vec{x})\sigma, P_{eq}|D)$. ■

Finally, we can prove the main theorem:

Theorem 1 *Let P be a finite set of rules, and F a finite set of ground facts about the extensional relations. Let D be the set of constants in $P \cup F$. A set M of ground facts is an answer set of $(P|D) \cup F$ iff M is a propositional model of $(COMP(P \cup F) \cup LF(P))|D$, where $COMP(P \cup F)$ is the completion of $P \cup F$ and $LF(P)$ the set of the loop formulas of P.*

Proof: In the following, we denote by $GCOMP_1(Q)$ the set of ground completions of the atoms not mentioning "*eq*" in Q:

$$GCOMP_1(Q) = \{GCOMP(A, Q)|A \text{ does not mention } eq\}.$$

Similarly, we let

$$COMP_1(Q) = \{COMP(p, Q) \mid p \text{ is not } eq\}.$$

We also let

$$IU(D) = \{eq(d, d)|d \in D\} \cup \{\neg eq(d, d')|d \text{ and } d' \text{ are two distinct constants of } D\}.$$

M is an answer set of $(P|D) \cup F$
iff
M is an answer set of $(P \cup F)|D$
iff
$M \cup EQ_D$ is an answer set of $(P \cup F)_{eq}|D \cup EQ_D$ (Proposition 3)
iff
$M \cup EQ_D$ is a propositional model of

$$GCOMP((P \cup F)_{eq}|D \cup EQ_D) \cup GLF((P \cup F)_{eq}|D \cup EQ_D)$$

(Theorem 1 of (Lin & Zhao 2004))
iff
$M \cup EQ_D$ is a propositional model of

$$GCOMP_1((P \cup F)_{eq}|D) \cup IU(D) \cup GLF(P_{eq}|D \cup F \cup EQ_D)$$

($COMP(eq, P_{eq} \cup F \cup EQ_D)$ is equivalent to $IU(D)$)
iff
$M \cup EQ_D$ is a propositional model of

$$GCOMP_1((P \cup F)_{eq}|D) \cup IU(D) \cup GLF(P_{eq}|D)$$

iff
$M \cup EQ_D$ is a propositional model of

$$COMP_1((P \cup F)_{eq})|D \cup IU(D) \cup LF(P_{eq})|D$$

(by Proposition 6 and Proposition 5)
iff
$M \cup EQ_D$ is a propositional model of

$$[COMP_1((P \cup F)_{eq}) \cup \{\forall x, y(eq(x,y) \equiv x = y)\} \cup LF(P_{eq})]|D$$

iff
$M \cup EQ_D$ is a propositional model of

$$[COMP(P \cup F) \cup \{\forall x, y(eq(x,y) \equiv x = y)\} \cup LF(P)]|D$$

(by Proposition 2 and that in first-order logic,

$$\{\forall x, y(eq(x,y) \equiv x = y)\} \models COMP(p, Q_{eq}) \equiv COMP(p, Q),$$

for any program Q and predicate p in Q, and

$$\{\forall x, y(eq(x,y) \equiv x = y)\} \models LF(L, Q_{eq}) \equiv LF(L, Q),$$

for any loop L)
iff
$M \cup EQ_D$ is a propositional model of
$[COMP(P \cup F) \cup LF(P)]|D \cup \{\forall x, y(eq(x,y) \equiv x = y)\}|D$
iff
$M \cup EQ_D$ is a propositional model of

$$[COMP(P \cup F) \cup LF(P)]|D \cup IU(D)$$

iff
M is a propositional model of

$$[COMP(P \cup F) \cup LF(P)]|D$$

as M and $[COMP(P \cup F) \cup LF(P)]|D$ do not mention eq. ■

Some properties of loops and loop formulas

It is clear from the definition that if L is a loop of P, and θ is a substitution such that $Const(\theta) \subseteq Const(P)$, then $L\theta$ is also a loop of P, where $Const(\theta)$ is the set of constants occurring in θ. Thus if P has a loop that contains a variable, then P has an infinite number of loops. But as we have seen from the examples many of these loops are "equivalent" in the sense that their loop formulas are logically equivalent in first-order logic. This motivates our following definition.

Let L_1 and L_2 be two sets of atoms. We say L_1 *subsumes* L_2 if there is substitution θ such that $L_1\theta = L_2$. We say that L_1 and L_2 are *equivalent* if they subsume each other. For instance, $\{p(x)\}$ and $\{p(y)\}$ are equivalent. The set $\{p(x_1), p(x_2)\}$ subsumes $\{p(x)\}$, but not the other way around. If L_1 subsumes L_2, then the loop formula of L_1 also entails the loop formula of L_2.

Proposition 7 *Let L_1 be a loop of P, and L_2 a set of atoms such that $Const(L_2) \subseteq Const(P)$. If L_1 subsumes L_2, then L_2 is also a loop of P. Furthermore,*

$$\models LF(L_1, P) \supset LF(L_2, P).$$

Proof: Let θ be a substitution of $Var(L_1)$ such that $L_1\theta = L_2$. Since L_1 is a loop, there is a cycle $A_1, ..., A_n, A_{n+1} = A_1$ in G_P, the dependency graph of P, such that $L_1 = \{A_1, ..., A_n\}$. By the definition of G_P, $A_1\theta, ..., A_n\theta, A_{n+1}\theta$ is also a cycle of G_P. Thus $L_2 = \{A_1\theta, ..., A_n\theta\}$ is also a loop of P.

Now $LF(L_1, P)$ is $\forall\vec{\xi}\Phi(\vec{\xi})$, where $\vec{\xi}$ is the tuple of variables in L_1, and $\Phi(\vec{\xi})$ is

$$\left[\bigvee_{A \in L_1} A \supset \bigvee_{A \in L_1} ES(A, L_1, P)\right]. \qquad (6)$$

We show that $LF(L_2, P)$ is equivalent to $\forall\vec{\zeta}\Phi(\vec{\xi})\theta$, where $\vec{\zeta}$ is the tuple of variables in $L_2 = L_1\theta$, i.e. $\vec{\xi}\theta$.

Notice that $LF(L_2, P)$ is

$$\forall\vec{\zeta}\left[\bigvee_{A \in L_1} A\theta \supset \bigvee_{A \in L_1} ES(A\theta, L_1\theta, P)\right]. \qquad (7)$$

Thus it is enough to show that, for any $p(\vec{t}) \in L_1$, $ES(p(\vec{t}), L_1, P)\theta \equiv ES(p(\vec{t})\theta, L_1\theta, P)$. That is

$$\bigvee_{1 \le i \le k} \exists\vec{y_i}\left[\widehat{B_i}\delta_1 \wedge \bigwedge_{\substack{q(\vec{u}) \in B_i\delta_1 \\ q(\vec{v}) \in L_1}} \vec{u} \neq \vec{v}\right]\theta$$

$$\equiv \bigvee_{1 \le i \le k} \exists\vec{y_i}\left[\widehat{B_i}\delta_2 \wedge \bigwedge_{\substack{q(\vec{u}) \in B_i\delta_2 \\ q(\vec{v}) \in L_1\theta}} \vec{u} \neq \vec{v}\right]$$

where

- $\vec{x}$ is a tuple of distinct variables that are not in P, and

  $$(p(\vec{x}) \leftarrow B_1), \cdots, (p(\vec{x}) \leftarrow B_k)$$

  are the normal forms on $\vec{x}$ of all the rules in P whose heads mention the predicate p.
- Let $\vec{t} = (t_1, ..., t_n)$ and $\vec{x} = (x_1, ..., x_n)$. Then $\delta_1 = \{x_1/t_1, ..., x_n/t_n\}$ (so that $\vec{x}\delta_1 = \vec{t}$), and similarly for δ_2. So $p(\vec{x})\delta_1 = p(\vec{t})$ and $p(\vec{x})\delta_2 = p(\vec{t})\theta$.
- For each $1 \le i \le k$, $\vec{y_i}$ is the tuple of local variables in $p(\vec{x}) \leftarrow B_i$. We assume that $\vec{y_i} \cap Var(L_1) = \emptyset$, by renaming local variables in B_i if necessary. For the same reason, we also assume that $\vec{y_i} \cap Var(L_2) = \emptyset$.

The equivalence follows because

$$\bigvee_{1 \le i \le k} \exists\vec{y_i}\left[\widehat{B_i}\delta_1 \wedge \bigwedge_{\substack{q(\vec{u}) \in B_i\delta_1 \\ q(\vec{v}) \in L_1}} \vec{u} \neq \vec{v}\right]\theta$$

$$\equiv \bigvee_{1 \le i \le k} \exists\vec{y_i}\left[\widehat{B_i}\delta_1\theta \wedge \bigwedge_{\substack{q(\vec{u}) \in B_i\delta_1 \\ q(\vec{v}) \in L_1}} \vec{u}\theta \neq \vec{v}\theta\right],$$

$\widehat{B_i}\delta_1\theta$ is the same as $\widehat{B_i}\delta_2$ and $\bigwedge_{\substack{q(\vec{u}) \in B_i\delta_1 \\ q(\vec{v}) \in L_1}} \vec{u}\theta \neq \vec{v}\theta$ is the same as $\bigwedge_{\substack{q(\vec{u}) \in B_i\delta_2 \\ q(\vec{v}) \in L_1\theta}} \vec{u} \neq \vec{v}$. ■

Given a program P, a set Δ of loops of P is said to be *complete* if for any loop L of P, there is a loop $L' \in \Delta$ such that L' subsumes L. Thus by Proposition 7, if Δ is a complete set of loops of P, then the set of loop formulas of the loops in Δ is logically equivalent to the set of the loop formulas of all the loops of P. Of special interests are programs that have finite complete sets of loops. For the programs in Example 1, $\{\{p(x)\}\}$ is a complete set of loops for P_1, $\{\{p(x), q(x)\}\}$ is a complete set of loops for P_2. But P_3 has no finite complete set of loops. One of its complete sets of loops is $\{\{p(x_1)\}, \cdots, \{p(x_1), \cdots, p(x_k)\}, \cdots\}$.

An interesting question then is how to decide whether a program has a finite complete set of loops. The following theorem answers this question.

Theorem 2 *Let P be a finite logic program, and*

$$D = \{c_1, c_2\} \cup Const(P),$$

where c_1 and c_2 are two new constants not in P. The following five assertions are equivalent:

1. *P has a finite complete set of loops.*
2. *There is a natural number N such that, for any loop L of P, the size of $Var(L)$ is smaller than N.*
3. *For any loop L of P, and any atoms A_1 and A_2 in L, $Var(A_1) = Var(A_2)$.*
4. *For any loop L of $P_{eq}|D$, there are no two atoms A_1 and A_2 in L such that either A_1 mentions c_1 but A_2 does not or A_1 mentions c_2 but A_2 does not.*
5. *For any maximal loop (w.r.t. to subset ordering) L of $P_{eq}|D$, there are no two atoms A_1 and A_2 in L such that either A_1 mentions c_1 but A_2 does not or A_1 mentions c_2 but A_2 does not.*

Proof: "1 $\Leftarrow$ 2": Let Δ be the set of loops of P that mention only variables in $V = \{x_1, ..., x_N\}$. Clearly, Δ is finite. We show that it is complete. Let L be any loop of P. Let L' be obtained from L by replacing in it variables in $Var(L) \setminus V$ by distinct variables in $V \setminus Var(L)$ - this is possible as L mentions at most N variables. Clearly $L' \in \Delta$, and subsumes L.

"2 $\Leftarrow$ 3": P is a finite program, so there are at most finite predicates occurring in it. Let N be the maximum arity of all the predicates in P. For any loop L in P, the number of variables in L is at most N.

"3 $\Leftarrow$ 4": Suppose otherwise, and L is a loop of P such that there are two atoms A_1 and A_2 in L such that $Var(A_1) \neq Var(A_2)$. Without loss of generality, let x be a variable in A_1 and not in A_2. Let θ be the substitution $\{x/c_1\} \cup \{x'/c_2 \mid x' \in Var(L) \setminus \{x\}\}$. Then $L\theta$ is a loop of $P_{eq}|D$. Moreover, we can see $A_1\theta$ mentions c_1 and $A_2\theta$ does not, and $A_1\theta$ and $A_2\theta$ are two atoms in $L\theta$, which is a contradiction.

"4 $\Leftarrow$ 1": Let $\Delta = \{L_1, ..., L_n\}$ be a finite complete set of loops of P. For every $L_i \in \Delta$, $Var(L_i)$ is finite since L_i is finite. Now let $N = max\{|Var(L_1)|, ..., |Var(L_n)|\}$. For any loop L of P, there is a loop $L^* \in \Delta$ such that L^* subsumes L, and consequently,

$$|Var(L)| \leq |Var(L^*)| \leq N.$$

Now suppose condition 4 does not hold, and let $L = \{A_1, A_2, \ldots, A_n\}$ be a ground loop such that, without loss of generality, c_1 occurs in A_1 and not in A_2. Let x and y be two new variables not in L, and for each i, A'_i be obtained from A_i by replacing c_1 with x and c_2 with y. Let $L^{[1]} = \{A'_1, A'_2, \ldots, A'_n\}$. Clearly, A'_1 mentions x but A'_2 does not. But by Proposition 4 (more precisely as shown in the proof of this proposition), $L^{[1]}$ is loop of P.

From loop $L^{[1]}$, we can construct another loop $L^{[2]}$ of P as following. Let $\delta = \{x/x^*\}$ be a substitution, where x^* is variable not mentioned in $L^{[1]}$. Then $L^{[1]}\delta$ is also a loop of P. Notice that A'_2 does not mention x, so $A'_2 = A'_2\delta$, thus $A'_2 \in L^{[1]} \cap L^{[1]}\delta$. Let $L^{[2]} = L^{[1]} \cup L^{[1]}\delta$, then $L^{[2]}$ is also a loop of P as the union of any two loops that have a common element is also a loop. Clearly,

$$|Var(L^{[2]})| = |Var(L^{[1]})| + 1.$$

This procedure can be iterated, and for any $L^{[i]}$, we can construct a loop $L^{[i+1]}$ such that

$$|Var(L^{[i+1]})| = |Var(L^{[i]})| + 1.$$

So, for any given nature number N^*, we can construct a loop L^* of P such that $|Var(L^*)| > N^*$, a contradiction with condition 1.

Finally, "4 $\Leftrightarrow$ 5" as $P_{eq}|D$ is a finite program, thus for any loop of this program, there is a maximal one containing it. ■

Notice that for the purpose of computing loops, EQ_D is not needed: $P_{eq}|D$ and $P_{eq}|D \cup EQ_D$ have the same loops. Notice also that since P is a finite set of rules, the set D is finite, thus $P_{eq}|D$ in the theorem is also finite. In this case, since $P_{eq}|D$ has no variables, its loops cannot have variables either. In fact, for $P_{eq}|D$, the definition of loops given here is the same as that in the propositional case given in (Lin & Zhao 2004). In particular, the set of loops of $P_{eq}|D$ is finite. From Theorem 2.5, we have the following proposition:

Proposition 8 *Let P be a finite program, M the number of rules in P, N the maximum length of the rules in P, n the number of predicates (including equality) in P, k the maximum arity of the predicates (counting equality) in P, and c the number of constants in P. Then there is a $O(MNn^2(c+2)^{2k})$ algorithm for checking whether P has a finite complete set of loops.*

Proof: $n(c+2)^k$ is the maximum number of atoms in $P_{eq}|D$, $n^2(c+2)^{2k}$ the maximum number of arcs in the dependency graph of $P_{eq}|D$, $O(MNn^2(c+2)^{2k})$ the worst case time for constructing the graph, and there is a linear time algorithm in the size of the dependency graph of $P_{eq}|D$ for computing all maximal loops of the program. ■

The exact complexity of deciding whether P has a finite complete set of loops is an open question.

Recall that a variable in a rule is said to be a local one if it occurs in the body of the rule but not in the head. A variable of a program is said to be a local one if it is a local variable in one of the rules in the program. From Theorem 2, we can easily prove the following result.

Proposition 9 *If P does not have any local variables, then P has a finite complete set of loops.*

Proof: If P has no local variables in any of its rules, then by the definition of dependency graph G_P, if there is an arc from A to B in G_P, then $Var(A) = Var(B)$. The result then follows from the definition of the loop and Theorem 2. ■

For instance, programs P_1 and P_2 in Example 1 do not have any local variables.

Concluding remarks

We have proposed notions of loops and loop formulas for normal logic programs with variables. Our main result is Theorem 1, which basically says that for a logic program with variables, we can first compute a first-order theory consisting of its completion and loop formulas so that when given a finite domain, the instantiation of the first-order theory on this domain yields a propositional theory whose models are the same as the answer sets of the ground logic program obtained from instantiating the first-order logic program.

This means that in principle, we can pre-compute the completion and loop formulas of a logic program with variables before we know the actual problem domain, and once the domain is known, all that is needed is to instantiate the first-order theory on the domain to yield a set of clauses to be given to a SAT solver. It is likely that for logic programs that have finite complete sets of loops, pre-computing all the loops is more efficient than ASSAT-like strategies that have to compute loop formulas on each ground program, especially when the performance is averaged over a large number of domains. However, for logic programs such as Niemelä's program for Hamiltonian Circuit problem (Niemelä 1999) that do not have a finite complete set of loops, it is not clear whether pre-computing first-order loops will be more effective computationally. To answer this question, an empirical study is needed.

Another future work is that as we have mentioned, in this paper, the semantics of logic programs with variables are defined according to the propositional answer set semantics of the the grounded programs. In other words, variables in programs are just place-holders, and rules with variables are schemas. In this sense, the logic programs considered in this paper are not truly first-order. Recently, Lifschitz[1] introduced a semantics of logic programs with variables that does not depend on grounding. Also the formulation of answer set semantics for normal logic programs in circumscription (Lin 1991) can be applied to programs with variables as well. How the notions of loops and loop formulas proposed in this paper relate to these semantics of logic programs with variables is an open question that is worth studying.

Acknowledgments

Yisong Wang and Mingyi Zhang were partially supported by the Natural Science Foundation of China under grant NSFC 60573009 and the stadholder foundation of Guizhou Province under grant 2005(212).

[1]Invited talk at ASP'05, Bath, UK, July 2005.

References

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logics and Databases*. New York: Plenum Press. 293–322.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2004. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Log. 5(2)* 206–263.

Erdem, E.; Lifschitz, V.; Nakhleh, L.; and Ringe, D. 2003. Reconstructing the evolutionary history of indo-european languages using answer set programming. In *Proceedings of PADL 2003*, 160–176.

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

Lee, J., and Lin, F. 2006. Loop formulas for circumscription. *Artificial Intelligence* 170(2):160–185.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In *Proceedings of IJCAI-05*, 503–508.

Lifschitz, V. 1999. Action languages, answer sets and planning. In *The Logic Programming Paradigm: A 25-Year Perspective*. K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren, eds, Springer-Verlag.

Lin, F., and Zhao, Y. 2004. ASSAT: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence* 157(1-2):115–137.

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

Marek, V. W., and Truszczynski, M. 1999. 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.

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

Nogueira, M.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. An A-Prolog decision support system for the space shuttle. In *Proceedings of PADL 2001*, 169–183.