Bisimilarity, Datalog and Negation

Antoun Yaacoub
Lebanese University, Lebanon
Email: antoun.yaacoub@ul.edu.lb

ABSTRACT
We extend the concept of bisimilarity relation between datalog goals from positive datalog programs to stratified and restricted Datalog programs with negation. The introduction of negation forced us to reconsider the search space and the semantics in order to guarantee and preserve soundness and completeness results. We address the problem of deciding whether two given goals are bisimilar with respect to a given program. When the given programs are stratified or restricted with negation, this problem is decidable.

KEYWORDS
Logic programming — Equivalence of goals — Datalog — Decision problem — Computational complexity — Deductive database
© 2016 by Orb Academic Publisher. All rights reserved.

1. Introduction
In [1, 2, 3, 4, 5], a formal framework is given for deciding the bisimilarity relation between Datalog goals. However, all of these papers deal with positive programs only. This paper extends the framework of [1] to interpreters for Datalog programs with negation, i.e. logic programs allowing negative literals in clauses bodies.

We restrict our attention to stratified and restricted Datalog programs, for which a clear semantics is available [6, 7, 8]. One of the well known problems is the occurrence of floundering [9, 10]: a nonground negative goal cannot be answered properly. Thus, goals of the form $\leftarrow \text{not } p(x)$ are not allowed. Since the SLDNF-trees do not present enough detail in the treatment of negative literals, these trees are augmented and show the construction of subsidiary SLDNF-trees of $\leftarrow G$ when $\text{not } G$ is selected [11].

The goal of this paper is to suggest the use of equivalence relations between logic programs that take into account the shape of the SLDNF-trees that these programs give rise to. This idea is not new: in automata theory, for instance, many variants of the equivalence relation of bisimilarity have been defined in order to promote the idea that automata with the same trace-based semantics should sometimes not be considered as equivalent if they are not bisimilar [12].

In this paper, we consider Datalog programs with negation. Furthermore, comparing two given Datalog programs with negation and taking into account the shape of the SLDNF-trees they give rise to, necessitates the comparison of infinitely many SLDNF-trees. Thus, we restrict our study to the comparison of two given Datalog goals. We will say that, with respect to a fixed Datalog program $P$ with negation, two given goals are equivalent when their SLDNF-trees are bisimilar.

In this paper, we investigate the computability of the equivalence problem between Datalog goals. In particular, we examine the following decision problems:

- given two Datalog goals $F, G$ and a stratified Datalog program $P$, determine if the SLDNF-trees of $P \cup F$ and $P \cup G$ are bisimilar.
- given two Datalog goals $F, G$ and a restricted Datalog program with negation $P$, determine if the SLDNF-trees of $P \cup F$ and $P \cup G$ are bisimilar.

In section 2 of this paper, we will present some basic notions about Datalog programs with negated literals, syntax and semantics. In section 3, we will introduce the concept of bisimulation between Datalog goals with negated literals. In section 4, we will address the problem of deciding whether two given goals are bisimilar with respect to a given stratified Datalog program. In section 5, we will address the same question as in section 4 by considering here restricted Datalog programs with negation. These programs allow a specific kind of recursion in the clauses. Section 6 concludes this paper.

2. Datalog Programs with negated literals
Datalog [13, 14, 15, 16] is a simplified version of Prolog. A Datalog program consists of a finite set of Horn clauses of the form $A_0 \leftarrow A_1, \ldots, A_n$, where each $A_i$ is a literal of the form $p(t_1, \ldots, t_k)$ such that $p$ is a predicate symbol of arity $k$ and the $t_1, \ldots, t_k$ are terms. A term is either a constant or a variable.
The left-hand side of a Datalog-clause is called its head and the right-hand side is called its body. Any Datalog program must satisfy the following condition: each variable which occurs in the head of a clause must also occur in the body of the same clause. A Datalog program $P$ is said to be stratified if it forbids recursion inside negation. For example, any program containing a clause of the form $p \leftarrow q, \neg p$ is not stratified. Nor is any program containing clauses of the form:

$p \leftarrow q, \neg r$

$r \leftarrow s, p$

We say that if $P$ contains a clause of the form $A \leftarrow \cdots, p(\cdots), \cdots$, then predicate $p$ occurs positively in the clause, and if $P$ contains a clause of the form $A \leftarrow \cdots, \neg p(\cdots), \cdots$ then predicate $p$ occurs negatively in the clause. Note that, a predicate could occur both positively and negatively, even in the same clause.

A Datalog program $P$ is stratified when there is a partition $P = P_0 \cup P_1 \cup \cdots \cup P_n$ ($P_i$ and $P_j$ disjoint for all $i \neq j$) such that, for every predicate $p$:

- the definition of $p$ (all clauses with $p$ in the head) is contained in one of the partitions/strata $P_i$

and, for each $1 \leq i \leq n$:

- if a predicate occurs positively in a clause of $P_i$ then its definition is contained within $\bigcup_{j \leq i} P_j$

- if a predicate occurs negatively in a clause of $P_i$ then its definition is contained within $\bigcup_{j > i} P_j$

A program $P$ is said to be stratified if there is any such partition.

**Example 2.1. Stratified program**

Let $P$ be the following program:

$p(X) \leftarrow q(X), \neg r(X)$;

$p(X) \leftarrow q(X), \neg t(X)$;

$r(X) \leftarrow s(X), \neg t(X)$;

$t(a) \leftarrow$;

$s(a) \leftarrow$;

$s(b) \leftarrow$;

$q(a) \leftarrow$

A possible stratification of $P$ is:

$P = \{ p(X) \leftarrow q(X), \neg r(X); p(X) \leftarrow q(X), \neg t(X) \} \cup$

$\{ r(X) \leftarrow s(X), \neg t(X) \} \cup$

$\{ t(a) \leftarrow; s(a) \leftarrow; s(b) \leftarrow; q(a) \leftarrow \}$

**Example 2.2. Non-stratified program**

Let $P$ be the following program:

$p \leftarrow q, \neg r$;

$r \leftarrow s, \neg p$;

$q \leftarrow$

This program cannot be stratified since it contains a recursion through negation.

The dependency graph of a stratified Datalog program $P$ is the graph $(N, E)$ where $N$ is the set of all predicate symbols occurring in $P$ and $E$ is the adjacency relation (edges labeled +/−) defined on $N$ as follows:

- $pE^+ q$ (p refers + to q) iff $P$ contains a clause of the form $p(\cdots) \leftarrow \cdots, q(\cdots), \cdots$

- $pE^- q$ (p refers − to q) iff $P$ contains a clause of the form $p(\cdots) \leftarrow \cdots, \neg q(\cdots), \cdots$

Let $E^*$ be the reflexive transitive closure of $E$.

A logic program $P$ is stratified iff the dependency graph for $P$ contains no cycles containing a negative edge.

A Datalog program $P$ is said to be restricted iff for all clauses $A_0 \leftarrow A_1, \cdots, A_n$ in $P$ and for all $1 \leq i \leq n - 1$, if $A_0$ is of the form $p(\cdots)$ and $A_i$ is of the form $\neg q(\cdots)$, then $\neg qE^* p$

**Example 2.3. Dependency Graph of a Stratified program**

The dependency graph of the following program is:

$p \leftarrow q, r$;

$p \leftarrow \neg q, s$;

$q \leftarrow q, not t$;

**Example 2.4. Dependency Graph of a non-stratified program**

The dependency graph of the following program is:

$p \leftarrow q, r$;

$q \leftarrow \not q, p, s$;

$q \leftarrow q, not t$;

The logic program $P$ is not stratified since the dependency graph for $P$ contains a cycle containing a negative edge (note the existence of the negative edge directed from node q to node $p$).

The inference rule we rely on in this paper, is the so-called SLDNF proof procedure. In fact, the computation of a goal/query $G \leftarrow L_1, \cdots, L_m$ is a series of derivation steps:
There are two kinds of derivation steps. Let $P$ be the following program:

$q \leftarrow \theta$

Let $P$ be the following program:

$q$

and let $G$ be the goal $\leftarrow p(a)$. The next figure shows the SLDNF-tree for this goal. The success and failure branches are discussed:

Since the goal $\leftarrow p(b)$ fails (its associated tree shows a failure branch), $\leftarrow \not p(b)$ succeeds and thus $\leftarrow p(a)$ succeeds.

The computation rule picks out one of the literals of the current goal (query). The computation rule must be safe: it must not pick a negative literal containing a variable. This is necessary for soundness. If the current goal contains only negative literals with variables then the computation cannot proceed: it flounders.

The SLDNF proof procedure, which is the most common way of executing normal logic programs is sound, but not complete (because of the possibility of floundering and of infinite computation trees) with respect to this semantics.

3. Bisimulation with negated literals

As stated in [1], a bisimulation is a binary relation between goals such that related goals, even NBF ones, have “equivalent” SLD-trees.

Let $P$ be a Datalog program with negated literals. A binary relation $\mathcal{Z}$ between Datalog goals is said to be a P-bisimulation if it satisfies the following conditions for all Datalog goals $F_1, G_1$ such that $F_1 \not\not\not \mathcal{Z} G_1$:

- $F_1 = \Box$ or $G_1 = \Box$.
- For each resolvent $F_2$ of $F_1$ and a clause in $P$, there exists a resolvent $G_2$ of $G_1$ and a clause in $P$ such that $F_2 \not\not\not \mathcal{Z} G_2$.
- For each resolvent $G_2$ of $G_1$ and a clause in $P$, there exists a resolvent $F_2$ of $F_1$ and a clause in $P$ such that $F_2 \not\not\not \mathcal{Z} G_2$.
- For the SLD-tree $\Phi$ associated to $F_1 = \leftarrow \not A_1, \cdots$, there exists an SLD-tree $\Phi'$ associated to $G_1 = \leftarrow \not B_1, \cdots$ such that $A_1 \not\not\not \not\not \not \mathcal{Z} B_1$.
- For the SLD-tree $\Phi'$ associated to $G_1 = \leftarrow \not B_1, \cdots$, there exists an SLD-tree $\Phi$ associated to $F_1 = \leftarrow \not A_1, \cdots$ such that $B_1 \not\not\not \not\not \not \mathcal{Z} A_1$.

Note that the main difference between this definition of bisimulation and the one proposed for datalog programs without negated literals is the introduction of two supplementary tests.
These tests check whether the SLD-trees of the two leftmost negated atoms (if they exist) in a goal are bisimilar. One can check easily that the set of all P-bisimulations is closed under taking arbitrary unions.

**Proposition 3.1.** There exists a maximal P-bisimulation, namely the binary relation \( \mathcal{Z}^p_{\text{max}} \) between Datalog goals with negated literals defined as follows: \( F_1 \mathcal{Z}^p_{\text{max}} G_1 \) if there exists a P-bisimulation \( \mathcal{Z} \) such that \( F_1 \mathcal{Z} G_1 \). \( \mathcal{Z}^p_{\text{max}} \) is an equivalence relation on the set of all Datalog goals.

**Example 3.2.** Let \( P \) be the following program:

\[
\begin{align*}
p(a, b) &\leftarrow \text{not } q(b, a), \\
q(a, b) &\leftarrow, \\
q(a, b) &\leftarrow q(a, b), \\
\text{and let } F &\leftarrow p(a, y) \text{ and } G &\leftarrow b(y).
\end{align*}
\]

\( \mathcal{Z} \) is not a P-bisimulation due the presence of an associated tree for the goal \( \leftarrow q(b, a) \) in the left tree and the absence of an associated tree for the goal \( \leftarrow q(a, b) \) in the right tree. Since \( F \not\mathcal{Z} G \), then \( F \not\mathcal{Z}^p_{\text{max}} G \).

It was shown in [1] that, in the general case, it is undecidable, given a Prolog program \( P \) and Prolog goals \( F_1, G_1 \), to determine whether \( F_1 \mathcal{Z}^p_{\text{max}} G_1 \). Thus, we will restrict our language and consider stratified and restricted Datalog programs with negation in order to restore the decidability of our decision problem.

**4. Decidability of Bisimulation for Stratified Datalog Programs with Negation**

We now study the computational decidability of the following decision problem: \( \pi_{\text{str,neg}} \) given an stratified Datalog program \( P \) with negation and Datalog goals \( F_1, G_1 \), determine whether \( F_1 \mathcal{Z}^p_{\text{max}} G_1 \). In this respect, let \( P \) be a stratified Datalog program with negation. In Algorithm 1, bothempty \( (F_1, G_1) \) is a Boolean function returning true iff \( F_1 = \emptyset \) and \( G_1 = \emptyset \), whereas bothfail \( (F_1, G_1) \) is a Boolean function returning true iff \( F_1 \neq \emptyset \), successor \( (F_1) = \emptyset \), \( G_1 \neq \emptyset \) and successor \( (G_1) = \emptyset \). Moreover, successor \( (.) \) is a function returning the set of all resolvents of its argument with a clause of \( P \) whereas get-element \( (.) \) is a function removing one element from the set of elements given as input and returning it.

In order to demonstrate the decidability of \( \pi_{\text{str,neg}} \), we need to prove the following lemmas for all Datalog goals \( F_1, G_1 \):

**Algorithm 1: function bisim1 \( (F_1, G_1) \)**
Lemma 4.1 (Termination). \( \text{bisiml}(F_1, G_1) \) terminates.

Lemma 4.2 (Completeness). If \( F_1 \overset{F}{\rightarrow}_{\max} G_1 \), then \( \text{bisiml}(F_1, G_1) \) returns true.

Lemma 4.3 (Soundness). If \( \text{bisiml}(F_1, G_1) \) returns true, then \( F_1 \overset{F}{\rightarrow}_{\max} G_1 \).

In order to prove the termination, we will introduce two notions. The first is the level of a goal and the second is a measure which calculates the maximum number of steps needed to reach a leaf from some node in a SLDF-tree.

Let \( F = \leftarrow A_1, \ldots, A_n, \not B_1, \ldots, \not B_m \), the level of \( F \) is defined as follows:

\[
\text{level}(F) = \begin{cases} 
\max\{\text{level}(A_1), \ldots, \text{level}(A_n), \text{level}(B_1), \ldots, \text{level}(B_m)\} & \text{if } A_i \in F, i = 1, \ldots, n; \text{ level}(A_i) = 0 \\
0 & \text{if } \forall B_j \in F, j = 1, \ldots, m; \text{ level}(b_j) = 0
\end{cases}
\]

Consider also \( H_1 \) a Datalog goal with negated atoms. Let:

\[
M(H_1) = \max\{L(D) | D = (H_1 \Rightarrow \cdots \Rightarrow H_n) \text{ a derivation from } H_1\}
\]

and let:

\[
L(D) = \begin{cases} 
0 & \text{if } D = (H_1) \\
\sum_{i=1}^{n} C(H_i, H_{i+1}) & \text{if } D = (H_1 \Rightarrow \cdots \Rightarrow H_n)
\end{cases}
\]

where

\[
C(H_i, H_{i+1}) = \begin{cases} 
1 & \text{if } H_i = \leftarrow A, \cdots \\
M(A) + 1 & \text{if } H_i = \not A, \cdots
\end{cases}
\]

We can prove by induction on the level of goals, that the M-function is well-defined.

Proof. Let \( F = \leftarrow H_1, \ldots, H_n \) be a goal of level zero, then for every \( i = 1, \ldots, n \), level\((H_i)\) = 0. Thus, each \( H_i \) is either a fact or/and an atom that appears in the body of some clause(s).

Thus, the function \( C \) for goals of level 0 is always equal to 1. Consequently, \( M(F) = n - 1 \).

Suppose that \( M(F) \) is defined for all goals of level \( \leq k \).

Consider a goal \( F \) of level \( k + 1 \). Consider also an atom \( A \) (positive or negative) in the goal \( F \) of level \( k + 1 \). Since \( A \) is of level \( k + 1 \), then \( A \) will be resolved into resolvents of level \( \leq k \).

Thus \( F \) of level \( k + 1 \) will be resolved to some goals of level \( \leq k \).

For the previous example, \( M(\leftarrow p(a, y)) = 3 \), and \( M(\leftarrow p(b, y)) = 2 \).

Let \( \ll \) be the binary relation on the set of all pairs of Datalog goals defined by: \((F_2, G_2) \ll (F_1, G_1)\) iff

- \( M(F_2) \leq M(F_1) \),
- \( M(G_2) < M(G_1) \),

\( \ll \) is a well-founded partial order on the set of all pairs of goals.

Proof of Lemma 1. The proof is done by \( \ll \)-induction on \( (F_1, G_1) \).

Let \((F_1, G_1) \) be such that for all \((F_2, G_2), \) if \((F_2, G_2) \ll (F_1, G_1) \) then \( \text{bisiml}(F_2, G_2) \) terminates. Since every recursive call to \( \text{bisiml} \) that is performed along the execution of \( \text{bisiml}(F_1, G_1) \) is done with respect to a pair \((F_2, G_2)\) of goals such that \((F_2, G_2) \ll (F_1, G_1) \), then \( \text{bisiml}(F_1, G_1) \) terminates.

Proof of Lemma 2. Let us consider the following property:

\((\text{Prop}_1(F_1, G_1)) \) if \((F_1, G_1) \) then \( \text{bisiml}(F_1, G_1) \) returns true. Again, we proceed by \( \ll \)-induction. Suppose \((F_1, G_1) \) is such that for all \((F_2, G_2), \) if \((F_2, G_2) \ll (F_1, G_1) \) then \( \text{Prop}_1(F_2, G_2) \).

Let us show that \( \text{Prop}_1(F_1, G_1) \). Suppose \((F_1, G_1) \). Hence, for all successors \( F_2 \) of \( F_1 \), there exists a successor \( G_2 \) of \( G_1 \) such that \( F_2 \overset{F}{\rightarrow}_{\max} G_2 \), and conversely. Seeing that the logic program is stratified, then \((F_2, G_2) \ll (F_1, G_1) \). By induction hypothesis, \( \text{Prop}_1(F_2, G_2) \). Since \( F_2 \overset{F}{\rightarrow}_{\max} G_2 \), then \( \text{bisiml}(F_2, G_2) \) returns true. As a result, one sees that \( \text{bisiml}(F_1, G_1) \) returns true.

Proof of Lemma 3. It suffices to demonstrate that the binary relation \( \ll \) defined as follows between Datalog goals is a bisimulation:

\( F_1 \overset{F}{\rightarrow}_{\max} G_1 \) if \( \text{bisiml}(F_1, G_1) \) returns true. Let \( F_1, G_1 \) be Datalog goals such that \( F_1 \overset{F}{\rightarrow}_{\max} G_1 \). Hence, \( \text{bisiml}(F_1, G_1) \) returns true. Thus, obviously, \( F_1 = \square \) iff \( G_1 = \square \), and the first condition characterizing bisimulations holds for \( \ll \). Now, suppose that \( F_2 \) is a resolvent of \( F_1 \) and a clause in \( P \). Since \( \text{bisiml}(F_1, G_1) \) returns true, then there exists a resolvent \( G_2 \) of \( G_1 \) and a clause in \( P \) such that \( \text{bisiml}(F_2, G_2) \) returns true, i.e. \( F_2 \overset{F}{\rightarrow}_{\max} G_2 \). As a result, the second condition characterizing bisimulations holds for \( \ll \). The third condition characterizing bisimulations holds for \( \ll \) too, as the reader can quickly check. The same discussion applies for any couple of associated trees to negated bisimilar goals, verifying thus the fourth and fifth conditions. Thus \( \ll \) is a bisimulation.

As a consequence of lemmas 1–3, we have:

Theorem 4.4. Algorithm 1 is a sound and complete decision procedure for \((\pi_{\text{str}})\).

It follows that \((\pi_{\text{str}})\) is decidable.

5. Decidability of Bisimulation for Restricted Datalog Programs with Negation

In this section, we will consider restricted programs with negation with a restriction that we will not allow any dependencies between the rightmost negative atom (if it is exists) in a clause and the atom in its head.

We study here the computational decidability of the following decision problem: \((\pi_{\text{str}})\) given a restricted Datalog program \( P \) with negation and Datalog goals \( F_1, G_1 \), determine whether \( F_1 \overset{F}{\rightarrow}_{\max} G_1 \). In this respect, let \( P \) be a restricted Datalog program with negation. In Algorithm 2, bothempty \((F_1, G_1)\), bothfail \((F_1, G_1)\) and occur \((F_1 \Rightarrow \cdots \Rightarrow F_i, (G_1 \Rightarrow \cdots \Rightarrow G_i))\) are similar to the corresponding functions used in
begin
if (SLDNF-tree($F_1$) exists and SLDNF-tree($G_1$) not exists) or (SLDNF-tree($F_1$) not exists and SLDNF-tree($G_1$) exists) or (SLDNF-tree($F_1$) exists with root $A$ and SLDNF-tree($G_1$) exists with root $B$ and bisim2($A, B = false$)) then
  return $false$
else
  if bothempty($F_1, G_1$) or bothfalse($F_1, G_1$) or occur(($F_1 \Rightarrow \cdots \Rightarrow F_i), (G_1 \Rightarrow \cdots \Rightarrow G_i)$) then
    $SF \leftarrow$ successor($F_1$)
    $SG \leftarrow$ successor($G_1$)
    if $SF \neq \emptyset$ and $SG \neq \emptyset$ then
      $SF' \leftarrow SF$
      while $SF' \neq \emptyset$ do
        $F' \leftarrow$ get-element($SF'$)
        found-bisim $\leftarrow$ false
        $SG' \leftarrow SG$
        while $SG' \neq \emptyset$ and
          found-bisim $\leftarrow$ false do
            $G' \leftarrow$ get-element($SG'$)
            found-bisim $\leftarrow$ bisim2($(F_1 \Rightarrow \cdots \Rightarrow F_i), (G_1 \Rightarrow \cdots \Rightarrow G_i)$)
          end
          if found-bisim $\leftarrow$ false then
            return $false$
        end
      end
      $SG' \leftarrow SG$
    end
    while $SG' \neq \emptyset$ do
      $G' \leftarrow$ get-element($SG'$)
      found-bisim $\leftarrow$ false
      $SF' \leftarrow SF$
      while $SF' \neq \emptyset$ and
        found-bisim $\leftarrow$ false do
          $F' \leftarrow$ get-element($SF'$)
          found-bisim $\leftarrow$ bisim2($(F_1 \Rightarrow \cdots \Rightarrow F_i), (G_1 \Rightarrow \cdots \Rightarrow G_i)$)
        end
        if found-bisim $\leftarrow$ false then
          return $false$
      end
    end
  end
else
  return $false$
end

Algorithm 2: function bisim2($F_1 \Rightarrow \cdots \Rightarrow F_i), (G_1 \Rightarrow \cdots \Rightarrow G_i)$)

As the reader can see, Algorithm 2 is very similar to the other algorithms. In order to demonstrate the decidability of $(\pi_{\text{resneg}})$, we need to prove the following lemmas for all Datalog goals $F_1, G_1$.

Lemma 5.1 (Termination). bisim2 ($(F_1), (G_1)$) terminates.

Lemma 5.2 (Completeness). If $F_1 \not\equiv_{\text{max}}^P G_1$, then bisim2 ($(F_1), (G_1)$) returns true.

Lemma 5.3 (Soundness). $\text{bisim2} ((F_1), (G_1)) \text{ returns true, then } F_1 \not\equiv_{\text{max}}^P G_1$.

One can follow the same proofs presented for the positive case [1] except that we need to prove that for a restricted program $P$ with negation and a goal $F_1$, all its derived subgoals in all the derivations must be bounded.

For this, suppose that the derived subgoals are not bounded and suppose that $F_1 = \leftarrow A$.

Consider a derivation $D$ of the form $D = (F_1 \Rightarrow F_2 \Rightarrow \cdots \Rightarrow F_n)$. One can decompose each $F_i (i \geq 2)$ into the concatenation of suffixes of body of clauses:

$F_i = L_1^i, \cdots, L_{k_1}^i, L_1^i, \cdots, L_{k_2}^i, L_{k_2+1}^i, \cdots, L_{k_3}^i, \cdots, L_{k_n}^i,$

where $1 \leq i < n$, $k_1 > 0, \ldots, k_n > 0$. Note that, for example, $L_1^i, \cdots, L_{k_1}^i$ is a suffix of the body of the clause $p_i \leftarrow Q_i$ in the program $P$, and that $L_{k_1+1}^i, \cdots, L_{k_n}^i$ is a suffix of the body of the clause $p_{i+1} \leftarrow Q_{i+1}$. Since $F_i$ can be decomposed as described, then the head atom $p_i$ of the clause $p_i \leftarrow Q_i$ can be unified with the atom $P_{k_1}^i$ which precedes $L_{k_1}^i$ contained in $Q_{i+1}$.

Since $\text{level}(L_{k_1}^i) < \text{level}(p_{i+1})$ (as $p_{i+1}$ depends on $L_{k_1}^i$), then $\text{level}(p_i) < \text{level}(p_{i+1})$. This contradicts the fact that $F_i$ is not bounded.

As a consequence of lemmas 4 – 6, we have:

Theorem 5.4. Algorithm 4 is a sound and complete decision procedure for $(\pi_{\text{resneg}})$.

It follows that $(\pi_{\text{resneg}})$ is decidable.

6. Conclusion

In this paper, we have introduced the concept of bisimulation between Datalog goals with negation: two Datalog goals are bisimilar with respect to a given program when their SLDNF-trees are bisimilar. As proved, when the given logic program is stratified or restricted with negation, the problem of deciding whether two given goals are bisimilar is decidable. The proof of decidability of bisimulation problem for restricted logic program with negation that we presented in Section 5 is based on techniques that were developed in [17] for detecting loops in logic programming.

We have shown the decidability of the bisimilarity relation between negated Datalog goals. We are currently working toward evaluating the complexity of our algorithms.

Future work can be dedicated also to the study of the adaptation of the flow definitions, presented in [1, 2, 3, 4] for positive Datalog programs, to Datalog programs with negation. These flows could also be studied for Datalog programs with negation equipped with bottom up evaluation techniques.
References


