Information flow in logic programming

Ali Awada[1], Philippe Balbiani[2], Antoun Yaacoub[2]

[1] Lebanese University

[2] Institut de Recherche en Informatique de Toulouse

Abstract : This paper proposes a theoretical foundation of what could be an information flow in logic programming. Several information flow definitions (based on success/failure, substitution answers, bisimulation between resolution trees of goals) are stated and compared. Decision procedures are given for each definition and complexity is studied for specific classes of logic programs.

Keywords : Logic programming, Information flow, Computational complexity

1 Introduction

Data security is the science and study of methods of protecting data in computer and communication systems from unauthorized disclosure and modification. One of the aspects of data security is the control of information flow in the system. In some sense, an information flow should describe controls that regulate the dissemination of information. These controls are needed to prevent programs from leaking confidential data, or from disseminating classified data to users with lower security clearances. The theory of information flow is well defined for imperative programming. Different models of information flow were proposed, namely, the Bell-LaPadula Model (Bell & LaPadula, 1973), nonlattice and nontransitive models (Foley, 1989; Denning, 1976) of information flow, and nondeducibility and noninterference (Goguen & Meseguer, 1982). Each model has rules about the conditions under which information can move throughout the system. For example, in the Bell-LaPadula Model which describes a lattice-based information flow policy, information can flow from an object in security level A to a subject in security level B if and only if B dominates A. Both compile-time mechanisms (Denning & Denning, 1977) and runtime mechanisms (Fenton, 1974) supporting the checking of information flows were also proposed.

Intuitively, information flows from an object x to an object y if the application of a sequence of commands causes the information initially in x to affect the information in y. For example, the sequence $tmp := x; y := tmp;$ has information flowing from x to y because the value of x at the beginning of the sequence is revealed when the value of y is determined at the end of the sequence. Several studies (Denning, 1982) addressed information flow in imperative programming, but none were concerned to bring answers of what could be an information flow in security systems for logic programming. In fact, logic programming is a well-known declarative method of knowledge representation and programming based on the idea that the language of first-order logic is

well-suited for both representing data and describing desired outputs. Logic programming was developed in the early 1970s based on work in automated theorem proving, in particular, on Robinson's resolution principle.

In this paper, we propose three definitions of information flows in logic programs. These definitions correspond to what can be observed by the user when a query $\leftarrow G(x, y)$ is run on a logic program P. In section 2 of this paper, we will present some basic notions about logic programming, syntax and semantics. In section 3, several definitions of information flow in logic programming are proposed relatively for a logic program P and a goal $\leftarrow G(x, y)$ of arity 2, (which stipulates the existence of a flow that passes from the variable x to the variable y in the goal $\leftarrow G(x, y)$). The implications between these definitions are then studied. Decision procedures are then given in section 4 for each of the previous definitions and computational issues studied for some types of logic programs.

2 Syntax and semantics

In this section, we introduce basic concepts of logic programming. See (Lloyd, 1984; Baral & Gelfond, 1994) for more details. In this article, we will use $p, q, \cdots$ for predicate symbols, $x, y, z, \cdots$ for variables, $f, g, h, \cdots$ for function symbols, and $a, b, c, \cdots$ for constants.The language L considered here is essentially that of first order predicate logic. It has countable sets of variables, function symbols and predicate symbols, these sets being mutually disjoint. Each function and predicate symbol is associated with a unique natural number called its *arity* ; a (function or predicate) symbol whose arity is n is said to be an *n-ary* symbol. A *0-ary* function symbol is referred to as a constant. A *term* is a variable, a constant, or a compound term $f(t_1, \cdots, t_n)$ where f is an *n-ary* function symbol and the t_i are terms, $1 \leq i \leq n$. A term is *ground* if no variable occurs in it. The *Herbrand universe* of L, denoted U_L, is the set of all ground terms that can be formed with the functions and constants in L. An *atom* is of the form $p(t_1, \cdots, t_n)$, where p is an *n-ary* predicate symbol and the t_i are terms, $1 \leq i \leq n$. An atom is *ground* if all t_i are ground. The *Herbrand base* of a language L, denoted B_L, is the set of all ground atoms that can be formed with predicates from L and terms from U_L. A *clause* is an expression of the form $A \leftarrow B_1, \cdots, B_n$ where $A, B_1, \cdots, B_n$ are atoms. A is called the head of the clause and $B_1, \cdots, B_n$ is called its body. A goal is an expression of the form $\leftarrow B_1, \cdots, B_n$. A clause r of the form $A \leftarrow$ (i.e., whose body is empty) is called a fact, and if A is a ground atom, then r is called a ground fact. The empty goal is denoted $\Box$. A *predicate definition* is assumed to consist of a finite set (possibly ordered) of clauses defining the same predicate. A *logic program* consists of a finite set of predicate definitions. With each logic program P, we associate the language $L(P)$ that consists of the predicates, functions, and constants occurring in P. If no constant occurs in P, we add some constant to $L(P)$ to have a nonempty domain. A *substitution* is an idempotent mapping from a finite set of variables to terms. The identity substitution will be denoted ϵ. A substitution σ_1 is said to be more general than a substitution σ_2 if there is a substitution θ such that $\sigma_2 = \theta\sigma_1$. Two terms t_1 and t_2 are said to be *unifiable* if there exists a substitution σ such that $\sigma(t_1) = \sigma(t_2)$; in this case σ is said to be a unifier for the terms. If two terms t_1 and t_2 have a unifier, then they have a *most*

general unifier $mgu(t_1, t_2)$ that is unique up to variable renaming. In this paper, we will be interested in

- Datalog programs, i.e. logic programs without function symbols and where each variable appearing in the head of the clause, must also appear in its body.
- Binary programs, i.e. logic programs such that, the body of every program statement is composed of at most one atom.
- Hierarchical programs, i.e. logic program having a level mapping such that, in every program statement $A(t_1, \cdots, t_n) \leftarrow B$, the level of every predicate symbol in B is less than the level of A.

Note that the level mapping of a program is a mapping from its set of predicate symbols to the non-negative integers. We refer to the value of the predicate symbol under this mapping as the level of that predicate symbol. The operational behavior of logic programs can be described by means of *SLD-derivations*. An SLD-derivation for a goal $G = \leftarrow A_1, \cdots, A_n$ with respect to a program P is a sequence of goals $G_0, \cdots, G_i, G_{i+1}, \cdots$, such that $G_0 = G$, and if $G_i = \leftarrow B_1, \cdots, B_m$, then $G_{i+1} = \leftarrow \theta B_1, \cdots, \theta B_{i-1}, \theta B'_1, \cdots, \theta B'_k, \theta B_{i+1}, \cdots, \theta B_m$ such that $1 \leq i \leq m$; $B \leftarrow B'_1, \cdots, B'_k$ is a variant of a clause in P that has no variable in common with any of the goals $G_0, \cdots, G_i$; and $\theta = mgu(B_i; B)$. The goal G_{i+1} is said to be obtained from G_i by means of *resolution step*, and B_i is said to be the *resolved atom*. Let $G_0, \cdots, G_n$ be an SLD-derivation for a goal G with respect to a program P, and let θ_i be the unifier obtained when resolving the goal G_{i-1} to obtain G_i, $1 \leq i \leq n$. If this derivation is finite and maximal, i.e., one in which it is not possible to resolve the goal G_n with any of the clauses in P, then it corresponds to a terminating computation for G : in this case, if G_n is the empty goal then we say that $P?G$ succeeds and the computation is said to succeed with answer substitution θ, where θ is the substitution obtained by restricting the substitution $\theta_n \cdots \theta_1$ to the variables occurring in G; if G_n is not the empty goal, then the computation is said to fail. We say that $P?G$ fails if all computations from G in P fail. If the derivation is infinite, the computation does not terminate. Given a program P and a goal G, let $\Theta(P?G)$ be the set of all answer substitutions of G in P.

3 Information flow

As the theory of information flow is well studied for imperative programming, it is tempting to see what could be an information flow in logic programming, especially given the fact that there are no notions of assignment, or variable of a program. In fact, variables in logic programs behave differently from variables in conventional programming languages. They stand for an unspecified but single entity rather than for a store location in memory. The following three definitions for information flow in logic programming are based on the following principle. The information flow that occurs when the user ask a goal to logic programs depends mainly on what parts of the computation the user sees. In the first definition, the user only sees whether goals succeed or fail. In the second definition, the user has access to the set of substitution answers computed by the program. In the third definition, the user obtains the shape of the computation trees produced by the program.

3.1 Successes and failures

Let P be a logic program, and $G(x, y)$ be a two variables goal. We shall say that there is a flow from x to y in $G(x, y)$ with respect to successes and failures in P (in symbols $x \xrightarrow{SF}{}^{P}_{G} y$) iff there exists $a, b \in U_{L(P)}$ such that $P?G(a, y)$ succeeds and $P?G(b, y)$ fails.
Example 1 Let P_1 be the following program : $p(a, b) \leftarrow$ and let $G_1(x, y)$ be the following goal : $\leftarrow p(x, y)$. Since $P_1?G_1(a, y)$ succeeds and $P_1?G_1(b, y)$ fails, then $x \xrightarrow{SF}{}^{P_1}_{G_1} y$.

3.2 Substitution answers

Let P be a logic program, and $G(x, y)$ be a two variables goal. We shall say that there is a flow from x to y in $G(x, y)$ with respect to substitution answers in P (in symbols $x \xrightarrow{SA}{}^{P}_{G} y$) iff there exists $a, b \in U_{L(P)}$ such that $\Theta(P?G(a, y)) \neq \Theta(P?G(b, y))$.
Example 2 Let P_2 be the following program : $p(a, y) \leftarrow$, and let $G_2(x, y)$ be the following goal : $\leftarrow p(x, y)$. Since $\Theta(P_2?G_2(a, y)) = \{\epsilon\}$ and $\Theta(P_2?G_2(b, y)) = \emptyset$, then $x \xrightarrow{SA}{}^{P_2}_{G_2} y$.

3.3 Bisimulation

Our third definition of flow is based on the notion of bisimulation between goals. Let P be a logic program and $\mathcal{Z}$ be a binary relation between goals. We shall say that $\mathcal{Z}$ is a $P - bisimulation$ iff for all goals G, H, if $G\mathcal{Z}H$ then :
- for all goals $G' \in succ_P(G)$, there exists $H' \in succ_P(H)$, such that $G'\mathcal{Z}H'$.
- for all goals $H' \in succ_P(H)$, there exists $G' \in succ_P(G)$, such that $G'\mathcal{Z}H'$.
- $G = \square$ iff $H = \square$.

Above, $succ_P(G)$ denotes the set of all goals obtained from a goal G by means of a resolution step in the program P. Obviously,

Lemma 1 The relation identity Id between goals is a $P - bisimulation$.
Lemma 2 If $\mathcal{Z}$ is a $P - bisimulation$, then $\mathcal{Z}^{-1}$ is also a $P - bisimulation$.
Lemma 3 If $\mathcal{Z}_1$, $\mathcal{Z}_2$ are two $P - bisimulations$, then the composition $\mathcal{Z}_1\mathcal{Z}_2$ defined by $\mathcal{Z}_1\mathcal{Z}_2 = \{(G, H)/\exists I, G\mathcal{Z}_1 I \text{ and } I\mathcal{Z}_2 H\}$ is also a $P - bisimulation$.
Lemma 4 Let $(\mathcal{Z}_i)_{i\in I}$ be a family of $P - bisimulations$, then $\bigcup_{i\in I} \mathcal{Z}_i$ is also a $P - bisimulation$.

By lemma 4, there exists a maximal $P - bisimulation$, denoted $\mathcal{Z}_{max}$.

Example 3 Let P be the following program formed of :
$p(a, y) \leftarrow q(y)$, $p(b, y) \leftarrow r(y)$ and $p(b, y) \leftarrow s(y)$
and let G, H be respectively the following goals $\leftarrow p(a, y)$ and $\leftarrow p(b, y)$
Let Z be the binary relation between goals such that :
- $\leftarrow p(a, y)\ Z \leftarrow p(b, y)$

– $\leftarrow q(y)\ Z \leftarrow r(y)$
– $\leftarrow q(y)\ Z \leftarrow s(y)$

Obviously, Z is a $P - bisimulation$. Since $G\ Z\ H$, then $G\ \mathcal{Z}_{max}\ H$.
By lemmas 1, 2 and 3, we have
Lemma 5 $\mathcal{Z}_{max}$ is an equivalence relation.

Let P be a logic program, and $G(x, y)$ be a two variables goal. We shall say that there is a flow from x to y in $G(x, y)$ with respect to the bisimulation in P (in symbols $x \xrightarrow{BI}{}^{P}_{G}\ y$) iff there exists $a, b \in U_{L(P)}$ such that *not* $G(a, y)\mathcal{Z}_{max}G(b, y)$.

Example 4 Let P_3 be the following program : $p(x, a) \leftarrow$ and $p(a, b) \leftarrow q(a)$ and let $G_3(x, y)$ be the goal : $\leftarrow p(x, y)$. Let us prove *not* $\leftarrow p(a, y)\mathcal{Z}_{max} \leftarrow p(b, y)$. Suppose that $\leftarrow p(a, y)\mathcal{Z}_{max} \leftarrow p(b, y)$. Since $\leftarrow q(a) \in succ_{P_3}(\leftarrow p(a, y))$, then there should be $G_3 \in succ_{P_3}(\leftarrow p(b, y))$ such that $\leftarrow q(a)\mathcal{Z}_{max}G_3$. The problem is that the only goal in $succ_{P_3}(\leftarrow p(b, y))$ is the empty goal, which cannot be bisimilar to $\leftarrow q(a)$. Hence, *not* $\leftarrow p(a, y)\mathcal{Z}_{max} \leftarrow p(b, y)$. Therefore $x \xrightarrow{BI}{}^{P_3}_{G_3}\ y$.

3.4 Links between the definitions of information flow

The existence of a flow with respect to substitution answers does not entail the existence of a flow with respect to successes and failures. To see this, it suffices to consider the following example.
Example 5 Let P be the following program : $p(a, b) \leftarrow$ and $p(Z, c) \leftarrow$ and let $G(x, y)$ be the goal : $\leftarrow p(x, y)$. Since $\Theta(P?G(a, y)) = \{y/b, y/c\}$ and $\Theta(P?G(b, y)) = \{y/c\}$, then $x \xrightarrow{SA}{}^{P}_{G}\ y$. Since $P?G(a, y)$ and $P?G(b, y)$ both succeed, then $x \overset{SF}{\not\longrightarrow}{}^{P}_{G}\ y$.

However, one can establish the following result.

Lemma 6 Let P be a logic program and $G(x, y)$ be a two variables goal. If $x \xrightarrow{SF}{}^{P}_{G}\ y$ then $x \xrightarrow{SA}{}^{P}_{G}\ y$.
Proof 1 Suppose that $x \xrightarrow{SF}{}^{P}_{G}\ y$, then there exists $a, b \in U_{L(P)}$ such that $P?G(a, y)$ succeeds and $P?G(b, y)$ fails. Therefore, $\Theta(P?G(a, y)) \neq \emptyset$ and $\Theta(P?G(b, y)) = \emptyset$. Consequently, $x \xrightarrow{SA}{}^{P}_{G}\ y$. □

The existence of a flow with respect to bisimulation does not entail the existence of a flow with respect to successes and failures. The next example explains why.
Example 6 Let P_3 and G_3 be the program and goal considered in example 4. We know that $x \xrightarrow{BI}{}^{P_3}_{G_3}\ y$. Nevertheless, since all the goals of the form $G_3(a, y)$, with $a \in U_{L(P)}$, succeed, thus $x \overset{SF}{\not\longrightarrow}{}^{P_3}_{G_3}\ y$.

Nevertheless, it is worth noting at this point the following.

Lemma 7 Let P be a logic program and $G(x, y)$ be a two variables goal. If $x \xrightarrow{SF}{}^{P}_{G} y$ then $x \xrightarrow{BI}{}^{P}_{G} y$.

Proof 2 Suppose that $x \xrightarrow{SF}{}^{P}_{G} y$. Thus, there exists $a, b \in U_{L(P)}$ such that $P?G(a, y)$ succeeds and $P?G(b, y)$ fails. Suppose that $G(a, y)\mathcal{Z}_{max}G(b, y)$. Since $P?G(a, y)$ succeeds, then there exists an SLD-refutation $G_0, \cdots, G_n$ of $G(a, y)$ in P. That is to say, $G_0 = G(a, y)$, $G_n = \square$ and G_i is a successor of G_{i-1} in P for $i = 1 \cdots n$.
Since $G(a, y)\mathcal{Z}_{max}G(b, y)$ in P, thus $P?G(b, y)$ succeeds : a contradiction. Thus, *not* $G(a, y)\mathcal{Z}_{max}G(b, y)$ and $x \xrightarrow{BI}{}^{P}_{G} y$. $\square$

4 Decidability / Complexity

We now study the computational complexity of the following decision problems :

$$\pi_{SF} \begin{cases} \text{Input : A logic program } P\text{, a two variables goal } G(x, y) \\ \text{Output : Determine whether } x \xrightarrow{SF}{}^{P}_{G} y \end{cases}$$

$$\pi_{SA} \begin{cases} \text{Input : A logic program } P\text{, a two variables goal } G(x, y) \\ \text{Output : Determine whether } x \xrightarrow{SA}{}^{P}_{G} y \end{cases}$$

$$\pi_{BI} \begin{cases} \text{Input : A logic program } P\text{, a two variables goal } G(x, y) \\ \text{Output : Determine whether } x \xrightarrow{BI}{}^{P}_{G} y \end{cases}$$

4.1 Undecidability

In the general setting, our decision problems are undecidable.

Proposition 1 The three decision problems above are undecidable.

Proof 3 (π_{SF}) We will reduce the following undecidable decision problem π_1 (Devienne *et al.*, 1996) to π_{SF} :

$$\pi_1 \begin{cases} \text{Input : A logic program } P\text{, a ground goal } q(a) \\ \text{Output : } P?q(a) \text{ succeeds} \end{cases}$$

Let $(P, q(a))$ be an instance of π_1 and let $(P', G(x, y))$ be the instance of π_{SF} defined by : $P' = P \cup \{G(a, y) \leftarrow q(a)\}$, where G is a new predicate symbol of arity 2. We need to show that, $P?q(a)$ succeeds iff $x \xrightarrow{SF}{}^{P'}_{G} y$.
($\Rightarrow$) Suppose that $P?q(a)$ succeeds. Thus $P'?G(a, y)$ succeeds and $P'?G(b, y)$ fails, consequently $x \xrightarrow{SF}{}^{P'}_{G} y$.
($\Leftarrow$) Suppose that $x \xrightarrow{SF}{}^{P'}_{G} y$, then there exists $a', b' \in U_{L(P)}$ such that $P'?G(a', y)$ succeeds and $P'?G(b', y)$ fails. Thus, $a' = a$ and $b' \neq a$. Thus, $P?q(a)$ succeeds.

(π_{SA}) A similar proof applies here.
(π_{BI}) We will reduce the following undecidable decision problem (Devienne *et al.*, 1993) to π_{BI} :

$$\pi_2 \begin{cases} \text{Input : A binary logic program } P\text{, a ground goal } q(a) \\ \text{Output : The SLD-tree of } P?q(a) \text{ contains a failure branch} \end{cases}$$

Let $(P, q(a))$ be an instance of π_2 and let $(P', G(x, y))$ be the instance of π_{BI} defined by :

$$P' = P \cup \begin{cases} G(a, y) \leftarrow q(a) \\ G(b, y) \leftarrow G(b, y) \text{ for all } b \text{ in } L(P) \text{ such that } a \neq b \\ G(f(x_1, \cdots, x_n), y) \leftarrow G(f(x_1, \cdots, x_n), y) \text{ for all } f \text{ in } L(P) \end{cases}$$

Remark that for all $a' \in U_{L(P)}$, the computation tree of $P'?G(a', y)$ consists of a unique infinite branch. We need to show that the SLD-tree of $P?q(a)$ contains a failure branch iff $x \xrightarrow{BI}{}^{P'}_{G} y$.
($\Rightarrow$) Suppose that the SLD-tree of $P?q(a)$ contains a failure branch. Thus the SLD-tree of $P'?G(a, y)$ will eventually contains this failure branch while the SLD-tree of $P'?G(b, y)$ will have infinite branche(s). Consequently $x \xrightarrow{BI}{}^{P'}_{G} y$.
($\Leftarrow$) Suppose that $x \xrightarrow{BI}{}^{P'}_{G} y$, then there exists $a', b' \in U_{L(P)}$ such that *not* $P'?G(a', y)\mathcal{Z}_{max}P'?G(b', y)$. Hence, either a' or b' is equal to a. Thus (in the case of $a' = a$) the SLD-tree of $P?q(a)$ contains a failure branch. □

4.2 Decidability

If one restricts the language to Datalog programs and goals then determining existence of information flows becomes decidable.

Proposition 2 π_{SF} is EXPTIME-complete for Datalog programs.
Proof 4 (*Membership*) The following algorithm decides the existence of the information flow in Datalog programs.

Require: A Datalog program P, a goal $\leftarrow G(x, y)$, finite Herbrand Universe $U_{L(P)} = \{a_1, \cdots, a_n\}$
Ensure: $x \xrightarrow{SF}{}^{P}_{G(x,y)} y$ for the Datalog program P and the goal g

```
answer = false ; i = 0 ;
while i < n and not answer do
    i = i + 1 ; j = i ;
    while j < n and not answer do
        j = j + 1 ;
        if (p?G(a_i, y) succeeds and p?G(a_j, y) fails) or (p?G(a_i, y) fails and
        p?G(a_j, y) succeeds) then
            answer = true ;
        end if
```

```
    end while
end while
return answer ;
```

This algorithm is deterministic and using the fact that Datalog is program complete for EXPTIME (Vardi, 1982; Immerman, 1986), it follows that it can be executed in EXPTIME.
(*Hardness*) In order to prove EXPTIME-hardness, we consider the following decision problem known to be EXPTIME-hard (Vardi, 1982) :

$$\pi_3 \begin{cases} \text{Input : A Datalog program } P\text{, a ground atom } A \\ \text{Output : } P?A \text{ (} A \text{ is a logical consequence of } P\text{)} \end{cases}$$

Let (P, A) an instance of π_3 and let $(P', g(x, y))$ be the instance of π_{SF} defined by $P' = P \cup \{g(a, y) \leftarrow A\}$, where g is a new predicate symbol. Thus $P?A$ iff $x \xrightarrow{SF}{}^{P'}_{g} y$.
($\Rightarrow$) Suppose that A is a logical consequence of P, thus $P'?g(a, y)$ succeeds and $P'?g(b, y)$ fails. Consequently $x \xrightarrow{SF}{}^{P'}_{g} y$.
($\Leftarrow$) Suppose that $x \xrightarrow{SF}{}^{P'}_{g} y$. Then there exists $a', b' \in U_{L(P)}$ such that $P'?g(a', y)$ succeeds and $P'?g(b', y)$ fails. Hence, it follows that $a' = a$ and $b' \neq a$. Thus, $P?A$. □

Proposition 3 π_{SA} is EXPTIME-complete for Datalog programs.
Proof 5 A proof similar to the previous one applies here. □

Concerning π_{SF}, determining existence of flows is even in $\Sigma_2 P$ if one considers binary hierarchical Datalog programs.

Proposition 4 π_{SF} is in $\Sigma_2 P$ for binary hierarchical Datalog programs.
Proof 6 Let us consider the following nondeterministic algorithm with oracle :
Procedure SF(P,$G(x, y)$**)**
Require: A binary hierarchical Datalog program P, a goal $G(x, y)$.
Ensure: $x \xrightarrow{SF}{}^{P}_{G} y$

```
choose a, b in U_{L(P)}
if (P?G(a, y) ∈ SUCCESSES and P?G(b, y) ∈ FAILURES) then
    Accept
else
    Reject
end if
```

The oracle SUCCESSES consists in the set of all pairs (P, G) such that G succeeds in P. Restricting P to binary hierarchical programs, one can show that SUCCESSES belongs to NP. The oracle FAILURES consists in the set of all pairs (P, G) such that G fails in P. Restricting P to binary hierarchical programs, one can show that FAILURES belongs to co-NP. Hence π_{SF} is in $\Sigma_2 P$. □

At the time of writing, we do not know if π_{SA} is in $\Sigma_2 P$ too for binary hierarchical programs. Now, let us address the complexity of deciding the existence of flows with

TABLE 1 – Complexity results

	General setting	Datalog programs	Binary hierarchical Datalog programs
π_{SF}	Undecidable	EXPTIME-complete	in $\Sigma_2 P$
π_{SA}	Undecidable	EXPTIME-complete	in EXPTIME
π_{BI}	Undecidable	?	in EXPTIME

respect to our third definition.

Proposition 5 π_{BI} is in EXPTIME for hierarchical binary Datalog programs.
Proof 7 Since $EXPTIME = APSPACE$, then it suffices to demonstrate that π_{BI} is in $APSPACE$ for binary hierarchical Datalog programs. In this respect, we consider the following alternating algorithm :
Procedure bisim(P,G_1**,**G_2**)**
Require: A hierarchical Datalog program P, two goals G_1 and G_2.
Ensure: Deciding whether $G_1 \mathcal{Z}_{max} G_2$

case $(succ(P, G_1), succ(P, G_2))$
- $(true, true)$ **:**
  (∀) choose $i, j \in \{1, 2\}$ such that $i \neq j$
  (∀) choose a successor G'_i of G_i in P
  (∃) choose a successor G'_j of G_j in P
  (.) call **bisim**(P, G'_i, G'_j)
- $(true, false)$ **:** reject
- $(false, true)$ **:** reject
- $(false, false)$ **:**
  if $(G_1 = \square$ iff $G_2 = \square)$ **then**
    accept
  else
    reject
  end if
endcase

The subprocedure $succ(.,.)$ produces, given a program P and a goal G a Boolean value. More precisely, $succ(P, G)$ is true iff there exists a goal G' such that G' is derived from G and P. Obviously, $succ(.,.)$ can be implemented in deterministic linear time. Concerning the procedure **bisim**, seeing that P is hierarchical, it accepts its inputs P, G_1, G_2 iff $G_1 \mathcal{Z}_{max} G_2$. Moreover, seeing that P is binary, **bisim** can be implemented in polynomial space. □

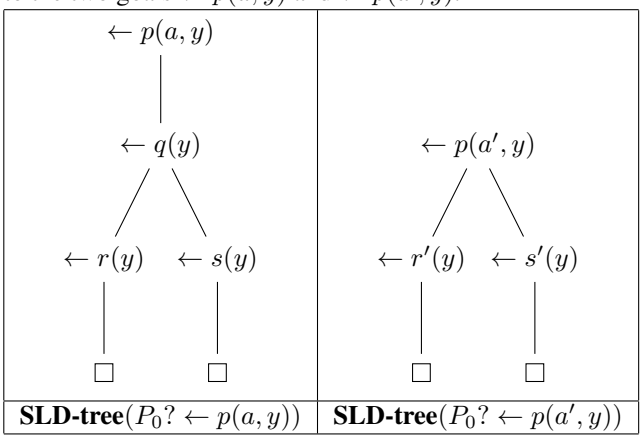
5 Conclusion

In this paper, we have proposed three definitions of information flow in logic programs. As proved in section 4.1, determining whether there exists an information flow is undecidable in the general setting. Hence, a natural question was to restrict the language of logic programming as done in section 4.2. Table 1 contains the results we

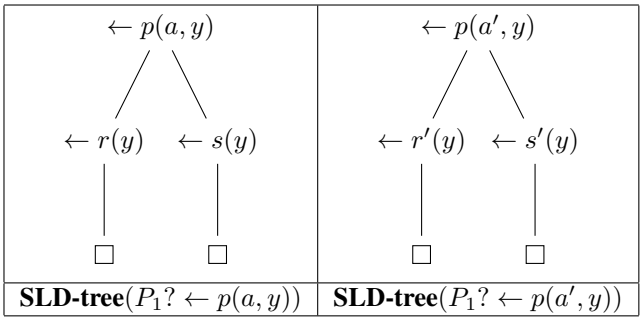
have obtained so far. Much remains to be done. Firstly, in the setting of Datalog programs, the main difficulty concerning π_{BI} comes from loops or infinite branches in SLD-refutation trees. Therefore, in order to determine, given a Datalog program P and two Datalog goals G_1 and G_2, whether $G_1 \mathcal{Z}_{max} G_2$, one can think about using loop checking techniques and considering either restricted programs, or nvi programs or svo programs. See (Bol, 1995) and (Bol *et al.*, 1991) for details. Secondly, considering the unfold/fold transformations introduced by Tamaki and Sato (Tamaki & Sato, 1984) within the context of logic programs optimization, one can ask whether these transformations introduce or eliminate information flows. Obviously, since folding or unfolding clauses in logic programs change neither its successes, nor its failures (Tamaki & Sato, 1984), nor its substitution answers (Kawamura & Kanamori, 1990), the information flows based either on successes and failures or on substitution answers are preserved after applying the transformations of Tamaki and Sato. The same cannot be said for information flows based on bisimulation. For example, let P_0 be the logic program containing the following clauses :

$C_1 : p(a, y) \leftarrow q(y)$	$C_4 : r(y) \leftarrow$	$C_7 : p(a', y) \leftarrow s'(y)$
$C_2 : q(y) \leftarrow r(y)$	$C_5 : s(y) \leftarrow$	$C_8 : r'(y) \leftarrow$
$C_3 : q(y) \leftarrow s(y)$	$C_6 : p(a', y) \leftarrow r'(y)$	$C_9 : s'(y) \leftarrow$

and let G be the goal $\leftarrow p(x, y)$. It is easy to verify that $x \xrightarrow{BI}{}^{P_0}_{G} y$. To see this, we sketch, by omitting the different substitutions, the SLD-refutation trees corresponding to the two goals $\leftarrow p(a, y)$ and $\leftarrow p(a', y)$.



Obviously, as *not* $\leftarrow p(a, y) \mathcal{Z}_{max} \leftarrow p(a', y)$, $x \xrightarrow{BI}{}^{P_0}_{G} y$. By unfolding C_1, the program P_1 is obtained from P_0 by replacing C_1 with the following clauses :
$C_{10} : p(a, y) \leftarrow r(y) \quad C_{11} : p(a, y) \leftarrow s(y)$
In the new transformed program P_1, the two SLD-refutation trees of the goals $\leftarrow p(a, y)$ and $\leftarrow p(a', y)$ are bisimilar as shown in the figure on the next page. Thus $x \not\xrightarrow{BI}{}^{P_1}_{G} y$. Hence, a general question concerns the definition of transformations of logic programs that never introduce or eliminate information flows.



Références

BARAL C. & GELFOND M. (1994). Logic programming and knowledge representation. *Journal of Logic Programming*, **19-20**, 73–148.

BELL D. & LAPADULA L. (1973). Secure computer systems : Mathematical foundations and model. *The MITRE Corporation Bedford MA Technical Report M74244*, **1**(M74-244), 42.

BOL R. (1995). *Loop Checking in Logic Programming*. Amsterdam, the Netherlands : Dissertation of the CWI (Centre for Mathematics and Computer Science).

BOL R., APT K. & KLOP J. (1991). An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, **86**, 35–79.

DENNING D. (1976). A lattice model of secure information flow. *Commun. ACM*, **19**, 236–243.

DENNING D. (1982). *Cryptography and Data Security*. Addison-Wesley.

DENNING D. & DENNING P. (1977). Certification of programs for secure information flow. *Commun. ACM*, **20**, 504–513.

DEVIENNE P., LEBÈGUE P., PARRAIN A., ROUTIER J.-C. & WÜRTZ J. (1996). Smallest Horn clause programs. *The Journal of Logic programming*, **27**, 227–267.

DEVIENNE P., LEBÈGUE P. & ROUTIER J.-C. (1993). Halting problem of one binary Horn clause is undecidable. *STACS 93*, **665**, 48–57.

FENTON J. (1974). Memoryless subsystems. *The Computer Journal*, **17**, 143–147.

FOLEY S. (1989). A model for secure information flow. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, p. 248–258 : IEEE Computer Society.

GOGUEN J. & MESEGUER J. (1982). Security policies and security models. *Security and Privacy, IEEE Symposium on*, **0**, 11.

IMMERMAN N. (1986). Relational queries computable in polynomial time. *Information and Control*, **68**, 86–104.

KAWAMURA T. & KANAMORI T. (1990). Preservation of stronger equivalence in unfold/fold logic program transformation. *Theoretical Computer Science*, **75**, 139–156.

LLOYD J. (1984). *Foundations of logic programming*. Springer-Verlag.

TAMAKI H. & SATO T. (1984). Unfold/fold transformations of logic programs. In *In S.-Å. Tärnlund, editor, Proceedings of The Second International Conference on Logic Programming*, p. 127–139.

VARDI M. Y. (1982). The complexity of relational query languages (extended abstract). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, p. 137–146 : ACM.