# Towards an information flow in logic programming

**Antoun Yaacoub**[1]

[1] **Institut de Recherche en Informatique de Toulouse, CNRS - Université de Toulouse**
**118 route de Narbonne, 31062 Toulouse Cedex 9, France**

**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 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 in security systems is well defined for imperative programming. Different models of information flow were proposed, namely, the Bell-LaPadula Model [2], nonlattice and nontransitive models [10, 4] of information flow, and nondeducibility and noninterference [11]. 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 [6] and runtime mechanisms [9] 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 (unknown) 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 [5] addressed information flow in security systems for 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 $P$.

Firstly, we consider that the user only sees whether her queries succeed or fail. In this respect, we say information flows from $x$ to $y$ in $G$ when there exists constants $a$, $b$ such that $\leftarrow G(a,y)$ succeeds whereas $\leftarrow G(b,y)$ fails. Secondly, we assume that the user has also access to the sets of substitution answers computed by the interpreter with respect to her queries. As a result, in this case, there is a flow of information from $x$ to $y$ in $G$ if there are constants $a$, $b$ such that the substitution answers of $\leftarrow G(a,y)$ and $\leftarrow G(b,y)$ are different. Thirdly, we suppose that the user, in addition to the substitution answers, also observes the SLD-refutation trees produced by the interpreter. If the SLD-trees of the queries $\leftarrow G(a,y)$ and $\leftarrow G(b,y)$ can be distinguished in one way or another by the user, then we will say that information flows from $x$ to $y$ in $G$. Of course, it remains to properly define what "distinguished" means in our setting. Following a traditional view in program semantics, we will base distinguishability of SLD-refutation trees on the notion of bisimilarity.

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 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 [14, 1] for more details. In the remainder of this article, we will use $p, q, \ldots$ for predicate symbols, $x, y, z, \ldots$ for variables, $f, g, h, \ldots$ for function symbols, and $a, b, c, \ldots$ 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 $f(t_1,\ldots,t_n)$ 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 $p(t_1,\ldots,t_n)$, 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 $A \leftarrow B_1,\ldots,B_n$ where $A, B_1, \ldots, B_n$ are atoms. $A$ is called the head of the clause and $B_1,\ldots,B_n$ is called its body. A goal is an expression of the form $\leftarrow B_1,\ldots,B_n$. 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 □. 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 $\varepsilon$. A substitution $\sigma_1$ is said to be more general than a substitution $\sigma_2$ if there is a substitution $\theta$ such that $\sigma_2 = \theta\sigma_1$. Two terms $t_1$ and $t_2$ are said to be *unifiable* if there exists a substitution $\sigma$ such that $\sigma(t1) = \sigma(t2)$, in this case $\sigma$ 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.

The operational behavior of logic programs can be described by means of SLD-derivations. An SLD-derivation for a goal $G=\leftarrow A_1,\ldots,A_n$ with respect to a program $P$ is a sequence of goals $G_0,\ldots,G_i,G_{i+1},\ldots$ , such that $G_0=G$, and if $G_i = B_1,\ldots,B_m$, then $G_{i+1} = \theta B_1,\ldots,\theta B_{i-1}, \theta B'_1,\ldots,\theta B'_k,\theta B_{i+1},\ldots,\theta B_m$ such that $1 \leq i \leq m$, $B \leftarrow B'_1,\ldots,B'_k$ is a variant of a clause in $P$ that has no variable in common with any of the goals $G_0,\ldots,G_i$, 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 $G_0,\ldots,G_n$ be an SLD-derivation for a goal $G$ with respect to a program $P$, and let $\theta_i$ 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 $\theta$, where $\theta$ is the substitution obtained by restricting the substitution $\theta_n\ldots\theta_1$ 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 $P$.

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 $A(t_1,\ldots,t_n) \leftarrow B$, the level of every predicate symbol in $B$ is less than the level of $A$.
- Restricted programs, i.e. logic programs such that, in every program statement $A_0 \leftarrow A_1,\ldots,A_k$, only $A_k$ can depend on $A_0$.
- Nonvariable introducing programs (in short nvi) i.e. logic programs such that, in every program statement $A \leftarrow B_1,\ldots,B_n$, if a variable appears in $B_1,\ldots,B_n$, it must also appears in $A$.
- Single variable occurrence (in short svo) i.e. logic programs such that, in the body of every program statement, no variable occurs more than once.

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.

## 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 asks 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.

It is now time to present our three definitions of information flows in logic programs.

### 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 $x \xrightarrow[G]{SF}{}^{P} y$) iff there exists $a, b \in U_{L(P)}$ such that $P?G(a,y)$ succeeds and $P?G(b,y)$ fails.

This intuitively means that when the user only sees the outputs of computations in terms of successes and failures, there exists two different $a, b \in U_{L(P)}$ such that this user can distinguish (without seeing what concerns $a$, $b$) between the output for $P?G(a,y)$ and the output for $P?G(b,y)$.

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 $x \xrightarrow[G_1]{SF}{}^{P_1} y$.

### 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 $x \xrightarrow[G]{SA}{}^{P} y$) iff there exists $a, b \in U_{L(P)}$ such that $\Theta(P?G(a,y)) \neq \Theta(P?G(b,y))$.

Roughly speaking, in this definition, the user only sees the outputs of computations in terms of substitution answers. As a result, there is a flow if this user can distinguish (without seeing what is about $a$, $b$) the output of $P?G(a,y)$ and the output of $P?G(b,y)$.

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 $\Theta(P_2?G_2(a,y)) = \{\varepsilon\}$ and $\Theta(P_2?G_2(b,y)) = \varnothing$, then $x \xrightarrow[G_2]{SA}{}^{P_2} y$.

### 3.3 Bisimulation

Our third definition of flow is based on the notion of bisimulation between goals. Let $P$ be a logic program and $Z$ be a binary relation between goals. We shall say that $Z$ is a *P-bisimulation* iff for all goals $G$, $H$, if $GZH$ then:

- for all goals $G' \in succ_P(G)$, there exists $H' \in succ_P(H)$, such that $G'ZH'$.
- for all goals $H' \in succ_P(H)$, there exists $G' \in succ_P(G)$, such that $G'ZH'$.
- $G = \square$ 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 $P$.

**Lemma 1** *The relation identity Id between goals is a P-bisimulation.*

**Lemma 2** *If Z is a P-bisimulation, then $Z^{-1}$ is also a P-bisimulation.*

**Lemma 3** *If $Z_1$, $Z_2$ are two P-bisimulations, then the composition $Z_1Z_2$ defined by $Z_1Z_2 = \{(G,H)/\exists I, GZ_1I$ and $IZ_2H\}$ is also a P- bisimulation.*

*Proof.* Suppose that $(G,H) \in Z_1Z_2$.
Then there exists $I$ such that $GZ_1I$ and $IZ_2H$. Let $G_0$ be a successor of $G$.
Since $GZ_1I$, then there exists $I'$ successor of $I$ such that $G'Z_1I'$.
Since $IZ_2H$, then, there exists $H'$ successor of $H$ such that $I'Z_2H'$. Thus, $(G',H') \in Z_1Z_2$.
With a similar reasoning, if $H'$ is a successor of $H$, we can find a $G'$ successor of $G$ such that $(G',H') \in Z_1Z_2$.□

**Lemma 4** *Let $(Z_i)_{i \in I}$ be a family of P-bisimulations, then $\bigcup_{i \in I} Z_i$ is also a P-bisimulation .*

By lemma 4, there exists a maximal *P-bisimulation*, denoted $Z_{max}$.

*Example 3*. Let *P* be the following program:
*p(a,y)←q(y)*
*p(b,y) ←r(y)*
*p(b,y) ←s(y)*
and let *G, H* be respectively the following goals *←p(a,y)* and *←p(b,y)*
Let *Z* be the binary relation between goals such that:
*←p(a,y) Z ←p(b,y)*
*←q(y) Z ←r(y)*
*←q(y) Z ←s(y)*
Obviously, *Z* is a *P-bisimulation*. Since *G Z H*, then $G\ Z_{max}\ H$.

**Lemma 5** *$Z_{max}$ is an equivalence relation.*
*Proof*. By lemmas 1, 2 and 3.□

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 $x \overset{BI}{\longrightarrow}_{G} y$ ) iff there exists *a, b* $\in U_{L(P)}$ such that *not[G(a,y)$Z_{max}$G(b,y)]*.

In this definition, there is a flow if the user, by only seeing the outputs of computations in terms of bisimulation between goals, can distinguish (without looking at *a, b*) the output of *P?G(a,y)* and the output of *P?G(b,y)*.

*Example 4*. Let $P_3$ be the following program:
*p(x,a)←*
*p(a,b) ←q(a)*
and let $G_3(x,y)$ be the goal: *←p(x,y)*.
Let us prove *not[←p(a,y)$Z_{max}$←p(b,y)]*.
Suppose that *←p(a,y)$Z_{max}$←p(b,y)*.
Since *←q(a)*$\in succ_{P3}$*(←p(a,y))*, then there should be $G_3 \in succ_{P3}$*(←p(b,y))* such that *←q(a)*$Z_{max}G_3$.
The problem is that the only goal in $succ_{P3}$ *(←p(b,y))* is the empty goal, which cannot be bisimilar to *←q(a)*.
Hence, *not[←p(a,y)$Z_{max}$←p(b,y)]*.
Therefore $x \overset{BI}{\longrightarrow}_{G_3} y$.

## 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)←*
*p(Z,c)←*
and let *G(x,y)* be the goal: *←p(x,y)*. Since *Θ(P?G(a,y))={y/b,y/c}* and *Θ(P?G(b,y))={y/c}*, then $x \overset{SA}{\longrightarrow}_{G} y$. Since *P?G(a,y)* and *P?G(b,y)* both succeed, then $x \overset{SF}{\nrightarrow}_{G} y$. 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 $x \overset{SF}{\longrightarrow}_{G} y$ then $x \overset{SA}{\longrightarrow}_{G} y$.
*Proof.* Suppose that $x \overset{SF}{\longrightarrow}_{G} y$, then there exists *a, b* $\in U_{L(P)}$ such that *P?G(a,y)* succeeds and *P?G(b,y)* fails. Therefore, *Θ(P?G(a,y))*$\neq\varnothing$, and *Θ(P?G(b,y))*$=\varnothing$. Consequently, $x \overset{SA}{\longrightarrow}_{G} y$.□

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 $x \overset{BI}{\longrightarrow}_{G_3} y$ . Nevertheless, since all the goals of the form $G_3(a,y)$, with *a*$\in U_{L(P)}$ succeed, thus $x \overset{SF}{\nrightarrow}_{G_3} y$.
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 $x \overset{SF}{\longrightarrow}_{G} y$ then $x \overset{BI}{\longrightarrow}_{G} y$.
*Proof.* Suppose that $x \overset{SF}{\longrightarrow}_{G} y$. 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)*$Z_{max}$*G(b,y)*. Since *P?G(a,y)* succeeds, then there exists an SLD-refutation $G_0,\ldots,G_n$ of *G(a,y)* in *P*. That is to say, $G_0=G(a,y)$, $G_n=$□ and $G_i$ is a successor of $G_{i-1}$ in *P* for *i=1,…,n*. Since *G(a,y)*$Z_{max}$*G(b,y)* in *P*, thus *P?G(b,y)* succeeds: a contradiction. Thus, *not[G(a,y)*$Z_{max}$*G(b,y)]* and $x \overset{BI}{\longrightarrow}_{G} y$.

## 3.5 Information flow definitions over goals with arity > 2

We now generalize the previous definitions by considering goals with arity higher than two. Firstly, we consider information flows between two variables. Secondly, we consider information flows between two sets of variables.
The generalization of the previous three definitions to goals with arity higher than two and by only considering information flows between two variables leads us to the following three definitions:

Definition 1 For a logic program *P* and a goal $G(x_1,\ldots,x_k,\ldots,x_m,\ldots,x_p)$ of arity *p*
$x \overset{SF}{\longrightarrow}_{P\ G(x_1,\ldots,x_k,\ldots,x_m,\ldots,x_p)} y$ iff
$\exists\, a,a' \in U_{L(P)},\ a \neq a'$

$\exists c_1,\dots,c_{k-1},c_{k+1},\dots,c_{m-1},c_{m+1},\dots,c_p \in U_{L(P)}$ such that
$P?G(c_1,\dots,c_{k-1},a,c_{k+1},\dots,c_{m-1},x_m,c_{m+1},\dots,c_p)$ succeeds and
$P?G(c_1,\dots,c_{k-1},a',c_{k+1},\dots,c_{m-1},x_m,c_{m+1},\dots,c_p)$ fails.

Definition 2 For a logic program $P$ and a goal $G(x_1,\dots,x_k,\dots,x_m,\dots,x_p)$ of arity $p$
$x \xrightarrow{SA} \overset{P}{G(x_1,\dots,x_k,\dots,x_m,\dots,x_p)} y$ iff
$\exists a,a' \in U_{L(P)}$, $a \neq a'$
$\exists c_1,\dots,c_{k-1},c_{k+1},\dots,c_{m-1},c_{m+1},\dots,c_p \in U_{L(P)}$ such that
$\Theta[P?G(c_1,\dots,c_{k-1},a,c_{k+1},\dots,c_{m-1},x_m,c_{m+1},\dots,c_p)] \neq$
$\Theta[P?G(c_1,\dots,c_{k-1},a',c_{k+1},\dots,c_{m-1},x_m,c_{m+1},\dots,c_p)]$.

Definition 3 For a logic program $P$ and a goal $G(x_1,\dots,x_k,\dots,x_m,\dots,x_p)$ of arity $p$
$x \xrightarrow{BI} \overset{P}{G(x_1,\dots,x_k,\dots,x_m,\dots,x_p)} y$ iff
$\exists a,a' \in U_{L(P)}$, $a \neq a'$
$\exists c_1,\dots,c_{k-1},c_{k+1},\dots,c_{m-1},c_{m+1},\dots,c_p \in U_{L(P)}$ such that
$not[P?G(c_1,\dots,c_{k-1},a,c_{k+1},\dots,c_{m-1},x_m,c_{m+1},\dots,c_p)\ Z_{max}$
$P?G(c_1,\dots,c_{k-1},a',c_{k+1},\dots,c_{m-1},x_m,c_{m+1},\dots,c_p)]$.

The idea behind the previous definitions is that in order to see if there is a flow from $x_k$ to $x_m$, one can try to instantiate the $p$-2 other variables to some constants and to find two constants $a$, $a'$ for which the instantiations of the variable $x_k$ by $a$ or $a'$ leads to a success and failure for the first definition, or different substitutions answers for the second definition or two different shapes of resolution trees for the third definition.

By considering $x_1 = x$, $x_m = y$ and $p = 2$, we find again the same information flow definitions for goals of arity two. In addition, with this generalization, the results of lemma 6 and 7 are also preserved.

Now we will generalize the previous notions to cover information flows from a set of variables to a set of variables. For this, we will proceed in 2 steps (Due to space limitation, we will consider only in this subsection the information flow based on success and failure).

**1. Information flow from a set of variables to a single variable**

For a program P and a goal $G(\boldsymbol{x_1,\dots,x_k},\underline{x_l},x_m,\dots,x_n)$, we say that there is a flow from $\{x_1,\dots,x_k\}$ to $x_l$ iff one can instantiate the variables $\{x_m,\dots,x_n\}$ by some constants and instantiate the variables $\{x_1,\dots,x_k\}$ in two different manners and thus lead to a success and failure.
$\{x_1,\dots,x_k\} \xrightarrow{SF} \overset{P}{G(x_1,\dots,x_k,x_l,x_m,\dots,x_n)} y$ iff
$\exists c_m,\dots,c_n \in U_{L(P)}, \exists a_1,\dots,a_k, \exists a'_1,\dots,a'_k \in U_{L(P)}$ such that
$(a_1,\dots,a_k) \neq (a'_1,\dots,a'_k)$ and
$P?G(a_1,\dots,a_k,x_l,c_m,\dots,c_n)$ succeeds
and
$P?G(a'_1,\dots,a'_k,x_l,c_m,\dots,c_n)$ fails.

**2. Generalization of the previous definition of information flow from a set of variables to a set of variables**

For a program P and a goal $G(\boldsymbol{x_1,\dots,x_k},\underline{x_l,\dots x_{m-1}},x_m,\dots,x_n)$ we say that there is a flow from $\{x_1,\dots,x_k\}$ to $\{x_l,\dots,x_{m-1}\}$ iff there is a flow from $\{x_1,\dots,x_k\}$ to every variable in $\{x_l,\dots,x_{m-1}\}$.
$\{x_1,\dots,x_k\} \xrightarrow{SF} \overset{P}{G(x_1,\dots,x_n)} \{x_l,\dots,x_{m-1}\}$ iff
$\forall j = 1,\dots,m-1, \{x_1,\dots,x_k\} \xrightarrow{SF} \overset{P}{G(x_1,\dots,x_n)} x_j$

A similar remark applies here too, by considering $k = 1$, and $l=m-1$, we find again the same definitions of information flows between single variables.

## 3.6 Non-transitivity of information flow in logic programs

Most of the policies of information flow in imperative programming are represented by a lattice structure, which means that if information flows from a variable $x$ to a variable $y$ and from $y$ to $z$, then there is a flow from $x$ to $z$. In such contexts, the information flow relation between program variables is transitive. It is interesting to investigate this property on the information flow of logic programs according to our definitions.

Several counter examples prove that the information flow relation according to our definitions is not transitive.

**Non transitivity of the information flow for the first definition based on success and failure**

For the following program $P_4$:
$p(a,a,a) \leftarrow$
$p(x,a,b) \leftarrow$
and the goal $G(x,y,z) : \leftarrow p(x,y,z)$, we have:
$x \xrightarrow{SF} \overset{P_4}{G(x,y,z)} y$ ($P_4?G(a,y,a)$ succeeds, $P_4?G(b,y,a)$ fails),
$y \xrightarrow{SF} \overset{P_4}{G(x,y,z)} z$ ($P_4?G(a,a,z)$ succeeds, $P_4?G(a,b,z)$ fails),
but we have not $x \xrightarrow{SF} \overset{P_4}{G(x,y,z)} z$ (both $P_4?G(a,a,z)$ and $P_4?G(b,a,z)$ succeed).

**Non transitivity of the information flow for the second definition based on substitutions answers**

For the following program:
$p(x,y,z) \leftarrow q(x,y), r(y,z)$
$q(x,y) \leftarrow$
$q(a,c) \leftarrow$
$r(y,c) \leftarrow$
$r(c,d) \leftarrow$
and the goal $G(x,y,z) : \leftarrow p(x,y,z)$, we have:
$x \xrightarrow{SA} \overset{P}{G(x,y,z)} y$ ($\Theta(P?G(a,y,c)) = \{y=c,\varepsilon\}$, $\Theta(P?G(b,y,c)) = \{\varepsilon\}$),

$y \xrightarrow[G(x,y,z)]{SA\ P} z$ ($\Theta(P?G(a,c,z))= \{z/c,z/d\}$, $\Theta(P?G(a,b,z)) =\{z/c\}$),
but we have not $x \xrightarrow[G(x,y,z)]{SA\ P} z$ ($\Theta(P?G(a,c,z))=\{z/c,z/d\}$, $\Theta(P?G(b,c,z)) =\{z/c,z/d\}$).

**Non transitivity of the information flow for the third definition based on bisimulation between goals**
For the same previous program $P_4$ and the goal $G(x,y,z)$ : $\leftarrow p(x,y,z)$, we have:
$x \xrightarrow[G(x,y,z)]{BI\ P_4} y$ ($not\ tree(P_4?G(a,y,a))\ Z_{max}\ tree(P_4?G(b,y,a))$),
$y \xrightarrow[G(x,y,z)]{BI\ P_4} z$ ($not\ tree(P_4?G(a,a,z))\ Z_{max}\ tree(P_4?G(a,b,z))$),
but we have not $x \xrightarrow[G(x,y,z)]{BI\ P_4} z$ ($tree(P_4?G(a,a,z))\ Z_{max}\ tree(P_4?G(b,a,z))$).

This non-transitivity of our information flow relation can be explained by the particular role of variables in logic programming. The truth is that in imperative programs, the basic instruction is the assignment operation, whereas in logic programs, the basic instructions are the resolution rule and the unification.

## 4. Decidability / Complexity

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

$\pi_{SF}$ { Input: A logic program $P$, a two variables goal $G(x,y)$
Output: Determine whether $x \xrightarrow[G]{SF\ P} y$

$\pi_{SA}$ { Input: A logic program $P$, a two variables goal $G(x,y)$
Output: Determine whether $x \xrightarrow[G]{SA\ P} y$

$\pi_{BI}$ { Input: A logic program $P$, a two variables goal $G(x,y)$
Output: Determine whether $x \xrightarrow[G]{BI\ P} y$

### 4.1 Undecidability

In the general setting, our decision problems are undecidable.
**Proposition 1**. The three decision problems above are undecidable.
*Proof.* ($\pi_{SF}$) We will reduce the following undecidable decision problem $\pi_1$ [7] to $\pi_{SF}$ :

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

Let $(P,q(a))$ be an instance of $\pi_1$ and let $(P',G(x,y))$ be the instance of $\pi_{SF}$ defined by: $P' = P \cup \{G(a,y) \leftarrow q(a)\}$, where $G$ is a new predicate symbol of arity 2 and $a$ is a new constant. We need to show that, $P?q(a)$ succeeds iff $x \xrightarrow[G]{SF\ P'} y$.

($\Leftarrow$) Suppose that $P?q(a)$ succeeds. Thus $P'?G(a,y)$ succeeds and $P'?G(b,y)$ fails, consequently $x \xrightarrow[G]{SF\ P'} y$.
($\Rightarrow$) Suppose that $x \xrightarrow[G]{SF\ P'} y$, 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.

($\pi_{SA}$) A similar proof applies here.

($\pi_{BI}$) We will reduce the following undecidable decision problem [8] to $\pi_{BI}$:

$\pi_2$ { Input: A binary logic program $P$, a ground goal $q(a)$
Output: The SLD − tree of $P?\,q(a)$ contains a failure branch

Let $(P,q(a))$ be an instance of $\pi_2$ and let $(P',G(x,y))$ be the instance of $\pi_{BI}$ defined by:
$P' = P \cup \{$
$G(a,y) \leftarrow q(a);$
$G(b,y) \leftarrow G(b,y)$ for all $b$ in $L(P)$ such that $a \neq b$,
$G(f(x_1,\ldots,x_n),y) \leftarrow G(f(x_1,\ldots,x_n),y)$ for all $f$ in $L(P)\}$
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 $x \xrightarrow[G]{BI\ P'} y$.
($\Rightarrow$) 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 $x \xrightarrow[G]{BI\ P'} y$.
($\Leftarrow$) Suppose that $x \xrightarrow[G]{BI\ P'} y$, then there exists $a',b' \in U_{L(P)}$ such that $not[P'?G(a',y)Z_{max}P'?G(b',y)]$. 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**. $\pi_{SF}$ is EXPTIME-complete for Datalog programs.
*Proof.* (*Membership*) The following algorithm decides the existence of the information flow in Datalog programs.
**Require:** A Datalog program $P$, a goal $G(x,y)$, finite Herbrand Universe $U_{L(P)} =\{a_1,\ldots,a_n\}$
**Ensure:** $x \xrightarrow[G(x,y)]{SF\ P} y$ for the Datalog program $P$ and the goal $g$

```
1: answer = false
2: i = 0
3: while i < n and not answer do
4:     i = i + 1; j = i
5:     while j < n and not answer do
6:         j = j + 1
```

```
7:          if (P?G(ai,y) succeeds and
            P?G(aj ,y) fails)
            or (P?G(ai,y) fails and
            P?G(aj ,y) succeeds) then
8:                answer = true
9:          end if
10:   end while
11: end while
12: return answer
```

This algorithm is deterministic and using the fact that Datalog is program complete for EXPTIME [16, 12], 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 [16]:

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

Let $(P,A)$ an instance of $\pi_3$ and let $(P',g(x,y))$ be the instance of $\pi_{SF}$ 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**. $\pi_{SA}$ is is EXPTIME-complete for Datalog programs.
*Proof.* A proof similar to the previous one applies here.□

Concerning $\pi_{SF}$, determining existence of flows is even in $\Delta_2$P if one considers binary hierarchical Datalog programs.

**Proposition 4**. $\pi_{SF}$ is in $\Delta_2 P$ for binary hierarchical Datalog programs.
*Proof.* Let us consider the following deterministic algorithm with oracle:
**Procedure SF(P,G(x,y))**
**Require:** A binary hierarchical Datalog program $P$, a goal $G(x,y)$.
**Ensure:** $x \xrightarrow{SF\ P}_{G} y$

```
1: For all a in U_L(P) do
2: For all b in U_L(P) do
3: if (P?G(a,y) ∈ SUCCESSES and
   P?G(b,y) ∈ FAILURES) then
4:    Accept
5: else
6:    Reject
7: 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 $\pi_{SF}$ is in $\Delta_2$P.□

At the time of writing, we do not know if $\pi_{SA}$ is in $\Delta_2$P too for binary hierarchical programs.
Now, let us address the complexity of deciding the existence of flows with respect to our third definition.

**Proposition 5**. $\pi_{BI}$ is in EXPTIME for hierarchical binary Datalog programs.
*Proof.* Since EXPTIME = APSPACE, then it suffices to demonstrate that $\pi_{BI}$ 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 $G_1 Z_{max} G_2$

```
1: case (succ(P,G1),succ(P,G2))
2:    - (true,true):
3:          (∀) choose i,j∈{1,2}
                  such that i≠j
4:          (∀) choose a successor G'i
                  of Gi in P
5:          (∃) choose a successor G'j
                  of Gj in P
6:          (.) call bisim(P,G'I,G'j)
7:    - (true,false): reject
8:    - (false,true): reject
9:    - (false,false):
10:         if (G1 = □ iff G2 = □) then
11:               accept
12:         else
13:               reject
14:         end if
15: endcase
```

The subprocedure *succ(.,.)* produces, given an hierarchical Datalog 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 $G_1 Z_{max} G_2$. Moreover, seeing that $P$ is binary, *bisim* can be implemented in polynomial space.

We mention that the different algorithms previously presented for goals of arity two can be generalized easily to goals with arity higher than two.

## 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 $\pi_{BI}$ 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 $G_1 Z_{max} G_2$, one can think about using loop checking techniques and considering either *restricted* programs, or *nvi* programs or *svo* programs. See Bol et al [3] for details.

Table 1: Margin Complexity results

| | General setting | Datalog programs | Binary hierarchical Datalog programs |
|---|---|---|---|
| $\pi_{SF}$ | Undecidable | EXPTIME-complete | In $\Delta_2$P |
| $\pi_{SA}$ | Undecidable | EXPTIME-complete | in EXPTIME |
| $\pi_{BI}$ | Undecidable | ? | in EXPTIME |

Secondly, considering the unfold/fold transformations introduced by Tamaki and Sato [15] 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 [15], nor its substitution answers [13], 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:
*C1: p(a,y)←q(y)*
*C2: q(y) ←r(y)*
*C3: q(y) ←s(y)*
*C4: r(y) ←*
*C5: s(y) ←*
*C6: p(a',y) ← r'(y)*
*C7: p(a',y) ← s'(y)*
*C8: r'(y) ←*
*C9: s'(y) ←*
and let $G$ be the goal *←p(x,y)*.

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

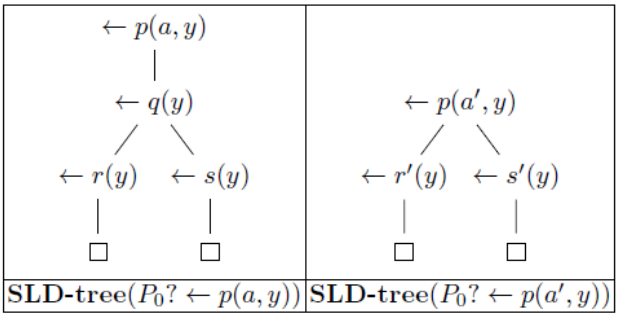


Fig. 1 SLD refutation trees to the goals $P_0$*?←p(a,y)* and $P_0$*?←p(a',y)*.

Obviously, as *not [←p(a,y)*$Z_{max}$*←p(a',y)]*, $x \overset{BI}{\longrightarrow}_{G}^{P_0} y$.
By unfolding *C1*, the program $P_1$ is obtained from $P_0$ by replacing *C1* with the following clauses:
*C10: p(a,y) ←r(y)*
*C11: p(a,y) ← s(y)*

In the new transformed program $P_1$, the two SLD-refutation trees of the goals *←p(a,y)* and *←p(a',y)* are bisimilar as shown in the next figure.

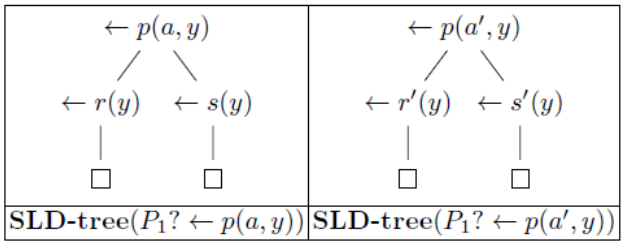


Fig. 2 SLD refutation trees to the goals $P_1$*?←p(a,y)* and $P_1$*?←p(a',y)*.

Thus $x \overset{BI}{\nrightarrow}_{G}^{P_0} y$.

A general question concerns the definition of transformations of logic programs that never introduce or eliminate information flows.

**Acknowledgments**

We would like to thank Philippe Balbiani and Ali Awada for valuable discussions regarding information flow and complexity.

## References

[1] C. Baral and M. Gelfond. Logic programming and knowledge representation. Journal of Logic Programming, 19-20:73-148, 1994.
[2] D. Bell, and L. LaPadula. Secure computer systems: Mathematical foundations and model. The MITRE Corporation Bedford MA Technical Report M74244, 1(M74-244):42, 1973.
[3] R. Bol, K. Apt, and J. Klop. An analysis of loop checking mechanisms for logic programs. Theoretical Computer Science, 86:35-79, 1991.
[4] D. Denning. A lattice model of secure information ow. Commun. ACM, 19:236-243, 1976.
[5] D. Denning. Cryptography and Data Security. Addison-Wesley, 1982.
[6] D. Denning and P. Denning. Certification of programs for secure information flow. Commun. ACM, 20:504-513, 1977.
[7] P. Devienne, P. Lebègue, A. Parrain, J.-C. Routier, and J. Wrtz. Smallest Horn clause programs. The Journal of Logic programming, 27:227-267, 1996.
[8] P. Devienne, P. Lebègue, and J.-C. Routier. Halting problem of one binary Horn clause is undecidable. STACS 93, 665:48-57, 1993.
[9] J. Fenton. Memoryless subsystems. The Computer Journal, 17:143-147, 1974.
[10] S. Foley. A model for secure information flow. In Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on, pages 248-258. IEEE Computer Society, 1989.
[11] J. Goguen and J. Meseguer. Security policies and security models. Security and Privacy, IEEE Symposium on, 0:11, 1982.
[12] N. Immerman. Relational queries computable in polynomial time. Information and Control, 68:86-104, 1986.
[13] T. Kawamura and T. Kanamori. Preservation of stronger equivalence in unfold/fold logic program transformation. Theoretical Computer Science, 75:139-156, 1990.
[14] J. Lloyd. Foundations of logic programming. Springer-Verlag, 1984.
[15] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In In S.-. Trnlund, editor, Proceedings of The Second International Conference on Logic Programming, pages 127-139, 1984.
[16] M. Y. Vardi. The complexity of relational query languages (extended abstract). In Proceedings of the fourteenth annual ACM symposium on Theory of computing, STOC '82, pages 137-146. ACM, 1982.

**Antoun Yaacoub** is currently a Ph.D. student in computer science in Université Paul Sabatier at Toulouse – France. He's conducting his research at the Institut de recherche informatique de Toulouse (IRIT) – France. His research focuses on defining, identifying and analyzing the flow of information (from a security point of view) in logic programs. He previously worked on various aspects of the French language and focused on the types of links existing between the words.