Loop Checks for Logic Programs with Functions

Yi-Dong Shen [a,1] Li-Yan Yuan [b,2] Jia-Huai You [b,3]

[a] *Department of Computer Science, Chongqing University, Chongqing 400044, P.R.China*

[b] *Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1*

Abstract

Two complete loop checking mechanisms have been presented in the literature for logic programs with functions: OS-check and EVA-check. OS-check is computationally efficient but quite unreliable in that it often mis-identifies infinite loops, whereas EVA-check is reliable for a majority of cases but quite expensive. In this paper, we develop a series of new complete loop checking mechanisms, called VAF-checks. The key technique we introduce is the notion of expanded variants, which captures a key structural characteristic of infinite loops. We show that our approach is superior to both OS-check and EVA-check in that it is as efficient as OS-check and as reliable as EVA-check.

Keywords: Logic programming, loop checking.

1 Introduction

The recursive nature of logic programs leads to possibilities of running into infinite loops with top-down query evaluation. By an infinite loop we refer to any infinite SLD-derivation. An illustrative example is the evaluation of the goal $\leftarrow p(a)$ against the logic program

$$P_1: \quad p(X) \leftarrow p(X) \qquad C_{11}$$

[1] Corresponding author. Email: ydshen@cs.ualberta.ca. Work performed during a visit to Department of Computing Science, University of Alberta, Canada.

[2] Email: yuan@cs.ualberta.ca.

[3] Email: you@cs.ualberta.ca.

which leads to the infinite loop

$$\leftarrow p(a) \Rightarrow_{C_{11}} \leftarrow p(a) \Rightarrow ... \Rightarrow_{C_{11}} \leftarrow p(a) \Rightarrow ... \qquad L_1$$

Another very representative logic program is

$$P_2: \quad p(X) \leftarrow p(f(X)) \qquad C_{21}$$

against which evaluating the query $\leftarrow p(g(a))$ generates the infinite loop

$$\leftarrow p(g(a)) \Rightarrow_{C_{21}} \leftarrow p(f(g(a))) \Rightarrow ... \Rightarrow_{C_{21}} \leftarrow p(f(f(...f(g(a))...))) \Rightarrow ... \qquad L_2$$

Loop checking is a long recognized problem in logic programming. [4] Although many loop checking mechanisms have been proposed during the last decade (e.g. [1,2,5,7,8,10,15,17,20,23,24,26,27]), a majority of them are suitable only for function-free logic programs because they determine infinite loops by checking if there are *variant* goals/subgoals in an SLD-derivation. Variant goals/subgoals are the same goals/subgoals up to variable renaming. Hence, an infinite loop like L_2 cannot be detected because no variant goals/subgoals occur in the derivation.

An important fact is that for function-free logic programs, infinite loops can be completely avoided by appealing to tabling techniques [4,6,21,22,25,27,28]. However, infinite loops with functions remain unresolved even in tabling systems [16].

To our best knowledge, among all existing loop checking mechanisms only two can deal with infinite loops like L_2. One is called *OS-check* (for OverSize loop check) [17] and the other *EVA-check* (for Extended Variant Atoms loop check) [20]. (Bruynooghe, De schreye and Martens [5,13,14] presents a framework for partial deduction with finite unfolding that, when applied to loop checking, is very similar to OS-check.)

OS-check, first introduced by Sahlin [17,18] and further formalized by Bol [3], determines infinite loops based on two parameters: a depth bound d and a size function *size*. Informally, OS-check says that an SLD-derivation may go into an infinite loop if it generates an OverSized subgoal. A subgoal A is said to be *OverSized* if it has d ancestor subgoals in the SLD-derivation that have the same predicate symbol as A and whose size is smaller than or equal to A. For example, if we choose $d = 1$, then $p(f(g(a)))$ in L_2 is OverSized, so L_2 is an infinite loop.

[4] There are two different topics on termination of logic programs. One is *termination analysis* (see [9] for a detailed survey), and the other is *loop checking* (see [1,27]). In this paper, we study loop checking.

It is proved that OS-check is *complete* in the sense that it cuts all infinite loops. However, because it merely takes the number of repeated predicate symbols and the size of subgoals as its decision parameters, without referring to the informative internal structure of the subgoals, the underlying decision is fairly unreliable; i.e. many non-loop derivations may be pruned unless the depth bound d is set sufficiently large.

EVA-check, proposed by Shen [20], determines infinite loops based on a depth bound d and generalized variants. Informally, EVA-check says that an SLD-derivation may go into an infinite loop if it generates a subgoal A' that is a generalized variant of all its d ancestor subgoals. A subgoal A' is said to be a *generalized variant* of a subgoal A if it is the same as A up to variable renaming except for some arguments whose size increases from A via a set of recursive clauses. Recursive clauses are of the form like C_{21} in P_2, one distinct property of which is that repeatedly applying them may lead to recursive increase in size of some subgoals.

Recursive increase in term size is a key feature of infinite loops with functions. That is, any infinite loops with infinitely large subgoals are generated by repeatedly applying a set of recursive clauses. Due to this fact, EVA-check is complete and much *more reliable* than OS-check in the sense that it is less likely to mis-identify infinite loops [20].

OS-check has the obvious advantage of simplicity, but it is unreliable. In contrast, EVA-check is reliable in a majority of cases, but it is computationally expensive. The main cost of EVA-check comes from the computation of recursive clauses. On the one hand, given a logic program we need to determine which clauses in it are recursive clauses. On the other hand, for any subgoals A and A' in an SLD-derivation, in order to determine if A' is a generalized variant of A, we need to check if A' is derived from A by applying some set of recursive clauses. Our observation shows that both processes are time-consuming.

In this paper, we continue to explore complete loop checking mechanisms, which have proved quite useful as stopping criteria for partial deduction in logic programming [12] (see [3] for the relation between stopping criteria for partial deduction and loop checking). On the one hand, unlike OS-check, we will fully employ the structural characteristics of infinite loops to design reliable loop checking mechanisms. On the other hand, instead of relying on the expensive recursive clauses, we extract structural information on infinite loops directly from individual subgoals. We will introduce a new concept – *expanded variants*, which captures a key structural characteristic of certain subgoals in an infinite loop. Informally, a subgoal A' is an expanded variant of a subgoal A if it is a variant of A except for some terms (i.e. variables or constants or functions) in A each of which grows in A' into a function containing the term.

The notion of expanded variants provides a very useful tool by which a series of complete loop checking mechanisms can be defined. In this paper, we develop four such *VAF-checks* (for Variant Atoms loop checks for logic programs with Functions) $-$ $VAF_{1-4}(d)$, where d is a depth bound. $VAF_1(d)$ identifies infinite loops based on expanded variants. $VAF_2(d)$ enhances $VAF_1(d)$ by taking into account one (infinitely) repeated clause. $VAF_3(d)$ enhances $VAF_2(d)$ with a constraint of a set of (infinitely) repeated clauses. And $VAF_4(d)$ enhances $VAF_3(d)$ with a constraint of recursive clauses. The reliability increases from $VAF_1(d)$ to $VAF_4(d)$, but the computational overhead increases, too. By balancing between the two key factors, we choose $VAF_2(d)$ as the best for practical applications. $VAF_2(d)$ has the same complexity as OS-check, but is far more reliable than OS-check. When $d \geq 2$, $VAF_2(d)$ is reliable for a vast majority of logic programs. Moreover, while no less reliable than EVA-check, $VAF_2(d)$ is much more efficient than EVA-check (because like OS-check it does not compute recursive clauses).

The plan of this paper is as follows. In Section 2, we review basic concepts concerning loop checking. In Section 3, we introduce expanded variants and examine their properties. In Section 4, we define four VAF-checks and prove their completeness. In Section 5, we make a comparison of the VAF-checks with OS-check and EVA-check.

2 Preliminaries

In this section, we review some basic concepts concerning loop checking. We assume familiarity with the basic concepts of logic programming, as presented in [11]. Here and throughout, by a logic program we always mean a positive logic program. Variables begin with a capital letter, and predicate symbols, function symbols and constants with a lower case letter. Let A be an atom/function. The size of A, denoted $|A|$, is the count of function symbols, variables and constants in A. We use $rel(A)$ to refer to the predicate/function symbol of A, and use $A[i]$ to refer to the i-th argument of A, $A[i][j]$ to refer to the j-th argument of the i-th argument, and $A[i]...[k]$ to refer to the k-th argument of ... of the i-th argument. For example, let $A = p(a, X, f(g, h(Y)))$, then $A[3] = f(g, h(Y))$, $A[3][2] = h(Y)$, and $A[3][2][1] = Y$.

Definition 2.1 By a *variant* of an SLD-derivation (resp. a goal, subgoal, atom or function) D we mean a derivation (resp. a goal, subgoal, atom or function) D' that is the same as D up to variable renaming.

For example, $p(X)$ is a variant of $p(Y)$ since these two atoms become the same after the variable Y is renamed by X.

Definition 2.2 ([1,3]) Let P be a logic program, G_0 a top goal and S a computation rule.

(i) Let L be a set of SLD-derivations of $P \cup \{G_0\}$ under S. Define
$RemSub(L) = \{\ D \in L \mid$ no $D' \in L$ that is a proper subderivation of D$\}$.
L is *subderivation free* if $L = RemSub(L)$.

(ii) A (simple) *loop check* is a computable set L of finite SLD-derivations such that L is closed under variants and is subderivation free.

Observe that a loop check L formally defines a certain type of infinite loops generated from $P \cup \{G_0\}$ under S; i.e. an SLD-derivation $G_0 \Rightarrow ... \Rightarrow_{C_k,\theta_k} G_k \Rightarrow ...$ is said to step into an infinite loop at G_k if $G_0 \Rightarrow ... \Rightarrow_{C_k,\theta_k} G_k$ is in L. Therefore, whenever such an infinite loop is detected, we should cut it immediately below G_k. This leads to the following definition.

Definition 2.3 Let T be the SLD-tree of $P \cup \{G_0\}$ under S and L a loop check. Let $CUT = \{G' \mid$ the SLD-derivation from the top goal G_0 to G' is in $L\}$. By applying L to T we obtain a new SLD-tree T_L which consists of T with all the nodes (goals) in CUT pruned. By pruning a node from an SLD-tree we mean removing all its descendants.

In order to justify a loop check, Bol et al. introduced the following criteria.

Definition 2.4 ([1]) Let S be a computation rule. A loop check L is *weakly sound* if the following condition holds: for every logic program P, top goal G_0 and SLD-tree T of $P \cup \{G_0\}$ under S, if T contains a successful branch, then T_L contains a successful branch. A loop check L is *complete* if every infinite SLD-derivation is pruned by L. (Put another way, a loop check L is complete if for any logic program P and top goal G_0 T_L is finite.)

An ideal loop check would be both weakly sound and complete. Unfortunately, since logic programs have the full power of the recursive theory, there is no loop check that is both weakly sound and complete even for function-free logic programs [1]. As mentioned in the Introduction, in this paper we explore complete loop checking mechanisms. So in order to compare different complete loop checks, we introduce the following concept.

Definition 2.5 A complete loop check L_1 is said to be *more reliable*[5] than a complete loop check L_2 if for any logic program P and top goal G_0, the successful SLD-derivations in T_{L_1} are not less than those in T_{L_2}, and not vice versa.

[5] In [20], it is phrased as *more sound*.

It is proved that EVA-check is more reliable than OS-check [20]. In the Introduction, we mentioned a notion of ancestor subgoals.

Definition 2.6 ([20]) For each subgoal A in an SLD-tree, its *ancestor list* AL_A is defined recursively as follows:

(i) If A is at the root, then $AL_A = \{\}$.
(ii) Let $M = \leftarrow A_1, ..., A_m$ be a node in the SLD-tree, with A_1 being selected to resolve against a clause $A'_1 \leftarrow B_1, ..., B_n$. Let $A_1\theta = A'_1\theta$. So M has a child node $N = \leftarrow B_1\theta, ..., B_n\theta, A_2\theta, ..., A_m\theta$. Let the ancestor list of each A_i at M be AL_{A_i}. Then the ancestor list $AL_{B_i\theta}$ of each $B_i\theta$ at N is $\{A_1\} \cup AL_{A_1}$ and the ancestor list $AL_{A_j\theta}$ of each $A_j\theta$ is AL_{A_j}.

Obviously, for any subgoals A and B, if A is in the ancestor list of B, i.e. $A \in AL_B$, the proof of A requires the proof of B.

Definition 2.7 Let G_i and G_k be two nodes in an SLD-derivation, and A and B be the selected subgoals in G_i and G_k, respectively. We say A is an *ancestor subgoal* of B, denoted $A \prec_{ANC} B$, if $A \in AL_B$.

The following result shows that the ancestor relation $\prec_{ANC}$ is transitive.

Theorem 2.1 *If* $A_1 \prec_{ANC} A_2$ *and* $A_2 \prec_{ANC} A_3$ *then* $A_1 \prec_{ANC} A_3$.

Proof. By the definition of ancestor lists, for any subgoal A if $A \in AL_{A'}$, then $AL_A \subset AL_{A'}$. So $A_2 \prec_{ANC} A_3$ implies $AL_{A_2} \subset AL_{A_3}$. Thus $A_1 \in AL_{A_2}$ (since $A_1 \prec_{ANC} A_2$) and $AL_{A_2} \subset AL_{A_3}$ implies $A_1 \in AL_{A_3}$. That is, $A_1 \prec_{ANC} A_3$. □

With no loss in generality, in the sequel we assume the *leftmost* computation rule. So the *selected* subgoal at each node is the leftmost subgoal. For convenience, for any node (goal) G_i, unless otherwise specified we use A_i to refer to the leftmost subgoal of G_i.

3 Expanded Variants

To design a complete and reliable loop check, we first need to determine what principal characteristics that an infinite loop possesses. Consider the infinite loop L_2 (see the Introduction) again. We notice that for any $i \geq 0$, the subgoal $p(f(..f(f(g(a)))..))$ at the $(i+1)$-th node G_{i+1} is a variant of the subgoal $p(f(..f(g(a))..))$ at the i-th node G_i except for the **function** $g(a)$ at G_i that grows into a function $f(g(a))$ at G_{i+1}. However, If we replace $g(a)$ with a

constant a in L_2, then $p(f(..f(f(a))..))$ at G_{i+1} is a variant of $p(f(..f(a)..))$ at G_i except for the **constant** a at G_i that grows into a function $f(a)$ at G_{i+1}. Furthermore, If we replace $g(a)$ with a variable X in L_2, then $p(f(..f(f(X))..))$ at G_{i+1} is a variant of $p(f(..f(X)..))$ at G_i except for the **variable** X at G_i that grows into a function $f(X)$ at G_{i+1}.

As another example, consider the program

$$\begin{array}{lll} P_3: & p(X,Y) \leftarrow q(X,Y). & C_{31} \\ & q(X,g(Y)) \leftarrow p(X,g(f(X,Y))). & C_{32} \end{array}$$

Let the top goal $G_0 = \leftarrow p(a,Z)$. Then we will get an infinite loop L_3 as depicted in Fig.1. Observe that for any $i > 0$, the subgoal at $G_{2(i+1)}$ is a variant of that at G_{2i} except that the variable Y at G_{2i} grows into $f(a,Y)$ at $G_{2(i+1)}$.

$$\begin{array}{rl} \leftarrow p(a,Z) & G_0 \\ \Downarrow_{C_{31}} & \\ \leftarrow q(a,Z) & G_1 \\ \Downarrow_{C_{32}} & \\ \leftarrow p(a,g(f(a,Y))) & G_2 \\ \Downarrow_{C_{31}} & \\ \vdots & \\ \leftarrow p(a,g(f(a,f(a,...f(a,Y))))) & G_{2i} \\ \Downarrow_{C_{31}} & \\ \vdots & \end{array}$$

Fig.1 The infinite loop L_3.

These observations reveal a key structural characteristic of some subgoals in an infinite loop with functions, which can be formalized as follows.

Definition 3.1 Let A and A' be two atoms/functions. A' is said to be an *expanded variant* of A, denoted $A' \sqsupseteq_{EV} A$, if after variable renaming A' becomes B that is the same as A except that there may be some terms at certain positions in A each $A[i]...[k]$ of which grows in B into a function $B[i]...[k] = f(...,A[i]...[k],...)$. Such terms like $A[i]...[k]$ in A are then called *growing terms* w.r.t. A'.

For example, $p(f(Y))$ is an expanded variant of $p(X)$ since by renaming the variable Y with X $p(f(Y))$ becomes $p(f(X))$, which is the same as $p(X)$ except for X growing into $f(X)$.

To simplify the presentation, in the sequel of the paper, when comparing atoms/functions we always assume their variables have already been renamed. The following result is immediate.

Theorem 3.1 *If A is a variant of B, then $A \sqsupseteq_{EV} B$.*

Example 3.1 At each of the following lines, A' is an expanded variant of A because it is the same as A except for the growing terms.

A' : $p(f(g(a)))$	A : $p(g(a))$	where	$A'[1] = f(A[1])$
$p(f(g(h(X))))$	$p(h(X))$		$A'[1] = f(g(A[1]))$ [6]
$p(a, g(f(a, f(a, Y))))$	$p(a, g(f(a, Y)))$		$A'[2][1] = f(a, A[2][1])$ or
			$A'[2][1][2] = f(a, A[2][1][2])$
$p(f(b, a), a, X)$	$p(a, a, Y)$		$A'[1] = f(b, A[1])$
$p(X, f(X))$	$p(X, X)$		$A'[2] = f(A[2])$
$p([X_1, X_2, X_3])$	$p([X_2, X_3])$		$A'[1] = [X_1 \mid A[1]]$

However, at the following lines A' is not an expanded variant of A.

A' : $p(f(a), b)$	A : $p(a, c)$	/*c and b are not unifiable
$p(f(X), Y)$	$p(X, f(Y))$	/*$f(Y)$ cannot be in Y
$p(X, f(X))$	$p(Y, X)$	/*since $p(X, X)$ is not a variant of $p(Y, X)$

In the above example, $p(X, f(X))$ is an expanded variant of $p(X, X)$. It might be doubtful how that would happen in an infinite loop. Here is an example.

Example 3.2 Let $P_4 = \{p(X, Y) \leftarrow p(X, f(Y)) \quad (C_{41})\}$ be a logic program and $G_0 = \leftarrow p(X, X)$ a top goal. We have the following infinite loop:

$$\leftarrow p(X, X) \Rightarrow_{C_{41}} \leftarrow p(X, f(X)) \Rightarrow_{C_{41}} ... \Rightarrow_{C_{41}} \leftarrow p(X, f(...f(X)...)) \Rightarrow ... \quad L_4$$

Clearly, for any $i \geq 0$, the subgoal A_{i+1} at G_{i+1} is the subgoal A_i at G_i with the second X growing to $f(X)$. That is, A_{i+1} is a variant of A_i except for $A_{i+1}[2] = f(A_i[2])$.

Any expanded variant has the following properties.

Theorem 3.2 *Let $A' \sqsupseteq_{EV} A$.*

(1) $|A| \leq |A'|$.
(2) For any i, ..., k, $|A[i]...[k]| \leq |A'[i]...[k]|$.
(3) When $|A| = |A'|$, *A and A' are variants.*
(4) When $|A| \neq |A'|$, *there exists i such that* $|A[i]| < |A'[i]|$.

Proof. (1) and (2) are immediate from Definition 3.1. By (2), when $|A| = |A'|$, for any i, ..., k $|A[i]...[k]| = |A'[i]...[k]|$. That is, there is no growing term in A, so by Definition 3.1 A' is a variant of A. This proves (3). Finally, (4) is immediate from (2). □

[6] This example is suggested by an anonymous referee.

These properties are useful for the computation of expanded variants. That is, if $|A'| < |A|$, we conclude A' is not an expanded variant of A. Otherwise, if $|A| = |A'|$, we determine if both are variants. Otherwise, we proceed to their arguments (recursively) to find growing terms and check if they are variants except for the growing terms.

The relation "*variant of*" defined in Definition 2.1 yields an equivalent relation; it is reflexive (i.e., A is a variant of itself), symmetric (i.e., A being a variant of B implies B is a variant of A), and transitive (i.e., if A is a variant of B and B is a variant of C, then A is a variant of C). However, the relation $\sqsupseteq_{EV}$ is not an equivalent relation.

Theorem 3.3 *The following properties hold:*

(1) $A \sqsupseteq_{EV} A$.
(2) $A \sqsupseteq_{EV} B$ *does not imply* $B \sqsupseteq_{EV} A$.
(3) $A \sqsupseteq_{EV} B$ *and* $A \sqsupseteq_{EV} C$ *does not imply* $B \sqsupseteq_{EV} C$.
(4) $A \sqsupseteq_{EV} B$ *and* $B \sqsupseteq_{EV} C$ *does not imply* $A \sqsupseteq_{EV} C$.

Proof. (1) Straightforward by Theorem 3.1. (2) Here is a counter-example: $p(f(X)) \sqsupseteq_{EV} p(X)$, but $p(X) \not\sqsupseteq_{EV} p(f(X))$. (3) Immediate by letting $A = p(f(X_1), g(Y_1))$, $B = p(f(X_2), Y_2)$, and $C = p(X_3, g(Y_3))$. (4) A counter-example: $p(g(f(g(a)))) \sqsupseteq_{EV} p(f(g(a))) \sqsupseteq_{EV} p(f(a))$, but $p(g(f(g(a)))) \not\sqsupseteq_{EV} p(f(a))$ $\Box$

The following result is immediate from Theorem 3.2, which states that the size of expanded variants is transitively decreasing.

Corollary 3.4 *If* $A \sqsupseteq_{EV} B$ *and* $B \sqsupseteq_{EV} C$ *then* $|A| \geq |C|$.

The concept of expanded variants provides a basis for designing loop checking mechanisms for logic programs with functions. This claim is supported by the following theorem.

Theorem 3.5 *Let* $D = (G_0 \Rightarrow_{C_1} G_1 \Rightarrow ... \Rightarrow_{C_i} G_i \Rightarrow ...)$ *be an infinite SLD-derivation with infinitely large subgoals. Then there are infinitely many goals* $G_{i_1}, G_{i_2}, ...$ *such that for any* $j \geq 1$, $A_{i_j} \prec_{ANC} A_{i_{j+1}}$ *and* $A_{i_{j+1}} \sqsupseteq_{EV} A_{i_j}$ *with* $|A_{i_{j+1}}| > |A_{i_j}|$.

Proof. Since D is infinite, by the justification given by Bol [3] (page 40) D has an infinite subderivation D' of the form

$$(\leftarrow A'_{i_1}, ...) \Rightarrow ... \Rightarrow (\leftarrow A'_{i_2}, ...) \Rightarrow ... \Rightarrow (\leftarrow A'_{i_j}, ...) \Rightarrow ...$$

where for any $j \geq 1$, $A'_{i_j} \prec_{ANC} A'_{i_{j+1}}$. Since any logic program has only a finite number of clauses, there must be a set of clauses in the program that are invoked an infinite number of times in D'. Let $S = \{C_1, ..., C_n\}$ be the set of all different clauses that are used an infinite number of times in D'. Then D' must have an infinite subderivation D'' of the form

$$(\leftarrow A''_{i_1}, ...) \Rightarrow_{C_{1_1}} ... \Rightarrow_{C_{1_{n_1}}} (\leftarrow A''_{i_2}, ...) \Rightarrow_{C_{2_1}} ... \Rightarrow_{C_{2_{n_2}}} (\leftarrow A''_{i_3}, ...) \Rightarrow_{C_{3_1}} ...$$

where for any $j \geq 1$, $A''_{i_j} \prec_{ANC} A''_{i_{j+1}}$ and $\{C_{j_1}, ..., C_{j_{n_j}}\} = S$.[7] Since any logic program has only a finite number of predicate/function/constant symbols and D contains infinitely large subgoals, there must be an infinite sequence of A''_{i_j}s in D'': $A_{i_1}, ..., A_{i_j}, ...$ such that for any $j \geq 1$, $A_{i_j} \prec_{ANC} A_{i_{j+1}}$ and A_{i_j} is a variant of $A_{i_{j+1}}$ except for a few terms in $A_{i_{j+1}}$ whose size increases. Note that such an infinite increase in term size in D'' must result from some clauses in S that cause some terms I to grow into functions of the form $f(...I...)$ each cycle S is applied. This means that A_{i_j} is a variant of $A_{i_{j+1}}$ except for some terms I that grow in $A_{i_{j+1}}$ into $f(...I...)$, i.e., $A_{i_{j+1}} \sqsupseteq_{EV} A_{i_j}$ with $|A_{i_{j+1}}| > |A_{i_j}|$. □

4 VAF-Checks

Based on expanded variants, we can define a series of loop checking mechanisms for logic programs with functions. In this section, we present four representative VAF-checks and prove their completeness.

Definition 4.1 Let P be a logic program, G_0 a top goal, and $d \geq 1$ a depth bound. Define

$VAF_1(d) = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_1} G_1 \Rightarrow ... \Rightarrow_{C_k} G_k)$ in which there are up to d goals $G_{i_1}, G_{i_2}, ..., G_{i_d}$ $(0 \leq i_1 < i_2 < ... < i_d < k = i_{d+1})$ that satisfy the following conditions:
(1) For each $j \leq d$, $A_{i_j} \prec_{ANC} A_{i_{j+1}}$ and $A_{i_{j+1}} \sqsupseteq_{EV} A_{i_j}$.
(2) For any $j \leq d$ $|A_{i_j}| = |A_{i_{j+1}}|$, or for any $j \leq d$ $|A_{i_j}| < |A_{i_{j+1}}|$. $\})$

Theorem 4.1 *(1) $VAF_1(d)$ is a (simple) loop check. (2) $VAF_1(d)$ is complete w.r.t. the leftmost computation rule.*

Proof. (1) Straightforward from Definition 2.2. (2) Let $D = (G_0 \Rightarrow_{C_1} G_1 \Rightarrow ... \Rightarrow_{C_i} G_i \Rightarrow ...)$ be an infinite SLD-derivation. Since P has only a finite

[7] Note that (1) the order of clauses in $\{C_{j_1}, ..., C_{j_{n_j}}\}$ is not necessarily the same as that in S, say $\{C_{j_1}, ..., C_{j_{n_j}}\} = \{C_2, C_n, ..., C_1\}$, and (2) $\{C_{j_1}, ..., C_{j_{n_j}}\}$ may contain duplicated clauses, say $\{C_{j_1}, ..., C_{j_{n_j}}\} = \{C_2, C_n, C_1, ..., C_1\}$.

number of clauses, there must be a set of clauses in P that are invoked an infinite number of times during the derivation. Let $S = \{C_1, ..., C_n\}$ be the set of all distinct clauses that are applied an infinite number of times in D. Then, by the proof of Theorem 3.5 D has an infinite sub-derivation of the form

$$(\leftarrow A_{i_1}, ...) \Rightarrow_{C_{1_1}} ... \Rightarrow_{C_{1_{n_1}}} (\leftarrow A_{i_2}, ...) \Rightarrow_{C_{2_1}} ... \Rightarrow_{C_{2_{n_2}}} (\leftarrow A_{i_3}, ...) \Rightarrow_{C_{3_1}} ...$$

where for any $j \geq 1$, $\{C_{j_1}, ..., C_{j_{n_j}}\} = S$ and $A_{i_j} \prec_{ANC} A_{i_{j+1}}$. Let $T = \{A_{i_1}, A_{i_2}, ...\}$. We distinguish between two cases.

(i) There is no subgoal in D whose size is infinitely large. Because any logic program has only a finite number of predicate symbols, function symbols and constants, there must be infinitely many atoms in T that are variants. Let $\{B_1, ..., B_d, B_{d+1}\}$ be the first $d+1$ atoms in T that are variants. Then, by Theorem 3.1, for each $1 \leq j \leq d$ $B_{j+1} \sqsupseteq_{EV} B_j$ with $|B_{j+1}| = |B_j|$, so the conditions of $VAF_1(d)$ are satisfied, which leads to the derivation D being pruned at the node with the leftmost subgoal B_{d+1}.

(ii) There is a subgoal in D with infinitely large size. Then by Theorem 3.5, there must be infinitely many atoms in T that are expanded variants with growing terms. Let $\{B_1, ..., B_d, B_{d+1}\}$ be the first $d+1$ atoms in T such that for each $1 \leq j \leq d$, $B_{j+1} \sqsupseteq_{EV} B_j$ with $|B_{j+1}| > |B_j|$. Again, the conditions of $VAF_1(d)$ are satisfied, so that the derivation D will be pruned. □

Since $VAF_1(d)$ is complete for any $d \geq 1$, taking $d \rightarrow \infty$ leads to the following immediate corollary to Theorem 4.1.

Corollary 4.2 *Any infinite SLD-derivation contains an infinite sub-derivation of the form*

$$(\leftarrow A_{i_1}, ...) \Rightarrow ... \Rightarrow (\leftarrow A_{i_2}, ...) \Rightarrow ... \Rightarrow (\leftarrow A_{i_j}, ...) \Rightarrow ...$$

such that all A_{i_j} satisfy the two conditions of $VAF_1(d)$ ($d \rightarrow \infty$).

Observe that $VAF_1(d)$ identifies infinite loops only based on expanded variants of selected subgoals. More reliable loop checks can be built by taking into account the clauses selected to generate those expanded variants.

Definition 4.2 Let P be a logic program, G_0 a top goal, and $d \geq 1$ a depth bound. Define

$VAF_2(d) = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_1} G_1 \Rightarrow ... \Rightarrow_{C_k} G_k)$ in which there are up to d goals $G_{i_1}, G_{i_2}, ..., G_{i_d}$ $(0 \leq i_1 < i_2 < ... < i_d < k = i_{d+1})$ that satisfy the following conditions:
(1) For each $j \leq d$, $A_{i_j} \prec_{ANC} A_{i_{j+1}}$ and $A_{i_{j+1}} \sqsupseteq_{EV} A_{i_j}$.

(2) For any $j \leq d$ $|A_{i_j}| = |A_{i_{j+1}}|$, or for any $j \leq d$ $|A_{i_j}| < |A_{i_{j+1}}|$.
(3) For all $j \leq d$, the clause selected to resolve with A_{i_j} is the same. })

Theorem 4.3 *(1) $VAF_2(d)$ is a (simple) loop check. (2) $VAF_2(d)$ is complete w.r.t. the leftmost computation rule.*

Proof. (1) Straightforward. (2) By Corollary 4.2, for any infinite SLD-derivation D, there is an infinite sub-derivation in D of the form

$$(\leftarrow A'_{i_1}, ...) \Rightarrow_{C_1} ... \Rightarrow (\leftarrow A'_{i_2}, ...) \Rightarrow_{C_2} ... \Rightarrow (\leftarrow A'_{i_j}, ...) \Rightarrow_{C_j} ...$$

such that all A'_{i_j} satisfy the first two conditions of $VAF_2(d)$. Since any logic program has only a finite number of clauses, there must be a clause C_k that resolves with infinitely many A'_{i_j}s in the sub-derivation. Let $A_{i_1}, ..., A_{i_d}$ be the first d A'_{i_j}s that resolve with C_k. The third condition of $VAF_2(d)$ is then satisfied, so we conclude the proof. □

Again, taking $d \to \infty$ leads to the following corollary to Theorem 4.3.

Corollary 4.4 *Any infinite SLD-derivation contains an infinite sub-derivation of the form*

$$(\leftarrow A_{i_1}, ...) \Rightarrow_{C_k} ... \Rightarrow (\leftarrow A_{i_2}, ...) \Rightarrow_{C_k} ... \Rightarrow (\leftarrow A_{i_j}, ...) \Rightarrow_{C_k} ...$$

such that all A_{i_j} satisfy the three conditions of $VAF_2(d)$ ($d \to \infty$).

Since $VAF_2(d)$ is a special case of $VAF_1(d)$, any SLD-derivation pruned by $VAF_2(d)$ must be pruned by $VAF_1(d)$, but the converse is not true. As an example, consider the SLD-derivation

$$\leftarrow p(a) \Rightarrow_{C_1} \leftarrow p(f_1(a)) \Rightarrow_{C_2} \leftarrow p(f_2(f_1(a))) \Rightarrow_{C_3} \square$$

It will be cut by $VAF_1(2)$ but not by $VAF_2(2)$ because condition (3) is not satisfied. This leads to the following.

Theorem 4.5 *$VAF_2(d)$ is more reliable than $VAF_1(d)$.*

$VAF_2(d)$ considers only the repetition of one clause in an infinite SLD-derivation. More constrained loop checks can be developed by considering the repetition of a set of clauses.

Definition 4.3 Let P be a logic program, G_0 a top goal, and $d \geq 1$ a depth bound. Define

$VAF_3(d) = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_1} G_1 \Rightarrow ... \Rightarrow_{C_k} G_k)$ in which there are up to d goals $G_{i_1}, G_{i_2}, ..., G_{i_d}$ $(0 \leq i_1 < i_2 < ... < i_d < k = i_{d+1})$ that satisfy the following conditions:
(1) For each $j \leq d$, $A_{i_j} \prec_{ANC} A_{i_{j+1}}$ and $A_{i_{j+1}} \sqsupseteq_{EV} A_{i_j}$.
(2) For any $j \leq d$ $|A_{i_j}| = |A_{i_{j+1}}|$, or for any $j \leq d$ $|A_{i_j}| < |A_{i_{j+1}}|$.
(3) For all $j \leq d$ the clause selected to resolve with A_{i_j} is the same.
(4) For all $j \leq d$ the set S of clauses used to derive $A_{i_{j+1}}$ from A_{i_j} is the same. $\})$

Theorem 4.6 *(1) $VAF_3(d)$ is a (simple) loop check. (2) $VAF_3(d)$ is complete w.r.t. the leftmost computation rule.*

Proof. (1) Straightforward. (2) By Corollary 4.4, for any infinite SLD-derivation D, there is an infinite sub-derivation in D of the form

$$(\leftarrow A'_{i_1}, ...) \Rightarrow_{C_k} ... \Rightarrow (\leftarrow A'_{i_2}, ...) \Rightarrow_{C_k} ... \Rightarrow (\leftarrow A'_{i_j}, ...) \Rightarrow_{C_k} ...$$

such that all A'_{i_j} satisfy the first two conditions of $VAF_3(d)$. Obviously, the third condition of $VAF_3(d)$ is satisfied as well. Since any logic program has only a finite number of clauses, there must be an infinite sequence, $A'_{l_1}, ..., A'_{l_j}, ...$, of A'_{i_j}s in the sub-derivation such that the set S of clauses used to derive $A'_{l_{j+1}}$ from A'_{l_j} is the same. Let $A_{i_1}, ..., A_{i_{d+1}}$ be the first $d+1$ such A'_{i_j}s. The fourth condition of $VAF_3(d)$ is then satisfied. $\square$

Taking $d \to \infty$ leads to the following immediate corollary to Theorem 4.6.

Corollary 4.7 *Any infinite SLD-derivation contains an infinite sub-derivation of the form*

$$(\leftarrow A_{i_1}, ...) \Rightarrow_{C_k} ... \Rightarrow_{C_{n_1}} (\leftarrow A_{i_2}, ...) \Rightarrow_{C_k} ... \Rightarrow_{C_{n_2}} ...(\leftarrow A_{i_j}, ...) \Rightarrow_{C_k} ... \Rightarrow_{C_{n_j}} ...$$

such that all A_{i_j} satisfy the three conditions of $VAF_2(d)$ $(d \to \infty)$, and that for any $j \geq 1$ $\{C_k, ..., C_{n_j}\} = \{C_k, ..., C_{n_{j+1}}\}$.

Obviously, any SLD-derivation pruned by $VAF_3(d)$ must be pruned by $VAF_2(d)$. But the converse is not true. Consider the SLD-derivation

$$\leftarrow p(a) \Rightarrow_{C_1} \leftarrow p(f_1(a)) \Rightarrow_{C_2} \leftarrow p(f_2(f_1(a))) \Rightarrow_{C_1} \leftarrow p(f_1(f_2(f_1(a)))) \Rightarrow_{C_4} \square$$

It will be cut by $VAF_2(2)$ but not by $VAF_3(2)$ because condition (4) is not satisfied. This leads to the following.

Theorem 4.8 *$VAF_3(d)$ is more reliable than $VAF_2(d)$.*

Before introducing another more constrained loop check, we recall a concept of *recursive clauses*, which was introduced in [19].

Definition 4.4 A set of clauses, $\{R_0, ..., R_m\}$, are called *recursive clauses* if they are of the form (or similar forms)

$$\begin{array}{ll} q_0(...X_0...) \leftarrow ..., q_1(...X_0...), ... & R_0 \\ q_1(...X_1...) \leftarrow ..., q_2(...X_1...), ... & R_1 \\ \quad \vdots & \\ q_{m-1}(...X_{m-1}...) \leftarrow ..., q_m(...X_{m-1}...), ... & R_{m-1} \\ q_m(...X_m...) \leftarrow ..., q_0(...f(...X_m...)...), ... & R_m \end{array}$$

where for any $0 < i < m$, $q_i(...X_{i-1}...)$ in R_{i-1} is unifiable with $q_i(...X_i...)$ in R_i with an mgu containing X_{i-1}/X_i, and $q_0(...f(...X_m...)...)$ in R_m is unifiable with $q_0(...X_0...)$ in R_0 with an mgu containing $f(...X_m...)/X_0$. Put another way, $\{R_0, ..., R_m\}$ is a set of recursive clauses if starting from the head of R_0 (replacing X_0 with X) applying them successively leads to an inference chain of the form

$$q_0(...X...) \Rightarrow_{R_0} q_1(...X...) \Rightarrow_{R_1} ... \Rightarrow_{R_{m-1}} q_m(...X...) \Rightarrow_{R_m} q_0(...f(...X...)...)$$

such that the last atom $q_0(...f(...X...)...)$ is unifiable with the head of R_0 with an mgu containing $f(...X...)/X_0$.

Example 4.1 The sets of clauses, $\{C_{11}\}$ in P_1, $\{C_{21}\}$ in P_2, $\{C_{31}, C_{32}\}$ in P_3, and $\{C_{41}\}$ in P_4, are all recursive clauses.

Recursive clauses cause some subgoals to increase their size recursively; i.e., each cycle $\{R_0, ..., R_m\}$ is applied, the size of $q_0(.)$ increases by a constant. If $\{R_0, ..., R_m\}$ can be repeatedly applied an infinite number of times, a subgoal $q_0(.)$ will be generated with infinitely large size (note that not any recursive clauses can be repeatedly applied). Since any logic program has only a finite number of clauses, if there exist no recursive clauses in a program, there will be no infinite SLD-derivations with infinitely large subgoals, because no subgoal can increase its size recursively. This means that any infinite SLD-derivation with infinitely large subgoals is generated by repeatedly applying a certain set of recursive clauses. This leads to the following.

Definition 4.5 Let P be a logic program, G_0 a top goal, and $d \geq 1$ a depth bound. Define

$VAF_4(d) = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_1} G_1 \Rightarrow ... \Rightarrow_{C_k} G_k)$ in which there are up to d goals $G_{i_1}, G_{i_2}, ..., G_{i_d}$ $(0 \leq i_1 < i_2 < ... < i_d < k = i_{d+1})$ that satisfy the following conditions:
(1) For each $j \leq d$, $A_{i_j} \prec_{ANC} A_{i_{j+1}}$ and $A_{i_{j+1}} \sqsupseteq_{EV} A_{i_j}$.
(2) For any $j \leq d$ $|A_{i_j}| = |A_{i_{j+1}}|$, or for any $j \leq d$ $|A_{i_j}| < |A_{i_{j+1}}|$.

(3) For all $j \leq d$ the clause selected to resolve with A_{i_j} is the same.
(4) For all $j \leq d$ the set S of clauses used to derive $A_{i_{j+1}}$ from A_{i_j} is the same.
(5) If for any $j \leq d$ $|A_{i_j}| < |A_{i_{j+1}}|$ then S contains recursive clauses that lead to the size increase. })

Theorem 4.9 *(1) $VAF_4(d)$ is a (simple) loop check. (2) $VAF_4(d)$ is complete w.r.t. the leftmost computation rule.*

Proof. (1) Straightforward. (2) By Corollary 4.7, for any infinite SLD-derivation D, there is an infinite sub-derivation E in D of the form

$$(\leftarrow A'_{i_1}, ...) \Rightarrow_{C_k} ... \Rightarrow_{C_{n_1}} (\leftarrow A'_{i_2}, ...) \Rightarrow_{C_k} ... \Rightarrow_{C_{n_2}} ...(\leftarrow A'_{i_j}, ...) \Rightarrow_{C_k} ... \Rightarrow_{C_{n_j}} ...$$

such that all A'_{i_j} satisfy the first four conditions of $VAF_4(d)$ $(d \to \infty)$. Now assume that for any $j \geq 1$ $|A'_{i_j}| < |A'_{i_{j+1}}|$. Then E contains A'_{i_∞} with infinitely large size. Such infinitely increase in term size in E must be generated by the repeated applications of some recursive clauses. This means that there must be an infinite sequence, $A'_{l_1}, ..., A'_{l_j}, ...$, of A'_{i_j}s in E such that the clauses used to derive $A'_{l_{j+1}}$ from A'_{l_j} contain recursive clauses that lead to the size increase from A'_{l_j} to $A'_{l_{j+1}}$. Let $A_{i_1}, ..., A_{i_{d+1}}$ be the first $d+1$ such A'_{i_j}s. Then all A_{i_j} satisfy the five conditions of $VAF_4(d)$. □

When $d \to \infty$, we obtain the following corollary to Theorem 4.9.

Corollary 4.10 *Any infinite SLD-derivation contains an infinite sub-derivation of the form*

$$(\leftarrow A_{i_1}, ...) \Rightarrow_{C_k} ... \Rightarrow_{C_{n_1}} (\leftarrow A_{i_2}, ...) \Rightarrow_{C_k} ... \Rightarrow_{C_{n_2}} ...(\leftarrow A_{i_j}, ...) \Rightarrow_{C_k} ... \Rightarrow_{C_{n_j}} ...$$

such that for any $j \geq 1$, $A_{i_j} \prec_{ANC} A_{i_{j+1}}$, $A_{i_{j+1}} \sqsupseteq_{EV} A_{i_j}$, $\{C_k, ..., C_{n_j}\} = \{C_k, ..., C_{n_{j+1}}\}$, and for all $j \geq 1$ $|A_{i_j}| = |A_{i_{j+1}}|$ or for all $j \geq 1$ $|A_{i_j}| < |A_{i_{j+1}}|$ where the size increase results from the application of a set of recursive clauses in $\{C_k, ..., C_{n_j}\}$.

Since $VAF_4(d)$ is an enhancement of $VAF_3(d)$, any SLD-derivation pruned by $VAF_4(d)$ must be pruned by $VAF_3(d)$. But the converse is not true. Consider the program that consists of the clauses $C_1 : p(a) \leftarrow p(f(a))$ and $C_2 : p(f(a))$. The SLD-derivation

$$\leftarrow p(a) \Rightarrow_{C_1} \leftarrow p(f(a)) \Rightarrow_{C_2} \square$$

will be cut by $VAF_3(1)$ but not by $VAF_4(1)$ because there are no recursive clauses in the program. So we have the following result.

Theorem 4.11 *$VAF_4(d)$ is more reliable than $VAF_3(d)$.*

Example 4.2 Let us choose the depth bound $d = 1$. Then by applying any one of the four VAF-checks, $VAF_{1-4}(1)$, all the four illustrating infinite loops introduced earlier, $L_1, ..., L_4$, will be cut at some node. That is, L_1, L_2 and L_4 will be pruned at G_1 (the second node from the root), and L_3 (Fig.1) pruned at G_4.

Example 4.3 Consider the following list-reversing program (borrowed from [3])

$$\begin{array}{lll} P_5: & reverse([], X, X). & C_{51} \\ & reverse([0|X], Y, Z) \leftarrow reverse(X, Y, Z). & C_{52} \\ & reverse([s(W)|X], Y, Z) \leftarrow reverse(X, [s(W)|Y], Z). & C_{53} \end{array}$$

and the top goal $G_0 = \leftarrow reverse([0, s(0), s(s(0))|X], [], Z)$. Note that C_{53} is a recursive clause. Again, let us choose $d = 1$. After successively applying the clauses C_{52}, C_{53} and C_{53}, we get the following SLD-derivation:

$$\begin{array}{ll} \leftarrow reverse([0, s(0), s(s(0))|X], [], Z) & G_0 \\ \quad \Downarrow_{C_{52}} & \\ \leftarrow reverse([s(0), s(s(0))|X], [], Z) & G_1 \\ \quad \Downarrow_{C_{53}} & \\ \leftarrow reverse([s(s(0))|X], [s(0)], Z) & G_2 \\ \quad \Downarrow_{C_{53}} & \\ \leftarrow reverse(X, [s(s(0)), s(0)], Z) & G_3 \end{array}$$

It is easy to check that there is no expanded variant, so we continue to expand G_3. We first apply C_{51} to G_3, generating a successful node $\Box$; we then apply C_{52} to G_3, generating a node

$$\leftarrow reverse(X', [s(s(0)), s(0)], Z) \qquad G_5$$

As $A_3 \prec_{ANC} A_5$ and $A_5 \sqsupseteq_{EV} A_3$ with $|A_5| = |A_3|$, $VAF_{1-4}(1)$ are satisfied, which stop expanding G_5. We then apply C_{53} to G_3, generating a node

$$\leftarrow reverse(X', [s(W')|[s(s(0)), s(0)]], Z) \qquad G_6$$

Obviously, $A_3 \prec_{ANC} A_6$ and $A_6 \sqsupseteq_{EV} A_3$ with $|A_6| > |A_3|$ where the size increase of A_6 is via the recursive clause C_{53}, so $VAF_{1-4}(1)$ are satisfied again, which stop expanding G_6. Since $VAF_{1-4}(1)$ cut all infinite branches while retaining the (shortest) successful SLD-derivation

$$G_0 \Rightarrow_{C_{52}} G_1 \Rightarrow_{C_{53}} G_2 \Rightarrow_{C_{53}} G_3 \Rightarrow_{C_{51}} \Box$$

they are weakly sound for $P_5 \cup \{G_0\}$.

Observe that each condition of the above VAF-checks captures one characteristic of an infinite loop. Obviously, except (1) and (5), all the conditions (2)−(4) make sense only when $d > 1$. Because expanded variants capture a key structural characteristic of subgoals in infinite loops, all the VAF-checks with $d = 1$ are weakly sound for a majority of representative logic programs (see the above examples). However, considering the undecidable nature of the loop checking problem, choosing $d > 1$ would be safer.[8] The following example, although quite artificial, illustrates this point.

Example 4.4 Consider the following logic program

$$\begin{array}{lll} P_6: & p(X) \leftarrow p(f(X)). & C_{61} \\ & p(f(a)). & C_{62} \end{array}$$

and the following successful SLD-derivation D for the top goal $G_0 = \leftarrow p(a)$:

$$\leftarrow p(a) \Rightarrow_{C_{61}} \leftarrow p(f(a)) \Rightarrow_{C_{62}} \square$$

Obviously, $p(a) \prec_{ANC} p(f(a))$, $p(f(a)) \sqsupseteq_{EV} p(a)$, and C_{61} is a recursive clause. If we choose $d = 1$, the derivation D will be pruned at G_1 by all the above four VAF-checks. That is, $VAF_{1-4}(1)$ are not weakly sound for this program. Apparently, $VAF_{1-4}(2)$ are weakly sound.

Observe that from $VAF_1(d)$ to $VAF_4(d)$, the reliability increases, but the computational overhead increases as well. Therefore, we need to consider a trade-off in choosing among these VAF-checks. For practical applications, when $d > 1$ we suggest choosing the VAF-checks in the following order: $VAF_2(d)$, $VAF_3(d)$, $VAF_1(d)$, and $VAF_4(d)$. The basic reasons for such a preference are (i) our experience shows that $VAF_2(2)$ is weakly sound for a vast majority of logic programs, and (ii) the check of condition (3) of $VAF_2(d)$ takes little time, whereas the check of recursive clauses (condition (5) of $VAF_4(d)$) is rather costly.

5 Comparison with OS-Check and EVA-Check

Because OS-check, EVA-check and $VAF_{1-4}(d)$ are complete loop checks, we make the comparison based on the two key factors: reliability and computational overhead.

[8] As mentioned by Bol [3], the question of which depth bound is optimal remains open. However, our experiments show that $VAF_2(2)$ is weakly sound for a vast majority of logic programs.

5.1 Comparison with OS-Check

We begin by recalling the formal definition of OS-check.

Definition 5.1 ([3,17]) Let P be a logic program, G_0 a top goal, and $d \geq 1$ a depth bound. Let *size* be a size-function on atoms. Define

$OSC(d,size) = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_1} G_1 \Rightarrow ... \Rightarrow_{C_k} G_k)$ in which there are up to d goals $G_{i_1}, G_{i_2}, ..., G_{i_d}$ $(0 \leq i_1 < i_2 < ... < i_d < k = i_{d+1})$ such that for any $1 \leq j \leq d$
(1) $A_{i_j} \prec_{ANC} A_{i_{j+1}}$ and $rel(A_{i_j}) = rel(A_k)$.
(2) $size(A_{i_j}) \leq size(A_{i_k})$.$\})$

There are three versions of OS-check, depending on how the size-function *size* is defined [17,3]. In the first version, $size(A) = size(B)$ for any atoms A and B, so condition (2) will always hold and thus can be ignored. In the second version, $size(A) = |A|$ for any atom A. And in the third version, for any atoms A and B with the same arity n, $size(A) \leq size(B)$ if for any $1 \leq i \leq n$ $|A[i]| \leq |B[i]|$. Obviously, the third version is more reliable than the first two versions so we can focus on the third version for the comparison.

OS-check is complete [3], but is too weak in that it identifies infinite loops mainly based on the size-function, regardless of what the internal structure of atoms is. Therefore, in order to increase its reliability, we have to choose the depth bound d as large as possible. For example, in [17] $d = 10$ is suggested. However, because the internal structure of atoms with functions may vary drastically in different application programs, using only a large depth bound together with the size-function as the loop checking criterion could in general be ineffective/inefficient. For example, when applying $OSC(10, size)$ to the programs $P_1, ..., P_5$, we would generate a lot of redundant nodes. The following example further illustrates this fact.

Example 5.1 Consider the following logic program and top goal:

$$
\begin{array}{lll}
P_7: & p(X,1) \leftarrow p(f(X),2). & C_{71} \\
& p(X,2) \leftarrow p(f(X),3). & C_{72} \\
& \vdots & \\
& p(X,99) \leftarrow p(f(X),100). & C_{7,99} \\
& p(X,100). & C_{7,100} \\
G_0 = & \leftarrow p(0,1). &
\end{array}
$$

The successful SLD-derivation for $P_7 \cup \{G_0\}$ is as follows:

$$\leftarrow p(0,1) \Rightarrow_{C_{71}} \leftarrow p(f(0),2) \Rightarrow_{C_{72}} ... \Rightarrow_{C_{7,99}} \leftarrow p(\underbrace{f(...f}_{99\ fs}(0)...),100) \Rightarrow_{C_{7,100}} \square$$

It is easy to see that $OSC(d, size)$ is not weakly sound for this program unless we choose $d \geq 100$.

In contrast, in our approach the common structural features of repeated subgoals in infinite loops are characterized by expanded variants. Based on expanded variants the VAF-checks $VAF_{1-4}(d)$ are weakly sound with **small depth bounds** (e.g. $d \leq 2$) for a majority of logic programs. For instance, $VAF_{1-4}(1)$ are weakly sound for P_7 in the above example, which shows a dramatical difference.

The above discussion is summarized in the following results

Theorem 5.1 *Let size be the size-function of the third version of OS-check. For any atoms A and B, $A \sqsupseteq_{EV} B$ implies $size(B) \leq size(A)$.*

Proof. Immediate from Theorem 3.2. □

Theorem 5.2 *For any $1 \leq i \leq 4$, $VAF_i(d)$ is more reliable than $OSC(d, size)$.*

Proof. By Theorem 5.1 and Corollary 3.4, $OSC(d, size)$ will be satisfied whenever condition (1) of $VAF_i(d)$ holds. So any SLD-derivations pruned by $VAF_i(d)$ will be pruned by $OSC(d, size)$ as well. But the reverse is not true. As a counter-example, when $d < 100$, the SLD-derivation in Example 5.1 will be pruned by $OSC(d, size)$ but not by $VAF_i(d)$. □

We now discuss computational overhead. First note that in both OS-check and the VAF-checks, the ancestor checking, $A_{i_j} \prec_{ANC} A_{i_{j+1}}$, is required. Moreover, for each ancestor subgoal A_{i_j} of A_k, in $OSC(d, size)$ we compute $size(A_{i_j}) \leq size(A_{i_k})$, whereas in $VAF_{1-4}(d)$ we compute $A_{i_{j+1}} \sqsupseteq_{EV} A_{i_j}$. Although the computation of expanded variants is a little more expensive than that of the size-function, both are processes of two strings (i.e. atoms). Since string processing is far faster than ancestor checking (which needs to scan the goal-stack), we can assume that the two kinds of string computations take constant time w.r.t. scanning the goal-stack. Under such an assumption, the complexity of $OSC(d, size)$ and $VAF_{1-2}(d)$ is the same (note that the check of conditions (2) and (3) of the VAF-checks takes little time).

Since the check of condition (4) of the VAF-checks requires scanning the goal-stack, $VAF_3(d)$ is more expensive than $OSC(d, size)$. Furthermore, condition (5) of the VAF-checks, i.e. the computation of recursive clauses, is quite expensive because on the one hand, given a logic program we need to determine which clauses in it are recursive clauses, and on the other hand, for two subgoals A_{i_j} and $A_{i_{j+1}}$ with $|A_{i_j}| < |A_{i_{j+1}}|$ in an SLD-derivation, we need to find

if the size increase from A_{i_j} to $A_{i_{j+1}}$ results from some recursive clauses. This means that $VAF_4(d)$ could be much more expensive than $OSC(d, size)$.

The above discussion further suggests that $VAF_2(d)$ is the best choice (balanced between reliability and overhead) among $OSC(d, size)$ and $VAF_{1-4}(d)$.

5.2 Comparison with EVA-Check

We begin by reproducing the definition of EVA-check.

Definition 5.2 ([20]) Let P be a logic program, G_0 a top goal, and $d \geq 1$ a depth bound. Define

$EVA(d) = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_1} G_1 \Rightarrow ... \Rightarrow_{C_k} G_k)$ in which there are up to d goals $G_{i_1}, G_{i_2}, ..., G_{i_d}$ $(0 \leq i_1 < i_2 < ... < i_d < k = i_{d+1})$ such that for any $1 \leq j \leq d$
(1) $A_{i_j} \prec_{ANC} A_{i_{j+1}}$.
(2) A_k is a generalized variant of A_{i_j}.$\})$

Here, a subgoal A' is said to be a *generalized variant* of a subgoal A if it is a variant of A except that there may be some arguments whose size increases from A via a set of recursive clauses.

The following characterization of generalized variants is immediate from the above definition and Definition 3.1.

Theorem 5.3 *For any subgoals A' and A in an SLD-derivation, A' is a generalized variant of A if and only if $A' \sqsupseteq_{EV} A$ and if $|A'| > |A|$ then the size increase is via a set of recursive clauses.*

$EVA(d)$ relies heavily on recursive clauses, so its complexity is similar to $VAF_4(d)$. Since the computation of recursive clauses is too expensive, we will not choose $EVA(d)$ in practical applications unless it is more reliable than some $VAF_i(d)$. However, the following example shows that $EVA(d)$ cannot be more reliable than any of the four VAF-checks.

Example 5.2 Consider the following logic program and top goal:

P_8 : $p(X) \leftarrow p(X).$ C_{81}
$p(X) \leftarrow p(f(X)).$ C_{82}
$p(f(a)).$ C_{83}
$G_0 = \leftarrow p(a).$

A successful SLD-derivation for $P_8 \cup \{G_0\}$ is as follows:

$\leftarrow p(a) \Rightarrow_{C_{81}} \leftarrow p(a) \Rightarrow_{C_{82}} \leftarrow p(f(a)) \Rightarrow_{C_{83}} \square$

It can be easily seen that $\{C_{81}, C_{82}\}$ and $\{C_{82}\}$ are two sets of recursive clauses. Let us choose $d = 2$. Then A_2 is a generalized variant of both A_0 and A_1, so $EVA(2)$ will cut the derivation at G_2. However, this SLD-derivation will never be cut by any $VAF_i(2)$ because condition (2) of the VAF-checks is not satisfied (i.e. we have $|A_0| = |A_1|$, but $|A_1| < |A_2|$).

6 Conclusions

We have developed four VAF-checks for logic programs with functions based on the notion of expanded variants. We observe that the key structural feature of infinite loops is repetition (of selected subgoals and clauses) and recursive increase (in term size). Repetition leads to variants (because a logic program has only a finite number of clauses and predicate/function/constant symbols), whereas recursive increase introduces growing terms. The notion of expanded variants exactly catches such a structural characteristic of certain subgoals in infinite loops. Due to this, the VAF-checks are much more reliable than OS-check and no less reliable than EVA-check even with small depth bounds (see Examples 5.1 and 5.2). On the other hand, since the structural information is extracted directly from individual subgoals, without appealing to recursive clauses, the VAF-checks (except $VAF_4(d)$) are much more efficient than EVA-check.

In balancing between the reliability and computational overhead, we choose $VAF_2(d)$ as the favorite one for practical applications. Although $VAF_2(2)$ is reliable for a vast majority of logic programs, due to the undecidability of the loop checking problem, like any other complete loop checks, $VAF_2(d)$ in general cannot be weakly sound for any fixed d. The only way to deal with this problem is by heuristically tuning the depth bound in practical situations. Methods of carrying out such a heuristic tuning then present an interesting open problem for further study.

Acknowledgements

We thank the anonymous referees for their constructive comments, which have greatly improved the presentation. The first author is supported in part by Chinese National Natural Science Foundation and Trans-Century Training Programme Foundation for the Talents by the Chinese Ministry of Education.

References

[1] R. N. Bol, K. R. Apt and J. W. Klop, An analysis of loop checking mechanisms for logic programs, *Theoretical Computer Science* 86(1):35-79 (1991).

[2] R. N. Bol, Towards more efficient loop checks, in: S. Debray and M. Hermenegildo (eds.), *Proc. of the 1990 North American Conference on Logic Programming*, MIT Press, Cambridge, MA, 1990, pp.343-358.

[3] R. N. Bol, Loop checking in partial deduction, *Journal of Logic Programming* 16:25-46 (1993).

[4] R. N. Bol and L. Degerstedt, Tabulated resolution for the Well-Founded semantics. *Journal of Logic Programming* 34(2):67-109 (1998).

[5] M. Bruynooghe, D. De Schreye and B. Martens, A general criterion for avoiding infinite unfolding during partial deduction, *New Generation Computing* 11(1):47-79 (1992).

[6] W. D. Chen and D. S. Warren, Tabled evaluation with delaying for general logic programs, *J. ACM* 43(1):20-74 (1996).

[7] M. A. Covington, Eliminating unwanted loops in Prolog, *SIGPLAN Notices* 20(1):20-26 (1985).

[8] M. A. Covington, A further note on loops in Prolog, *SIGPLAN Notices* 20(8):28-31 (1985).

[9] D. De Schreye and S. Decorte, Termination of logic programs: the never-ending story, *Journal of Logic Programming* 19/20:199-260 (1993).

[10] F. Ferrucci, G. Pacini and M. Sessa, Redundancy elimination and loop checks for logic programs, *Information and Computation* 119:137-153 (1995).

[11] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed., Springer-Verlag, Berlin, 1987.

[12] J. W. Lloyd and J. C. Shepherdson, Partial evaluation in logic programming, *Journal of Logic Programming* 11:217-242 (1991).

[13] B. Martens and D. De Schreye, Automatic finite unfolding using well-founded measures, *Journal of Logic Programming* 28(2):89-146 (1996).

[14] B. Martens, D. De Schreye and M. Bruynooghe, Sound and complete partial deduction with unfolding based on well-founded measures, in: *Proc. of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, IOS Press, 1992, pp. 473-480.

[15] D. Pool and R. Goebel, On eliminating loops in Prolog, *SIGPLAN Notices* 20(8):38-40 (1985).

[16] K. Sagonas, T. Swift, D. S. Warren, J. Freire and P. Rao, *The XSB Programmer's Manual (Version 1.8)*, Department of Computer Science, SUNY at Stony Brook, 1998. Available from http://www.cs.sunysb.edu/ sbprolog/xsb-page.html.

[17] D. Sahlin, The mixtus approach to automatic partial evaluation of full Prolog, in: S. Debray and M. Hermenegildo (eds.), *Proc. of the 1990 North American Conference on Logic Programming*, MIT Press, Cambridge, MA, 1990, pp.377-398.

[18] D. Sahlin, Mixtus: an automatic partial evaluator for full Prolog, *New Generation Computing* 12(1):7-51 (1993).

[19] Y. D. Shen, Verifying local stratifiability of logic programs and databases II, *New Generation Computing* 14(3):317-341 (1996).

[20] Y. D. Shen, An extended variant of atoms loop check for positive logic programs, *New Generation Computing* 15(2):187-204 (1997).

[21] Y. D. Shen, L. Y. Yuan and J. H. You and N. F. Zhou, Linear tabulated resolution for the well founded semantics, in: M. Gelfond, N. Leone and G. Pfeifer (eds.), *Proc. of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning,* Springer, Berlin, 1999, pp. 192-205.

[22] Y. D. Shen, L. Y. Yuan, J. H. You and N. F. Zhou, Linear tabulated resolution based on Prolog control strategy, *Theory and Practice of Logic Programming*, to appear.

[23] D. Skordev, An abstract approach to some loop detection problems, *Fundamenta Informaticae* 31:195-212 (1997).

[24] D. Smith, M. Genesereth and M. Ginsberg, Controlling recursive inference, *Artificial Intelligence* 30:343-389 (1986).

[25] H. Tamaki and T. Sato, OLD resolution with tabulation, in: E. Shapiro (ed.), *Proceedings of the Third International Conference on Logic Programming*, pp. 84-98. Springer, Berlin, 1986.

[26] A. Van Gelder, Efficient loop detection in Prolog, *Journal of Logic Programming* 4:23-31 (1987).

[27] L. Vieille, Recursive query processing: the power of logic, *Theoretical Computer Science* 69:1-53 (1989).

[28] D. S. Warren, Memoing for logic programs, *CACM* 35(3):93-111 (1992).