On the safe termination of PROLOG programs

Krzysztof R. Apt[1,2]

Roland N. Bol[1]

Jan Willem Klop[1,3]

Abstract

We systematically study loop checking mechanisms for logic programs by considering their soundness, completeness, relative strength and related concepts. We introduce a natural concept of a *simple loop check* and prove that no sound and complete simple loop check exists, even for programs without function symbols. Then we introduce a number of sound simple loop checks and identify a natural class of PROLOG programs for which they are complete. In this class a limited form of recursion is allowed. As a by-product we obtain an implementation of the closed world assumption of Reiter [R] and a query evaluation algorithm for a class of logic programs without function symbols.

1. Introduction

PROLOG has been advocated as a programming language which allows us to write executable specifications. Unfortunately, when interpreting correct specifications written in the form of a logic program as a PROLOG program, a divergence usually arises... This is due to the fact that the PROLOG interpreter uses a depth-first search and consequently can enter an infinite branch and miss a solution.

The problem of detecting such a possibility of divergence is obviously undecidable as PROLOG has the full power of recursion theory. Consequently this problem has been taken care of by developing a number of useful heuristics on how to avoid a possibility of non-termination.

Another possible approach to this problem has been based on modifying the underlying computation mechanism that searches through the corresponding SLD-trees by adding a capability of pruning. Pruning an SLD-tree means that at some point the interpreter is forced to stop its search through a certain part of the tree, typically an infinite branch. Every method of pruning SLD-trees considered so far has been based on excluding some kind of repetition in the SLD-derivations, because such a repetition makes the interpreter enter an infinite loop.

[1] *Centre for Mathematics and Computer Science*
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands

[2] *Department of Computer Sciences, University of Texas at Austin,*
Austin, Texas 78712-1188, USA

[3] *Department of Computer Sciences, Free University of Amsterdam*
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

That is why pruning SLD-trees has been called *loop checking*. Such modifications of PROLOG interpreters were considered in the literature (see e.g. [B], [BW], [Co], [PG], [SGG]), but no results were proved about them, with a notable exception of [SGG].

In this paper we systematically study loop checking mechanisms by analyzing their soundness (no computed answer substitution to a goal is missed), completeness (all resulting derivations are finite), relative strength and related properties. We introduce a natural subclass of loop checking mechanisms, called *simple* loop checks, obtained when their definition does not depend on the analyzed logic programs. We prove among others that no sound and complete simple loop check exists even in the absence of function symbols.

Then we introduce a number of intuitive simple loop checks which are all sound and identify a natural class of *restricted* programs without function symbols for which these loop checks are complete. Restricted programs allow a restricted form of recursion (hence the name).

To better understand the relevance of the problems studied here, consider the following example. Let P be the following simple-minded PROLOG program computing in the relation *tc* the transitive closure of the relation *r*:

P = { tc(x,y) ← r(x,y).
     tc(x,y) ← r(x,z),tc(z,y). }

Suppose we add to P the following facts about r: r(a,a)←, r(a,b)←, r(b,c)←, r(d,a)←. Then if we ask:

- tc(a,b) we get the answer 'yes';
- tc(a,c) the program gets into an infinite loop (whereas we should get the answer 'yes');
- tc(a,d) the program gets into an infinite loop (whereas we should get the answer 'no');
- tc(b,d) we get the answer 'no'.

Thus P is not the right program for computing the transitive closure. One solution is to write a different program, which is not straightforward - see for example the program in [CM], section 7.2. In fact, Kunen [K] recently proved that any such program must use either function symbols or negated literals.

In our solution, we change the underlying interpreter by adding to it a simple loop check, and retain the above program, which turns out to be restricted. (In contrast, this solution cannot be applied to an alternative version of P obtained by replacing the second clause by tc(x,y) ← tc(x,z),tc(z,y), as the resulting program is not any more restricted.)

As a by-product of these considerations we obtain an implementation of the *closed world assumption* of Reiter [R] and of a query evaluation mechanism for definite deductive databases which are restricted programs. The closed world assumption (CWA in short) is a way of inferring negative information in deductive databases. Reiter [R] showed that in the case of definite deductive databases (DB in short) it does not introduce inconsistency. However, even though CWA is correctly defined for DB, there is still the problem of how it can be implemented, since it calls for the use of the following rule (or rather metarule):

$$\text{if DB} \nvdash \varphi \text{ then DB} \vdash \neg\varphi,$$

that is: deduce $\neg\varphi$ if φ cannot be proved from DB using first order logic.

The problem is how to determine for a particular ground atom (or *fact* in short) that there is no proof of it. When DB is a restricted program, to infer $\neg$A for a fact A it suffices to use Clark's [Cl] *negation as (finite) failure* rule augmented with an appropriate loop check.

A more general problem is that of query processing in DB: given an atom A, compute the set $[A]_{DB}$ of all its ground instances Aθ such that DB ⊢ Aθ. Indeed, when A is ground and DB ⊬ A, the query processing problem reduces to the problem of deducing ¬A by means of CWA. When DB is a restricted program, to compute $[A]_{DB}$ for an atom A, it suffices to collect all computed answer substitutions in the SLD-tree with leftmost selection rule and ←A as root, pruned by a sound and complete loop check.

In the full version of this paper we shall analyze several other loop checks, including those based on a subsumption check (see [CL], [SGG]). These loop checks are complete for different classes of programs.

2. Loop checking

Throughout this paper we assume familiarity with the basic concepts and notations of logic programming as described in [L]. For two substitutions σ and τ, we write σ≤τ when σ is more general than τ and for two expressions E and F, we write E≤F if F is an instance of E. We then say that F is *less general* than E. An SLD-derivation step from a goal G, using a clause C and an mgu θ, to a goal H is denoted as $G \Rightarrow_{C,\theta} H$.

The purpose of a loop check is to prune every infinite SLD-tree to a finite subtree of it containing the root. One might define a loop check as a function from SLD-trees to SLD-trees, directly giving the pruned tree. However, this would be a very general definition, allowing practically everything. We shall use therefore a more restricted definition according to which for a program P:

- a node in an SLD-tree of P∪{G} (for some goal G) is *pruned* if all its descendants have been removed. (Note the terminology: the pruned node itself remains in the tree.)
- by pruning some of the nodes we obtain a pruned version of the SLD-tree.
- whether a node is pruned or not only depends upon its ancestors in the SLD-tree, that is on the SLD-derivation from the root up to this node. (Note: throughout the paper, by an SLD-derivation we mean an SLD-derivation in the sense of [L] or an initial fragment of it.)

Therefore, we can define a loop check as a function on the SLD-derivations instead of on the SLD-trees. However, for convenience we do not define it as a function from derivations to derivations, but as a set of derivations (depending on the program): the derivations that are pruned exactly at their last node. Such a set of SLD-derivations L(P) can be extended in a canonical way to a function $f_{L(P)}$ from SLD-trees to SLD-trees by removing from an SLD-tree all the descendants of the nodes in {G | the SLD-derivation from the root to G is in L(P)}. In the remainder of this article, we shall usually make this conversion implicitly.

We shall also study an even more restricted form of loop check, called simple loop check, in which the set of pruned derivations is independent of the program P. In other words, a loop check is a function, having a program as input and a simple loop check as output. This leads us to the following definitions.

DEFINITION 2.1.
Let L be a set of SLD-derivations.
RemSub(L) = {D∈ L | L does not contain a proper subderivation of D}
L is *subderivation free* if L=RemSub(L). □

In order to render the intuitive meaning of a loop check L: 'every derivation $D \in L$ is pruned *exactly* at its last node', we need that L is subderivation free. Note that RemSub(RemSub(L)) = RemSub(L).

In the following definition, by a *variant* of a derivation D we mean a derivation D' in which in every derivation step, atoms in the same positions are selected and the same program clause is used. D' may differ from D in the renaming that is applied to these program clauses for reasons of standardizing apart and in the mgu used. It has been shown that in this case every goal in D' is a variant of the corresponding goal in D (see [LS]).

DEFINITION 2.2.
A *simple loop check* is a computable set L of SLD-derivations such that
- for every derivation D: if $D \in L$ then for every variant D' of D: $D' \in L$;
- L is subderivation free. □

The first condition here ensures that the choice of variables in the input clauses in an SLD-derivation does not influence its pruning. This is a reasonable demand since we are not interested in the choice of the names of the variables in the derivations.

DEFINITION 2.3.
A *loop check* is a computable function L from programs to sets of SLD-derivations such that for every program P, L(P) is a simple loop check. □

DEFINITION 2.4.
Let L be a loop check. An SLD-derivation D of $P \cup \{G\}$ is *pruned by L* if L(P) contains a subderivation D' of D. □

EXAMPLE 2.5 (based on Example 8 in [B], see also [vG1]).
A first attempt to formulate the *Contains a Variant of Atom* (*CVA*) check might be: 'A derivation is pruned at the first goal that contains a variant A of an atom A' that occurred in an earlier goal.' Note that we have to allow here that A and A' are variants: if we required A=A' then we would violate the first condition in definition 2.2.

The intuition behind this loop check is the following. We wish to prove A' by resolution. If we find out after some resolution steps that in order to prove A' we need to prove a variant A of A', then there are two possibilities. One is that there is a proof for A. Then this proof could also be used as a proof for A', by applying an appropriate renaming on it. So we do not need the proof of A' that goes via A. The other possibility is that there is no proof for A. In that case, the attempt to prove A' via A cannot be successful. So in both cases there is no reason to continue the attempt to prove A' via A.

The derivation step $\leftarrow B, A \Rightarrow_{B\leftarrow} \leftarrow A$ shows that the first formulation of the CVA check is not precise enough: it does not capture the intuition that the proof of A' *goes via* A. A should be the result (after one or more derivation steps) of resolving A', or a further instantiated version of A' (if A' is not immediately selected).

Therefore we define CVA = RemSub({ D | D = $(G_0 \Rightarrow_{C1,\theta1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{Ck,\theta k} G_k)$ such that for some i and j, $0 \le i \le j < k$, G_k contains an atom A that is
- a variant of an atom A' in G_i and
- the result of an attempt to resolve $A'\theta_{i+1} \ldots \theta_j$, the further instantiated version of A' that is selected in G_j }).

We shall now give an illustration of the use of this loop check.

Let P = { A(0) ← (C1)
B(1) ← (C2)
A(x) ← A(y) (C3)
C ← A(x),B(x) (C4) },
let G = ← C.

That the informal justification of the loop check CVA is incorrect, is shown by applying it to two SLD-trees of P∪{G}, via leftmost and rightmost selection rule respectively, which gives us:

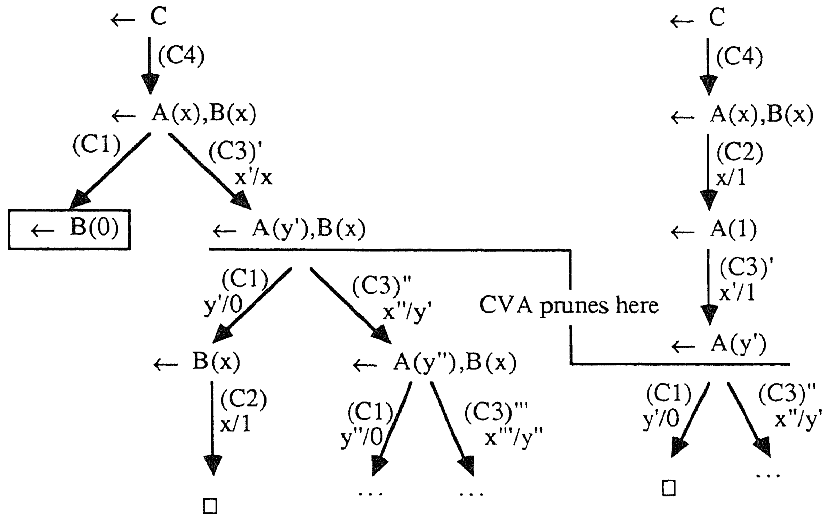

figure 1

Here and elsewhere a substitution θ acting on one variable only, say x, is denoted by x/xθ. A failed node, i.e. a node without a successor in the SLD-tree, is marked by a box around it.

A detailed analysis shows why the goal G_3=←A(y') in the rightmost tree is pruned by the CVA check. Clearly, a variant of A(y') occurs in an earlier goal: A(x) in G_1. So we take i=1. In G_1, A(x) is not yet selected, so j>i. In fact j=2, for in G_2 the atom A(1), which is a further instantiated version of A(x), is selected. Indeed, A(y') is the result of resolving A(1). Therefore the derivation is pruned at G_3 by the CVA check. (In this case, A(y') is the direct result of resolving A(1), but in general there may be any number of derivation steps between G_j and G_k.) □

Indeed, this loop check has not worked properly here: all successful derivations have been pruned. Clearly, this is an undesirable property for loop checks. On the other hand, all infinite derivations are pruned, as intended. In the next section, we shall give formal definitions of these and related properties of loop checks.

3. Some general considerations

In this section some basic properties of loop checks are introduced and some natural results concerning them are established.

3.1. Soundness and completeness

The most important property is definitely that using a loop check does not result in a loss of success. Since we intend to use pruned trees instead of the original ones, we need at least that pruning a successful tree yields again a successful tree.

Even stronger, because we use here a PROLOG-like interpreter augmented with a loop check as the *only* inference mechanism, we do not want to lose any individual solution. That is, if the original tree contains a successful branch (with some computed answer substitution), then we require that the pruned tree contains a successful branch with a more general answer substitution.

Finally, we would like to retain only shorter derivations and prune the longer ones that give the same result. This leads to the following definitions, where for a derivation D, |D| stands for its length, i.e. the number of goals in it.

DEFINITION 3.1.1.
i) A loop check L is *weakly sound* if for every program P and every goal G, and for every SLD-tree T of $P\cup\{G\}$: if T contains a successful branch, then $f_{L(P)}(T)$ contains a successful branch.
ii) A loop check L is *sound* if for every program P and every goal G, and for every SLD-tree T of $P\cup\{G\}$: if T contains a successful branch with a computed answer substitution σ, then $f_{L(P)}(T)$ contains a successful branch with a computed answer substitution σ' such that $G\sigma' \leq G\sigma$.
iii) A loop check L is *shortening* if for every program P and every goal G, and for every SLD-tree T of $P\cup\{G\}$: if T contains a successful branch D with a computed answer substitution σ, then either $f_{L(P)}(T)$ contains D or $f_{L(P)}(T)$ contains a successful branch D' with a computed answer substitution σ' such that $G\sigma' \leq G\sigma$ and $|D'|<|D|$. □

The following lemma is an immediate consequence of these definitions.

LEMMA 3.1.2. *Let L be a loop check.*
i) If L is shortening, then L is sound.
ii) If L is sound, then L is weakly sound. □

The purpose of a loop check is to reduce the search space for top-down interpreters. We would like to end up with a finite search space. This is the case when every infinite derivation is pruned.

DEFINITION 3.1.3.
A loop check L is *complete* if every infinite SLD-derivation is pruned by L. □

We must point out here that in these definitions we have overloaded the terms 'soundness' and 'completeness'. These terms do not refer here only to loop checks, but also to interpreters for logic programs (with or without a loop check). Such an interpreter is sound if the answer it gives (if it gives one) is correct w.r.t. the intended model or the intended theory of the program. An interpreter is complete if it finds every correct answer within a finite time.

3.2. Interpreters and loop checks

When a top-down interpreter is augmented with a loop check, we obtain a new interpreter. The soundness and completeness of this new interpreter depends on the soundness and completeness of the old one, as well as on the soundness and completeness of the loop check. However, these relations are not trivial. In particular, it is not true that adding a complete loop check to a complete interpreter yields again a complete interpreter.

These relationships are expressed by the following lemma's. For each of them, an intuitive meaning is provided in terms of interpreters. We refer here to two interpreters: one searching the SLD-tree depth-first left-to-right (as the PROLOG interpreter does), and one searching breadth-first. Without a loop check, both interpreters are sound w.r.t. CWA. The breadth-first interpreter is also complete.

The (quite simple) proofs are omitted in this section. They will appear in [BAK], the full version of this paper.

LEMMA 3.2.1. *Let P be a program, A a ground atom and L a weakly sound loop check. Then for every SLD-tree T of* $P \cup \{\leftarrow A\}$, $P \vdash_{CWA} \neg A$ *iff* $f_{L(P)}(T)$ *contains no successful branches.* □

Thus an interpreter augmented with a weakly sound loop check remains sound w.r.t. CWA. Since $f_{L(P)}(T)$ may be infinite, nothing can be said about completeness.

LEMMA 3.2.2. *Let P be a program, A an atom and L a sound loop check. Then for every SLD-tree T of* $P \cup \{\leftarrow A\}$ *and for every ground substitution* θ, $P \vdash A\theta$ *iff* $f_{L(P)}(T)$ *contains a successful branch with a computed answer substitution* τ *such that* $\tau \leq \theta$. □

Thus an interpreter augmented with a sound loop check remains sound. Moreover, a breadth-first interpreter remains complete .

COROLLARY 3.2.3. *Let P be a program, A a ground atom and L a weakly sound and complete loop check. Then for every SLD-tree T of* $P \cup \{\leftarrow A\}$, $P \vdash_{CWA} \neg A$ *iff* $f_{L(P)}(T)$ *is finite and contains no successful branches.* □

Thus an interpreter augmented with a weakly sound and complete loop check becomes complete w.r.t. CWA.

COROLLARY 3.2.4. *Let P be a program, A an atom and L a sound and complete loop check. Then for every SLD-tree T of* $P \cup \{\leftarrow A\}$ *and for every ground substitution* θ, $P \vdash A\theta$ *iff* $f_{L(P)}(T)$ *is finite and contains a successful branch with a computed answer substitution* τ *such that* $\tau \leq \theta$. □

Thus a depth-first interpreter augmented with a sound and complete loop check becomes complete . This also means that a sound and complete loop check can be used to implement query processing as defined in the introduction. Indeed, given a program P and an atom A with an SLD-tree T of $P \cup \{\leftarrow A\}$, it suffices to traverse the finite tree $f_{L(P)}(T)$ and collect all computed answer substitutions.

3.3. Comparing loop checks
After studying the relationships between loop checks and interpreters, we shall now analyze a relationship between loop checks. In general, it can be quite difficult to compare loop checks. However, some of them can be compared in a natural way: if every loop that is detected by one loop check, is detected at the same derivation step or earlier by another loop check, then the latter one is *stronger* than the former.

DEFINITION 3.3.1.
Let L_1 and L_2 be loop checks.
L_1 is *stronger than* L_2 if for every program P every SLD-derivation $D_2 \in L_2(P)$ contains a subderivation D_1 such that $D_1 \in L_1(P)$. □

In other words, L_1 is stronger than L_2 if every SLD-derivation that is pruned by L_2 is also pruned by L_1. Note that the definition implies that L_1 is stronger than itself.

The following theorem will prove to be very useful. It will enable us to obtain soundness and completeness results for loop checks which are related by the 'stronger than' relation, by proving soundness and completeness for only one of them.

THEOREM 3.3.2. *Let L_1 and L_2 be loop checks, and let L_1 be stronger than L_2.*
i) If L_1 is weakly sound, then L_2 is weakly sound.
ii) If L_1 is sound, then L_2 is sound.
iii) If L_1 is shortening, then L_2 is shortening.
iv) If L_2 is complete then L_1 is complete.
PROOF. Straightforward. □

Now we have a more clear view of the situation. Very strong loop checks prune derivations in an 'early stage'. If they prune too early, then they are unsound. Since this is undesirable, we must look for weaker loop checks. But a loop check should preferably be not too weak, for then it might fail to prune some infinite derivations (in other words, it might be incomplete). Of course, the 'stronger than' relation is not linear. Moreover, loop checks exist that are neither sound nor complete.

3.4. Sound and complete loop checks
The question is now: do there exist sound and complete loop checks? Obviously, there cannot be such a loop check for logic programs in general, as logic programming has the full power of recursion theory. (Remember that according to the definition, a loop check is computable.) So our first step is to rule out programs that compute over an infinite domain. We shall do so by restricting our attention to programs without function symbols. This restriction leads to a finite Herbrand Universe, but other solutions (typed functions, bounded term-size property [vG2]) are also possible here.

Note that our definitions so far referred to arbitrary programs and SLD-derivations. In the sequel, we shall consider only certain classes of programs (like the ones with a finite Herbrand Universe) and SLD-derivations (like the derivations via leftmost selection rule). The definitions we introduced can be extended in an obvious way so that we can use terminology like 'complete w.r.t. leftmost selection rule'.

In the sequel, we shall write 'complete' instead of 'complete in the absence of function symbols'. So our question can be reformulated as: is there a sound

and complete loop check? Before answering this question for loop checks in general, we shall answer it for simple loop checks.

THEOREM 3.4.1. *There is no weakly sound and complete simple loop check.*
PROOF. For every $n>0$, let $P_n = \{ S(i,i+1)\leftarrow \mid 0\leq i<n \} \cup \{ A(0)\leftarrow, A(x)\leftarrow A(y),S(y,x), B(n)\leftarrow \}$ and let $G=\leftarrow A(x_0),B(x_0)$. Let L be a complete loop check and T_n an SLD-tree of $P_n\cup\{G\}$ via leftmost selection rule. (T_n is fixed modulo the names of the variables.)
T_n has an infinite branch with goals of the form $\leftarrow A(x_i),S(x_i,x_{i-1}),\ldots,S(x_1,x_0),B(x_0)$ ($i\geq 0$). The side-branches that are the result of applying the clause $A(0)\leftarrow$ instead of $A(x)\leftarrow A(y),S(y,x)$ are all finitely failed, except for the one successful branch that begins at $\leftarrow A(x_n),S(x_n,x_{n-1}),\ldots,S(x_1,x_0),B(x_0)$.

L is complete, so the infinite derivation that is the result of always selecting the recursive clause $A(x)\leftarrow A(y),S(y,x)$ is pruned by L. Since L is simple, the goal at which pruning takes place is independent of P_n. In particular it is independent of n, since n does not occur in this derivation. Suppose that the pruned goal is $\leftarrow A(x_i), S(x_i,x_{i-1}),\ldots,S(x_1,x),B(x)$. (Note that according to the definition of a loop check, taking other variable names does not influence the level at which pruning takes place.) Then for $n\geq i$: T_n contains a successful branch and $f_L(T_n)$ does not. Hence L is not weakly sound. □

Taking the whole program into account gives us an opportunity to define a shortening (so a fortiori sound) loop check which is complete. Moreover, this loop check is stronger than *every* other shortening loop check. Strange as it may seem, this one is also impractical. It is like solving a puzzle by trial and error. You can save effort if you can avoid the trials that lead to an error. In order to know exactly which trials to avoid you decide to solve the puzzle first. Then you know.

DEFINITION 3.4.2.
$STRONG(P) = RemSub(\{D = G\Rightarrow\ldots \mid$ for no σ, D is an initial fragment of a shortest refutation of $P\cup\{G\}$ with a computed answer substitution $\sigma\})$. □

THEOREM 3.4.3. *i) STRONG is a shortening loop check.*
ii) STRONG is stronger than any shortening loop check.
iii) STRONG is complete.
PROOF. The proof will appear in [BAK]. □

So far, we have not been very successful in defining useful sound and complete loop checks. In the next section, we shall restrict our attention to simple loop checks. They will be shortening, but as shown above, they cannot be complete (in the absence of function symbols). Nevertheless, we shall introduce a natural class of programs for which they are complete.

4. Some simple loop checks

In this section, we introduce some simple loop checks. For each of them, there exist two versions: the first one is weakly sound, the second one shortening. The second, shortening version is obtained by adding an extra condition to the first one. By this construction, the first one is always stronger than the second one.

Starting with the Contains a Variant of Atom check (defined for arbitrary selection rules), we can make three independent modifications of it.

1. Adding this extra condition, dealing with the computed answer substitution 'generated so far'. A neat formulation of this condition can be obtained by the use of *resultants* instead of goals in SLD-derivations. When considering a derivation $G_0 \Rightarrow_{C1,\theta 1} G_1 \Rightarrow \ldots$, to every goal $G_i = \leftarrow S_i$ there corresponds the resultant $R_i = S_0\theta_1 \ldots \theta_i \leftarrow S_i$. Resultants were introduced in [LS].
2. Replace *variant* by *instance*. This yields the *Contains an Instance of Atom* (*CIA*) check. This check is still unsound: it is even stronger than the CVA check. Besnard [B] has introduced a weakly sound version of this loop check. This check and related ones (derived from CVA; shortening versions) are discussed in [BAK].
3. Replace *atom* by *goal*. This yields the *Equals Variant of Goal* (*EVG*) check. Informally, this loop check prunes a derivation as soon as a *goal* occurs that is a variant of an earlier goal. Replacing 'variant' by 'instance' again yields the *Equals Instance of Goal* (*EIG*) check. The shortening versions are called *Equals Variant of Resultant* (*EVR*) and *Equals Instance of Resultant* (*EIR*). These checks are discussed below.

   Taking goals instead of atoms as a basis for a loop check yields two independent choices again.

   3a. Whereas equality between atoms is unambiguous, equality between goals is much less clear. In SLD-derivations, we regard goals as lists, so both the number and the order of occurrences of atoms is important. However, we may also regard them as multisets, where the order of the occurrences is unimportant. We might even consider regarding them as sets, but that proves to be impractical: the difference between the derivation steps $\leftarrow A, A \Rightarrow \leftarrow A$ and $\leftarrow A \Rightarrow \leftarrow A$ is then no longer visible. Regarding goals as sets in our loop checks would require regarding goals as sets in SLD-derivations, which would result in too many undesirable effects.

   So we shall consider *two* EVG checks: EVG_L (for list) and EVG_M (for multiset). The same holds for EIG, EVR and EIR. We shall refer to these eight loop checks as the *equality* checks.

   3b. Finally, we may replace "G_2 *is* a variant/instance of G_1" by "G_2 *is subsumed by* a variant/instance of G_1". We define 'G_1 subsumes G_2' as '$G_1 \subseteq G_2$'. Thus we can make a distinction between 'subsumed by a variant' and 'subsumed by an instance'. Usually in literature, 'subsumed by a variant' is not considered, 'subsumed by an instance' is simply called 'subsumed'. See e.g. [CL]. Subsumption can also be defined for resolvents.

   This yields the *subsumption* check. Since this modification is again independent of the others, there are in total 2x2x2=8 subsumption checks. These checks are discussed in [BAK].

We now study the equality checks in more detail. At first we give a formal definition of the weakly sound versions. Then we introduce an extra condition that makes these checks shortening. Finally we identify a natural class of programs for which the equality checks are complete.

In fact, we should give a definition for each equality check. This would yield eight almost identical definitions. Therefore we compress them into two definitions, trusting that the reader is willing to understand our notation. The equality relation between goals regarded as lists is denoted by $=_L$; similarly $=_M$ for multisets. We begin with the weakly sound versions.

DEFINITION 4.1.
For Type $\in$ {L,M}, the *Equals Variant/Instance of Goal$_{Type}$* check is the set of SLD-derivations EVG/EIG$_{Type}$ = RemSub({D | D = ($G_0 \Rightarrow_{C1,\theta1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{Ck,\theta k} G_k$) such that for some i, $0 \le i < k$, there is a renaming/substitution τ such that $G_k =_{Type} G_i\tau$}). □

We shall prove later that these loop checks are weakly sound. However, they are not sound (see Example 4.3). We can make them sound, and even shortening, by adding the condition that τ and $\theta_{i+1}\ldots\theta_k$ also agree on the variables of the intermediate goal $G_0\theta_1\ldots\theta_i$. So the extra condition is: $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$. (Note: in this equality it is irrelevant whether goals are lists or multisets.) It will appear that this condition works not only for EVG and EIG, but for all other loop checks studied in this section, as well.

Note that adding this condition is equivalent to the replacement of the condition $G_k =_{Type} G_i\tau$ by the condition $R_k =_{Type} R_i\tau$, where R_k and R_i are the resultants corresponding to the goals G_k and G_i.

DEFINITION 4.2.
For Type $\in$ {L,M}, the *Equals Variant/Instance of Resultant$_{Type}$* check is the set of SLD-derivations EVR/EIR$_{Type}$ = RemSub({D | D = ($G_0 \Rightarrow_{C1,\theta1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{Ck,\theta k} G_k$) such that for some i, $0 \le i < k$, there is a renaming/substitution τ such that $G_k =_{Type} G_i\tau$ and $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$}). □

The following example shows the difference between the goal-based and resultant-based equality checks. The example is such that the other variations (variants or instances, goals regarded as lists or as multisets) do not play a role here.

EXAMPLE 4.3.
Let P = { $p(a) \leftarrow$, (C1)
$p(y) \leftarrow p(z)$ (C2) },
let G = $\leftarrow p(x)$.

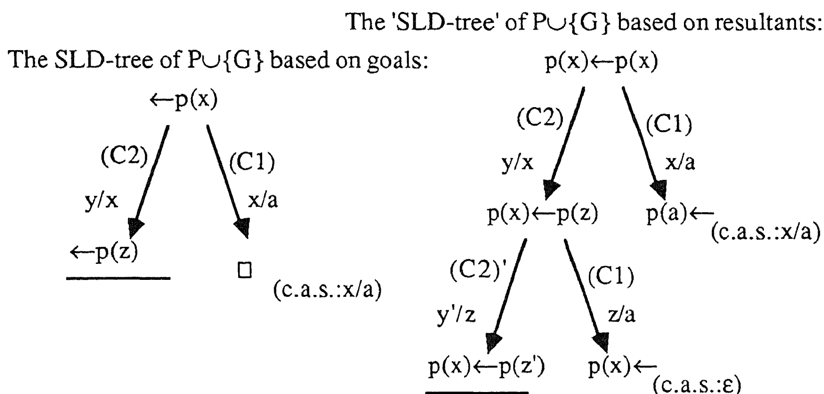

figure 2

Without the condition $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$ we would only obtain x/a as the computed answer substitution, whereas we also should obtain the empty substitution. This shows that the EVG and EIG loop checks are not sound.

In the leftmost tree ←p(z) is a variant of ←p(x), so the derivation is pruned by EVG at that goal. However, the corresponding resultant p(x)←p(z) is clearly not a variant of p(x)←p(x), therefore the derivation is not yet pruned by EVR. However, after another application of (C2), the resultant p(x)←p(z') occurs, which is a variant of p(x)←p(z). At that point the derivation is pruned by EVR.

The rightmost tree in figure 2 shows an 'SLD-tree' in which the goals are replaced by the corresponding resultants. Note that a successful branch in a resultant-based SLD-tree does not end by the empty goal □, but by the instance of the initial goal that was 'proved' by this branch. □

LEMMA 4.4. *All equality checks are simple loop checks.*
PROOF. Straightforward. □

We now prove that the equality checks based on resultants are shortening and that the equality checks based on goals are weakly sound. According to Theorem 3.3.2 it is sufficient to focus on the strongest checks in both classes: the EIR_M and the EIG_M checks. We need the following lemma.

LEMMA 4.5. *Let P be a program. Let G_1 and G_2 be goals such that $G_1 =_M G_2$. Suppose D_1 is an SLD-derivation of $P \cup \{G_1\}$ with computed answer substitution σ. Then there exists an SLD-derivation D_2 of $P \cup \{G_2\}$ with computed answer substitution σ and $|D_1| = |D_2|$ via every selection rule.*
PROOF. By the soundness and strong completeness of SLD-resolution, see [L]. □

THEOREM 4.6. *i) The loop check EIR_M is shortening.*
ii) The loop check EIG_M is weakly sound.

PROOF. i) Let D be an SLD-refutation of G_0 with computed answer substitution σ. If D is pruned by EIR_M then we have to find in every SLD-tree containing D an SLD-refutation D' of G_0 with computed answer substitution σ' such that $G_0\sigma' \leq G_0\sigma$, $|D'|<|D|$ and D' is not pruned by EIR_M. We prove this by induction on the length l of the refutation D. We have $l \geq 1$. For $l=1$, D cannot be pruned. Now suppose the theorem is true for every refutation of G_0 of length $\leq l$. Let D be a refutation of length $l+1$. Suppose that D is pruned by EIR_M. Then we have $D = (G_0 \Rightarrow_{C1,\theta 1} G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow_{Ci,\theta i} G_i \Rightarrow_{Ci+1,\theta i+1} G_{i+1} \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{Ck,\theta k} G_k \Rightarrow_{Ck+1,\theta k+1} G_{k+1} \Rightarrow \ldots \Rightarrow \Box)$, and for some substitution τ: $G_k =_M G_i\tau$ and $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$.

By Lemma 4.5 we have a refutation of $G_i\tau$ with a computed answer substitution $\theta_{k+1}\ldots\theta_l$. Now we can obtain an unrestricted (in the sense of [L]) SLD-refutation D_1 of G_0 (that is in the step $G_{i-1} \Rightarrow_{Ci,\theta i\tau} G_i\tau$ we do not use an mgu), which is shorter than D. Using the Mgu Lemma of [L], we have an SLD-refutation D_2 of G_0 with the same length as D_1 and a computed answer substitution $\sigma_2 \leq \theta_1\ldots\theta_i\tau\theta_{k+1}\ldots\theta_l$. (Lemma 4.5 and the proof of the Mgu Lemma show that for every SLD-tree containing D, such a derivation D_2 can be constructed.) D_2 is an SLD-refutation of G_0 which is shorter than D, so by the induction hypothesis there exists an SLD-derivation D_3 of G_0 with computed answer substitution σ_3 such that $G_0\sigma_3 \leq G_0\sigma_2$ and D_3 is not pruned by EIR_M. Now we can take $D'=D_3$ and we have $G_0\sigma' = G_0\sigma_3 \leq G_0\sigma_2 \leq G_0\theta_1\ldots\theta_i\tau\theta_{k+1}\ldots\theta_l = G_0\theta_1\ldots\theta_i\theta_{i+1}\ldots\theta_k\theta_{k+1}\ldots\theta_l = G_0\sigma$.

ii) Note that the extra condition $G_0\theta_1\ldots\theta_k=G_0\theta_1\ldots\theta_i\tau$ was only used to prove that $G_0\sigma' \leq G_0\sigma$. □

COROLLARY 4.7. *i) EVR and EIR are shortening.*
ii) EVG and EIG are weakly sound.
PROOF. By Theorem 4.6 and Theorem 3.3.2. □

For completeness issues, it is sufficient to consider the weakest of the equality checks: the EVR_L check. We know that EVR_L is not complete - Theorem 3.4.1 presents a counterexample that holds for every simple loop check. However, for the EVR_L check this counterexample can be simplified. The program in Theorem 3.4.1 consists of a collection of ground facts and one recursive clause. Clearly, this clause is the 'core' of the counterexample. It appears that for EVR_L, we need only this clause for a demonstration of its incompleteness. Moreover, we need only the propositional structure of the clause: i.e. we may remove the arguments.

EXAMPLE 4.8.
Let $P = \{ A \leftarrow A,S \}$.
Then for 'the' SLD-tree T of $P\cup\{\leftarrow A\}$ via leftmost selection rule, $f_{EVRL}(T)$ is infinite. Indeed, every descendant of the initial goal has one occurrence of S more than its parent goal, so it cannot be a variant of any of its ancestors. □

Obviously, the problem is that the atom A in the goal is allowed to generate infinitely many S-atoms, which are never selected, thereby making the goal wider and wider. We now introduce a class of programs for which this phenomenon cannot occur and we prove that EVR_L is complete for these programs. The necessary restriction is obtained by allowing at most one recursive call per clause and allowing such a call only after all other atoms in the body of the clause have been completely resolved. In order to avoid unnecessary complications, we shall put the atom that causes the recursive call (if present) at the end of the body of the clause, and consider only derivations via the leftmost selection rule. For a formal definition, we use the notion of the *dependency graph* D_P of a program P.

DEFINITION 4.9.
The *dependency graph* D_P of a program P is a directed graph whose nodes are the predicate symbols appearing in P and
$(p,q) \in D_P$ iff there is a clause in P using p in its head and q in its body.
D_P^* is the reflexive, transitive closure of D_P. When $(p,q) \in D_P^*$, we say that p *depends on* q. For a predicate symbol p, the *class of p* is the set of predicate symbols p 'mutually depends' on: $cl_P(p) = \{q \mid (p,q)\in D_P^* \text{ and } (q,p)\in D_P^*\}$. □

DEFINITION 4.10.
Given an atom A, let *rel(A)* denote its predicate symbol.
A program P is called *restricted* if in every clause $A_0\leftarrow A_1,\ldots,A_n$ $(n\geq 0)$ of P, $rel(A_i)$ does not depend on $rel(A_0)$ for $i = 1,\ldots,n-1$. □

Note that this definition allows at most one recursive call per clause. Thus (disregarding the order of atoms in the bodies) restricted programs include so called linear programs, which contain only one recursive clause and in this clause only a single recursive call occurs. The 'transitive closure' program from

the introduction is restricted. Note also that programs of which all clauses have a body with at most one atom are restricted.

We now prove that EVR_L is complete w.r.t. leftmost selection rule for restricted programs. An interesting feature of restricted programs is that in each SLD-derivation, goals have a number of atoms which is bounded in advance. We shall show that this implies that modulo the 'being a variant of' relation, the number of possible goals in a given SLD-derivation is finite.

In the rest of this section, P is a restricted program without function symbols and G is a goal in L_P. By the *length* of a goal G, $|G|$, we mean the number of atoms of G. The maximum length of the goals in a derivation is easily predictable from the program and the initial goal. This can be done by defining (simultaneously) the *weight*-function on goals and predicate symbols (or rather the classes of predicate symbols).

DEFINITION 4.11.
Let P be a restricted program.
Then the function *weight* is defined as:
i) for a goal $G=\leftarrow A_1,\ldots,A_n$ $(n\geq 1)$ in L_P,
$\quad$ $\text{weight}(G) = \max\{\text{weight}(\text{rel}(A_i))+n-i \mid i=1,\ldots,n\}$;
ii) for a predicate symbol p of P, weight(p) =
$$\max(\{\text{weight}(\leftarrow A_1,\ldots,A_n) \mid A\leftarrow A_1,\ldots,A_n \in P,\ n>0,\ \text{rel}(A)\in \text{cl}_P(p),\ (\text{rel}(A_n),p)\notin D_P^*\} \cup \{1+\text{weight}(\leftarrow A_1,\ldots,A_{n-1}) \mid A\leftarrow A_1,\ldots,A_n \in P,\ n>1,\ \text{rel}(A)\in \text{cl}_P(p),\ (\text{rel}(A_n),p)\in D_P^*\} \cup \{1\}).$$ □

Note that in the definition of weight(p), clauses of the form $A\leftarrow B$, with $\text{cl}(\text{rel}(A))=\text{cl}(\text{rel}(B))$ are not considered - they do not affect the length of goals appearing in a derivation. Moreover, if the predicate symbols p and q are mutually dependent, then weight(p)=weight(q).

The fact that P is restricted ensures that the weight-function is well-defined: if weight(p) is defined in terms of weight(q), then $(q,p)\notin D_P^*$, hence weight(q) is not defined in terms of weight(p). Intuitively, the weight of a goal G majorizes the length of all goals which appear in an SLD-derivation of $P\cup\{G\}$ using leftmost selection rule. More precisely, we have the following lemma's.

LEMMA 4.12. *$|G|\leq$weight(G).*
PROOF. Let $G=\leftarrow A_1,\ldots,A_n$ $(n\geq 1)$. Then $\text{weight}(G) \geq \text{weight}(\text{rel}(A_1))+n-1 \geq n = |G|$. □

LEMMA 4.13. *Let $G \Rightarrow_C H$ be a derivation step w.r.t. P. Then weight(G) $\geq$ weight(H).*
PROOF. Since the weight of a goal does only depend on the predicates appearing in it, and not on the arguments of these predicates, we prove this fact for the case of programs written in the propositional logic. Let $G = \leftarrow A_1,\ldots,A_n$; then $\text{weight}(G) = \max\{\text{weight}(A_i)+n-i \mid i=1,\ldots,n\}$, and let $C = A_1\leftarrow B_1,\ldots,B_m$. Then the goal $H = \leftarrow B_1,\ldots,B_m,A_2,\ldots,A_n$ and therefore $\text{weight}(H) = \max(\{\text{weight}(B_i)+m+n-1-i \mid i=1,\ldots,m\} \cup \{\text{weight}(A_{i-m+1})+m+n-1-i \mid i=m+1,\ldots,m+n-1\}) = \max(\{\text{weight}(B_i)+m+n-1-i \mid i=1,\ldots,m\} \cup \{\text{weight}(A_i)+n-i \mid i=2,\ldots,n\})$. Two cases arise.
i) $\text{weight}(H) = \max\{\text{weight}(A_i)+n-i \mid i=2,\ldots,n\}$.
$\quad$ Then clearly $\text{weight}(H) \leq \text{weight}(G)$.

ii) weight(H) = max{weight(B_i)+m+n-1-i | i=1,...,m} (hence m>0). We will show that in this case weight(H) ≤ weight(A_1)+n-1 (which is ≤ weight(G)). Subtracting n-1, it suffices to show that max{weight(B_i)+m-i | i=1,...,m} ≤ weight(A_1). Again two cases arise.

iia) (rel(B_m),rel(A_1))∉D_P^*. Then because of the existence of C, weight(A_1) ≥ weight(←B_1,...,B_m) = max{weight(B_i)+m-i | i=1,...,m}.

iib) (rel(B_m),rel(A_1))∈D_P^*. Then weight(A_1) ≥ 1+weight(←B_1,...,B_{m-1}) = 1+max{weight(B_i)+m-1-i | i=1,...,m-1} = max{weight(B_i)+m-i | i=1,...,m-1}. Also weight(A_1) = weight(B_m)+m-m, since rel(B_m) ∈ clp(rel(A_1)). Now we have proven the claim that max{weight(B_i)+m-i | i=1,...,m} ≤ weight(A_1). □

COROLLARY 4.14. *Let D = $G_0 \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \ldots \Rightarrow G_i \Rightarrow \ldots$ be an SLD-derivation.Then for every goal G_i in D: $|G_i| \leq weight(G_0)$.*

PROOF. By induction on i. The induction basis is provided by Lemma 4.12, the induction step by Lemma 4.13. □

So weight(G_0) is the desired maximum length of goals occurring in any SLD-derivation of P∪{G_0}. Now we shall formalize the 'being a variant of' relation on resultants.

DEFINITION 4.15.
We define the relation ~ as the 'being a variant of' relation on resultants. Let G be a goal and let k≥1. Then $\sim_{G,k}$ stands for the restriction of the relation ~ to resultants G_1←G_2 such that G_1 is an instance of G and $|G_2|$≤k. □

LEMMA 4.16. *For every goal G and k≥1, $\sim_{G,k}$ is an equivalence relation.*
PROOF. Straightforward. □

The following lemma is crucial for our considerations.

LEMMA 4.17. *Suppose that the language L has no function symbols and finitely many predicate symbols. Then for every goal G and k≥1, the relation $\sim_{G,k}$ has only finitely many equivalence classes.*
PROOF. The proof is straightforward. It will appear in [BAK]. □

We can now prove the desired theorem.

THEOREM 4.18. *The loop check EVR_L is complete w.r.t. leftmost selection rule for restricted programs.*
PROOF. Let P be a restricted program and let G_0 be a goal in L_P. Let k=weight(G_0). Consider an infinite SLD-derivation D = $G_0 \Rightarrow_{C1,\theta 1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{Ck,\theta k} G_k \Rightarrow \ldots$ of P∪{G_0}. By Corollary 4.14 for every i≥0: $|G_i|$≤k. Every goal G_i is a goal in L_P and hence every resultant $G_0\theta_1\ldots\theta_i$←$G_i$ belongs to an equivalence class of $\sim_{G0,k}$. Since L_P satisfies the conditions of Lemma 4.17, $\sim_{G0,k}$ has only finitely many equivalence classes, so for some i≥0 and j>i ($G_0\theta_1\ldots\theta_i$←$G_i$) and $G_0\theta_1\ldots\theta_j$←$G_j$ are variants. This implies that D is pruned by EVR_L. □

COROLLARY 4.19. *All equality checks are complete w.r.t. leftmost selection rule for restricted programs.*
PROOF. By Theorem 4.18 and Theorem 3.3.2. □

Now combining Corollary 3.2.3 and Corollary 3.2.4 with Corollary 4.7 and Corollary 4.19, we conclude that all equality checks lead to an implementation of CWA for restricted programs without function symbols. Moreover, the equality checks based on resultants also lead to an implementation of query processing for these programs.

References

[AvE] K.R. APT and M.H. VAN EMDEN, *Contributions to the Theory of Logic Programming*, J. ACM, vol. 29, No. 3, 1982, 841-862.

[B] Ph. BESNARD, *Sur la Detection des Boucles Infinies en Programmation en Logique*, in: Actes "Séminaire de Programmation en Logique", Trégastel, 1985 (in French).

[BAK] R.N. BOL, K.R. APT and J.W. KLOP, *An Analysis of Loop Checking Mechanisms for Logic Programs*, Technical Report, Centre of Mathematics and Computer Science, Amsterdam, 1989. (In preparation)

[BW] D.R. BROUGH and A. WALKER, *Some Practical Properties of Logic Programming Interpreters*, in: Proceedings of the International Conference on Fifth Generation Computer Systems, (ICOT eds), 1984, 149-156.

[Cl] K.L. CLARK, *Negation as Failure*, in: Logic and Data Bases, (H. Gallaire and J. Minker, eds), Plenum Press, New York, 1978, 293-322.

[CL] C.L. CHANG and R.C. LEE, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[CM] W. CLOCKSIN and C. MELLISH, *Programming in PROLOG*, Springer-Verlag, New York, 1981.

[Co] M.A. COVINGTON, *Eliminating Unwanted Loops in PROLOG*, SIGPLAN Notices, Vol. 20, No. 1, 1985, 20-26.

[vG1] A. VAN GELDER, *Efficient Loop Detection in PROLOG using the Tortoise-and-Hare Technique*, J. Logic Programming 4:23-31 (1987).

[vG2] A. VAN GELDER, *Negation as Failure Using Tight Derivations for General Logic Programs*, in: Foundations of Deductive Databases and Logic Programming (J. Minker ed), Morgan Kaufmann, Los Altos, 1988, 149-176.

[K] K. KUNEN, *Some remarks on the Completed Database*, Technical report, Computer Sciences Department, University of Wisconsin, Madison, U.S.A., 1988.

[L] J.W. LLOYD, *Foundations of Logic Programming*, Second Edition, Springer-Verlag, Berlin, 1987.

[LS] J.W. LLOYD and J.C. SHEPHERDSON, *Partial Evaluation in Logic Programming*, Technical Report CS-87-09, Dept. of Computer Science, University of Bristol, 1987.

[PG] D POOLE and R. GOEBEL, *On Eliminating Loops in PROLOG*, SIGPLAN Notices, Vol. 20, No. 8, 1985, 38-40.

[R] R. REITER, *On Closed World Data Bases*, in: Logic and Data Bases, (H. Gallaire and J. Minker, eds), Plenum Press, New York, 1978, 55-76.

[SGG] D.E. SMITH, M.R. GENESERETH and M.L. GINSBERG, *Controlling Recursive Inference*, Artificial Intelligence 30:343-389 (1986).