AN ANALYSIS OF LOOP CHECKING MECHANISMS FOR LOGIC PROGRAMS

Roland N. Bol,¹ Krzysztof R. Apt,¹,²
and Jan Willem Klop¹,³

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188

TR-89-32    October 1989

¹Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.
²Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712-1188, USA.
³Department of Computer Sciences, Free University of Amsterdam, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands.
An Analysis of Loop Checking Mechanisms for Logic Programs

Roland N. Bol\textsuperscript{1}
Krzysztof R. Apt\textsuperscript{1,2}
Jan Willem Klop\textsuperscript{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 \textit{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 natural classes of PROLOG programs without function symbols for which they are complete. In these classes a limited form of recursion is allowed. As a by-product we obtain an implementation of the closed world assumption of Reiter [19] and a query evaluation algorithm for these classes of logic programs.

1. Introduction

1.1. Motivation
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

\textsuperscript{1} Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands
\textsuperscript{2} Department of Computer Sciences, University of Texas at Austin,
Austin, Texas 78712-1188, USA
\textsuperscript{3} Department of Computer Sciences, Free University of Amsterdam
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
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. However, the resulting program can be very different from the original specification.

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. That is why pruning SLD-trees has been called loop checking. Such modifications of PROLOG interpreters were considered in the literature (see e.g. [3], [4], [8], [18], [20], [21] and [22]), but no results were proved about them, with notable exceptions of [20], [21] and [22].

1.2. Plan of the paper
In this paper we systematically study loop checking mechanisms. To this end, after providing in section 2 a sufficiently general definition of a loop check, we introduce in section 3 the relevant concepts, like soundness (no computed answer substitution to a goal is missed), completeness (all resulting derivations are finite) and relative strength. We also introduce there a natural subclass of loop checks, 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.

In the remainder of the paper we study a number of intuitive simple loop checks. We can divide them into three groups, which are studied in the sections 4, 5 and 6 respectively. For each group we prove the appropriate soundness results and identify one or more natural classes of programs without function symbols for which the loop checks in the group are complete. The loop checks in all three groups appear to be complete for restricted programs without function
symbols. Restricted programs allow a restricted form of recursion (hence the name).

The first group consists of loop checks based on the equality between goals, respectively resultants, of the derivations and is studied in section 4. We call these loop checks equality checks.

The second group loop checks based on the inclusion between goals, respectively resultants, of the derivations and is studied in section 5. We call these loop checks subsumption checks. Subsumption checks are stronger than the corresponding equality checks and therefore they prune SLD-derivations earlier than their counterparts. This makes it more difficult to establish their soundness but opens a possibility for completeness for a larger class of programs than restricted ones.

We show that subsumption checks are complete for logic programs without function symbols in which no variables are introduced in the clause bodies (so called nvi programs). Also, the subsumption checks are complete for logic programs without function symbols in which a variable occurs at most once in every clause body (so called svo programs). These completeness theorems make use of a simple version of Kruskal's Tree Theorem, called Higman's Lemma [12]. While the use of this theorem to establish termination of term rewriting systems is well-known (see e.g. [9] or [13]), we have not encountered any applications of this theorem in the area of logic programming.

The third group is based on a simple loop check introduced by Besnard [3] and is studied in section 6. These checks test for equality of atoms in a certain context (a goal or a resultant). Therefore we call them context checks. We prove that for certain selection rules, the subsumption checks are stronger than the context checks.

As mentioned above, we prove that context checks are complete for restricted programs without function symbols. Besnard [3] claims without proof that a context check is also complete for nvi programs without function symbols.

**1.3. Example**

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:
\[ P = \{ \text{tc}(x,y) \leftarrow r(x,y). \]
\[ \quad \text{tc}(x,y) \leftarrow r(x,z), \text{tc}(z,y). \} \]

Suppose we add to \( P \) the following facts about \( r \): \( r(a,a) \leftarrow, r(a,b) \leftarrow, r(b,c) \leftarrow, r(d,a) \leftarrow \). Then if we ask:
- \( \text{tc}(a,b) \) we get the answer 'yes';
- \( \text{tc}(a,c) \) the program gets into an infinite loop (whereas we should get the answer 'yes');
- \( \text{tc}(a,d) \) the program gets into an infinite loop (whereas we should get the answer 'no');
- \( \text{tc}(b,d) \) 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 [7], section 7.2. In fact, Kunen [14] 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 an equality 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 \( \text{tc}(x,y) \leftarrow \text{tc}(x,z), \text{tc}(z,y) \), as the resulting program is not any more restricted.)

### 1.4. Applications

As a by-product of these considerations we obtain an implementation of the closed world assumption of Reiter [19] and of a query evaluation mechanism for various classes of definite deductive databases. The closed world assumption (CWA in short) is a way of inferring negative information in deductive databases. Reiter [19] 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 \( \varphi \) 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. The soundness and completeness results proved in section 4 show that when DB is a restricted program, to infer \( \neg A \) for a
fact A it suffices to use Clark's [5] negation as (finite) failure rule augmented with an appropriate equality 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. The results proved in section 4 imply that 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 \(\leftarrow A\) as root, pruned by an appropriate equality check.

Similar results concerning CWA and query processing hold for the subsumption and context checks and the corresponding classes of programs for which they are complete.

This paper is an extension of Apt, Bol & Klop [1], where exclusively equality checks were studied.

2. Loop checking

Throughout this paper we assume familiarity with the basic concepts and notations of logic programming as described in [15]. 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 \(E ≤ F\) when F is an instance of E. We then say that F is less general than E.

Throughout this paper, by an SLD-derivation we mean an SLD-derivation in the sense of [15] or an initial fragment of it. In SLD-derivations we shall only use idempotent mgu's. It is known that any idempotent mgu is relevant, i.e. its domain contains only variables of the atoms that are unified. An SLD-derivation step from a goal G, using a clause C and an idempotent mgu θ, to a goal H is denoted as \(G ⇒_{C,θ} H\).

2.1. Definitions

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 here a more restricted definition according to which for a program P:
- a node in an SLD-tree of \( P \cup \{ G \} \) (for some goal \( G \)) is \textit{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 depends only upon its ancestors in the SLD-tree, that is on the SLD-derivation from the root up to this node.

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 pruning in an SLD-tree the nodes in \( \{ G \mid \text{the SLD-derivation from the root to } G \text{ is in } L(P) \} \). In the remainder of this article, we shall usually make this conversion implicitly.

It is useful to note here that our definition of a loop check excludes more complicated pruning mechanisms for which the decision whether a node in a tree is pruned depends on the so far traversed fragment of the considered tree. Such mechanisms are for example studied in Vieille [22] and Seki & Itoh [21].

We shall also study an even more restricted form of a 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.

\textbf{Definition 2.1.}

Let \( L \) be a set of SLD-derivations.

\[ RemSub(L) = \{ D \in L \mid \text{L does not contain a proper subderivation of } D \} \]

\( L \) is \textit{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 \textit{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 \textit{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 programs clause are 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 [16]). Thus any variant of an SLD-refutation is also an SLD-refutation and yields the same computed answer substitution up to a renaming.

**DEFINITION 2.2.**

A *simple loop check* is a computable set L of finite SLD-derivations such that
- for every derivation D: if D ∈ L then for every variant D' of D: D' ∈ 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. □

Of course, we can treat a simple loop check L as a loop check, namely as the constant function λP.L.

**DEFINITION 2.4.**

Let L be a loop check. An SLD-derivation D of P∪{G} is pruned by L if L(P) contains a subderivation D' of D. □

2.2. Example

**EXAMPLE 2.5 (Variant of Atom check).**

(This example is based on Example 8 in [3], see also [10]).

A first attempt to formulate the *Variant of Atom (VA)* 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 $\leftarrow B, A \Rightarrow B \leftarrow A$ shows that the first formulation of the VA check is not precise enough: it does not capture the intuition that the proof of $A'$ goes via $A$. The atom $A$ 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 $VA =$ RemSub($\{ D | D = (G_0 \Rightarrow C_1, \theta_1 G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow C_k, \theta_k G_k) \}$ such that for some $i$ and $j$, $0 \leq i \leq j < k$, $G_k$ 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 now illustrate the use of this loop check.

Let $P = \{ A(0) \leftarrow (C1), B(1) \leftarrow (C2), A(x) \leftarrow A(y) (C3), C \leftarrow A(x), B(x) (C4) \}$,

let $G = \leftarrow C$.

That the informal justification of the loop check $VA$ is incorrect, is shown by applying it to two SLD-trees of $P \cup \{ G \}$, via the leftmost and rightmost selection rule respectively, which gives us Figure 2.1. (In this figure and elsewhere 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 $G_3 = \leftarrow A(y')$ in the rightmost tree is pruned by the VA 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 VA 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 (Soundness).**

i) A loop check $L$ is weakly sound if for every program $P$ and goal $G$, and 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 goal $G$, and SLD-tree $T$ of $P \cup \{G\}$: if $T$ contains a successful branch with a computed answer substitution $\sigma$, then $f_{L(P)}(T)$ contains a successful branch with a computed answer substitution $\sigma'$ such that $\sigma' \leq \sigma$.

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

The following lemma is an immediate consequence of these definitions.
LEMMA 3.2. 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 3.3 (Completeness).
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 any answer it gives 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. For
example, it is not true that adding a complete loop check to a complete interpreter
yields again a complete interpreter.

These relationships are expressed in the following lemma’s. 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 (but not complete w.r.t. CWA).

LEMMA 3.4. 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.
PROOF. We know by the soundness and strong completeness of SLD-resolution (see [2],[15]) that \( P \vdash_{\text{CWA}} \neg A \iff P \not\vdash A \iff T \) contains no successful branch.
\[ \Rightarrow T \] contains no successful branch and \( f_L(P)(T) \) is a subtree of \( T \), so \( f_L(P)(T) \) contains no successful branch either.
\[ \Leftarrow \] Since \( L \) is weakly sound, a successful branch in \( T \) would yield a successful branch in \( f_L(P)(T) \). But \( f_L(P)(T) \) contains no successful branch, hence \( T \) contains no successful branch either. \( \square \)

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 3.5.** 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 \( \theta \), \( P \vdash A \theta \) iff \( f_L(P)(T) \) contains a successful branch with a computed answer substitution \( \tau \) such that \( \tau \leq \theta \).

PROOF. We have by the strong completeness of SLD-resolution \( P \vdash A \theta \iff T \) contains a successful branch with a computed answer substitution \( \sigma \) such that \( \sigma \leq \theta \).
\[ \Rightarrow T \] contains this successful branch, and since \( L \) is sound, \( f_L(P)(T) \) contains a successful branch with a computed answer substitution \( \tau \) such that \( \tau \leq \sigma \). Hence \( \tau \leq \theta \).
\[ \Leftarrow f_L(P)(T) \] contains a successful branch with a computed answer substitution \( \tau \leq \theta \), so \( T \) contains this branch as well. \( \square \)

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

**Corollary 3.6.** 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 \( P \cup \{ \leftarrow A \} \), \( P \vdash_{\text{CWA}} \neg A \) iff \( f_L(P)(T) \) is finite and contains no successful branches.

PROOF. By Lemma 3.4 and the Completeness Definition 3.3. \( \square \)

Thus an interpreter augmented with a weakly sound and complete loop check becomes complete w.r.t. CWA.
**Corollary 3.7.** 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 $\theta$, $P \vdash A\theta$ iff $f_L(P)(T)$ is finite and contains a successful branch with a computed answer substitution $\tau$ such that $\tau \leq \theta$.

**Proof.** By Lemma 3.5 and the Completeness Definition 3.3. \hfill \Box

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 themselves. 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 3.8.**

Let $L_1$ and $L_2$ be loop checks.

$L_1$ is stronger than $L_2$ if for every program $P$ and goal $G$, every SLD-derivation $D_2 \in L_2(P)$ of $P \cup \{G\}$ contains a subderivation $D_1$ such that $D_1 \in L_1(P)$. \hfill \Box

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 every loop check 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 3.9 (Relative Strength).** 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.** i)-iii) If an SLD-tree \( T \) contains a successful branch, then \( f_{L_1}(T) \) contains a successful branch that satisfies the conditions of Definition 3.1. Since \( L_1 \) is stronger than \( L_2 \), \( f_{L_1}(T) \) is a subtree of \( f_{L_2}(T) \), so this branch is also contained in \( f_{L_2}(T) \).

iv) Every infinite SLD-derivation is pruned by \( L_2 \), so it is also pruned by \( L_1 \). □

Now we have a clearer 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

A question now arises: 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 when studying completeness we shall rule out programs that compute over an infinite domain. We shall do so by restricting our attention to programs without function symbols, so called function-free programs. This restriction leads to a finite Herbrand Universe, but other solutions (typed functions, bounded term-size property [11]) are also possible here.

Note that our definitions so far referred to arbitrary programs and SLD-derivations. In the sequel, we shall consider certain classes of programs (like function-free programs) and SLD-derivations (like the derivations via the 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. the leftmost selection rule for function-free restricted programs’.

As stated above, we shall study completeness only for function-free programs. So our question can be reformulated as: is there a sound and complete loop check for function-free programs? Before answering this question for loop checks in general, we shall answer it for simple loop checks.

**Theorem 3.10.** There is no weakly sound and complete simple loop check for function-free programs.

**Proof.** Let L be a simple loop check that is complete for function free programs. Consider the following infinite SLD-derivation D, obtained by repeatedly using the clause \( A(x) \leftarrow A(y), S(y, x) \) (using the leftmost selection rule).

\[
\begin{align*}
\leftarrow & A(x_0), B(x_0) \\
\downarrow \\
\leftarrow & A(x_1), S(x_1, x_0), B(x_0) \\
\downarrow \\
\leftarrow & A(x_2), S(x_2, x_1), S(x_1, x_0), B(x_0) \\
\downarrow \\
\leftarrow & A(x_3), S(x_3, x_2), S(x_2, x_1), S(x_1, x_0), B(x_0) \\
\downarrow \\
\vdots
\end{align*}
\]

**Figure 3.1**

Since L is a complete loop check, this derivation is pruned by L and since L is simple, the goal at which pruning takes place is independent of the program used for this derivation. Suppose this derivation is pruned by L at the goal \( \leftarrow A(x_n), S(x_n, x_{n-1}), \ldots, S(x_1, x_0), B(x_0) \).

Now let \( P = \{ S(i, i+1) \leftarrow \mid 0 \leq i < n \} \cup \{ A(0) \leftarrow, A(x) \leftarrow A(y), S(y, x), B(n) \leftarrow \} \). Extending the above derivation to an SLD-tree of \( P \cup \{ G \} \) (still using the leftmost selection rule, see Figure 3.2), we see that every goal of the derivation has two descendents, obtained by applying the clauses \( A(x) \leftarrow A(y), S(y, x) \) and \( A(0) \leftarrow \) respectively. The derivation of Figure 3.1 shows
the effect of repeatedly applying \( A(x) \leftarrow A(y), S(y, x) \). After applying \( A(0) \leftarrow \) at some goal, a derivation becomes deterministic: if there are initially \( m \) \( S \)-atoms, then these atoms are resolved from left to right by the clauses \( S(0,1) \leftarrow \ldots, S(m-1,m) \leftarrow \).

\[
\begin{align*}
&\leftarrow A(x_0), B(x_0) \quad \Rightarrow \quad \leftarrow B(0) \\
&\downarrow \\
&\leftarrow A(x_1), S(x_1,x_0), B(x_0) \quad \Rightarrow \quad \leftarrow S(0,x_0), B(x_0) \quad \Rightarrow \quad \leftarrow B(1) \\
&\downarrow \\
&\leftarrow A(x_2), S(x_2,x_1), S(x_1,x_0), B(x_0) \quad \Rightarrow \quad \leftarrow S(0,x_1), S(x_1,x_0), B(x_0) \\
&\quad \quad \quad \quad \Rightarrow \quad \leftarrow S(1,x_0), B(x_0) \quad \Rightarrow \quad \leftarrow B(2) \\
&\downarrow \\
&\ldots \\
&\downarrow \\
&\leftarrow A(x_n), S(x_n,x_{n-1}), \ldots, S(x_1,x_0), B(x_0) \Rightarrow \ldots \text{n intermediate goals...} \Rightarrow \quad \leftarrow B(n) \\
&\downarrow \\
&\ldots
\end{align*}
\]

**FIGURE 3.2**

Finally, the goal \( \leftarrow B(m) \) is left. Since of all goals of the form \( \leftarrow B(i) \) \((i \geq 0)\) only the goal \( \leftarrow B(n) \) can be refuted, exactly \( n \) \( S \)-atoms are needed. Therefore the only successful branch of this SLD-tree of \( P \cup \{G\} \) goes via the goal \( \leftarrow A(x_n), S(x_n,x_{n-1}), \ldots, S(x_1,x_0), B(x_0) \). As exactly this goal is pruned by \( L \), \( L \) has pruned the only successful branch of this SLD-tree. Hence \( L \) is not weakly sound.

However, taking the program into account gives us an opportunity to define for function-free programs 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. One can save effort if one can avoid the trials that lead to an error. Assuming that the puzzle is solvable (as our 'puzzle', finding the correct answers to a given goal, is), it is possible to find out exactly which trials
to avoid. How this can be done is formalized in the proof of Theorem 3.13 (1). However, solving the puzzle is the first step of the method described, so it can only be of theoretical importance.

For convenience, we shall write $S(P,G,\sigma)$ for the set of successful SLD-derivations of $P \cup \{G\}$ with a computed answer substitution $\tau$ such that $\tau \leq \sigma$. We say that a derivation $D$ is a shortest derivation in $S(P,G,\sigma)$ if $D \in S(P,G,\sigma)$ and $|D| = \min \{ |D'| \mid D' \in S(P,G,\sigma) \}$.

**Definition 3.11 (STRONG check).**
For a function-free program $P$, $\text{STRONG}(P) = \text{RemSub}(\{ D = G \Rightarrow \ldots \mid \text{for no } \sigma, D \text{ is an initial fragment of a shortest derivation in } S(P,G,\sigma) \})$. $\Box$

Note that an SLD-tree pruned by STRONG consists not only of the shortest refutation(s) of $P \cup \{G\}$ for any computed answer substitution $\sigma$, but also of the derivations that follow the path of such a derivation but 'make a wrong decision', that is a step deviating from such a refutation. After such a step, the derivation is immediately pruned by STRONG. This effect is a consequence of the fact that pruning a node in a tree implies removing all descendents, so we cannot remove the descendents caused by a 'wrong step' while retaining the others. The following example shows the effect of pruning an SLD-tree by STRONG.

**Example 3.12.**
Let $P = \{ A(1) \leftarrow \text{(C1)},
A(y) \leftarrow B(y,z), A(z) \text{(C2)},
B(w,0) \leftarrow \text{(C3)},
B(0,1) \leftarrow \text{(C4)} \}$,
and let $G = \leftarrow A(x)$.

Consider an SLD-tree of $P \cup \{G\}$ displayed in Figure 3.3. In $S(P,G,\{x/1\})$ a shortest derivation has 2 goals, in $S(P,G,\{x/0\})$ a shortest derivation has 4 goals and in $S(P,G,e)$ a shortest derivation has 6 goals. These derivations are retained by STRONG in the considered SLD-tree, the others are pruned (at the horizontal lines in the figure). Among these are successful ones, but not shortest ones. (The tree in Figure 3.3 is extended beyond the sixth level to show this effect.) $\Box$
THEOREM 3.13. For function-free programs:

i) STRONG is a shortening loop check.

ii) STRONG is stronger than any shortening loop check.

iii) STRONG is complete.

PROOF. i a) STRONG is a loop check.

The non-trivial point here is to prove that for every function-free program P, STRONG(P) is computable. Can we, given a derivation \( D = G \Rightarrow \ldots \), decide whether or not \( D \) is pruned by STRONG and if so, at which node? Indeed we can, using the following procedure.

1. Compute the set of correct answer substitutions for \( P \cup \{ G \} \) (e.g. bottom up). Since \( P \) has no function symbols, this set is finite. Construct (breadth first) an initial fragment of an SLD-tree of \( P \cup \{ G \} \) that contains (an initial part of) \( D \) and for each correct answer substitution a successful branch with a more general computed answer substitution. Such a fragment exists by the strong completeness of SLD-resolution. It has been shown in [17] that a length preserving bijection exists between the successful branches of two different SLD-trees for \( P \cup \{ G \} \). Therefore in every SLD-tree of \( P \cup \{ G \} \), for every correct answer substitution \( \sigma \) there exists a derivation \( D' \in S(P,G,\sigma) \) with \( |D'| = \min \{ |D''| \mid D'' \in S(P,G,\sigma) \} \).

2. For each computed answer substitution, mark the nodes of the shortest successful branches with this computed answer substitution.

3. Prune \( D \) at the first node in the tree that is not marked. If such a node does not exist, then \( D \) is a subderivation of a shortest successful branch.

   i b) STRONG is shortening.

If a successful derivation \( D \) of \( P \cup \{ G \} \) with computed answer substitution \( \sigma \) is pruned by STRONG, then it is not a shortest derivation in \( S(P,G,\sigma) \). By construction, there exists a shortest derivation \( D' \in S(P,G,\sigma) \) in the SLD-tree. \( D' \) is shorter than \( D \) and not pruned by STRONG.

   ii) STRONG is stronger than any shortening loop check.

Let \( L \) be a loop check and let \( D \) be a derivation of \( P \cup \{ G \} \) that is pruned by \( L \). If \( D \) is a subderivation of a shortest successful derivation \( D' \), then \( L \) is not shortening. Otherwise, \( D \) is pruned by STRONG.

   iii) STRONG is complete.

If \( D \) is an infinite SLD-derivation, then only an initial fragment of \( D \) is contained in the constructed (finite) part of the SLD-tree. Since the last goal of \( D \) that is in
the tree is not successful, D contains a ‘wrong step’ there or earlier. Hence D is pruned by STRONG. □

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 (not even for function-free programs). Nevertheless, for each of these loop checks we shall introduce one or more natural classes of programs for which they are complete.

4. Equality checks

4.1. Overview
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 additional condition to the first one. By this construction, the first version is always stronger than the corresponding second version.

Starting with the Variant of Atom check, we can make three independent modifications of it.

1. Adding this additional 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 $G_0 \Rightarrow_{C_1, \theta_1} G_1 \Rightarrow \ldots$, 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 [16].

2. Replace variant by instance. This yields the Instance of Atom (IA) check. This check is still unsound: it is even stronger than the VA check. Besnard [3] has introduced a weakly sound version of this loop check. This check and related ones (derived from VA; shortening versions) are discussed in section 6.

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).
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. They are discussed in the remainder of this section.

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. [6]). Subsumption can also be defined for resolvents.

This yields the subsumption check. Since this modification is again independent of the others, there are also eight subsumption checks. These checks are discussed in section 5.

4.2. Formal definitions

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 additional 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 (Equality checks for Goals).**
For Type $\in \{L,M\}$, the $Equals$ Variant/Instance of GoalType check is the set of SLD-derivations

$$EVG/EIG_{Type} = \text{RemSub}(\{ D \mid D = ( G_{0} \Rightarrow_{C_{1},\theta_{1}} G_{1} \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_{k},\theta_{k}} G_{k} ) \text{ such that for some } i, 0 \leq i < k, \text{ there is a renaming/substitution } \tau \text{ such that } G_{k} =_{Type} G_{i}\tau \} ). \square$$

For example,

$$EIG_{M} = \text{RemSub}(\{ D \mid D = ( G_{0} \Rightarrow_{C_{1},\theta_{1}} G_{1} \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_{k},\theta_{k}} G_{k} ) \text{ such that for some } i, 0 \leq i < k, \text{ there is a substitution } \tau \text{ such that } G_{k} =_{M} G_{i}\tau \} ).$$

The informal justification for these loop checks is similar to the one given for the VA check. Suppose that we want to refute a goal $G$. If we find that in order to refute $G$ we need to refute a variant or instance of $G$, say $G\tau$, then two cases arise. If there is no solution for $G\tau$, then pruning $G\tau$ is clearly safe. On the other hand, if there is a solution for $G\tau$, then the derivation giving this solution might be used (possibly in a more general form) directly from $G$.

We shall prove later in this section that these loop checks are weakly sound. However, they are not sound. To see this, suppose that we find for $G\tau$ a successful derivation $D$ with a computed answer substitution $\sigma$. Then using $D$ directly from $G$ gives a computed answer substitution $\tau\sigma$ (maybe a more general substitution, but not necessarily). Therefore success is not lost.

However, the derivation $G = G_{i} \Rightarrow_{C_{i+1},\theta_{i+1}} \ldots \Rightarrow_{C_{k},\theta_{k}} G_{k} = G\tau$, followed by $D$, yields a possibly different computed answer substitution: $\theta_{i+1} \ldots \theta_{k}\sigma$, thus possibly affecting soundness. (In Example 4.3, we show a specific program and goal for which this difference arises.) Of course, we are only interested in the effect of this difference on the variables of the initial goal $G_{0}$. In $G_{i}$ these variables are renamed to $G_{0}\theta_{1} \ldots \theta_{i}$. So $\tau$ and $\theta_{i+1} \ldots \theta_{k}$ should coincide on the variables of $G_{0}\theta_{1} \ldots \theta_{i}$.

Hence we can make these loop checks sound, and even shortening, by adding the condition $G_{0}\theta_{1} \ldots \theta_{k} = G_{0}\theta_{1} \ldots \theta_{i}\tau$. (Note that 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.

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

**Definition 4.2 (Equality checks for Resultants).**

For $Type \in \{L,M\}$, the **Equals Variant/Instance of Resultant**$_{Type}$ check is the set of SLD-derivations

$$EVR/EIR_{Type} = \text{RemSub} \{ D \mid D = ( G_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow G_k )$$

such that for some $i$, $0 \leq i < k$, there is a renaming/ substitution $\tau$ such that $G_k \models_{Type} G_i \tau$
and $G_0 \gamma_1 \ldots \gamma_k = G_0 \gamma_1 \ldots \gamma_i \tau \}$. \Box

For example,

$$EVR_L = \text{RemSub} \{ D \mid D = ( G_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow \gamma_k )$$
such that for some $i$, $0 \leq i < k$, there is a renaming $\tau$ such that $G_k = L, G_i \tau$ and $G_0 \gamma_1 \ldots \gamma_k = G_0 \gamma_1 \ldots \gamma_i \tau \}$$

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

**Example 4.3.**

Let $P = \{ p(a) \leftarrow, \quad (C1)$

$p(y) \leftarrow p(z) \quad (C2) \}$,

let $G = \leftarrow p(x)$.

Without the condition $G_0 \gamma_1 \ldots \gamma_k = G_0 \gamma_1 \ldots \gamma_i \tau$ we would only obtain the
computed answer substitution $\{ x/a \}$, whereas we should also obtain the empty
substitution. This shows that the EVG and EIG loop checks are not sound.

In the leftmost tree $\leftarrow p(z)$ is a variant of $\leftarrow p(x)$, so the derivation is pruned
by EVG at that goal. However, the corresponding resultant $p(x) \leftarrow p(z)$ is clearly
not a variant of $p(x) \leftarrow p(x)$, therefore the derivation is not yet pruned by EVR.
After another application of (C2), the resultant $p(x) \leftarrow p(z')$ occurs, which is a
variant of $p(x) \leftarrow p(z)$. At that point the derivation is pruned by EVR.
The rightmost tree in Figure 4.1 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 \( \Box \), but by the instance of the initial goal that was 'proved' by this branch.

\[ \text{An SLD-tree of } \mathcal{P} \cup \{G\} \text{ based on goals:} \]
\[ \text{An 'SLD-tree' of } \mathcal{P} \cup \{G\} \text{ based on resultants:} \]

\[ \text{LEMMA 4.4. All equality checks are simple loop checks.} \]
\[ \text{PROOF. Straightforward.} \]

Figure 4.2. shows the 'stronger than' relationships between the equality checks (and the VA and IA checks) and summarizes their properties. In this figure, an arrow \( L_1 \rightarrow L_2 \) means that \( L_2 \) is stronger than \( L_1 \). Proving these 'stronger than' relations is straightforward.
4.3. Soundness

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 the Relative Strength Theorem 3.9 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 \( D \) is an SLD-derivation of \( P \cup \{G_1\} \) with computed answer substitution \( \sigma \). Then every SLD-tree of \( P \cup \{G_2\} \) contains a successful branch of length \( |D| \) with a computed answer substitution \( \sigma \).

**Proof.** By the soundness and strong completeness of SLD-resolution, see [15].

**Theorem 4.6.**

i) The loop check EIR_M is shortening.

ii) The loop check EIG_M is weakly sound.

**Proof.** Let \( P \) be a program and \( G_0 \) a goal.

i) Let \( D \) be an SLD-refutation of \( P \cup \{G_0\} \) with computed answer substitution \( \sigma \). If \( D \) is pruned by EIR_M then we must find in every SLD-tree containing \( D \) an
SLD-refutation $D'$ of $P \cup \{G_0\}$ with computed answer substitution $\sigma'$ such that $\sigma' \leq \sigma$, $|D'| < |D|$ and $D'$ is not pruned by $EIR_M$. We prove that $D'$ exists 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 $P \cup \{G_0\}$ of length $\leq l$. Let $D$ be a refutation of length $l+1$. Suppose that $D$ is pruned by $EIR_M$. Then we have $D = (G_0 \Rightarrow C_1, \theta_1 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow C_i, \theta_i \Rightarrow G_i \Rightarrow C_{i+1}, \theta_{i+1} \Rightarrow G_{i+1} \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow C_k, \theta_k \Rightarrow G_k \Rightarrow C_{k+1}, \theta_{k+1} \Rightarrow G_{k+1} \Rightarrow \ldots \Rightarrow \Box)$, and for some substitution $\tau$: $G_k = M \Rightarrow G_i \tau$ 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 computed answer substitution $\theta_{k+1} \ldots \theta_i$. Now we can obtain an unrestricted (in the sense of [15]) SLD-refutation $D_1 = (G_0 \Rightarrow C_1, \theta_1 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow C_i, \theta_i \Rightarrow G_i \Rightarrow \ldots \Rightarrow \Box)$ of $P \cup \{G_0\}$ (that is in the step $G_i \Rightarrow C_i, \theta_i \tau \Rightarrow G_i \tau$ we do not use an mgu), which is shorter than $D$. Using the Mgu Lemma of [15], we obtain an SLD-refutation $D_2$ of $P \cup \{G_0\}$ with the same length as $D_1$ and a computed answer substitution $\sigma_2 \leq \theta_1 \ldots \theta_i \tau \theta_{k+1} \ldots \theta_i$. (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 $P \cup \{G_0\}$ which is shorter than $D$, so by the induction hypothesis there exists an SLD-refutation $D_3$ of $P \cup \{G_0\}$ with computed answer substitution $\sigma_3$ such that $\sigma_3 \leq \sigma_2$ and $D_3$ is not pruned by $EIR_M$. Now we can take $D' = D_3$ and we have $G_0 \sigma' = G_0 \sigma_3 \leq G_0 \sigma_2 \leq G_0 \theta_1 \ldots \theta_i \tau \theta_{k+1} \ldots \theta_i = G_0 \theta_1 \ldots \theta_i \tau \theta_{k+1} \ldots \theta_i = G_0 \sigma$, so $\sigma' \leq \sigma$.

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

COROLLARY 4.7 (Equality Soundness).

i) All equality checks based on resultants are shortening. A fortiori they are sound.

ii) All equality checks based on goals are weakly sound.

PROOF. By Theorem 4.6 and the Relative Strength Theorem 3.9.

4.4. Completeness

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 3.10 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
3.10 consists of a collection of ground facts and one recursive clause. Clearly, this clause is the 'core' of the counterexample. It appears that for EVRL, 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 the 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. \( \Box \)

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 EVRL 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 place the atom that causes the recursive call (if present) at the right 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

\[(p,q) \in D_P \text{ iff there is a clause in } P \text{ using } p \text{ in its head and } q \text{ in its body.}\]

\( D_P^* \) 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: \( cl_P(p) = \{ q \mid (p,q) \in D_P^* \text{ and } (q,p) \in D_P^* \} \). \( \Box \)
DEFINITION 4.10 (Restricted Program).

Given an atom A, let rel(A) denote its predicate symbol. Let P be a program. A clause $A_0 \leftarrow A_1, \ldots, A_n$ ($n \geq 0$) is called restricted w.r.t. P if for $i = 1, \ldots, n-1$, rel($A_i$) does not depend on rel($A_0$) in P. The atoms $A_1, \ldots, A_{n-1}$ are called the non-recursive atoms of the clause $A_0 \leftarrow A_1, \ldots, A_n$.

A program P is called restricted if every clause in P is restricted w.r.t. P. \hfill \Box

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 given in 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 EVRL is complete w.r.t. the leftmost selection rule for restricted programs. An interesting feature of restricted programs is that in each SLD-derivation via the leftmost selection rule, goals have a number of atoms which is bounded by a value depending only on the program and the initial goal. We shall show that this implies that modulo the ‘being a variant of’ relation, the number of possible goals in such an SLD-derivation is finite.

In the rest of this section, P is a function-free restricted program and G is a goal in L_P. With the length of G, |G|, we mean the number of atoms in G. The maximum length of the goals in a derivation of P \cup \{G\} can be computed by means of the following weight-function, which is defined on goals and predicate symbols (by mutual induction).

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,

   weight(G) = \max\{weight(rel(A_i)) + n - i \mid i = 1, \ldots, n\}.

ii) for a predicate symbol p of P, weight(p) =

   \max( \{ weight(\leftarrow A_1, \ldots, A_n) \mid
             A \leftarrow A_1, \ldots, A_n \in P, n > 0, rel(A) \in clp(p), rel(A_n) \notin clp(p) \} \cup
   \{ 1 + weight(\leftarrow A_1, \ldots, A_{n-1}) \mid
             A \leftarrow A_1, \ldots, A_n \in P, n > 1, rel(A) \in clp(p), rel(A_n) \in clp(p) \} \cup
   \{ 1 \} ). \hfill \Box
Note that in the definition of weight(p), clauses of the form A↔B, with cl(rel(A)) = cl(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) ∉ 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∪{G} using leftmost selection rule. More precisely, we have the following lemma's.

**Lemma 4.12.** |G| ≤ weight(G).

**Proof.** Let G = ←A_1,...,A_n (n≥1). Then weight(G) ≥ weight(rel(A_1)) + n-1 ≥ n = |G|. \Box

**Lemma 4.13.** Let G ⇒_C H be a derivation step w.r.t. P. Then weight(G) ≥ weight(H).

**Proof.** Since the weight of a goal depends only 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 = ←A_1,...,A_n; then weight(G) = max{weight(A_i)+n-i | i=1,...,n}, and let C = A_1←B_1,...,B_m.

Then the goal H = ←B_1,...,B_m,A_2,...,A_n and therefore weight(H) = max({weight(B_i)+m+n-1-i | i=1,...,m} ∪ {weight(A_i+m+1)+m+n-1-i | i=m+1,...,m+n-1}) = max({weight(B_i)+m+n-1-i | i=1,...,m} ∪ {weight(A_i)+n-i | i=2,...,n}). Two cases arise.

i) weight(H) = max{weight(A_i)+n-i | i=2,...,n}.

Then clearly weight(H) ≤ weight(G).

ii) weight(H) = max{weight(B_i)+m+n-1-i | i=1,...,m} (hence m>0). We 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) \((\text{rel}(B_m), \text{rel}(A_1)) \in Dp^*\). Then weight\((A_1) \geq 1 + \text{weight}\((B_1, \ldots, B_{m-1}) - 1 + \max\{\text{weight}(B_i) + m-i \mid i=1, \ldots, m-1\} = \max\{\text{weight}(B_i) + m-i \mid i=1, \ldots, m-1\}\). Also weight\((B_m) + m-m = \text{weight}(A_1)\), since \text{rel}(B_m) \in \text{clp}(\text{rel}(A_1)). This proves the claim that \(\max\{\text{weight}(B_i) + m-i \mid i=1, \ldots, m\} \leq \text{weight}(A_1)\). \qed

**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 \(D\): \(|G_i| \leq \text{weight}(G_0)\).

**Proof.** By induction on \(i\). The induction basis is provided by Lemma 4.12, the induction step by Lemma 4.13. \qed

So weight\((G_0)\) is indeed the desired maximum length of goals occurring in any SLD-derivation of \(P \cup \{G_0\}\).

We now present a formalization of the "being a variant of" relation on resultants. Our presentation here is more general than needed for the completeness proof for the equality checks. However, we need these results in full generality to prove the completeness of the subsumption checks and the context checks.

**Definition 4.15.**
Let \(X\) be a set of variables. We define the relation \(\sim_X\) on resultants as \(R_1 \sim_X R_2\) if for some renaming \(\rho\), \(R_1 \rho = R_2\) and for every \(x \in X\), \(x \rho = x\). Now let \(G\) be a goal and let \(k \geq 1\). Then the relation \(\sim_{X,G,k}\) stands for the restriction of the relation \(\sim_X\) to resultants \(G_1 \leftarrow G_2\) such that \(G_1\) is an instance of \(G\) and \(|G_2| \leq k\). \qed

**Lemma 4.16.** For every set of variables \(X\), goal \(G\) and \(k \geq 1\), \(\sim_{X,G,k}\) is an equivalence relation.

**Proof.** Straightforward. \qed

For a resultant \(R\), the equivalence class of \(R\) w.r.t. the relation \(\sim_{X,G,k}\) will be denoted as \([R]_{X,G,k}\), or just \([R]\) whenever \(X\), \(G\) and \(k\) are clear from the context. 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 and constants. Then for every finite set of variables $X$, goal $G$ and $k \geq 1$, the relation $\sim_{X,G,k}$ has only finitely many equivalence classes.

PROOF. Let $C$ be the number of constants in $L$, $R$ the number of predicate symbols and let $a$ be the maximum arity of the predicate symbols in $L$. Let $G$ be a goal of the form $\leftarrow p_1(...), p_2(...), ..., p_m(\ldots)$ with $m \geq 1$ and let $V$ be the number of distinct variables in $G$.

A resultant in an equivalence class of $\sim_{X,G,k}$ is then of the form $p_1(...), p_2(...), ..., p_m(\ldots) \leftarrow q_1(...), q_2(...), ..., q_n(\ldots)$ with $0 \leq n \leq k$. An equivalence class of $\sim_{X,G,k}$ is completely described by the predicate symbols $q_1, ..., q_n$, the arguments of $p_1, ..., p_m$ (in accordance with $G$) and the arguments of $q_1, ..., q_n$.

The number of arguments that must be specified in this resultant is $V$ for $p_1, ..., p_m$, plus at most $n \cdot a$ for $q_1, ..., q_n$. For every argument we may choose either a constant, a variable from $X$ or another (fresh) variable. However, we need at most $V + n \cdot a$ different fresh variables. Therefore the choice of the arguments is limited to $(C + \#X + V + n \cdot a)^{V + n \cdot a}$ possibilities.

Since for a fixed $n$, the choice of the predicate symbols $q_1, ..., q_n$ is limited to $R^n$ possibilities, we have at most \[ \sum_{n=0}^{k} R^n \cdot (C + \#X + V + n \cdot a)^{V + n \cdot a} \] equivalence classes of $\sim_{X,G,k}$. \qed

We can now prove the desired theorem.

THEOREM 4.18. The loop check EVRL is complete w.r.t. the leftmost selection rule for function-free restricted programs.

PROOF. Let $P$ be a function-free restricted program and let $G_0$ be a goal in $L_P$. Let $k = \text{weight}(G_0)$. Consider an infinite SLD-derivation $D = G_0 \Rightarrow_{C_1, \theta_1} G_1 \Rightarrow ... \Rightarrow G_{i-1} \Rightarrow_{C_i, \theta_i} G_i \Rightarrow ...$ of $P \cup \{G_0\}$. By Corollary 4.14 for every $i \geq 0$: $|G_i| \leq k$. Every goal $G_i$ is a goal in $L_P$ and hence every resultant $G_0 \theta_1 ... \theta_i \leftarrow G_i$ belongs to an equivalence class of $\sim_{\emptyset, G_0,k}$. Since $L_P$ satisfies the conditions of Lemma 4.17, $\sim_{\emptyset, G_0,k}$ has only finitely many equivalence classes, so for some $i \geq 0$ and $j > i$, $(G_0 \theta_1 ... \theta_i \leftarrow G_i)$ and $G_0 \theta_1 ... \theta_j \leftarrow G_j$ are variants. This implies that $D$ is pruned by EVRL. \qed
COROLLARY 4.19 (Equality Completeness). All equality checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

PROOF. By Theorem 4.18 and the Relative Strength Theorem 3.9. □

Now combining Corollary 3.6 and Corollary 3.7 with the Equality Soundness Corollary 4.7 and the Equality Completeness Corollary 4.19, we conclude that all equality checks lead to an implementation of CWA for function-free restricted programs. Moreover, a depth first interpreter augmented by any of the equality checks based on resultants yields an implementation of query processing for these programs.

5. Subsumption checks

As already stated, there are eight subsumption checks. We shall define them by means of two parametrized definitions, again trusting that the reader is willing to understand our notation. The inclusion relation between goals regarded as lists is denoted by $\subseteq_L$; similarly $\subseteq_M$ for multisets. Note: $L_1 \subseteq_L L_2$ if all elements of $L_1$ occur in the same order in $L_2$; they need not to occur on adjacent positions. For example, $(a,c) \subseteq_L (a,b,c)$.

5.1. Definitions

DEFINITION 5.1 (Subsumption checks for Goals).
For Type $\in \{L,M\}$, the Subsumes Variant/Instance of GoalType check is the set of SLD-derivations

$\text{SVG/SIG}_{\text{Type}} = \text{RemSub}(\{ D \mid D = ( G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k) \text{ such that for some } i, 0 \leq i < k, \text{ there is a renaming/substitution } \tau \text{ with } G_k \models_{\text{Type}} G_i \tau \})$. □

DEFINITION 5.2 (Subsumption checks for Resultants).
For Type $\in \{L,M\}$, the Subsumes Variant/Instance of ResultantType check is the set of SLD-derivations
SVR/SIRType = RemSub({ D | D = ( G₀ ⇒ c₁,θ₁ G₁ ⇒ ... ⇒ Gₖ₋₁ ⇒ cₖ,θₖ Gₖ ) \\
    such that for some i, 0 ≤ i < k, there is a 
    renaming/substitution τ with Gₖ ≅Type Gᵢτ and 
    G₀θ₁...θₖ = G₀θ₁...θᵢτ }). □

**LEMMA 5.3.** All subsumption checks are simple loop checks.

**PROOF.** Straightforward. □

The following example shows the differences between the behaviour of various subsumption checks and the equality checks.

**EXAMPLE 5.4.**
Let P = { A(y) ← A(0), C(y) (C1), 
    A(0) ← (C2), 
    B(1) ← (C3), 
    C(z) ← B(z), A(w) (C4) },
and let G = ← A(x).

Figure 4.3 shows an SLD-tree of P ∪ {G} using the leftmost selection rule.
It also shows how this tree is pruned by different loop checks. First we explain the behaviour of the loop checks with respect to this tree. Then we shall make some generalizing comments on this behaviour. In this example, the distinction between list versus multiset based loop checks does not play a role.

Starting at the root, the first loop check that prunes the tree is the SIG check. It prunes the goal ← A(0), C(x), because it contains A(0), an instance of A(x).
Following the lefmost infinite branch two steps down, the SVG check prunes the goal ← B(x), A(w), because it contains A(w), a variant of A(x). One step later, the atom B(x) is resolved, so the EIG and EVG checks prune the goal ← A(w) for the same reason.

However, the loop checks based on resultants do not yet prune the tree. The computed answer substitution built up so far maps x to x after the first three steps and to 1 later on. This is clearly different from the substitutions {x/0} and {x/w}, which are used to show that A(0) resp. A(w) are an instance resp. a variant of A(x).
Now the derivation repeats itself, but with x replaced by w. Therefore the loop checks based on resultants prune the tree during this second phase, exactly where the corresponding loop checks based on goals pruned during the first phase.

The side branch that is obtained by repeatedly applying the first clause (and corresponding side branches later on) is pruned by the subsumption checks at the goal ←A(0),C(0),C(x). This goal contains the previous goal ←A(0),C(x). Therefore both the resultant based and the goal based loop checks prune this goal. In contrast, the equality checks do not prune this infinite branch because the goals in it become longer in every derivation step (analogously to Example 4.8).

The loop checks based on goals all prune the solution \{x/1\}, so they are not sound. Among these loop checks, the SIG check prunes as soon as possible for a weakly sound loop check. Conversely, the SIR check prunes this tree as soon as possible for a shortening loop check. So on this tree, it behaves exactly like STRONG, which exhibits such a behaviour by definition. □

Another example shows that there can be a non-trivial difference between the behaviour of subsumption checks based on list subsumption and those based on multiset subsumption.

**Example 5.5.**

Let P = \{ A(x) ← A(y),S(x),T(y) \}. (Note the similarity between this clause and the clause A(x) ← A(y),S(x,y) in Theorem 3.10.)

Let G = ←A(x_0),B(x_0).

An SLD-derivation (and SLD-tree) of P∪\{G\} via the leftmost selection rule is depicted in Figure 4.4. This infinite SLD-derivation is pruned by the SVRM check at the goal ←A(x_2),S(x_1),T(x_2),S(x_0),T(x_1),B(x_0), since a variant of an earlier goal, namely ( ←A(x_1),S(x_0),T(x_1),B(x_0 ) \{x_1/x_2\}, is ‘multiset-contained’ in it.
\[ \leftarrow A(x_0), B(x_0) \]
\[ \downarrow \]
\[ \leftarrow A(x_1), S(x_0), T(x_1), B(x_0) \]
\[ \downarrow \]
\[ \leftarrow A(x_2), S(x_1), T(x_2), S(x_0), T(x_1), B(x_0) \]
\[ \downarrow \]
\[ \leftarrow A(x_3), S(x_2), T(x_3), S(x_1), T(x_2), S(x_0), T(x_1), B(x_0) \]
\[ \downarrow \]
\[ \ldots \]

**Figure 4.4**

However, this derivation is *not* pruned by the SVRL check, nor by the stronger SIGL check. For, assume that the SIGL check prunes this derivation at the goal \( G_k = \leftarrow A(x_k), S(x_{k-1}), T(x_k), S(x_{k-2}), T(x_{k-1}), \ldots, S(x_0), T(x_1), B(x_0), \)

because \( G_i \tau = ( \leftarrow A(x_i), S(x_{i-1}), T(x_i), S(x_{i-2}), T(x_{i-1}), \ldots, S(x_0), T(x_1), B(x_0) ) \tau, \) an instance of an earlier goal \( G_i, \) is list-contained in it.

Clearly, the presence of the B-Atoms in \( G_i \tau \) and \( G_k \) requires \( x_0 \tau = x_0. \) So the atom \( S(x_0) \tau \) in \( G_i \tau \) corresponds to the atom \( S(x_0) \) in \( G_k. \) Then, because \( G_i \tau \) is list-contained in \( G_k, \) \( T(x_1) \tau \) can only correspond to \( T(x_1), \) the only atom between \( S(x_0) \) and \( B(x_0). \) Therefore \( x_1 \tau = x_1. \) Using induction, we can derive \( x_2 \tau = x_2, \ldots, x_i \tau = x_i. \) However, the presence of the A-atoms in \( G_i \tau \) and \( G_k \) requires \( x_i \tau = x_k. \) Since \( i < k, \) this is a contradiction. So the assumption that the SIGL check prunes the derivation is refuted.

The above examples suggest some 'stronger than' relationships (although an example can only prove the absence of such a relationship). Figure 4.5 shows the relationships between the subsumption checks, the equality checks, VA and IA. The arrows between the 'cubes' mean that every subsumption check is stronger than the corresponding equality check in the other 'cube'. So the structure of 'stronger than' relations between equality checks and subsumption checks is a four-dimensional hypercube. Again, proving these 'stronger than' relations is straightforward.
5.2. Soundness
To prove the desired soundness results, we prove that the SIR\textsubscript{M} check is shortening and that the SIG\textsubscript{M} check is weakly sound, since these are the strongest loop checks based on resultants, respectively goals, in our scheme. First we need the following lemma.
Lemma 5.6. Let $P$ be a program. Let $G_1$ and $G_2$ be goals such that $G_2 \subseteq_M G_1$. Suppose $D$ is an SLD-derivation of $P \cup \{G_1\}$ with computed answer substitution $\sigma$. Then every SLD-tree of $P \cup \{G_2\}$ contains a successful branch of length $\leq |D|$ with a computed answer substitution more general than $\sigma$.

Proof. Let $D = (G_1 \Rightarrow C_1, \theta_1 \ldots \Rightarrow C_n, \theta_n \sqcap)$ and let $C_{n_1}, \ldots, C_{n_m}$ be those clauses from $C_1, \ldots, C_n$ that are used (directly or indirectly) to resolve atoms belonging to $G_2$, with $1 \leq n_1 < \ldots < n_m \leq n$. Then there exists an unrestricted (in the sense of [15]) SLD-derivation $G_2 \theta_1 \ldots \theta_{n_1-1} \Rightarrow C_{n_1}, \theta_{n_1} \ldots \theta_{n_2-1} \ldots \Rightarrow C_{n_m}, \theta_{n_m} \ldots \theta_n \sqcap$. Now apply the Mgu Lemma, the Lifting Lemma of [15] and the independence of the computation rule. $\square$

We can now prove the desired theorem.

Theorem 5.7. i) The $SIR_M$ check is shortening.

ii) The $SIG_M$ check is weakly sound.

Proof. Let $P$ be a program and let $G_0$ be a goal.

i) Let $D$ be an SLD-refutation of $P \cup \{G_0\}$ with a computed answer substitution $\sigma$. If $D$ is pruned by $SIR_M$, then we must find in every SLD-tree containing $D$ an SLD-refutation $D'$ of $P \cup \{G_0\}$ with computed answer substitution $\sigma'$ such that $\sigma' \leq \sigma, |D'| < |D|$ and $D'$ is not pruned by $SIR_M$. We prove that $D'$ exists by induction on the length $l$ of the refutation $D$.

We have $l \geq 1$. For $l = 1$, $D$ cannot be pruned by $SIR_M$. Now suppose the theorem is true for every refutation of $P \cup \{G_0\}$ of length $\leq l$. Let $D$ be a refutation of length $l+1$. Suppose that $D$ is pruned by $SIR_M$. Then we have $D = (G_0 \Rightarrow C_1, \theta_1 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow C_i, \theta_i \Rightarrow G_i \Rightarrow \ldots \Rightarrow G_{i+1} \Rightarrow \ldots \Rightarrow G_k \Rightarrow C_{k+1}, \theta_{k+1} \Rightarrow G_{k+1} \Rightarrow \ldots \Rightarrow \sqcap).$, with $G_k \subseteq_M G_i \tau$ and $G_0 \theta_1 \ldots \theta_k = G_0 \theta_1 \ldots \theta_i \tau$ for some substitution $\tau$.

We now proceed as follows.

1. Thus there exists an SLD-refutation of $P \cup \{G_k\}$ with the computed answer substitution $\theta_{k+1} \ldots \theta_i$, of length $l-k+1$. $G_i \tau \subseteq_M G_k$, so we conclude using Lemma 5.6 that there exists an SLD-refutation of $P \cup \{G_i \tau\}$ with a computed answer substitution $\sigma_j \leq \theta_{k+1} \ldots \theta_i$, of length at most $l-k+1$. 


2. Replacing \( G_{i-1} \Rightarrow C_{i,1} \theta_i G_i \) by \( G_{i-1} \Rightarrow C_{i,1} \theta_i \tau G_i \), we get an unrestricted SLD-derivation \( D_1 = (G_{i-1} \Rightarrow \ldots \Rightarrow \square) \) with a computed answer substitution \( \theta_i \tau \sigma_1 \leq \theta_i \tau \sigma_{k+1} \ldots \theta_i \).

3. Using the Mgu Lemma, this yields an SLD-derivation \( D_2 = (G_{i-1} \Rightarrow \ldots \Rightarrow \square) \) with a computed answer substitution \( \sigma_2 \leq \theta_i \tau \sigma_1 \leq \theta_i \tau \theta_{k+1} \ldots \theta_i \) and \( |D_2| \leq l-k+2 \leq l-i+1 \) (since \( k > i \)).

4. There exists an SLD-derivation \( D_3 = (G_0 \Rightarrow \ldots \Rightarrow \square) \) with a computed answer substitution \( \sigma_3 \leq \theta_1 \ldots \theta_{i-1} \sigma_2 \leq \theta_1 \ldots \theta_{i-1} \theta_i \tau \theta_{k+1} \ldots \theta_i \) and \( |D_2| \leq l \) (combining \( G_0 \Rightarrow \ldots \Rightarrow G_{i-1} \) of \( D \) with \( D_2 \)). Indeed, Lemma 4.5, Lemma 5.6 and the proofs of the Mgu Lemma and the Lifting Lemma show that for every SLD-tree containing \( D \), such a derivation \( D_3 \) can be constructed.

5. By induction hypothesis, either \( D_3 \) is not pruned by \( \text{SIR}_M \) or an SLD-refutation \( D_4 \) of \( P \cup \{ G_0 \} \) exists with a computed answer substitution \( \sigma_4 \) such that \( G_0 \sigma_4 \leq G_0 \sigma_3, |D_4| < |D_3| \) and \( D_4 \) is not pruned by \( \text{SIR}_M \). In the first case, we take \( D' = D_3 \), in the second case \( D' = D_4 \).

6. \( G_0 \sigma' \leq G_0 \sigma_3 \leq G_0 \theta_1 \ldots \theta_i \tau \theta_{k+1} \ldots \theta_i = G_0 \theta_1 \ldots \theta_i = G_0 \sigma \), so \( \sigma' \leq \sigma \).

   ii) Note that the additional condition \( G_0 \theta_1 \ldots \theta_k = G_0 \theta_1 \ldots \theta_i \tau \) was only used in step 6.

COROLLARY 5.8 (Subsumption Soundness).

i) All subsumption checks based on resultants are shortening. A fortiori they are sound.

ii) All subsumption checks based on goals are weakly sound.

PROOF. By Theorem 5.7 and the Relative Strength Theorem 3.9.

5.3 Completeness

We now shift our attention to completeness issues. From the results of the previous section we can immediately deduce the following result.

COROLLARY 5.9 (Subsumption Completeness 1). All subsumption checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

PROOF. By the Equality Completeness Corollary 4.19 and the Relative Strength Theorem 3.9.
However, the subsumption checks are stronger than the corresponding equality checks. So we can try to find other classes of programs for which the subsumption checks are complete. We know that the subsumption checks are not complete for all programs, not even for all function-free programs. For \( P = \{ A(x) \leftarrow A(y), S(y,x) \} \), a derivation of \( P \cup \{ \leftarrow A(x), B(x) \} \) is not pruned by any of the subsumption checks, as was shown in Theorem 3.10.

A close analysis of the proof of this theorem shows that the problem is caused by three 'events' occurring simultaneously, namely:
1. A new variable, \( y \), is introduced by a 'recursive' atom, \( A(y) \).
2. There is a relation between this new variable, \( y \), and an old variable, \( x \), namely via the atom \( S(y,x) \).
3. The 'recursive' atom \( A(y) \) is selected before the 'relating' atom \( S(y,x) \).

It appears that, in order to obtain the completeness of the subsumption checks, it is enough to prevent any of these events. Clearly, the use of restricted programs and the leftmost selection rule prevents the third event. We now introduce two new classes of programs, preventing the first and the second event, respectively.

**DEFINITION 5.10 (Nvi Program).**
A clause \( C \) is *non-variable introducing* (in short *nvi*) if every variable that appears in the body of \( C \) also appears in the head of \( C \).
A program \( P \) is *nvi* if every clause in \( P \) is nvi.  \( \Box \)

**DEFINITION 5.11 (Svo Program).**
A clause \( C \) has the *single variable occurrence* property (in short *svo*) if in the body of \( C \), no variable occurs more than once.
A program \( P \) is *svo* if every clause in \( P \) is svo.  \( \Box \)

Clearly, in nvi programs the first event cannot occur, whereas in svo programs the second event is prevented. We would rather have used the terminology *right-linear* instead of svo, which is common in the area of term rewriting systems. However, in the area of deductive databases this term is already in use for a completely different notion.
**Example 5.12.**
The following program is an nvi program and a svo program, but not a restricted
program. It computes in the relation ‘add’ the sum of two two-digit binary
numbers (the first four arguments of ‘add’); this sum is a three-digit binary
number, stored in the last three arguments of ‘add’.
\[
\text{ADD} = \{ \begin{align*}
\text{add}( & 0,0, A,B, 0,A,B ) \leftarrow ; \\
\text{add}( & A,B, 0,0, 0,A,B ) \leftarrow ; \\
\text{add}( & A,B, A,B, A,B,0 ) \leftarrow ; \\
\text{add}( & A_1,B_1, A_2,B_2, C,A_3,B_3 ) \leftarrow \text{add}( 0,B_1, 0,B_2, 0,0,B_3 ), \\
& \quad \quad \quad \quad \quad \text{add}( 0,A_1, 0,A_2, 0,C,A_3 ) ; \\
\text{add}( & A_1,1, A_2,1, 1,0,0) \leftarrow \text{add}( 0,A_1, 0,A_2, 0,0,1) \}.
\end{align*}
\]
The first three clauses are evidently correct; every addition of the form \(0X + 0Y\)
is taken care of by them. The fourth clause deals with the case where adding the
last digits of both numbers does not give a carry (ensured by the first atom in the
body). The fifth clause deals with the case where there is such a carry. Only the
case \(A_1 \neq A_2\) (or equivalently, \(A_1 + A_2 = 1\)) has to be considered there: if \(A_1 = A_2\) then the third clause applies.

Note that this program yields infinite derivations that are not pruned by any
of the equality checks. Indeed, starting with the goal \(\leftarrow \text{add}(0,B_1,0,B_2,0,0,B_3)\),
the first recursive clause applies, giving the goal \(\leftarrow \text{add}(0,B_1,0,B_2,0,0,B_3)\),
\(\text{add}(0,0,0,0,0,0,0)\). Repeatedly selecting \(\text{add}(0,B_1,0,B_2,0,0,B_3)\) and applying
the first recursive clause yields an infinite derivation containing goals of
increasing length, which is not pruned by any of the equality checks.

We now prove that the weakest of the subsumption checks, the SVR\(_L\)
check, is complete for function-free nvi programs. To this end we use the
following (weakened) version of Kruskal’s Tree Theorem, called Higman’s
Lemma. (See [12]; for a formulation of the full version of Kruskal’s Tree
Theorem, see [9] or [13].)

**Lemma 5.13 (Higman’s Lemma).** Let \(w_0,w_1,w_2,\ldots\) be an infinite sequence of
(finite) words over a finite alphabet \(\Sigma\). Then for some \(i\) and \(k > i\), \(w_i \preceq_L w_k\). □

In order to prove that the SVR\(_L\) check is complete for function-free nvi
programs, we prove that, in the absence of function symbols, infinite derivations
in which no new variables are introduced are pruned by the $\text{SVR}_L$ check. Then we prove that every derivation of a function-free nvi program (and an arbitrary goal) has a variant that indeed does not introduce new variables.

**Definition 5.14.**
An SLD-derivation $D = (G_0 \Rightarrow C_1, \theta_1 G_1 \Rightarrow \ldots)$ is non-variable introducing (in short nvi) if $\text{var}(G_0) \supseteq \text{var}(G_1) \supseteq \text{var}(G_2) \supseteq \ldots$. □

**Lemma 5.15.** In the absence of function symbols, every infinite nvi SLD-derivation is pruned by $\text{SVR}_L$.

**Proof.** Let $D = (G_0 \Rightarrow C_1, \theta_1 G_1 \Rightarrow \ldots)$ be an infinite nvi SLD-derivation.
We take for $\Sigma$ the set of equivalence classes of $\sim_{\text{var}(G_0), G_0, 1}$ as defined in Definition 4.15. By Lemma 4.17, $\Sigma$ is finite. To apply Higman's Lemma 5.13 we represent for $j \geq 0$ a goal $G_j = A_{1j}, \ldots, A_{nj}$ (or rather the corresponding resultant $G_0 \theta_1 \ldots \theta_j \leftarrow G_j$) as the word $[G_0 \theta_1 \ldots \theta_j \leftarrow A_{1j}], \ldots, [G_0 \theta_1 \ldots \theta_j \leftarrow A_{nj}]$ over $\Sigma$. (Recall that for a resultant R, $[R]$ denotes its equivalence class.) The sequence of representations of $G_0, G_1, G_2, \ldots$ yields an infinite sequence of words $w_0, w_1, w_2, \ldots$ over $\Sigma$.

Now by Higman's Lemma 5.13, for some $j$ and $k > j$: $[G_0 \theta_1 \ldots \theta_j \leftarrow A_{1j}], \ldots, [G_0 \theta_1 \ldots \theta_j \leftarrow A_{nj}] \subseteq [G_0 \theta_1 \ldots \theta_k \leftarrow A_{1k}], \ldots, [G_0 \theta_1 \ldots \theta_k \leftarrow A_{nk}]$. So by the definition of $\sim_{\text{var}(G_0), G_0, 1}$, there exist renamings $\rho_1, \ldots, \rho_n$ which do not act on the variables of $G_0$ such that $(G_0 \theta_1 \ldots \theta_j \leftarrow A_{1j})\rho_1, \ldots, (G_0 \theta_1 \ldots \theta_j \leftarrow A_{nj})\rho_n \subseteq (G_0 \theta_1 \ldots \theta_k \leftarrow A_{1k}), \ldots, (G_0 \theta_1 \ldots \theta_k \leftarrow A_{nk})$.

However, $D$ is nvi, so $\text{var}(G_j) \subseteq \text{var}(G_0)$ and therefore the renamings $\rho_h$ do not act on the atoms $A_{ij}$ of $G_j$ ($1 \leq h, i \leq n_j$). Thus $G_j = G_j\rho_1 \subseteq G_k$ and $G_0 \theta_1 \ldots \theta_j \rho_1 = G_0 \theta_1 \ldots \theta_k$. So $D$ is pruned by $\text{SVR}_L$. □

**Lemma 5.16.** Let $P$ be a function-free nvi program and let $G_0$ be a goal in $L_P$.

Let $D$ be an infinite SLD-derivation of $P \cup \{G_0\}$. Then a variant $D'$ of $D$ is an infinite nvi derivation.

**Proof.** Suppose that $D = (G_0 \Rightarrow C_1, \theta_1 G_1 \Rightarrow C_2, \theta_2 G_2 \Rightarrow \ldots)$. We show that there exists an infinite nvi derivation $D' = (G_0' \Rightarrow C_1, \theta_1' G_1' \Rightarrow C_2, \theta_2' G_2' \Rightarrow \ldots)$ that is a variant of $D$. Note that $D'$ uses the same input clauses as $D$. 

We give an inductive construction of $D'$. By definition, $G_0' = G_0$. Suppose we have constructed $D'$ up to a goal $G_{i-1}'$ ($i > 0$). $G_{i-1}'$ and $G_{i-1}$ are variants, say $G_{i-1} = G_{i-1}'\rho$. $G_0' = G_0$ and the clauses $C_1, \ldots, C_{i-1}$ are the same as in $D$, so $C_i$ is well standardized apart and we may assume that $C_i\rho = C_i$. Therefore $\rho\theta_1\rho^{-1}$ is an applicable (idempotent) mgu.

Now we obtain $\theta_1'$ by replacing every pure variable binding $x/y$ within $\rho\theta_1\rho^{-1}$ by $y/x$ whenever $x \in \text{var}(G_{i-1}')$ and $y \in \text{var}(C_i)$, and replacing for such $x$ and $y$ every other binding $z/y$ within $\rho\theta_1\rho^{-1}$ by $z/x$.

Since no function symbols appear in $P$, this yields that for every variable $x \in \text{var}(G_{i-1}')$ either $x\theta_1' \in \text{var}(G_{i-1}')$ or $x\theta_1'$ is a constant. Hence $\text{var}(G_{i-1}'\theta_1') \subseteq \text{var}(G_{i-1}')$. Now let $A$ be the selected atom in $G_{i-1}'$, let $R$ be the rest of $G_{i-1}'$ and let $x \in \text{var}(G_i')$. Two cases arise.

- $x$ is introduced by $C_i$, that is $x \in \text{var(\text{body}(C_i)\theta_1')}$. Then, since $P$ is an nvi program, $x \in \text{var(\text{head}(C_i)\theta_1')}$. $\theta_1'$ is a unifier of $\text{head}(C_i)$ and $A$, so $x \in \text{var}(A\theta_1') \subseteq \text{var}(G_{i-1}\theta_1') \subseteq \text{var}(G_{i-1}')$.
- $x$ is introduced by $G_{i-1}'$, that is $x \in \text{var(\text{R}\theta_1')}$. Then $x \in \text{var}(G_{i-1}\theta_1') \subseteq \text{var}(G_{i-1}')$.

This proves the induction hypothesis for $D'$ up to the goal $G_i'$.

**Theorem 5.17.** The $\text{SVRL}$ loop check is complete for function-free nvi programs.

**Proof.** By Lemma 5.3, 5.15 and 5.16.

**Corollary 5.18 (Subsumption Completeness 2).** All subsumption checks are complete for function-free nvi programs.

**Proof.** By Theorem 5.17 and the Relative Strength Theorem 3.9.

We now prove that the $\text{SVR}_L$ check (and hence all subsumption checks) are complete for function-free svo programs. By a construction similar to the one used in the proof of Lemma 5.16, we may assume that in an SLD-derivation $D = (G_0 \Rightarrow C_1, \theta_1 \ G_1 \Rightarrow \ldots)$, $\text{var}(G_0\theta_i) \subseteq \text{var}(G_0)$ for $i > 0$. (Note that for this construction, only the absence of function symbols was needed, and not the nvi property.) Under this assumption we can prove the following lemma.
LEMMA 5.19. Let $P$ be a function-free svo program and let $G_0$ be a goal in $L_P$.
Let $D = (G_0 \Rightarrow C_{1, \theta_1} \Rightarrow C_{2, \theta_2} G_2 \Rightarrow \ldots)$ be an SLD-derivation of
$P \cup \{G_0\}$. Then for every goal $G_i$ ($i \geq 0$): if $x$ occurs more than once in $G_i$,
then $x \in \text{var}(G_0)$.

PROOF. By induction. For $i=0$, the claim is trivial.
Now suppose $x$ occurs more than once in $G_{i+1}$ ($i \geq 0$) and $x \notin \text{var}(G_0)$.

Let $G_i = (A, S)$, where $A$ is the selected atom (not necessarily the leftmost atom) and let $C_{i+1} = H \leftarrow X$. Then $\theta_{i+1}$ is an idempotent mgu of $A$ and $H$ and
$G_{i+1} = (X, S)\theta_{i+1}$. There are two ways in which we can obtain a variable $x$ occurring more than once in $G_{i+1}$.
1. A variable $y$ occurs more than once in $(X, S)$ and $y\theta_{i+1} = x$.
By standardizing apart, $\text{var}(S) \cap \text{var}(X) = \emptyset$, so $y$ occurs either only in $S$ or
only in $X$. Since $C_{i+1}$ is svo, $y$ does not occur more than once in $X$. Therefore $y$
ocite{iskakov1995}
occurs more than once in $S$. Then by the induction hypothesis, $y \in \text{var}(G_0)$. But
then $x = y\theta_{i+1} \in \text{var}(G_0 \theta_{i+1}) \subseteq \text{var}(G_0)$.
2. There are two variables $y_1$ and $y_2$ in $(X, S)$ such that $y_1\theta_{i+1} = y_2\theta_{i+1} = x$ and
$y_1 \neq y_2$.
In this case, $y_1, y_2 \in \text{var}(A, H)$, since $\text{dom}(\theta_{i+1}) \subseteq \text{var}(A, H)$.
If $y_1 \in \text{var}(S)$, then by standardizing apart $y_1 \notin \text{var}(H)$, so $y_1 \in \text{var}(A)$.
Therefore $y_1$ occurs more than once in $G_i$ (in $A$ and in $S$), and we can apply the
induction hypothesis again. Since the same argument holds for $y_2 \in \text{var}(S)$, only
the case $y_1, y_2 \in \text{var}(X)$ is left. In this case, since $y_1, y_2 \in \text{var}(A, H)$, by
standardizing apart $y_1, y_2 \in \text{var}(H)$.

Since $y_1\theta_{i+1} = y_2\theta_{i+1} = x$, the sets
$Z_1 = \{ z \in \text{var}(A) \mid z \text{ occurs in } A \text{ at the position of an occurrence of } y_1 \text{ in } H \}$
and $Z_2 = \{ z \in \text{var}(A) \mid z \text{ occurs in } A \text{ at the position of an occurrence of } y_2 \text{ in } H \}$
are not disjoint. (Otherwise, a more general unifier of $A$ and $H$ than $\theta$ would exist,
mapping $y_1$ to an element of $Z_1$ and $y_2$ to an element of $Z_2$.) Let $z \in Z_1 \cap Z_2$. $z$
ocite{iskakov1996}
ocite{iskakov1997}
ocite{iskakov1998}
ocite{iskakov1999}
ocite{iskakov2000}
occurs at least twice in $A$, so $z \in \text{var}(G_0)$. Thus $x = z\theta_{i+1} \in \text{var}(G_0 \theta_{i+1}) \subseteq
\text{var}(G_0)$.

We can now prove the desired theorem.
**Theorem 5.20.** The SVRL loop check is complete for function-free svo programs.

**Proof.** Let P be a function-free svo program and let G₀ be a goal in Lₚ. Let D = (G₀ ⇒ C₁, θ₁ G₁ ⇒ C₂, θ₂ G₂ ⇒ ...) be an infinite SLD-derivation of P ∪ {G₀}.

Again, we take for Σ the set of equivalence classes of ~_{var(G₀)}, G₀, 1 as defined in Definition 4.15. By Lemma 4.17, Σ is finite. To apply Higman’s Lemma 5.13 we represent a goal G_j = A_{1j}, ..., A_{nj} in D as the word w_j = [G₀θ₁...θ_j←A_{1j}], ..., [G₀θ₁...θ_j←A_{nj}] over Σ. The sequence of representations of G₀, G₁, G₂, ... yields an infinite sequence of words w₀, w₁, w₂, ... over Σ.

Now by Higman’s Lemma 5.13, for some j and k > j: [G₀θ₁...θ_j←A_{1j}], ..., [G₀θ₁...θ_j←A_{nj}] ≺ₗ [G₀θ₁...θ_k←A_{1k}], ..., [G₀θ₁...θ_k←A_{nk}]. So there are renamings ρ₁, ..., ρ_{nj} such that (G₀θ₁...θ_j←A_{1j})ρ₁,...,(G₀θ₁...θ_j←A_{nj})ρ_{nj} ≺ₗ (G₀θ₁...θ_k←A_{1k}), ..., (G₀θ₁...θ_k←A_{nk}).

Now we can define the renaming ρ by

\[ xρ = \begin{cases} x & \text{if } x \in \text{var}(G₀) \text{ or } x \notin \text{var}(G_j), \\ xρ_k & \text{if } x \in \text{var}(A_{kj}), \end{cases} \]

since by Lemma 5.19, if x \notin \text{var}(G₀), x occurs at most once in G_j, and therefore k is uniquely defined. Applying Lemma 5.19 on G_k yields that ρ is again a renaming.

We have assumed that \text{var}(G₀θ₁...θ_j) \subseteq \text{var}(G₀θ₂...θ_j) \subseteq \ldots \subseteq \text{var}(G₀θ_j) \subseteq \text{var}(G₀), and since by the definition of ~_{var(G₀), G₀, 1}, the renamings ρ_j do not act on variables in \text{var}(G₀), we have ((G₀θ₁...θ_j←A_{1j}), ..., (G₀θ₁...θ_j←A_{nj}))ρ \subseteqₗ (G₀θ₁...θ_k←A_{1k}), ..., (G₀θ₁...θ_k←A_{nk}). So G_jρ \subseteqₗ G_k and G₀θ₁...θ_jρ = G₀θ₁...θ_k, and D is pruned by SVRL.

\[ \Box \]

**Corollary 5.21 (Subsumption Completeness 3).** All subsumption checks are complete for function-free svo programs.

**Proof.** By Theorem 5.20 and the Relative Strength Theorem 3.9. \[ \Box \]

Now combining Corollary 3.6 and Corollary 3.7 with the Subsumption Soundness Corollary 5.8 and the Subsumption Completeness Corollaries 5.9, 5.18 and 5.21, we conclude that all subsumption checks lead to an implementation of CWA for restricted programs, nvi programs and svo programs without function symbols. Moreover, the subsumption checks based
on resultants also lead to an implementation of query processing for these programs.

6. Context checks

The problem with the Instance of Atom check is that it does not take into account the context of the atom. This is incorrect: whereas solving \( \leftarrow A(x) \) or \( \leftarrow A(y) \) makes no difference, solving \( \leftarrow A(x), B(x) \) is essentially more difficult than solving \( \leftarrow A(y), B(x) \). To remedy this problem we should keep track of the links between the variables in the atom and those in the rest of the goal.

Roughly speaking, the IA check prunes a derivation as soon as a goal \( G_k \) occurs that contains an instance \( A\tau \) of an atom \( A \) that occurred in an earlier goal \( G_i \). But when a variable occurs both inside and outside of \( A \) in \( G_i \), we should not prune the derivation if this link has been altered. Such a variable \( x \) in \( G_i \) is substituted by \( x\theta_{i+1}\ldots\theta_k \) when \( G_k \) is reached. Therefore \( \tau \) and \( \theta_{i+1}\ldots\theta_k \) should agree on \( x \). This leads us to a loop check introduced by Besnard [3].

6.1. Definitions

**DEFINITION 6.1 (Context checks for Goals).**

The Variant/Instance Context check on Goals is the set of SLD-derivations

\[
CVG/CIG = \text{RemSub}(\{ D \mid D = (G_0 \Rightarrow C_1, \theta_1 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow C_k, \theta_k \Rightarrow G_k) \}
\]

such that for some \( i \) and \( j \), \( 0 \leq i \leq j < k \), there is a renaming/substitution \( \tau \) such that for some atom \( A \) in \( G_i \): \( A\tau \) appears in \( G_k \) as the result of resolving \( A\theta_{i+1}\ldots\theta_j \) in \( G_j \) and for every variable \( x \) that occurs both inside and outside of \( A \) in \( G_i \), \( x\theta_{i+1}\ldots\theta_k = x\tau \).

Besnard describes the condition on the substitutions as follows: ‘When \( A\tau \) is substituted for \( A\theta_{i+1}\ldots\theta_k \) in \( G_i\theta_{i+1}\ldots\theta_k \), this should give an instance of \( G_i \).’ We show that this formulation is equivalent to ours. Let \( G_i = (A, S) \), that is \( A \) occurs in \( G_i \) and \( S \) is the list of other atoms in \( G_i \). Then \( (A\tau, S\theta_{i+1}\ldots\theta_k) \) should be an instance of \( (A, S) \), say \( (A\sigma, S\sigma) \).

Clearly, \( x\sigma = -x\tau \) for \( x \in \text{var}(A) \)

\[-x\theta_{i+1}\ldots\theta_k \] for \( x \in \text{var}(S) \),

so for \( x \in \text{var}(A) \cap \text{var}(S) \), \( x\tau = x\theta_{i+1}\ldots\theta_k \).
The following example clarifies the use of the context checks.

**Example 6.2.**
We use the program P and the goal G of Variant of Atom check Example 2.5 and apply the CIG check on two SLD-trees of \( P \cup \{ G \} \), via the leftmost and rightmost selection rule, respectively. This yields the trees in Figure 4.6.

![Figure 4.6](image)

The goal \( G_3 = \leftarrow A(y') \) in the rightmost tree that was incorrectly pruned by the VA check, is not pruned by the CIG check. Certainly, \( A(y') \) is the result of resolving \( A(1) \) in \( G_2 \), the further instantiated version of \( A(x) \) in \( G_1 \). But replacing \( A(x) \theta_2 \theta_3 \) by \( A(y') \) in \( G_1 \theta_2 \theta_3 \) yields \( \leftarrow A(y'), B(1) \), which is not an instance of \( \leftarrow A(x), B(x) \).

**Claim 6.3.** CVG and CIG are weakly sound simple loop checks.
Proving that CVG and CIG are simple loop checks is straightforward. Besnard claims that CIG is weakly sound. From this it follows that the weaker CVG check is also weakly sound. See also Theorem 6.6.
In Example 4.3, the context checks act exactly in the same way as the corresponding equality checks. This shows that CVG and CIG are not sound. Again we can obtain sound, even shortening, versions by using resultants instead of goals.

**Definition 6.4 (Context checks for Resultants).**

The *Variant/Instance Context check on Resultants* is the set of SLD-derivations

\[ CVR/CIR = \text{RemSub}(\{ D \mid D = (G_0 \Rightarrow \theta_1 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow \theta_k \Rightarrow G_k) \}
\]

such that for some \( i \) and \( j \), \( 0 \leq i \leq j < k \), there is a renaming/substitution \( \tau \) such that \( G_0 \theta_1 \ldots \theta_k = G_0 \theta_1 \ldots \theta_i \tau \) and for some atom \( A \) in \( G_j \): \( A \tau \) appears in \( G_k \) as the result of resolving \( A \theta_{i+1} \ldots \theta_j \) in \( G_j \) and for every variable \( x \) that occurs both inside and outside of \( A \) in \( G_j \): \( x \theta_{i+1} \ldots \theta_k = x \tau \}).

Using Besnard's phrasing, the conditions on the substitutions can be summarized as: 'When \( A \tau \) is substituted for \( A \theta_{i+1} \ldots \theta_k \) in the resultant \( R_i \theta_{i+1} \ldots \theta_k \), this should give an instance of \( R_i \).

**Lemma 6.5.** CVR and CIR are simple loop checks.

**Proof.** Straightforward.

\[ \square \]

### 6.2. Soundness

Now we shall prove that the CIR check is shortening. From this it follows that the weaker loop check CVR is also shortening.

**Theorem 6.6. The CIR check is shortening.**

**Proof.** Let \( P \) be a program and let \( G_0 \) be a goal. Let \( D \) be an SLD-refutation of \( P \cup \{G_0\} \) with a computed answer substitution \( \sigma \). If \( D \) is pruned by CIR, then we must find in every SLD-tree containing \( D \) an SLD-refutation \( D' \) of \( P \cup \{G_0\} \) with computed answer substitution \( \sigma' \) such that \( \sigma' \leq \sigma \), \(|D'| \leq |D|\) and \( D' \) is not pruned by CIR. We prove the existence of such a refutation by induction on the length \( l \) of the refutation \( D \).

We have \( l \geq 1 \). For \( l = 1 \), \( D \) cannot be pruned by CIR. Now suppose the theorem is true for every refutation of \( P \cup \{G_0\} \) of length \( \leq l \). Let \( D \) be a refutation of length \( l+1 \). Suppose that \( D \) is pruned by CIR. Then we have
D = (G_0 \Rightarrow C_i \theta_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow C_i \theta_i \Rightarrow C_{i+1} \theta_{i+1} \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow C_k \theta_k \Rightarrow \ldots \Rightarrow \Box), \) with for some \( \tau: G_i = (A, S_i), G_k = (A \tau, S_k), S_i \theta_{i+1} \ldots \theta_k = S_i \tau \) and \( G_0 \theta_1 \ldots \theta_k = G_0 \theta_1 \ldots \theta_1 \tau. \)

Therefore \( S_i \triangleleft_M G_i \) and \( \{ A \tau \} \triangleleft_M G_k. \) By Lemma 5.6 we have SLD-refutations \( D_1 \) of \( P \cup \{ \leftarrow S_i \} \) and \( D_2 \) of \( P \cup \{ \leftarrow A \tau \} \) via every selection rule. An inspection of the proof of Lemma 5.6 shows that every derivation step in \( D_1 \) and \( D_2 \) has a corresponding derivation step in the tail \( (G_i \Rightarrow \ldots \Rightarrow \Box) \) of \( D. \) This tail consists of \( l-i \) derivation steps. On the other hand, at least one step in this tail has no corresponding step in \( D_1 \) or \( D_2: \) the step in which \( A \theta_{i+1} \ldots \theta_i \) is selected. Therefore \( |D_1| + |D_2| < l-i. \)

\( D_1 \) has a computed answer substitution more general than \( \theta_{i+1} \ldots \theta_k \theta_{k+1} \ldots \theta_l. \) So there exists an SLD-refutation \( D_1' \) of \( P \cup \{ \leftarrow S_i \theta_{i+1} \ldots \theta_k \} \) with \( |D_1'| = |D_1| \) and a computed answer substitution more general than \( \theta_{k+1} \ldots \theta_l. \) The computed answer substitution of \( D_2 \) is also more general than \( \theta_{k+1} \ldots \theta_l. \) So we can combine \( D_1' \) and \( D_2 \) into an SLD-refutation \( D_3 \) of \( P \cup \{ \leftarrow (A, S_i) \tau \} \).

Finally we combine \( G_0 \Rightarrow \ldots \Rightarrow G_{i-1} \) from \( D, \) the unrestricted derivation step \( G_{i-1} \Rightarrow C_i \theta_i \tau G_i \\tau \) and \( D_3 \) into an unrestricted SLD-refutation \( D_4 \) of \( P \cup \{ G_0 \}. \)

\( |D_4| = i+1+|D_1|+|D_2| < i+1+l-i = l+1. \)

By the Mgu Lemma, an SLD-refutation \( D_5 \) exists with \( |D_5| = |D_4| \) and a computed answer substitution \( \sigma_5 \leq \theta_1 \ldots \theta_{i-1} \theta_i \tau \theta_{k+1} \ldots \theta_l. \) \( D_5 \) is an SLD-refutation of \( P \cup \{ G_0 \} \) which is shorter than \( D, \) so by the induction hypothesis there exists an SLD-refutation \( D' \) of \( P \cup \{ G_0 \} \) with computed answer substitution \( \sigma' \) such that \( G_0 \sigma' \leq G_0 \sigma_5 \leq G_0 \tilde{\sigma}_5 \leq G_0 \theta_1 \ldots \theta_i \theta_{k+1} \ldots \theta_l = G_0 \theta_1 \ldots \theta_l = G_0 \sigma, \) so \( \sigma' \leq \sigma. \)

\hfill \Box

**Corollary 6.7 (Context Soundness).**

i) The context checks based on resultants are shortening. A fortiori they are sound.

ii) The context checks based on goals are weakly sound.

**Proof.** By Theorem 6.6 and the Relative Strength Theorem 3.9. Note that omitting the considerations about computed answer substitutions from this proof yields a proof for ii), i.e. for Claim 6.3. \hfill \Box
For derivations via certain selection rules (including leftmost and rightmost selection rule), a much easier soundness proof exists. Namely, w.r.t. these selection rules, the subsumption checks are stronger than the corresponding (w.r.t. variant vs. instance and goal vs. resultant distinctions) context checks.

**DEFINITION 6.8.**

(This definition is equivalent to the definition of local selection functions in [22].) A selection rule \( R \) is *most recent* if every SLD-derivation \( D = (G_0 \Rightarrow C_1, \theta_1 \quad G_1 \Rightarrow \ldots) \) via \( R \) satisfies the following property. If in a goal \( G_i \), an atom \( A \) is selected and in a goal \( G_j \) (j>i) the further instantiated version \( B\theta_{i+1}\ldots\theta_j \) of the atom \( B \) in \( G_i \) is selected, then \( A \) is resolved completely between \( G_i \) and \( G_j \). \( \square \)

**LEMMA 6.9.** The subsumption checks are stronger than the corresponding context checks w.r.t. most recent selection rules.

**PROOF.** Suppose \( D = (G_0 \Rightarrow C_1, \theta_1 \quad G_1 \Rightarrow \ldots \Rightarrow C_k, \theta_k \quad G_k) \) is pruned at \( G_k \) by a context check. We show that \( D \) is pruned by the corresponding subsumption check based on lists at \( G_k \) (or earlier).

We have an atom \( A \) in \( G_i \), \( A\theta_{i+1}\ldots\theta_j \) in \( G_j \) as the selected atom and \( A\tau \) as the result of resolving \( A\theta_{i+1}\ldots\theta_j \). Let \( G_i = (A,S,T) \), where \( S \) consists of those atoms in \( G_i \) that are completely resolved between \( G_i \) and \( G_j \). The use of a most recent selection rule yields that \( G_j = (A\theta_{i+1}\ldots\theta_j, T\theta_{i+1}\ldots\theta_j) \) and \( G_k = (A\tau, U, T\theta_{i+1}\ldots\theta_k) \) (\( U \) consists of the other atoms in \( G_k \) that are the result of resolving \( A\theta_{i+1}\ldots\theta_j \)). Finally \( (S,T)\theta_{i+1}\ldots\theta_k = (S,T)\tau \).

![Diagram](image.png)

**Figure 4.7**
First we show that for every x, y, if \( x\theta_{i+1} \ldots \theta_j = y\theta_{i+1} \ldots \theta_j \) then \( x\tau = y\tau \). Suppose \( x\theta_{i+1} \ldots \theta_j = y\theta_{i+1} \ldots \theta_j \) and \( x \neq y \). Then clearly \( x, y \in \text{var}(S) \). So \( x\tau = x\theta_{i+1} \ldots \theta_k = y\theta_{i+1} \ldots \theta_k = y\tau \).

Now we can define \( \sigma \) on \( \text{var}(G_j) \) as follows:

- if \( x \in \text{var}(A\theta_{i+1} \ldots \theta_j) \) then \( x\sigma = y\tau \), where \( y \in \text{var}(A) \) such that \( y\theta_{i+1} \ldots \theta_j = x \). (Although \( y \) may not be unique, \( y\tau \) is.) (case a),

- if \( x \in \text{var}(T\theta_{i+1} \ldots \theta_j) \) then \( x\sigma = x\theta_{i+1} \ldots \theta_k \). (Hence for some \( y \in \text{var}(T) \) such that \( y\theta_{i+1} \ldots \theta_j = x \), we have \( x\sigma = y\theta_{i+1} \ldots \theta_k = y\tau \). (case b),

and we show that:

1) \( G_j \sigma \subseteq L \).

We show that \( A\theta_{i+1} \ldots \theta_j \sigma = A\tau \) and \( T\theta_{i+1} \ldots \theta_j \sigma = T\theta_{i+1} \ldots \theta_k \).

If \( y \in \text{var}(A) \), then either \( y\theta_{i+1} \ldots \theta_j \in \text{var}(A\theta_{i+1} \ldots \theta_j) \), so \( y\theta_{i+1} \ldots \theta_j \sigma = y\tau \), or

\( y\theta_{i+1} \ldots \theta_j \) is a constant, so \( y\theta_{i+1} \ldots \theta_j \sigma = y\theta_{i+1} \ldots \theta_j = y\tau \), since in this case \( y \in \text{var}(S) \).

If \( y \in \text{var}(T) \), then if \( y \in \text{var}(A) \), \( y\theta_{i+1} \ldots \theta_j \sigma = y\tau = y\theta_{i+1} \ldots \theta_k \),

otherwise \( y\theta_{i+1} \ldots \theta_j \sigma = y\theta_{i+1} \ldots \theta_j \theta_j \theta_{j+1} \ldots \theta_k = y\theta_{i+1} \ldots \theta_k \).

2) If \( G_0\theta_{i+1} \ldots \theta_k = G_0\theta_1 \ldots \theta_2 \tau \) then \( G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_j \sigma \) (goal vs. resultant).

Let \( x \in \text{var}(G_0\theta_1 \ldots \theta_j) \).

In case a, \( x = y\theta_{i+1} \ldots \theta_j \) for some \( y \in \text{var}(G_0\theta_1 \ldots \theta_j) \). Then \( x\sigma = y\tau = y\theta_{i+1} \ldots \theta_k = x\theta_{i+1} \ldots \theta_k \).

In case b, \( x\sigma = x\theta_{i+1} \ldots \theta_k \) by definition.

3) If \( \tau \) is a renaming, then \( \sigma \) is a renaming (variant vs. instance).

For every \( x \in \text{var}(G_j) \) there exists a \( y \in \text{var}(A,T) \) such that \( y\theta_{i+1} \ldots \theta_j = x \) and \( x\sigma = y\tau \). \( \Box \)

The following example shows that the previous result does not hold for selection rules that are not most recent.

**EXAMPLE 6.10** (based on Example 10 in [3]).

Let \( P = \{ A \leftarrow B \quad \text{(C1)}, \right. \)

\( B \leftarrow A \quad \text{(C2)}, \)

\( C \leftarrow D \quad \text{(C3)} \}, \)

and let \( G = \leftarrow A, C \).

Then the derivation \( \leftarrow A, C \Rightarrow \text{(C1)} \leftarrow B, C \Rightarrow \text{(C3)} \leftarrow B, D \Rightarrow \text{(C2)} \leftarrow A, D \) (in which the selected atoms are underlined) is pruned by the context checks (the \( A \) in the
fourth goal is the result of resolving the A in the first goal), but not by the subsumption checks.

\[ \text{shortening} \quad \text{weakly sound} \]

\[ \begin{align*}
\text{EVR}_L & \rightarrow \text{EVG}_L \\
\text{EIR}_L & \rightarrow \text{EIG}_L \\
\text{EVR}_M & \rightarrow \text{EVM}_M \\
\text{EIR}_M & \rightarrow \text{EIM}_M
\end{align*} \]

\[ \begin{align*}
\text{CVR} & \rightarrow \text{SVR}_L \\
\text{CIR} & \rightarrow \text{SIR}_L \\
\text{SVR}_M & \rightarrow \text{SVG}_M \\
\text{SIR}_M & \rightarrow \text{SIG}_M \\
\text{VA} & \rightarrow \text{IA}
\end{align*} \]

\[ \text{not weakly sound} \]

\textbf{FIGURE 4.8}
Now we can add the context checks to our ‘stronger than’ scheme. The dotted arrows are only valid for most recent selection rules.

6.3. Completeness
Again we shift our attention to completeness issues. Besnard [3] claims without a proof that his loop check is complete for function-free nvi programs.

CLAIM 6.11 (Context Completeness 1).

The CIG check is complete for function-free nvi programs.

We conjecture that even the weakest of the four context checks, CVR, is complete for function-free nvi programs. We now prove that, like the equality checks and the subsumption checks, the context checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

THEOREM 6.12. The CVR check is complete w.r.t. the leftmost selection rule for function-free restricted programs.

PROOF. Let \( P \) be a function-free restricted program and let \( G_0 \) be a goal in \( L_P \). Let \( k = \text{weight}(G_0) \). Consider an infinite SLD-derivation \( D = G_0 \Rightarrow C_1, \theta_1 G_1 \Rightarrow \ldots \Rightarrow G_{i-1} \Rightarrow C_i, \theta_i G_i \Rightarrow \ldots \) of \( P \cup \{ G_0 \} \). By Corollary 4.14 for every \( i \geq 0 \): 
\( |G_i| \leq k \). Every goal \( G_i \) is a goal in \( L_P \) and hence every resultant \( G_0 \theta_1 \ldots \theta_i \leftarrow G_i \) belongs to an equivalence class of \( \sim \emptyset, G_0, k \). \( L_P \) satisfies the conditions of Lemma 4.17, so \( \sim \emptyset, G_0, k \) has only finitely many equivalence classes. Thus the set \( E = \{ e \mid e \text{ is an equivalence class of } \sim \emptyset, G_0, k \text{ and for infinitely many resultants } R \text{ in } D: R \in e \} \) is non-empty. For simplicity, we shall say that the goal \( G_i \) is in an equivalence class \( e \), when in fact \( (G_0 \theta_1 \ldots \theta_i \leftarrow G_i) \in e \).

For every equivalence class \( e \) of \( \sim \emptyset, G_0, k \), we define the length of \( e \), denoted by \( |e| \), as the length of the goals in \( e \). Since \( E \neq \emptyset \), we can define \( l = \min \{ |e| \mid e \in E \} \). Now we choose an equivalence class \( e \in E \) with \( |e| = l \). According to the choice of \( e \), \( D \) contains infinitely many goals in \( e \) and a finite number of shorter goals (since the number of equivalence classes of \( \sim \emptyset, G_0, k \) is finite).

Let \( G_i \) and \( G_k \) be (the first) two goals in \( D \) that are in \( e \) such that no goal lying in \( D \) between them is shorter. Since \( G_i \) and \( G_k \) are in the same equivalence class \( e \), we have \( G_k = G_i \tau \) and \( G_0 \theta_1 \ldots \theta_i \leftarrow G_i \) and \( G_0 \theta_1 \ldots \theta_k \leftarrow G_k \tau \) for some renaming \( \tau \).
Let A be the leftmost atom in \( G_i \) and let S be the rest of \( G_i \). A is selected in \( G_i \). However, A is not completely resolved between \( G_i \) and \( G_k \), otherwise a goal shorter than \( G_i \), namely an instance of S, would appear between \( G_i \) and \( G_k \) in D. Therefore the atom \( A \tau \) in \( G_k \) is the result of resolving A. Furthermore, no atom of S is selected between \( G_i \) and \( G_k \), so \( G_k = (A \tau, S \theta_{i+1} \ldots \theta_k) \). Hence \( S \theta_{i+1} \ldots \theta_k = S \tau \).

When in the resultant \( R_i \theta_{i+1} \ldots \theta_k \), we replace \( A \theta_{i+1} \ldots \theta_k \) by \( A \tau \), we obtain \( (G_0 \theta_1 \ldots \theta_k \leftarrow A \tau, S \theta_{i+1} \ldots \theta_k) = (G_0 \theta_1 \ldots \theta_i \leftarrow A \tau, S \tau) \), which is a variant of \( R_i \). Therefore D is pruned by the CVR check.

\[ \square \]

**COROLLARY 6.13 (Context Completeness 2).** All context checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

**PROOF.** By Theorem 6.12 and the Relative Strength Theorem 3.9. \[ \square \]

Now combining Corollary 3.6 and Corollary 3.7 with the Context Soundness Corollary 6.7 and the Context Completeness Corollary 6.13, we conclude that all context checks lead to an implementation of CWA for restricted programs without function symbols. Moreover, the context checks based on resultants also lead to an implementation of query processing for these programs. Finally, if the Context Completeness Claim 6.11 can be confirmed, the same holds for function-free nvi programs.
References


