On the Power of Subsumption and Context Checks

Roland N. Bol[1]

Krzysztof R. Apt[1,2]

Jan Willem Klop[1,3]

Abstract

Loop checking is a mechanism used to prune infinite SLD-derivations. Here we study two classes of loop checking mechanisms - subsumption checks and context checks. We analyze their soundness, completeness relative strength and related concepts. We prove their soundness (no computed answer substitution to a goal is missed) and demonstrate their completeness (all resulting derivations are finite) for some classes of logic programs. The completeness theorems for the subsumption checks make use of a simple version of Kruskal's Tree Theorem [K], called Higman's Lemma [H].

This paper is a sequel to Apt, Bol and Klop [ABK] where a formal framework for studying loop checking mechanisms was introduced and where so-called equality checks were studied.

1. Introduction

Logic programming is advocated as a formalism for writing executable specifications. However, even when these specifications are correct in the logical sense, their execution by means of a PROLOG interpreter may lead to divergence. This problem motivated the study of loop checking mechanisms which are used to discover some form of looping in SLD-derivations (see [B], [BW], [C], [PG], [SGG], [SI], [V]).

The use of a PROLOG interpreter augmented with a loop check allows us to use a larger class of logic programs as correct *executable* specifications. Which class it is depends on the selected loop check. To study such problems in a rigorous way, we introduced in our previous paper Apt, Bol and Klop [ABK] a number of natural concepts like soundness, completeness and relative strength of loop checks. We also introduced there the concept of a *simple loop check* arising when the loop checking mechanism does not depend on the analyzed logic program and showed that no sound and complete simple loop check exists, not even for programs without function symbols. Then we analyzed a number of natural simple loop checks based on the *equality* between goals, respectively resultants, of the derivations. Finally we introduced a class of logic programs, called *restricted programs*, in which a restricted form of recursion is allowed, and established the completeness of these loop checks for restricted programs without function symbols.

In this paper we study a more powerful class of simple loop checks based on the *inclusion* between goals, respectively resultants, of the derivations. 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 three natural classes of logic programs without function symbols. These completeness theorems make use of a simple version of Kruskal's Tree Theorem, called Higman's Lemma ([H]). While the use of this theorem to establish termination of term rewriting systems is well-known (see e.g. [D] or [K]), we have not encountered any applications of this theorem in the area of logic programming.

Finally we study a simple loop check introduced by Besnard [B], which we call a *context check*. As for the equality and subsumption checks, some variations on this context check can be made. It appears that the context checks are sound and complete for restricted programs without function symbols.

---

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

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

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

This research was partly supported by Esprit BRA-project 3020 Integration.

For reasons of space, most proofs are omitted.

2. Basic notions

In this section we recall some basic concepts presented in [ABK]. Throughout this paper we assume familiarity with the basic concepts and notations of logic programming as described in [L]. For two substitutions σ and τ, we write $\sigma \leq \tau$ when σ is more general than τ and for two expressions E and F, we write $E \leq F$ when F is an instance of E. We then say that F is *less general* than E. An SLD-derivation step from a goal G, using a clause C and an idempotent mgu θ, to a goal H is denoted as $G \Rightarrow_{C,\theta} H$. By an SLD-derivation we mean an SLD-derivation in the sense of [L] *or an initial fragment (subderivation) of it.*

2.1 Loop checks

The purpose of a loop check is to prune every infinite SLD-tree to a finite subtree of it containing the root. We define a loop check as a set of SLD-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 | the SLD-derivation from the root to G is in L(P) }. We shall usually make this conversion implicitly.

In this paper, we shall restrict ourselves to 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. This leads us to the following definitions.

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

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

In the following definition, by a *variant* of a derivation D we mean a derivation D' in which in every derivation step, atoms in the same positions are selected and the same program clauses 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 [LS]). 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. □

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

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

EXAMPLE 2.5 (see also [B] and [vG]).
We define the *Variant of Atom* (*VA*) check as:
$VA = RemSub(\{ D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$ 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
- introduced while resolving $A'\theta_{i+1}\ldots\theta_j$, the further instantiated version of A', that is selected in G_j }). □

2.2 Soundness and completeness

The most important property of a loop check is definitely that using it does not result in a loss of success. Even the loss of solutions is undesirable. 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 2.6.
i) A loop check L is *weakly sound* if for every program P and goal G, and SLD-tree T of P∪{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∪{G}: if T contains a successful branch with a computed answer substitution σ, then $f_{L(P)}(T)$ contains a successful branch with a computed answer substitution σ' such that $\sigma' \leq \sigma$.
iii) A loop check L is *shortening* if for every program P and goal G, and SLD-tree T of P∪{G}: if T contains a successful branch D with a computed answer substitution σ, then either $f_{L(P)}(T)$ contains D or $f_{L(P)}(T)$ contains a successful branch D' with a computed answer substitution σ' such that $\sigma' \leq \sigma$ and $|D'| < |D|$. □

Obviously, a shortening loop check is sound, and a sound loop check is weakly sound. In [ABK] it is shown that the VA check of Example 2.5, although intuitively appealing, is not even 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 2.7.
A loop check L is *complete* (*w.r.t. a selection rule* ***R***) if every infinite SLD-derivation (via **R**) is pruned by L. □

In general, comparing loop checks is difficult. The following relation comparing loop checks is not very general: most loop check will be incomparable with respect to it. Nevertheless it turns out to be very useful.

DEFINITION 2.8.
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∪{G} contains a subderivation $D_1 \in L_1(P)$. □

In other words, L_1 is stronger than L_2 if every SLD-derivation that is pruned by L_2 is also pruned by L_1. Note that the definition implies that every loop check is stronger than itself. The following theorem 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 2.9 (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. Straightforward. □

2.3 The existence of sound and complete loop checks

The undecidability of the halting problem implies that there cannot be a sound and complete loop check for logic programs in general, as logic programming has the full power of recursion theory. So our first step is to rule out programs that compute over an infinite domain. We shall do so by restricting our attention to programs without function symbols, so called *function-free* programs.

So our question can be reformulated as: is there a sound and complete loop check for function-free programs? In this paper, we shall only address this question for simple loop checks.

THEOREM 2.10. *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): $D = \leftarrow A(x_0),B(x_0) \Rightarrow \leftarrow A(x_1),S(x_1,x_0),B(x_0) \Rightarrow \leftarrow A(x_2),S(x_2,x_1),S(x_1,x_0),B(x_0) \Rightarrow \ldots$.

Since L is a complete loop check, D 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 that D 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 D to an SLD-tree of $P\cup\{G\}$ (still using the leftmost selection rule), we see that the only successful branch of this SLD-tree of $P\cup\{G\}$ goes via the goal that is pruned by L. Hence L is not weakly sound. □

3. Equality checks

In this section, we introduce some simple loop checks. For each of them, there exist two versions: the first one is weakly sound, the second one shortening. The second, shortening version is obtained by adding an 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 $G_0 \Rightarrow_{C1,\theta 1} G_1 \Rightarrow \ldots$, to every goal $G_i = \leftarrow S_i$ there corresponds the resultant $R_i = S_0\theta_1\ldots\theta_i\leftarrow S_i$. Resultants were introduced in [LS].
2. Replace *variant* by *instance*. This yields the *Instance of Atom* (*IA*) check. This check is still unsound: it is even stronger than the VA check. Besnard [B] has introduced a weakly sound version of this loop check. This check and related ones (derived from VA; shortening versions) are discussed in section 5. We shall refer to these checks as the *context checks*.
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.

   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. These checks 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. [CL]). 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 4.

The equality checks are studied in detail in [ABK]. In the rest of this section we recall the basic definitions and results. 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.

DEFINITION 3.1.

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

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

DEFINITION 3.2.
For Type ∈ {L,M}, the *Equals Variant/Instance of Resultant*$_{Type}$ check is the set of SLD-derivations
$EVR/EIR_{Type} = RemSub(\{ D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$ such that for some i,
$0 \le i < k$, there is a renaming/substitution τ such that $G_k =_{Type} G_i\tau$ and
$G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau \})$. □

Usually, once a loop check is defined, we shall present two kinds of results. First, we prove it soundness (in fact, the loop checks we present are either weakly sound or shortening). After that, we define one or more classes of programs, for which we then prove that the loop check is complete.

THEOREM 3.3 (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. See [ABK]. □

DEFINITION 3.4.
The *dependency graph* D_P of a program P is a directed graph whose nodes are the predicate symbols of P and
$(p,q) \in D_P$ iff there is a clause in P using p in its head and q in its body.
D_P^* is the reflexive, transitive closure of D_P. When $(p,q) \in D_P^*$, we say that p *depends on* q.
For a predicate symbol p, the *class of p* is the set of predicate symbols p 'mutually depends' on:
$cl_P(p) = \{ q \mid (p,q) \in D_P^* \text{ and } (q,p) \in D_P^* \}$. □

DEFINITION 3.5.
Given an atom A, let *rel(A)* denote its predicate symbol. Let P be a program.
A clause $A_0 \leftarrow A_1,\ldots,A_n$ ($n \ge 0$) is called *restricted w.r.t. P* if for i = 1,...,n-1, $rel(A_i)$ does not depend on $rel(A_0)$ in P. The atoms $A_1,\ldots,A_{n-1}$ are called *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. □

THEOREM 3.6 (Equality Completeness).
All equality checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.
PROOF. See [ABK]. □

4. Subsumption checks

As already stated, there are eight subsumption checks. We 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, $(a,c) \subseteq_L (a,b,c)$.

DEFINITION 4.1.
For Type ∈ {L,M}, the *Subsumes Variant/Instance of Goal*$_{Type}$ check is the set of SLD-derivations
$SVG/SIG_{Type} = RemSub(\{ D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$ such that for some i,
$0 \le i < k$, there is a renaming/substitution τ with $G_k \supseteq_{Type} G_i\tau \})$. □

DEFINITION 4.2.
For Type ∈ {L,M}, the *Subsumes Variant/Instance of Resultant*$_{Type}$ check is the set of SLD-derivations
$SVR/SIR_{Type} = RemSub(\{ D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$ such that for some i,
$0 \le i < k$, there is a renaming/substitution τ with $G_k \supseteq_{Type} G_i\tau$ and
$G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau \})$. □

LEMMA 4.3. *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 4.4.
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).

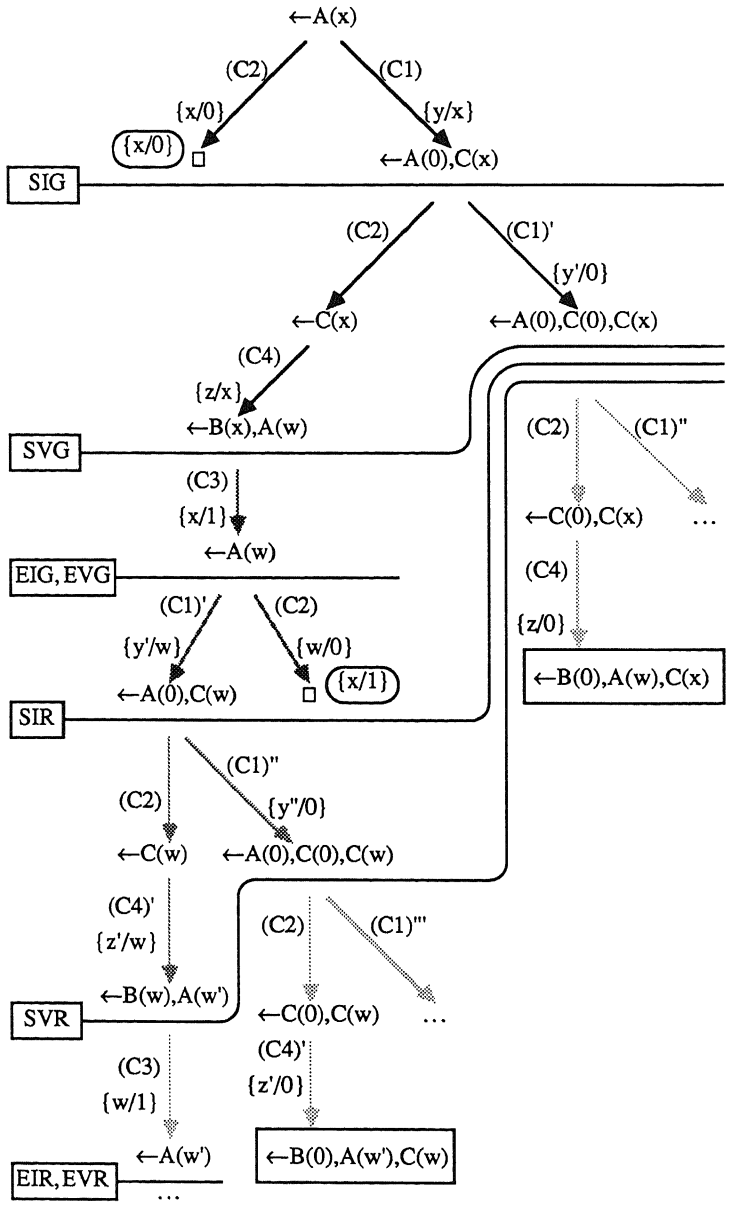

FIGURE 4.1

Figure 4.1 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 leftmost 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 clearly differs 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 there 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 every derivation step.

The loop checks based on goals all prune out 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. □

We have the following soundness results.

THEOREM 4.5 (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. □

We now shift our attention to the completeness issues. From the results of the previous section and the fact that subsumption checks are stronger than the corresponding equality checks we deduce the following result.

COROLLARY 4.6 (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 Theorem 3.6 and the Relative Strength Theorem 2.9. □

However, since the subsumption checks are stronger than the corresponding equality checks, 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 in the absence of function symbols. For P = {A(x)←A(y),S(y,x)}, a derivation of P∪{←A(x),B(x)} is not pruned by any of the subsumption checks, as was shown in Theorem 2.10.

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 shall now introduce two new classes of programs, preventing the first and the second event, respectively.

DEFINITION 4.7.
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. □

DEFINITION 4.8.
A clause C has the *single variable occurrence* property (in short *is 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. □

Clearly, in nvi programs the first event cannot occur, whereas in svo programs the second event is prevented. 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 [H]; for a formulation of the full version of Kruskal's Tree Theorem, see [D] or [K].) We also need a specialized formalization of the 'being a variant of' relation.

LEMMA 4.9 (Higman's Lemma). *Let $w_0,w_1,w_2,\ldots$ be an infinite sequence of (finite) words over a finite alphabet Σ. Then for some i and $k > i$, $w_i \subseteq_L w_k$.* □

DEFINITION 4.10.
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 ρ, $R_1\rho = R_2$ 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 $|G_2| \leq k$. □

LEMMA 4.11. *Suppose that the language L has no function symbols and finitely many predicate symbols and constants. Then for every finite set of variables X, every goal G and $k \geq 1$, the relation $\sim_{X,G,k}$ is an equivalence relation with only finitely many equivalence classes.*
PROOF. The proof is straightforward. □

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. In order to prove that the SVR_L check is complete for nvi programs, we prove that infinite derivations in which no new variables are introduced are pruned by the SVR_L check. We omit the proof that every derivation of an nvi program (and an arbitrary goal) has a variant that indeed does not introduce new variables.

DEFINITION 4.12.
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 4.13. *In the absence of function symbols, every infinite nvi SLD-derivation is pruned by 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 Σ the set of equivalence classes of $\sim_{\text{var}(G_0),G_0,1}$ as defined in Definition 4.10. By Lemma 4.11, Σ is finite. To apply Higman's Lemma 4.9 we represent for $j \geq 0$ a goal $G_j = \leftarrow A_{1j},\ldots,A_{n_j j}$ (or rather the corresponding resultant $G_0\theta_1\ldots\theta_j \leftarrow G_j$) as the word $[G_0\theta_1\ldots\theta_j \leftarrow A_{1j}],\ldots,[G_0\theta_1\ldots\theta_j \leftarrow A_{n_j j}]$ over Σ. The sequence of representations of G_0, G_1, G_2, ... yields an infinite sequence of words w_0, w_1, w_2, ... over Σ.

Now by Higman's Lemma 4.9, for some j and $k > j$: $[G_0\theta_1\ldots\theta_j \leftarrow A_{1j}],\ldots,[G_0\theta_1\ldots\theta_j \leftarrow A_{n_j j}] \subseteq_L [G_0\theta_1\ldots\theta_k \leftarrow A_{1k}],\ldots,[G_0\theta_1\ldots\theta_k \leftarrow A_{n_k k}]$. So by the definition of $\sim_{\text{var}(G_0),G_0,1}$ there exist renamings $\rho_1,\ldots,\rho_{n_j}$ which do not act on the variables in G_0, such that $(G_0\theta_1\ldots\theta_j \leftarrow A_{1j})\rho_1,\ldots,(G_0\theta_1\ldots\theta_j \leftarrow A_{n_j j})\rho_{n_j} \subseteq_L (G_0\theta_1\ldots\theta_k \leftarrow A_{1k}),\ldots,(G_0\theta_1\ldots\theta_k \leftarrow A_{n_k k})$.

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

LEMMA 4.14. *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.* □

Now we have the following completeness results.

THEOREM 4.15. *The SVR_L loop check is complete for function-free nvi programs.*
PROOF. By Lemma 4.3, 4.13 and 4.14. □

COROLLARY 4.16 (Subsumption Completeness 2).
All subsumption checks are complete for function-free nvi programs.
PROOF. By Theorem 4.15 and the Relative Strength Theorem 2.9. □

THEOREM 4.17 (Subsumption Completeness 3).
All subsumption checks are complete for function-free svo programs.
PROOF. The proof resembles the proof of Lemma 4.13. □

5. 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 ←A(x) or ←A(y) makes no difference, solving ←A(x),B(x) is essentially more difficult than solving ←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 τ and $\theta_{i+1}\ldots\theta_k$ should agree on x. This leads us to a loop check introduced by Besnard [B].

DEFINITION 5.1.
The *Variant/Instance Context check on Goals* is the set of SLD-derivations

CVG/CIG = 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, $0 \le i \le j < k$, there is a renaming / substitution τ 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 $G_i = \leftarrow(A,S)$, 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 $x \in$ var(A),
$-\ x\theta_{i+1}\ldots\theta_k$ for $x \in$ var(S),
so for $x \in$ var(A) ∩ var(S), we have $x\tau = x\theta_{i+1}\ldots\theta_k$.

The following example clarifies the use of the context checks.

EXAMPLE 5.2.
Let P = { A(0) ← (C1),
B(1) ← (C2),
A(x) ← A(y) (C3),
C ← A(x),B(x) (C4) },
let G = ← C.
We apply the CIG check on two SLD-trees of P∪{G}, via the leftmost and rightmost selection rule, respectively. This yields the trees in Figure 5.1.

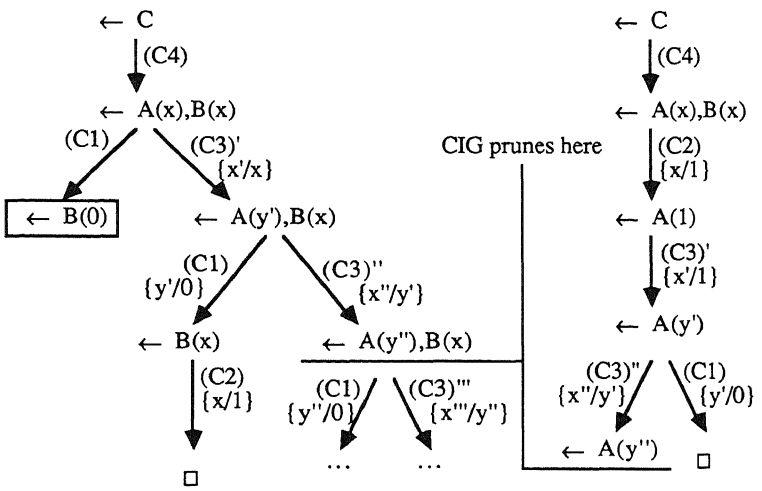

FIGURE 5.1

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 $\leftarrow A(y'),B(1)$: *not* an instance of $\leftarrow A(x),B(x)$. □

In Example 4.4, the context checks act exactly in the same way as the corresponding subsumption 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 5.3.
The *Variant/Instance Context check on Resultants* is the set of SLD-derivations
$CVR/CIR = RemSub(\{ D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$ such that for some i and j, $0 \le i \le j < k$, there is a renaming/substitution τ such that $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$ and 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: $x\theta_{i+1}\ldots\theta_k = x\tau$ $\})$. □

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.'

The following theorems state the soundness and completeness results for the context checks.

THEOREM 5.4 (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. □

THEOREM 5.5 (Context Completeness 1).
All context checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.
PROOF. The proof resembles the proof for the equality checks as presented in [ABK], but is more complex. □

THEOREM 5.6 (Context Completeness 2). *All context checks are complete for function-free nvi programs.*
PROOF. The proof resembles the proof of Lemma 4.13, but is more complex. Then Lemma 4.14 can be used. □

References

[ABK] K.R. APT, R.N. BOL and J.W. KLOP, *On the Safe Termination of PROLOG Programs*, in: Proceedings of the Sixth International Conference on Logic Programming, (G. Levi and M. Martelli eds.), MIT Press, Cambridge Massachusetts, 1989, 353-368.

[B] Ph. BESNARD, *On Infinite Loops in Logic Programming*, Internal Report 488, IRISA, Rennes, 1989.

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

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

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

[D] N. DERSHOWITZ, *A note on Simplification Orderings*, Information Processing Letters 9, 1979, 212-215.

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

[H] G. HIGMAN, *Ordering by divisibility in abstract algebra's*, Proceedings of the London Mathematical Society (3) 2 (7), 1952, 215-221.

[K] J.B. KRUSKAL, *Well-Quasi-Ordering, the Tree Theorem, and Vazsonyi's Conjecture*, Transactions of the AMS 95, 1960, 210-225.

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

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

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

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

[SI] H. SEKI and H. ITOH, *A Query Evaluation Method for Stratified Programs under the Extended CWA*, in: Proceedings of the Fifth International Conference on Logic Programming, MIT Press, Cambridge Massachusetts, 1988, 195-211.

[V] L. VIEILLE, *Recursive Query Processing: The Power of Logic*, Theoretical Computer Science 68, No. 2, 1989.