Theoretical Computer Science 104 (1992) 3-28
Elsevier

Generalizing completeness results for loop checks in logic programming*

Roland N. Bol

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

Abstract

Bol, R.N., Generalizing completeness results for loop checks in logic programming, Theoretical Computer Science 104 (1992) 3-28.

Loop checking is a mechanism for pruning infinite SLD-derivations. In (Bol, Apt and Klop, 1991) *simple loop checks* were introduced and their soundness, completeness and relative strength was studied. Since no sound and complete simple loop check exists even in the absence of function symbols, subclasses of programs were determined for which the (sound) loop checks introduced by Bol et al. are complete.

In this paper, the *Generalization Theorem* is proved. This theorem presents a method to extend (under certain conditions) a class of programs for which a given loop check is complete to a larger class, for which the loop check is still complete. Then this theorem is applied to the results of Bol et al., giving rise to stronger completeness theorems.

It appears that unnecessary complications in the proof of the theorem can be avoided by introducing a *normal form* for SLD-derivations, allowing only certain most general unifiers. This normal form might have other applications than those in the area of loop checking.

1. Introduction

Logic programming is advocated as a formalism for writing executable specifications. However, even when such 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 stop loops in SLD-derivations (see [3, 6, 7, 8, 13, 14, 16, 17]).

The loop checking mechanisms studied in this paper are the *simple loop checks* introduced by Apt, Bol and Klop [2]. Simple loop checks have the following properties:

Correspondence to: R.N. Bol, Department of Computer and Information Science, University of Linköping, S-58183 Linköping, Sweden. Email: rolbo@ida.liu.se.

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

0304-3975/92/$05.00 © 1992—Elsevier Science Publishers B.V. All rights reserved

- the search space is reduced by *pruning* goals; pruning a goal means that *all* its descendants are removed;
- whether a goal is pruned depends only on the derivation leading to that goal (i.e., not on other parts of the search space, and not on the program).

This excludes more complicated techniques such as tabulation, which are studied in [16, 17].

To study simple loop checks in a rigorous way, Apt, Bol and Klop introduced a number of natural concepts like soundness (no answers are lost), completeness (the resulting search space is finite) and relative strength of loop checks. It is obvious that a sound loop check cannot be complete for all logic programs. It was even shown that a sound simple loop check cannot be complete for all programs without function symbols.

A number of natural simple loop checks was introduced in [5]. These loop checks were proven to be sound, but only complete for certain classes of function-free programs. For each of these loop checks, one or more such classes were determined.

Here, the problem of finding classes of programs for which a simple loop check is complete is addressed in more generality. The main theorem of this paper is called the *Generalization Theorem*, since it allows us to generalize certain completeness results: given that a loop check L is complete for a class of programs $\mathscr{C}$, we may conclude that L is also complete (w.r.t. the leftmost selection rule) for a class of programs extending $\mathscr{C}$, provided that L and $\mathscr{C}$ satisfy some natural conditions.

Basically, the theorem is only applicable to a class of programs $\mathscr{C}$ if $\mathscr{C} = \{P \mid \text{every clause in program } P \text{ satisfies } Pr\}$, for some property Pr of clauses that is "local" to clauses (that is, whether a clause satisfies Pr is independent of the rest of the program). We say that $\mathscr{C}$ is the class of *Pr-programs.* By allowing the addition of atoms in clauses that cannot give rise to recursive calls to the head of the clause (so called *nonrecursive* atoms), the class of *nr-extended Pr-programs* is obtained.

The Generalization Theorem states that if the loop check L is complete for Pr-programs, then L is also complete for function-free nr-extended Pr-programs, provided that the nonrecursive atoms are resolved before other atoms are selected. For simplicity, this is achieved by using the leftmost selection rule, and putting the nonrecursive atoms to the left of the other atoms in the clause. Notice that the property of being a nonrecursive atom is *not* local to clauses; therefore the theorem cannot be applied repeatedly.

In the proof of the Generalization Theorem, we make use of certain properties of SLD-derivations that are in a normal form, tentatively called *normal* SLD-derivations. In normal SLD-derivations, only certain mgu's may be used. This normal form may well have other applications than those in the area of loop checking.

Once the proof of the Generalization Theorem is given, it is applied to several completeness results presented in [5] concerning two loop checks that indeed satisfy the conditions of the Generalization Theorem. The extension of some of these completeness results is straightforward, whereas for others a more elaborate analysis is needed.

2. Basic notions

In this section we recall the basic notions concerning loop checking, as presented in [5]. Throughout this paper we assume familiarity with the concepts and notations of logic programming as described in [9]. For two substitutions σ and τ, we write $\sigma \leqslant \tau$ when σ is more general than τ and for two expressions E and F, we write $E \leqslant F$ if F is an instance of E. An SLD-derivation step from a goal G, using a clause C and an mgu θ, to a goal H is denoted as $G \Rightarrow_{C,\theta} H$. By an SLD-derivation we mean an SLD-derivation in the sense of [9] *or an initial segment of it.* For a program P, L_P denotes the language of P.

2.1. Loop checks

The purpose of a loop check is to prune every infinite SLD-tree to a subtree of it containing the root. We define a loop check as a set of SLD-derivations: the derivations that are pruned exactly at their last node. Such a set of SLD-derivations L can be extended in a canonical way to a function f_L from SLD-trees to SLD-trees by pruning in an SLD-tree T the nodes in $\{G \mid \text{the SLD-derivation from the root of } T \text{ to } G \text{ is in } L\}$. We shall usually make this conversion implicitly.

Definition 2.1. Let L be a set of SLD-derivations.

$$\mathrm{RemSub}(L) = \{D \in L \mid L \text{ does not contain a proper subderivation of } D\}.$$

L is *subderivation free* if $L = \mathrm{RemSub}(L)$.

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

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 [10]). 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 L is closed under variants and subderivation free.

In [2], loop checks are treated in a more general way. There nonsimple loop checks occur: their behaviour may depend on the program the interpreter is confronted with. In this paper, we shall only consider simple loop checks. Therefore we shall usually omit the qualification "simple".

Definition 2.3. Let L be a loop check. An SLD-derivation D of $P \cup \{G\}$ is *pruned by* L if L contains a subderivation D' of D.

2.2. Soundness and completeness

Using a loop check should definitely not result in a loss of success. Even losing individual solutions is usually undesirable. On the other hand, 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. This leads to the following definitions.

Definition 2.4 (*Soundness*). (i) A loop check L is *weakly sound* if for every program P, goal G, and SLD-tree T of $P \cup \{G\}$ we have: if T contains a successful branch, then $f_L(T)$ contains a successful branch.

(ii) A loop check L is *sound* if for every program P, goal G, and SLD-tree T of $P \cup \{G\}$ we have: if T contains a successful branch with a computed answer $G\sigma$, then $f_L(T)$ contains a successful branch with a computed answer $G\sigma' \leq G\sigma$.

Definition 2.5 (*Completeness*). A loop check L is *complete w.r.t. a selection rule* $\boldsymbol{R}$ *for a class of programs* $\mathscr{C}$, if for every program $P \in \mathscr{C}$ and goal G in L_P, every infinite SLD-derivation of $P \cup \{G\}$ via $\boldsymbol{R}$ is pruned by L.

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

Definition 2.6. Let L_1 and L_2 be loop checks. L_1 is *stronger than* L_2 if every SLD-derivation $D_2 \in L_2$ contains a subderivation $D_1 \in L_1$.

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. Notice that the definition implies that every loop check is stronger than itself. The following theorem enables 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 2.7 (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_2 is complete (w.r.t. a selection rule $\boldsymbol{R}$ for a class of programs $\mathscr{C}$), then L_1 is complete (w.r.t. $\boldsymbol{R}$ for the class of programs $\mathscr{C}$).*

Proof. Straightforward. □

The undecidability of the halting problem implies that there cannot be a weakly 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, for which the Herbrand Universe is finite. However, it appears that even with this restriction, there is no weakly sound and complete loop check.

Theorem 2.8. *There is no weakly sound and complete simple loop check for function-free programs.*

Proof. See [2]. See also [6, Theorem 4.7]. □

It was shown in [2] that weakly sound and complete nonsimple loop checks exist for function-free programs, but that nonsimple loop checks are in general too powerful. A loop check that depends only on "syntactical properties" of the program could be useful, but this restriction is hard to formalize. So a nonsimple loop check could be based on (for example) the set of correct answers of the program (as the program is function-free, this set is finite modulo variants). Once this set is computed by the loop check in some way, there is no point in reconstructing it by building an SLD-tree pruned by this loop check.

Therefore, it is more useful to develop some simple loop checks, and to find classes of programs for which these loop checks are complete.

2.3. Some simple loop checks

In this section we introduce three groups of weakly sound simple loop checks. How we arrived at these loop checks and why we thought them to be interesting was discussed in [5]. Here we restrict ourselves to giving the definitions and basic theorems (without proofs).

The first group of loop checks we consider consists of the so-called "equality checks". In fact, each equality check should be defined separately. This would yield 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 $=_L$. (In [5], also variants of these loop checks are considered, regarding goals as *multisets.*)

Definition 2.9 (*Equality checks based on goals*). The *Equals Variant/Instance of* $Goal_{List}$ check is the set of SLD-derivations

$$\mathrm{EVG/EIG}_L = \mathrm{RemSub}(\{D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k) \text{ such that for some } i,\ 0 \leq i < k, \text{ there is a renaming/substitution } \tau \text{ such that } G_k =_L G_i\tau\}).$$

Definition 2.10 (*Equality checks based on resultants*). The *Equals Variant/Instance of Resultant*$_{List}$ check is the set of SLD-derivations

$$\mathrm{EVR/EIR}_L = \mathrm{RemSub}(\{D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$$

such that for some i, $0 \leq i < k$, there is a renaming/substitution τ such that $G_k =_L G_i\tau$ and $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau\})$.

Theorem 2.11 (Equality soundness). *The equality checks are weakly sound loop checks. Moreover, the equality checks based on resultants are sound.*

Proof. See [2]. □

We now define a class of programs for which the equality checks are complete in the absence of function symbols (as was shown in [2]). This class of programs is closely related to the class of programs discussed in [15]. For a formal definition, we use the notion of the *dependency graph* D_P of a program P.

Definition 2.12. The *dependency graph* D_P of a program P is a directed graph whose nodes are the predicate symbols appearing in P and $(p, q) \in D_P$ iff there is a clause in P using p in its head and q in its body.

D_P^* is the reflexive, transitive closure of D_P. When $(p, q) \in D_P^*$, we say that p *depends on* q in P. For a predicate symbol p, the *class of* p is the set of predicate symbols p "mutually depends" on:

$$\mathrm{cl}_P(p) = \{q \mid (p, q) \in D_P^* \text{ and } (q, p) \in D_P^*\}.$$

Definition 2.13. Given an atom A, let rel(A) denote its predicate symbol. Let P be a program. In a clause $H \leftarrow A_1, \ldots, A_n$ ($n \geq 0$) of P, an atom A_i ($1 \leq i \leq n$) is called *recursive* if rel(A_i) depends on rel(H) in P. Otherwise, the atom is called nonrecursive.

A clause $H \leftarrow A_1, \ldots, A_n$ is *restricted w.r.t. P* if $A_1, \ldots, A_{n-1}$ are *nonrecursive*.

A program P is called *restricted* if every clause in P is restricted w.r.t. P.

Theorem 2.14 (Equality completeness). *All equality checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.*

Proof. See [2]. □

The second group of loop checks we consider consists of the so-called "subsumption checks". Again, we define them by means of two parametrized definitions. The inclusion relation between goals regarded as lists is denoted by $\subseteq_L$. Note: $L_1 \subseteq_L L_2$ if all elements of L_1 occur in the same order in L_2; they do not need to occur on adjacent positions. For example, $(a, c) \subseteq_L (a, b, c)$.

Definition 2.15 (*Subsumption checks based on goals*). The *Subsumes Variant/Instance of* $Goal_{List}$ check is the set of SLD-derivations

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

Definition 2.16 (*Subsumption checks based on resultants*). The *Subsumes Variant/Instance of* $Resultant_{List}$ check is the set of SLD-derivations

$$\mathrm{SVR/SIR}_L = \mathrm{RemSub}(\{D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k) \text{ such that for some } i,\ 0 \leq i < k, \text{ there is a renaming/substitution } \tau \text{ with } G_k \supseteq_L G_i\tau \text{ and } G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau\}).$$

Theorem 2.17 (Subsumption soundness). *The subsumption checks are weakly sound loop checks. Moreover, the subsumption checks based on resultants are sound.*

Proof. See [5]. □

We now show three classes of programs for which the subsumption checks are complete in the absence of function symbols. Since the subsumption checks are stronger than the "corresponding" equality checks, the first result follows immediately.

Theorem 2.18 (Subsumption completeness 1). *All subsumption checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.*

Proof. By the Relative Strength Theorem 2.7 and the Equality Completeness Theorem 2.14. □

The remaining two classes of programs for which the subsumption checks are complete in the absence of function symbols are the following.

Definition 2.19. A clause C is *nonvariable 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 2.20. 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.

Theorem 2.21 (Subsumption completeness 2). *All subsumption checks are complete for function-free nvi programs.*

Proof. See [5]. □

Theorem 2.22 (Subsumption completeness 3). *All subsumption checks are complete for function-free svo programs.*

Proof. See [5]. □

The third group of loop checks we consider are based on a loop check introduced by Besnard [3]. They are called "context checks" in [5]. Again we have weakly sound versions based on goals and sound versions based on resultants.

Definition 2.23 (*Context checks based on goals*). The *Variant/Instance Context check based on Goals* is the set of SLD-derivations

$$\mathrm{CVG/CIG} = \mathrm{RemSub}(\{D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$$

such that for some i and j, $0 \leq i \leq j < k$, 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 an attempt to resolve $A\theta_{i+1} \ldots \theta_j$, the further instantiated version of A in G_j and for every variable x that occurs both inside and outside of A in G_i, $x\theta_{i+1} \ldots \theta_k = x\tau\})$.

Definition 2.24 (*Context checks based on resultants*). The *Variant/Instance Context check based on Resultants* is the set of SLD-derivations

$$\mathrm{CVR/CIR} = \mathrm{RemSub}(\{D \mid D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{k-1} \Rightarrow_{C_k,\theta_k} G_k)$$

such that for some i and j, $0 \leq i \leq j < k$, 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 an attempt to resolve $A\theta_{i+1} \ldots \theta_j$, the further instantiated version of A in G_j and for every variable x that occurs both inside and outside of A in G_i, $x\theta_{i+1} \ldots \theta_k = x\tau\})$.

Theorem 2.25 (Context soundness). *The context checks are weakly sound loop checks. Moreover, the context checks based on resultants are sound.*

Proof. See [5]. □

For the context checks the same completeness results have been proven as for the subsumption checks.

Theorem 2.26 (Context completeness). *All context checks are complete for function-free restricted programs, nvi programs and svo programs.*

Proof. See [5]. □

3. The choice of most general unifiers

We now divert for a moment from the subject of loop checking. It appears that, in order to prove the Generalization Theorem in Section 4, we need some auxiliary results regarding SLD-derivations. These results can be obtained by putting extra requirements on the most general unifiers in those derivations. In this section we introduce these requirements and show why we consider them to be justifiable. Finally we prove the lemmas needed in Section 4.

3.1. Relevant and idempotent mgu's

The general feeling is that, in order to obtain mathematical elegancy, the definition of an SLD-derivation must leave the choice of variables as free as possible. However, during the evolution of this definition, the allowable freedom was continuously overestimated. For example, in the first edition of [9], the input clause was only standardized apart from the current goal, and not from the goals and clauses preceding it. Thereby, the undesirable derivation of Fig. 1 was allowed. In the second edition of [9], this has been corrected. However, yet another anomalous derivation is shown in Fig. 2.

It is not clear whether or not this derivation is allowed in [9] (does z *appear* in the derivation before the goal $\leftarrow r$?), but in [1] it definitely is, although later on in [1] it is assumed that all mgu's are relevant (a unifier of A and B is *relevant* if it

$$\begin{array}{l}
\leftarrow p(x) \\
\quad \Big\downarrow \begin{array}{l} p(y) \leftarrow q(a) \\ \{x/y\} \end{array} \\
\leftarrow q(a) \\
\quad \Big\downarrow \begin{array}{l} q(y) \leftarrow \\ \{y/a\} \end{array} \\
\square
\end{array}$$

Fig. 1.

$$\begin{array}{l}
\leftarrow p(x) \\
\quad \Big\downarrow \begin{array}{l} p(y) \leftarrow q \\ \{x/y\} \end{array} \\
\leftarrow q \\
\quad \Big\downarrow \begin{array}{l} q \leftarrow r \\ \{y/z,\ z/y\} \end{array} \\
\leftarrow r \\
\quad \Big\downarrow \begin{array}{l} r \leftarrow s(z) \\ \varepsilon \end{array} \\
\leftarrow s(z) \\
\quad \Big\downarrow \begin{array}{l} s(a) \leftarrow \\ \{z/a\} \end{array} \\
\square
\end{array}$$

Fig.2.

acts only on variables in A and B) and idempotent. It appears that the requirement that the mgu is relevant is redundant, as idempotent mgu's are always relevant. First of all, from now on we assume that only idempotent mgu's are used. Under this assumption we prove some properties of SLD-derivations. The first property we prove is that a variable cannot occur somewhere in the derivation, disappear and later reappear. (For an SLD-derivation D, $|D|$ denotes its length, i.e. the number of goals in it.)

Lemma 3.1. *Let* $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1} \Rightarrow \cdots)$ *be an SLD-derivation and let* $0 \leq i < k$ $(<|D|)$. *If* $x \in \mathrm{var}(C_{i+1}) \cup \mathrm{var}(G_i)$ *and* $x \in \mathrm{var}(G_k)$, *then for all* j, $i < j \leq k$, $x \in \mathrm{var}(G_j)$ *and* $x\theta_j = x$.

Proof. We use induction on j from k down to i. $x \in \mathrm{var}(G_k)$ is given. Now assume that $i \leq j < k$ and $x \in \mathrm{var}(G_{j+1})$. We prove that $x\theta_{j+1} = x$ and that if $j > i$, $x \in \mathrm{var}(G_j)$. Let $G_j = \leftarrow(S_1, A, S_2)$, where A is the selected atom in G_j. Let $C_{j+1} = H \leftarrow S_3$. ($S_1$, S_2 and S_3 are possibly empty sequences of atoms.) Then θ_{j+1} is an idempotent mgu of A and H and $G_{j+1} = \leftarrow(S_1, S_3, S_2)\theta_{j+1}$. So $x \in \mathrm{var}(S_1, S_3, S_2)\theta_{j+1}$, hence for some $y \in \mathrm{var}(S_1, S_3, S_2)$, $x \in \mathrm{var}(y\theta_{j+1})$. Two cases arise.

- $x = y$. Thus $x\theta_{j+1} = x$. Also, if $j > i$, $x \notin \mathrm{var}(S_3)$ since $x \in \mathrm{var}(C_{i+1}) \cup \mathrm{var}(G_i)$ and S_3 is standardized apart. So $x \in \mathrm{var}(S_1, S_2) \subseteq \mathrm{var}(G_j)$.
- $x \neq y$. Then $x \in \mathrm{var}(\mathrm{ran}(\theta_{j+1}))$, and since θ_{j+1} is idempotent, $x \notin \mathrm{dom}(\theta_{j+1})$, so $x\theta_{j+1} = x$. Also, since θ_{j+1} is relevant, $x \in \mathrm{var}(A, H)$. If $j > i$, $x \notin \mathrm{var}(H)$ since $x \in \mathrm{var}(C_{i+1}) \cup \mathrm{var}(G_i)$ and H is standardized apart. So $x \in \mathrm{var}(A) \subseteq \mathrm{var}(G_j)$.

So in both cases we have $x\theta_{j+1} = x$ and if $j > i$ also $x \in \mathrm{var}(G_j)$. □

The following definition captures the notion that two variables in a goal are related, i.e. that they might be unified in an attempt to refute the goal. (This notion can be compared with the notion of *connected (sets of) predicate instances* in [12].) We then prove that when two variables occur unrelated in a certain goal, they cannot be related in any goal later in the derivation.

Definition 3.2. Let S be a set of atoms. We define the relation $\sim_S$ on variables as:

$$x \sim_S y \text{ if there is an atom } A \text{ in } S \text{ such that } x, y \in \mathrm{var}(A).$$

Obviously, $\sim_S$ is a symmetrical relation. Now we define the relation $\approx_S$ to be the transitive and reflexive closure of $\sim_S$. Then $\approx_S$ is an equivalence relation.

An equivalence class of $\approx_S$ is called a *chain (in S)*. For $x \in \mathrm{var}(S)$, the chain of x is denoted by $C_S(x)$, or $C(x)$ whenever S is clear from the context.

Lemma 3.3. *Let* $D = G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots$ *be an SLD-derivation and let* $0 < i$ $(<|D|)$. *If* $x \approx_{G_i} y$ *and* $x, y \in \mathrm{var}(G_{i-1})$, *then* $x \approx_{G_{i-1}} y$.

Proof. Let $G_{i-1} = \leftarrow(A, R)$, where A is the selected atom in G_{i-1}. Let $C_i = H \leftarrow S$ and let θ_i be an mgu of A and H. Then $G_i = \leftarrow(S, R)\theta_i$. Assume $x \neq y$ (for $x = y$

the claim is trivial). Since $x \approx_{G_i} y$, there is a sequence of variables $x = w_1, w_2, \ldots, w_{2n} = y$ in G_i such that $w_{2j-1} \approx_{S\theta_i} w_{2j}$ for $1 \leq j \leq n$ and $w_{2j} \sim_{R\theta_i} w_{2j+1}$ for $1 \leq j < n$.

For $1 < j < 2n$, every variable $w_j \in \mathrm{var}(R\theta_i)$, so we can choose for it a corresponding variable $z_j \in \mathrm{var}(R) \subseteq \mathrm{var}(G_{i-1})$ such that $w_j \in \mathrm{var}(z_j\theta_i)$. Since θ_i is idempotent, and $x, y \in \mathrm{var}(G_{i-1}) \cap \mathrm{var}(G_i)$, we can choose $z_1 = w_1 = x = x\theta_i$ and $z_{2n} = w_{2n} = y = y\theta_i$. Now let $1 \leq j < 2n$.

We prove that $z_j \approx_{G_{i-1}} z_{j+1}$. Two cases arise.

- j is even, so $w_j \sim_{R\theta_i} w_{j+1}$. Then there is an atom B in R such that $w_j, w_{j+1} \in \mathrm{var}(B\theta_i)$. So we have variables $v_j, v_{j+1} \in \mathrm{var}(B)$ such that $w_j \in \mathrm{var}(v_j\theta_i)$ and $w_{j+1} \in \mathrm{var}(v_{j+1}\theta_i)$. So $v_j \sim_B v_{j+1}$, and hence $v_j \sim_R v_{j+1}$. For v_j (and analogously for v_{j+1}) two subcases arise.
  - $v_j = z_j$. Then $v_j \approx_A z_j$.
  - $v_j \neq z_j$. Then, since $w_j \in \mathrm{var}(v_j\theta_i) \cap \mathrm{var}(z_j\theta_i)$ and θ_i is relevant, we have $v_j, z_j \in \mathrm{var}(A)$. Hence $v_j \approx_A z_j$.

  Therefore $z_j \approx_A v_j \sim_R v_{j+1} \approx_A z_{j+1}$, so $z_j \approx_{G_{i-1}} z_{j+1}$.
- j is odd, so $w_j \approx_{S\theta_i} w_{j+1}$. If $w_j = w_{j+1}$, then $z_j = z_{j+1}$, so $z_j \approx_{G_{i-1}} z_{j+1}$. Otherwise, we can prove that $z_j \in \mathrm{var}(A)$ (and analogously $z_{j+1} \in \mathrm{var}(A)$). Again two subcases arise.
  - $z_j\theta_i \neq z_j$. Then $z_j \in \mathrm{var}(A)$: θ_i is relevant and $z_j \in \mathrm{var}(G_{i-1})$, so $z_j \notin \mathrm{var}(H)$.
  - $z_j\theta_i = z_j$. Then $w_j = z_j \in \mathrm{var}(S\theta_i)$, say $v_j \in \mathrm{var}(S)$ such that $z_j \in \mathrm{var}(v_j\theta_i)$. Then $v_j\theta_i \neq v_j$, since $v_j \in \mathrm{var}(S)$, $z_j \in \mathrm{var}(G_{i-1})$ and S is standardized apart. Therefore $v_j \in \mathrm{var}(H)$, and hence $z_j \in \mathrm{var}(A)$.

  Now $z_j \sim_A z_{j+1}$, so $z_j \approx_{G_{i-1}} z_{j+1}$.

Therefore we have $x = z_1 \approx_{G_{i-1}} z_2 \approx_{G_{i-1}} z_3 \approx_{G_{i-1}} \cdots \approx_{G_{i-1}} z_{2n} = y$. $\square$

3.2. *Normal SLD-derivations*

In fact, it appears to be convenient to restrict the choice of the mgu even more by disallowing the "needless renaming of variables in a derivation". We explain this now. When we have a variable x in the selected atom of the goal which is to be unified with a variable y in the input clause, then two idempotent mgu's are available: $\{x/y\}$ and $\{y/x\}$.

When $\{x/y\}$ is chosen, it is likely that the variable y occurs further on in the derivation as a substitute for x, whereas x itself does not occur any more. On the other hand, if $\{y/x\}$ is chosen, the variable x is retained and the variable y will not occur in any goal of the derivation. Therefore the renaming from x to y is considered to be a needless renaming. So we choose $\{y/x\}$, thereby retaining the "older" variables x and adjusting the "newer" variable y.

A more indirect instance of the same principle is shown in the derivation

$$\leftarrow A(x) \Rightarrow_{A(x') \leftarrow B(x',y), \{x'/x\}} \leftarrow B(x, y) \Rightarrow_{B(z,z) \leftarrow, \{y/x, z/x\}} \square.$$

In the first step $\{x'/x\}$ is chosen for the reason described above. In the second step, the choice of $\{x/z, y/z\}$ is out of the question for the same reason. However, this

still leaves the choice between $\{x/y, z/y\}$ and $\{y/x, z/x\}$. Although x and y occur both in $B(x, y)$, x appears earlier in the derivation than y. Therefore we choose $\{y/x, z/x\}$, thereby again retaining the older variables x and adjusting the newer variable y.

It is important to note two things. Firstly, Lemma 3.1 says that a variable cannot be introduced, disappear, and reappear later on in the derivation, which would complicate the decision criterion given above. Secondly, the choice of the mgu is still nondeterministic, as is shown in the derivation

$$\leftarrow A \Rightarrow_{A \leftarrow B(x,y),\varepsilon} \leftarrow B(x, y) \Rightarrow_{B(z,z) \leftarrow, \{y/x, z/x\}} \square.$$

Here the choice between $\{y/x, z/x\}$ and $\{x/y, z/y\}$ is arbitrary.

We now formalize these intuitions.

Definition 3.4. Let $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots)$ be an SLD-derivation. For every variable x occurring in D, we define

$$\mathrm{tag}(x) = \begin{cases} 0 & \text{if } x \in \mathrm{var}(G_0), \\ i & \text{if } x \in \mathrm{var}(C_i). \end{cases}$$

D is a *normal* SLD-derivation if for every $i > 0$ (and $i < |D|$ when D is finite),
- θ_i is idempotent and
- for every variable $x \in \mathrm{var}(G_{i-1})$: if $x\theta_i$ is a variable, then $\mathrm{tag}(x) \geqslant \mathrm{tag}(x\theta_i)$.

Intuitively, the lower the tag of a variable is, the "older" it is. The following lemma shows that we may restrict our attention to normal SLD-derivations.

Lemma 3.5. *Every SLD-derivation has a normal variant.*

Proof. We introduce a slightly changed version of the unification algorithm of Martelli and Montanari [11]. Using this algorithm for computing the mgu yields a normal SLD-derivation.

When $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$ are to be unified, first the set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$ is constructed. This set is then transformed according to the following six rules:

(a) $E \dot\cup \{t = x\} \Rightarrow E \cup \{x = t\}$ if $t \notin \mathrm{VAR}$ or $\mathrm{tag}(t) < \mathrm{tag}(x)$,

(b) $E \dot\cup \{x = x\} \Rightarrow E$,

(c1) $E \dot\cup \{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\} \Rightarrow E \cup \{s_1 = t_1, \ldots, s_n = t_n\}$ $(n \geqslant 0)$,

(c2) $E \dot\cup \{f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)\} \Rightarrow$ failure if $f \neq g$,

(d1) $E \dot\cup \{x = t\} \Rightarrow E\{x/t\} \cup \{x = t\}$ if $x \notin \mathrm{var}(t)$ and $\mathrm{var}(E)$,

(d2) $E \dot\cup \{x = t\} \Rightarrow$ failure if $x \neq t$ and $x \in \mathrm{var}(t)$,

until none of these rules is applicable. (Here $\dot\cup$ denotes the disjoint union.) Now we take $\theta = \{x/t \mid (x = t) \in E\}$.

The change w.r.t. the original algorithm is in rule (a), where now tags are taken into account. Whenever $x\theta = y \neq x$, we have that $(x = y) \in E$ and no rules are applicable on E, hence $\mathrm{tag}(x) \geqslant \mathrm{tag}(y)$ (otherwise rule (a) would be applicable). Showing that the algorithm terminates and that a resulting substitution is indeed an idempotent mgu of $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$ is straightforward. □

3.3. Properties of normal SLD-derivations

In this section we prove some properties of normal SLD-derivations that appear to be needed in the next section. The reader who is not interested in such technical details is encouraged to skip this section.

Lemma 3.6. *Let $D = G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots$ be a normal SLD-derivation and let $0 \leqslant j < k$ $(<|D|)$. Let C be a chain in G_j. Then $C\theta_k \cap \mathrm{VAR} \subseteq C$.*

Proof. Let $x \in C$ and assume that $x\theta_k$ is a variable. We prove that $x\theta_k \in C$.

If $x\theta_k = x$ then clearly $x\theta_k \in C$.

Otherwise, $x \in \mathrm{var}(G_{k-1})$ $(x \notin \mathrm{var}(C_k)$ since θ_k is relevant and by standardizing apart, D is normal, $x \in \mathrm{var}(G_{k-1})$ and $x\theta_k$ is a variable, so $\mathrm{tag}(x) \geqslant \mathrm{tag}(x\theta_k)$. Hence $x\theta_k \notin \mathrm{var}(C_k)$. $x\theta_k \neq x$ and θ_k is relevant, so since θ_k is relevant and by standardizing apart $x\theta_k \in \mathrm{var}(G_{k-1})$. Thus x and $x\theta_k$ occur both in the selected atom of G_{k-1}. Therefore $x \approx_{G_{k-1}} x\theta_k$.

Also $\mathrm{tag}(x\theta_k) \leqslant \mathrm{tag}(x) \leqslant j$, thus by Lemma 3.1, for every i such that $j \leqslant i < k$, $x \in \mathrm{var}(G_i)$ and $x\theta_k \in \mathrm{var}(G_i)$. Applying Lemma 3.3 $k-1-j$ times yields that $x \approx_{G_j} x\theta_k$. Hence $x\theta_k \in C$. □

Corollary 3.7. *Let $D = G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots$ be a normal SLD-derivation of a function-free program P and G_0 and let $0 \leqslant j < k$ $(<|D|)$. Then* $\mathrm{var}(G_j\theta_k) \subseteq \mathrm{var}(G_j)$.

Proof. Let $x \in \mathrm{var}(G_j\theta_k)$. P is function-free, so for some $y \in \mathrm{var}(G_j)$, $x = y\theta_k$. Now by Lemma 3.6, $x = y\theta_k \in C_{G_j}(y)\theta_k \cap \mathrm{VAR} \subseteq C_{G_j}(y) \subseteq \mathrm{var}(G_j)$. □

Corollary 3.8. *Let $D = G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots$ be a normal SLD-derivation of a function-free program P and G_0 and let $0 \leqslant j < k$ $(<|D|)$. Then* $\mathrm{var}(G_j\theta_{j+1} \ldots \theta_k) \subseteq \mathrm{var}(G_j)$.

Proof. Repeatedly using Corollary 3.7, we have $\mathrm{var}((G_j\theta_{j+1})\theta_{j+2} \ldots \theta_k) \subseteq \mathrm{var}(G_j\theta_{j+2} \ldots \theta_k) \subseteq \cdots \subseteq \mathrm{var}(G_j\theta_k) \subseteq \mathrm{var}(G_j)$. □

Corollary 3.9. *Let $D = G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots$ be a normal SLD-derivation and let $0 \leqslant j < k$ $(<|D|)$. Let C be a chain in G_j. Then $C\theta_{j+1}\theta_k \cap \mathrm{VAR} \subseteq C\theta_{j+1}$ and $C\theta_{j+1} \ldots \theta_k \cap \mathrm{VAR} \subseteq C\theta_{j+1}$.*

Proof. If $j+1=k$, then the claim is trivial. So assume $j+1<k$.

Let $x \in C\theta_{j+1}$ and assume that $x\theta_k$ is a variable. We prove that $x\theta_k \in C\theta_{j+1}$. By Lemma 3.6, $x \in C\theta_{j+1} \cap \mathrm{VAR}$ implies $x \in C$. Therefore, again by Lemma 3.6, $x\theta_k \in C\theta_k \cap \mathrm{VAR} \subseteq C$. Two cases arise.

- $x\theta_k\theta_{j+1} = x\theta_k$. Then $x\theta_k \in C$ implies $x\theta_k = x\theta_k\theta_{j+1} \in C\theta_{j+1}$.
- $x\theta_k\theta_{j+1} \neq x\theta_k$. Then $x\theta_k \notin \mathrm{var}(G_{j+1})$, since θ_{j+1} is idempotent. As we have $x\theta_k \in C \subseteq \mathrm{var}(G_j)$, $x\theta_k \notin \mathrm{var}(G_{k-1})$ by Lemma 3.1 and $x\theta_k \notin \mathrm{var}(C_k)$ by standardizing apart. Thus $x\theta_k = x \in C\theta_{j+1}$.

Now $((C\theta_{j+1})\theta_{j+2})\ldots\theta_k \cap \mathrm{VAR} \subseteq (C\theta_{j+1})\theta_{j+3}\ldots\theta_k) \cap \mathrm{VAR} \subseteq \cdots \subseteq (C\theta_{j+1})\theta_k \cap \mathrm{VAR} \subseteq C\theta_{j+1}$. □

In order to formulate the final property of normal derivations we prove in this section, we need the following definition.

Definition 3.10. (This definition is equivalent to the definition of local selection functions in [17].) A selection rule $\boldsymbol{R}$ is *local* if every SLD-derivation $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots)$ via $\boldsymbol{R}$ 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.

It is easy to see that the leftmost selection rule and the rightmost selection rule are examples of local selection rules.

Corollary 3.11. *Let* $D = G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{i-1} \Rightarrow_{C_i,\theta_i} G_i \Rightarrow \cdots$ *be a normal SLD-derivation of a function-free program P and* G_0 *and let* $0 \leq j < k$ $(<|D|)$. *Let A be the selected atom in* G_j. *Suppose a local selection rule is used between* G_j *and* G_k *and A is not completely resolved before* G_k. *Then* $\mathrm{var}(A\theta_k) \subseteq \mathrm{var}(A)$ *and* $\mathrm{var}(A\theta_{j+1}\ldots\theta_k) \subseteq \mathrm{var}(A)$.

Proof. Let $x \in \mathrm{var}(A)$ and assume that $x\theta_k$ is a variable. We prove that $x\theta_k \in \mathrm{var}(A)$. Let $G_j = (A, R)$ and consider the derivation $\leftarrow A = H_j \Rightarrow_{C_{j+1},\theta_{j+1}} H_{j+1} \Rightarrow \cdots \Rightarrow_{C_k,\theta_k} H_k$ (hence for $j \leq i \leq k$, $G_i = (H_i, R\theta_{j+1}\ldots\theta_i)$). Note that this derivation exists, since a local selection rule is used and A is not completely resolved before G_k, and note that the derivation is normal. Now $x \in \mathrm{var}(A) = \mathrm{var}(H_j)$ implies $x\theta_k \in \mathrm{var}(H_j) = \mathrm{var}(A)$ by Corollary 3.7. Hence, $\mathrm{var}((A\theta_{j+1})\theta_{j+2}\ldots\theta_k) \subseteq \mathrm{var}(A\theta_{j+2}\ldots\theta_k) \subseteq \cdots \subseteq \mathrm{var}(A\theta_k) \subseteq \mathrm{var}(A)$. □

4. Generalizing completeness results

The rest of this paper discusses the completeness of loop checks. *Therefore we assume from now on the absence of function symbols.* In this section we shall prepare, formulate and prove the Generalization Theorem, the main theorem of this paper.

This theorem states that, given a loop check, and given a class of programs for which this loop check is complete, the loop check is (under certain conditions) also complete w.r.t. the leftmost selection rule for a larger class of programs.

4.1. Preparation

The formulation of the Generalization Theorem requires the formalization of the classes of programs for which it is applicable. Roughly, these classes of programs are characterized by the condition that all clauses in the program satisfy some (preferably decidable) property. We do not go into details about these properties; we assume that the notion "a clause C satisfies a property Pr" is given.

Definition 4.1. Let Pr be a property of clauses. A program P *satisfies* Pr (P *is a* Pr*-program*) if every clause in P satisfies Pr.

Definition 4.2. A property of clauses Pr is *closed under instantiation* if for every clause C that satisfies Pr and for every substitution σ, $C\sigma$ satisfies Pr.

Note that $C\sigma$ is not necessarily a ground instance of C. The Generalization Theorem is only valid for properties that are closed under instantiation. However, in the next section, where we shall give some examples of the use of the Generalization Theorem, we shall also consider a property that is *not* closed under instantiation. A detailed inspection of the proof of the Generalization Theorem enables us to derive useful results for this property as well.

The Generalization Theorem is only valid for loop checks satisfying certain conditions. These conditions are formalized here. The first condition is that the loop check is "safe for goal extension". Informally, this means that when we have a derivation that is pruned by the loop check, adding some atoms to the initial goal that are never selected (before the derivation is pruned), yields again a pruned derivation.

Definition 4.3. A loop check L is *safe for goal extension* if for every SLD-derivation D of $P \cup \{\leftarrow G_0\}$ that is pruned by L, an SLD-derivation of $P \cup \{\leftarrow (G_0, H_0)\}$ which selects the same atoms, and uses the same input clauses and mgu's as D is also pruned by L.

The second condition is that the loop check is "safe for initialization". Informally, this means that when we have a derivation that is pruned by the loop check, adding some derivation steps in front of it ("initialization steps"), yields again a pruned derivation.

Definition 4.4. A loop check L is *safe for initialization* if for every SLD-derivation $D = (G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1} \Rightarrow_{C_{i+2},\theta_{i+2}} G_{i+2} \Rightarrow \cdots)$ that is pruned by L ($i > 0$), every derivation $(G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1} \Rightarrow_{C_{i+2},\theta_{i+2}} G_{i+2} \Rightarrow \cdots)$ in which in G_i, $G_{i+1}, \ldots$ the same atoms are selected as in D, is pruned by L.

The third condition is that the loop check is "safe for detailing". Informally, this means that when we have a derivation that is pruned by the loop check, replacing every derivation step by one or more steps giving the same computed answer ("showing the details of one step in several steps"), yields again a pruned derivation.

Definition 4.5. A loop check L is *safe for detailing* if for every SLD-derivation $D=(G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots)$ that is pruned by L, every derivation of the form

$$(G_0 \Rightarrow_{C_1^1,\tau_1^1} H_1^1 \Rightarrow \cdots \Rightarrow H_{n_1-1}^1 \Rightarrow_{C_{n_1}^1,\tau_{n_1}^1} G_1$$
$$\Rightarrow_{C_1^2,\tau_1^2} H_1^2 \Rightarrow \cdots \Rightarrow H_{n_2-1}^2 \Rightarrow_{C_{n_2}^2,\tau_{n_2}^2} G_2 \Rightarrow \cdots)$$

with for every $i>0$:

$$\tau_1^i \ldots \tau_{n_i}^i|_{\mathrm{var}(G_0,G_1,\ldots,G_{i-1})} = \theta_i|_{\mathrm{var}(G_0,G_1,\ldots,G_{i-1})}$$

and in which in $G_0, G_1, \ldots$ the same atoms are selected as in D, is pruned by L.

Finally, for a certain property Pr, we describe the larger class of programs for which the loop check is complete according to the Generalization Theorem (so-called nr-extended Pr-programs), given that the loop check is complete for Pr-programs. In Section 5 it will appear that the resemblance between the following definition and Definition 2.13 is not a coincidence.

Definition 4.6. Let P be a program. A clause $C=(H \leftarrow NR, R)$ is *nr-extended Pr w.r.t. P* if the clause $H \leftarrow R$ satisfies Pr and for every atom A in NR, $\mathrm{rel}(A)$ does not depend on $\mathrm{rel}(H)$ in P. NR is called the *nonrecursive part* of C and R is called the *Pr-part.*

A program P is *nr-extended Pr* if every clause in P is nr-extended Pr w.r.t. P.

4.2. The Generalization Theorem

We can now formulate the Generalization Theorem.

Theorem 4.7 (Generalization Theorem). *Let Pr be a property of clauses that is closed under instantiation. Let L be a loop check such that*
- *L is complete for Pr-programs,*
- *L is safe for goal extension,*
- *L is safe for initialization,*
- *L is safe for detailing.*

Then L is complete w.r.t. the leftmost selection rule for nr-extended Pr programs.

In the rest of this section, we shall assume that Pr is a property and L is a loop check satisfying the above conditions. In order to prove this theorem, we use the following lemma.

Lemma 4.8. *Let P be a nr-extended Pr-program and G_0 a goal in L_P. Let D be an infinite SLD-derivation of $P \cup \{G_0\}$ via the leftmost selection rule. Suppose that*

> *for no goal $G_i = (G, H)$ in D $(i \geq 0)$, the derivation of $P \cup \{G\}$ (using the same input clauses, mgu's and selection rule as D) is pruned by L.* (*)

Then D is pruned by L.

Before proving this lemma, we show that the Generalization Theorem is an immediate consequence of it.

Proof of the Generalization Theorem. Let P be an nr-extended *Pr*-program, G_0 a goal in L_P and D an infinite SLD-derivation of $P \cup \{G_0\}$. Two cases arise.

(i) For no goal (G, H) in D, the derivation of G (using the same input clauses, mgu's and selection rule as D) is pruned by L. Then by Lemma 4.8, D is pruned by L.

(ii) Otherwise, there is a goal (G, H) in D for which the derivation of G (using the same input clauses, mgu's and selection rule as D) is pruned by L. Then the tail of D starting at this goal (G, H) is pruned, since L is safe for goal extension. So D is pruned by L too, since L is also safe for initialization. □

Proof of Lemma 4.8. The dependency graph D_P defines a (well founded) partial ordering $\leq$ of the set $\{\mathrm{cl}_P(p) \mid p$ is a predicate symbol in $L_P\}$. Therefore we may assume as induction hypothesis (by a complete induction on $\leq$), that this lemma has been proved for every derivation of $P \cup \{G\}$ where G contains only strict $\leq$-smaller predicate symbols than the $\leq$-largest predicate symbol in G_0.

Claim 1. *D is of the form*

$$(G_0 \Rightarrow_{C_1^1, \tau_1^1} H_1^1 \Rightarrow \cdots \Rightarrow H_{n_1-1}^1 \Rightarrow_{C_{n_1}^1, \tau_{n_1}^1} G_1$$
$$\Rightarrow_{C_1^2, \tau_1^2} H_1^2 \Rightarrow \cdots \Rightarrow H_{n_2-1}^2 \Rightarrow_{C_{n_2}^2, \tau_{n_2}^2} G_2 \Rightarrow \cdots)$$

for some derivation $D' = (G_0 \Rightarrow_{C_1, \theta_1} G_1 \Rightarrow_{C_2, \theta_2} G_2 \Rightarrow \cdots)$, with for every $i > 0$:

$$\tau_1^i \ldots \tau_{n_i}^i|_{\mathrm{var}(G_0, G_1, \ldots, G_{i-1})} = \theta_i|_{\mathrm{var}(G_0, G_1, \ldots, G_{i-1})},$$

and where $C_1, C_2, \ldots$ all satisfy Pr. Moreover, in the goals $G_0, G_1, \ldots$ the same atoms are selected in D and D'.

The lemma follows from Claim 1: D' is a derivation of $\{G_0, C_1, C_2, \ldots\}$, $\{C_1, C_2, \ldots\}$ is a *Pr*-program, and L is complete for *Pr*-programs, therefore D' is pruned by L. Hence D is pruned by L, since L is safe for detailing.

Proof of Claim 1. We prove the claim by induction. Suppose we have constructed D' and proved the claim up to the goal G_i. (Up to G_0, the claim is trivial.)

Let $G_i = \leftarrow A_1, \ldots, A_n$, let $C = C_1^{i+1} = (A \leftarrow NR, R)$ and let $\tau = \tau_1^{i+1}$. Suppose that NR is the nonrecursive part of the body of C and that R is the Pr-part. The next step in D is $G_i \Rightarrow_{C,\tau} \leftarrow (NR, R, A_2, \ldots, A_n)\tau$. Let D_1 be the SLD-derivation of $P \cup \{\leftarrow NR\tau\}$ that uses the same input clauses, mgu's and selection rule as the tail of D starting at $\leftarrow (NR, R, A_2, \ldots, A_n)\tau$. Four cases arise.

(i) NR is *empty*. This is a special case of case (iv): $P \cup \{\leftarrow NR\tau\}$ is immediately successfully refuted. (If G_0 is $\leqslant$-minimal, then this is the only possible case, since then $\mathrm{rel}(A_1) = \mathrm{rel}(A)$ is $\leqslant$-minimal and by definition every predicate symbol in NR is strict $\leqslant$-smaller than $\mathrm{rel}(A)$.)

(ii) D_1 is *failed*. Then D is failed too, which contradicts the assumption that D is infinite.

(iii) D_1 is *infinite*. By definition, every predicate symbol in NR is strictly $\leqslant$-smaller than $\mathrm{rel}(A_1)$, which is $\leqslant$-smaller than the $\leqslant$-largest predicate symbol in G_i (hence in G_0), so we may assume that Lemma 4.8 holds for $D = D_1$. Now it follows that the Generalization Theorem can be applied on $D = D_1$. Hence D_1 should be pruned by L. However, this contradicts the assumption (*), for $G = NR\tau$ and $H = (R, A_2, \ldots, A_n)\tau$.

(iv) D_1 is *successful*, yielding a computed answer substitution σ (if NR is empty then $\sigma = \varepsilon$).

Case (iv) is the only remaining case. In this case we have in D the goal $G_{i+1} = \leftarrow (R, A_2, \ldots, A_n)\tau\sigma$, immediately after NR is completely resolved.

Claim 2. *The sequence of resolution steps between G_i and G_{i+1} in D can be mimiced by one resolution step $G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1}$ in D', where C_{i+1} is an instance of $A \leftarrow R$ and $\tau\sigma|_{\mathrm{var}(G_0,G_1,\ldots,G_i)} = \theta_{i+1}|_{\mathrm{var}(G_0,G_1,\ldots,G_i)}$.*

Claim 1 follows from Claim 2: since Pr is closed under instantiation, C_{i+1} satisfies Pr. So we have constructed D' and proved Claim 1 up to the goal G_{i+1}.

Now the construction of the resolution step $G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1}$ remains.

Proof of Claim 2. First, we define C_{i+1} and θ_{i+1}, then we prove that $G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1}$ is indeed a derivation step. Finally, we check the other requirements on C_{i+1} and θ_{i+1}. By Lemma 3.5 we may assume that D is normal.

For every chain C in NR, we fix a substitution ρ_C such that for every $x \in C\tau$, $x\rho_C \in C$ and $x\rho_C\tau = x$. Moreover, if $x \in (\mathrm{var}(R) \cap C)\tau$, then $x\rho_C \in \mathrm{var}(R)$. For every chain, such a substitution exists: if $x \in C\tau$, then $\{y \in C \mid y\tau = x\} \neq \emptyset$. If $\{y \in \mathrm{var}(R) \cap C \mid y\tau = x\} \neq \emptyset$, then $x\rho_C$ must be chosen from the latter set, otherwise any element of the former set will do.

Now we can define ψ by:

$$x\psi = \begin{cases} x & \text{if } x \notin \mathrm{var}(NR), \\ x\tau\sigma\rho_{C(x)} & \text{if } x \in C \subseteq \mathrm{var}(NR). \end{cases}$$

Notice that $x\tau\sigma \in C(x)\tau\sigma \subseteq C(x)\tau$ by Corollary 3.9, since D is normal. Finally, we define $C_{i+1} = (A \leftarrow R)\psi$ and $\theta_{i+1} = \tau\sigma|_{\mathrm{var}(A_1, A\psi)}$. Now we must prove that $G_i \Rightarrow_{C_{i+1}, \theta_{i+1}} G_{i+1}$ is indeed a resolution step. That is:

Claim 3. *$(A \leftarrow R)\psi$ is properly standardized apart.*

Claim 4. *θ_{i+1} is an idempotent mgu of $A\psi$ and A_1.*

Claim 5. *$(R\psi, A_2, \ldots, A_n)\theta_{i+1} = (R, A_2, \ldots, A_n)\tau\sigma$.*

In the proofs of these claims, we take $C(x) = C_{NR}(x)$.

Proof of Claim 3. We prove that $\mathrm{var}((A \leftarrow R)\psi) \subseteq \mathrm{var}(A \leftarrow NR, R)$. Let $x \in \mathrm{var}(A \leftarrow R)$. Then: if $x\psi = x$, then $x\psi \in \mathrm{var}(A \leftarrow R)$; if $x\psi \neq x$, then $x \in C(x) \subseteq \mathrm{var}(NR)$, so $x\psi = x\tau\sigma\rho_{C(x)} \in C(x) \subseteq \mathrm{var}(NR)$. □

Before proving Claim 4, we prove an additional claim.

Claim 6. *ψ is idempotent.*

Proof. Let x be a variable. If $x\psi = x$, then $x\psi\psi = x\psi$. Otherwise, $x\psi\psi = x\tau\sigma\rho_{C(x)}\psi =$ (since $x\tau\sigma\rho_{C(x)} \in C(x) \subseteq \mathrm{var}(NR)$) $= x\tau\sigma\rho_{C(x)}\tau\sigma\rho_{C(x)} = x\tau\sigma\sigma\rho_{C(x)} =$ (as σ is idempotent) $= x\tau\sigma\rho_{C(x)} = x\psi$. □

Proof of Claim 4. We prove that for every unifier η of A_1 and $A\psi$: $\eta = \theta_{i+1}\eta$. Let η be a unifier of A_1 and $A\psi$: $A_1\eta = A\psi\eta$.

By standardizing apart, $\mathrm{var}(A_1) \cap \mathrm{var}(NR) = \emptyset$, so we have $A_1 = A_1\psi$. Therefore, $\psi\eta$ is a unifier of A_1 and A. Since τ is an idempotent mgu of A_1 and A, we have $\psi\eta = \tau\omega = \tau\tau\omega = \tau\psi\eta$ ($\tau \leqslant \psi\eta$, so for some ω: $\tau\omega = \psi\eta$).

Let x be a variable. If $x \notin \mathrm{var}(A_1, A\psi)$, then $x = x\theta_{i+1}$, so $x\eta = x\theta_{i+1}\eta$. If $x \in \mathrm{var}(A_1)$, then at the corresponding position in A, we find a term (constant or variable) t such that $x\eta = t\psi\eta$ and $x\tau = t\tau$. Two cases arise.

- $x\tau = x\tau\sigma$. Then $x\eta = t\psi\eta = t\tau\psi\eta = x\tau\psi\eta \overset{*}{=} x\tau\eta = x\tau\sigma\eta = x\theta_{i+1}\eta$. $\overset{*}{=}$: $x\tau \notin \mathrm{var}(NR)$, since either $x\tau$ is ground, or $x\tau \in \mathrm{var}(A_1\tau) \subseteq \mathrm{var}(A_1)$ (the latter inclusion by Corollary 3.11, since D is normal).
- $x\tau \neq x\tau\sigma$. Then $x\tau \in \mathrm{var}(NR\tau)$, so for some $v \in \mathrm{var}(NR)$: $v\tau = x\tau$ and $v\psi = v\tau\sigma\rho_{C(v)}$. Now $x\eta = t\psi\eta = t\tau\psi\eta = x\tau\psi\eta = v\tau\psi\eta = v\psi\eta =$ (by Claim 6) $= v\psi\psi\eta = v\psi\tau\psi\eta = v\tau\sigma\rho_{C(v)}\tau\psi\eta = v\tau\sigma\psi\eta = x\tau\sigma\psi\eta \overset{*}{=} x\tau\sigma\eta = x\theta_{i+1}\eta$. $\overset{*}{=}$: $x\tau\sigma \notin \mathrm{var}(NR)$, since either $x\tau\sigma$ is ground, or $x\tau\sigma \in \mathrm{var}(A_1\tau\sigma) \subseteq \mathrm{var}(A_1)$. (the latter inclusion by Corollary 3.11, since D is normal).

If $x \in \mathrm{var}(A\psi)$, then for some $y \in \mathrm{var}(A)$ we have $y\psi = x$. At the corresponding position in A_1, we find a term t such that $x\eta = t\eta$ and $y\tau = t\tau$. Again two cases arise.

- $y \notin \mathrm{var}(NR)$. Then $y\psi = y$ and $y\tau\sigma = y\tau$. Therefore we have $x\eta = y\psi\eta = y\tau\psi\eta \stackrel{*}{=} y\tau\eta = y\tau\sigma\eta = y\psi\tau\sigma\eta = x\tau\sigma\eta = x\theta_{i+1}\eta$. $\stackrel{*}{=}$: $y\tau \notin \mathrm{var}(NR)$, since either $y\tau$ is ground, or $y\tau \in \mathrm{var}(A\tau) = \mathrm{var}(A_1\tau) \subseteq \mathrm{var}(A_1)$.
- $y \in \mathrm{var}(NR)$. Then $y\psi = y\tau\sigma\rho_{C(y)}$, so (see Claim 6), $y\psi = y\psi\psi = y\psi\tau\sigma\rho_{C(y\psi)} = x\tau\sigma\rho_{C(x)}$. Therefore we have $x\eta = y\psi\eta =$ (by Claim 6) $y\psi\psi\eta = y\psi\tau\psi\eta = x\tau\sigma\rho_{C(x)}\tau\psi\eta = x\tau\sigma\psi\eta \stackrel{*}{=} x\tau\sigma\eta = x\theta_{i+1}\eta$. $\stackrel{*}{=}$: again, $x\tau\sigma \notin \mathrm{var}(NR)$. □

Proof of Claim 5. If $x \in \mathrm{var}(A_i)$ $(2 \leq i \leq n)$, then
- if $x \notin \mathrm{var}(A_1)$ then $x\theta_{i+1} = x = x\tau\sigma$;
- if $x \in \mathrm{var}(A_1)$ then by definition $x\theta_{i+1} = x\tau\sigma$.

If $x \in \mathrm{var}(R)$, then two cases arise.
- $x\psi \in \mathrm{var}(A\psi)$. Then

$$x\psi\theta_{i+1} = x\psi\tau\sigma = \begin{cases} x\tau\sigma & \text{if } x \notin \mathrm{var}(NR), \\ x\tau\sigma\rho_{C(x)}\tau\sigma = x\tau\sigma\sigma = x\tau\sigma & \text{if } x \in \mathrm{var}(NR). \end{cases}$$

- $x\psi \notin \mathrm{var}(A\psi)$. Then either $x\psi$ is ground or for no $y \in \mathrm{var}(A)$: $y\psi = x\psi$. If $x\psi$ is ground, then $x\tau\sigma$ is ground, so $x\psi = x\tau\sigma\rho_{C(x)} = x\tau\sigma$. If for no $y \in \mathrm{var}(A)$: $y\psi = x\psi$, then in particular, $x \notin \mathrm{var}(A)$, so $x\tau = x$. Then if $x \notin \mathrm{var}(NR)$, then $x\psi = x = x\tau = x\tau\sigma$; if $x \in \mathrm{var}(NR)$, then $x\psi = x\tau\sigma\rho_{C(x)}$. Also, $x\tau\sigma \in C(x)\tau\sigma \subseteq C(x)\tau$ (by Corollary 3.9, since D is normal), so for some $z \in C(x)$: $z\tau = x\tau\sigma$ (and $C(z) = C(x)$). Then $z\tau\sigma = x\tau\sigma\sigma = x\tau\sigma$, so $z\psi = x\psi$. Hence $z \notin \mathrm{var}(A)$, so $z\tau = z = z\rho_{C(z)}\tau$, so $z = z\rho_{C(z)} = z\rho_{C(x)}$. Therefore $x\tau\sigma\rho_{C(x)} = z\tau\rho_{C(x)} = z\rho_{C(x)} = z = z\tau = x\tau\sigma$. □

Obviously, C_{i+1} is an instrance of $A \leftarrow R$. Also,

$$\theta_{i+1}|_{\mathrm{var}(G_0,G_1,\ldots,G_i)} = \tau\sigma|_{\mathrm{var}(A_1,A_\psi)\cap\mathrm{var}(G_0,G_1,\ldots,G_i)} = \tau\sigma|_{\mathrm{var}(A_1)} = \tau\sigma|_{\mathrm{var}(G_0,G_1,\ldots,G_i)}),$$

by Corollary 3.11, since D is normal and a local selection rule is used. This concludes the proof of Claim 2 and thereby the proof of Lemma 4.8. □

5. Applications of the Generalization Theorem

A simple example of the application of the Generalization Theorem is the following.

Corollary 5.1. *If P is a function-free hierarchical program, then every SLD-derivation of $P \cup \{G\}$ via the leftmost selection rule is finite.*

Proof. We prove an equivalent proposition, namely that the empty loop check is complete w.r.t. the leftmost selection rule for function-free hierarchical programs. This follows from the Generalization Theorem and the following observations.

- The empty loop check is complete for "unit-programs", programs that consist solely of unit clauses.
- The "unit" property is closed under instantiation.
- The empty loop check is safe for goal extension, initialization and detailing.
- Nr-extended unit-programs are known as hierarchical programs. □

Of course, this result is well known, even for arbitrary selection rules and programs with function symbols. More interesting results can be obtained by using the Generalization Theorem to extend the completeness results presented in Section 2. The first result presented there is the completeness of equality checks for function-free restricted programs w.r.t. the leftmost selection rule. The Generalization Theorem cannot be applied on this proposition. In contrast, the Generalization Theorem provides an alternative proof for this proposition, based on the lemma "the equality checks are complete for function-free programs in which the body of each clause contains at most one atom".

The other results of Section 2 are only valid for the subsumption and context checks. Therefore we shall now prove that the weakest of those checks, the SVR_L check and the CVR check, satisfy the conditions of the Generalization Theorem, i.e. that they are safe for goal extension, initialization and detailing.

Lemma 5.2. *The* SVR_L *check and the* CVR *check are safe for goal extension.*

Proof. Let D be an SLD-derivation of $P \cup \{\leftarrow G_0\}$. Let D' be an SLD-derivation of $P \cup \{\leftarrow (G_0, H_0)\}$, in which the same atoms are selected and the same input cluases and mgu's are used as in D. Thus D cannot contain any variable occurring in H_0 but not in G_0. Denote by θ_n the mgu used in the nth resolution step of D and D' $(n \geqslant 1)$.

If D is pruned by the SVR_L check resp. the CVR check, then we have for some renaming τ two goals G_i and G_k in D with $G_0\theta_1 \ldots \theta_k = G_0\theta_1 \ldots \theta_i\tau$ and $G_k \supseteq_L G_i\tau$ resp. (A in G_i "produces" $A\tau$ in G_k and τ and $\theta_{i+1} \ldots \theta_k$ agree on $\mathrm{var}(G_i) \cap \mathrm{var}(A)$). Assuming that τ acts only on the variables in D, we have that $\theta_1 \ldots \theta_k$ and $\theta_1 \ldots \theta_i\tau$ coincide on *all* variables of H_0. So $(G_0, H_0)\theta_1 \ldots \theta_k = (G_0, H_0)\theta_1 \ldots \theta_i\tau$ and $(G_k, H_0\theta_1 \ldots \theta_k) \supseteq_L (G_i, H_0\theta_1 \ldots \theta_i)\tau$, resp. τ and $\theta_{i+1} \ldots \theta_k$ agree also $\mathrm{var}(H_0\theta_1 \ldots \theta_i) \cap \mathrm{var}(A)$. This means that D' is pruned by SVR_L, respectively CVR, as well. □

Notice that it is essential to consider loop checks based on resultants here. It is easy to see that the loop checks based on goals are *not* safe for goal extension.

Lemma 5.3. *The* SVR_L *check and the* CVR *check are safe for initialization.*

Proof. Let $D' = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow_{C_2,\theta_2} G_2 \Rightarrow \cdots)$ be an SLD-derivation. Suppose that for some $i > 0$ the derivation $D = (G_i \Rightarrow_{C_{i+1},\theta_{i+1}} G_{i+1} \Rightarrow_{C_{i+2},\theta_{i+2}} G_{i+2} \Rightarrow \cdots)$ is pruned

by SVR_L resp. CVR. Clearly for some j, $k>j$ and renaming τ (acting only on variables in D): τ "proves" that G_j and G_k are "sufficiently similar" for SVR_L, resp. CVR, and $G_i\theta_{i+1}\ldots\theta_k = G_i\theta_{i+1}\ldots\theta_j\tau$. So it remains to prove that $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_j\tau$.

Let $x \in \mathrm{var}(G_0\theta_1\ldots\theta_i)$. Two cases arise.

(i) $x \notin \mathrm{var}(G_i)$. Then x does not occur in D, hence $x\theta_{i+1}\ldots\theta_k = x\theta_{i+1}\ldots\theta_j\tau = x$.

(ii) $x \in \mathrm{var}(G_i)$. Then $G_i\theta_{i+1}\ldots\theta_k = G_i\theta_{i+1}\ldots\theta_j\tau$ yields $x\theta_{i+1}\ldots\theta_k = x\theta_{i+1}\ldots\theta_j\tau$.

Hence D' is pruned by SVR_L, respectively CVR, as well. □

Lemma 5.4. *The* SVR_L *check and the* CVR *check are safe for detailing.*

Proof. Let $D = (G_0 \Rightarrow_{C_1,\theta_1} G_1 \Rightarrow \cdots)$ be an SLD-derivation that is pruned by SVR_L resp. CVR and let D' be an SLD-derivation of the form

$$(G_0 \Rightarrow_{C_1^1,\tau_1^1} H_1^1 \Rightarrow \cdots \Rightarrow H_{n_1-1}^1 \Rightarrow_{C_{n_1}^1,\tau_{n_1}^1} G_1$$
$$\Rightarrow_{C_1^2,\tau_1^2} H_1^2 \Rightarrow \cdots \Rightarrow H_{n_2-1}^2 \Rightarrow_{C_{n_2}^2,\tau_{n_2}^2} G_2 \Rightarrow \cdots)$$

with for every $i>0$:

$$\tau_1^i\ldots\tau_{n_i}^i|_{\mathrm{var}(G_0,G_1,\ldots,G_{i-1})} = \theta_i|_{\mathrm{var}(G_0,G_1,\ldots,G_{i-1})}$$

in which in $G_0, G_1, \ldots$ the same atoms are selected as in D. Since D is pruned by SVR_L resp. CVR, we have for some j, $k>j$ and renaming τ: τ "proves" that G_j and G_k are "sufficiently similar" for SVR_L, resp. CVR and $G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_j\tau$. For CVR this proof includes that "for every variable x that occurs both inside and outside of A in G_i, $x\theta_{i+1}\ldots\theta_k = x\tau$". It follows immediately that

$$G_0\tau_1^1\ldots\tau_{n_1}^1\tau_1^2\ldots\tau_{n_2}^2\ldots\tau_1^k\ldots\tau_{n_k}^k = G_0\tau_1^1\ldots\tau_{n_1}^1\tau_1^2\ldots\tau_{n_2}^2\ldots\tau_1^j\ldots\tau_{n_j}^j\tau,$$

and for CVR that "for every variable x that occurs both inside and outside of A in G_i, $x\tau_1^{i+1}\ldots\tau_{n_2}^{i+1}\ldots\tau_1^k\ldots\tau_{n_k}^k = x\tau$".

Hence D' is pruned by SVR_L, respectively CVR, as well. □

Now we can use the Generalization Theorem together with the fact that the subsumption and context checks are complete for function-free nvi programs.

Corollary 5.5. *The subsumption and context checks are complete w.r.t. the leftmost selection rule for function-free nr-extended nvi programs.*

Proof. The nvi property is obviously closed under instantiation. Therefore by Subsumption Completeness Theorem 2.21 respectively Context Completeness Theorem 2.26, the Generalization Theorem, and Lemmas 5.2, 5.3 and 5.4, the SVR_L check and the CVR check are complete w.r.t. the leftmost selection rule for function-free nr-extended nvi programs. Since the SVR_L check is the weakest of the subsumption

checks and the CVR check is the weakest of the context checks, by the Relative Strength Theorem 2.7, the same holds for the other subsumption and context checks. □

Finally, in Section 2 it was mentioned that the subsumption and context checks are also complete for function-free svo programs. However, the property "svo" is *not* closed under instantiation, so we cannot immediately use the Generalization Theorem. In fact, this should not come as a surprise, since *every* program can be converted into a "computationally equivalent" nr-extended svo program. This can be done by replacing the $k > 1$ occurrences of a variable x in the body of a clause by $x_1, \ldots, x_k$ and adding the nonrecursive atoms $\mathrm{eq}(x, x_1), \ldots, \mathrm{eq}(x, x_k)$ in the body of the clause. Finally the clause $\mathrm{eq}(x, x)$ is added to the program (assuming that eq is a new predicate symbol in P).

In the proof of Lemma 4.8, we need that the clause $C_{i+1} = (A \leftarrow R)\psi$ satisfies the property of clauses considered, given that the clause $A \leftarrow R$ satisfies the property. Up till now, this was derived immediately from the assumption that the property should be closed under instantiation. Since for the svo property this is not true, we shall derive conditions that ensure directly that C_{i+1} satisfies the svo property, i.e. that every variable in $R\psi$ occurs only once (provided that every variable in R occurs only once).

Formally, let $x, y \in \mathrm{var}(R)$ such that $x \neq y$ and $x\psi$, $y\psi \in \mathrm{VAR}$. We shall derive conditions on the program ensuring that $x\psi \neq y\psi$.

If $x \notin \mathrm{var}(NR)$, then $x\psi = x$. Then,

- if $y \notin \mathrm{var}(NR)$, $y\psi = y \neq x$, and
- if $y \in \mathrm{var}(NR)$, $y\psi = y\tau\sigma\rho_{C(y)} \in C(y) \subseteq \mathrm{var}(NR)$, so $y\psi \neq x$.

The same argument holds if $y \notin (NR)$. So a problem can only arise in the case that $x, y \in \mathrm{var}(NR)$. Then we have $x\psi = x\tau\sigma\rho_{C(x)} \subseteq C(x)$ and $y\psi = y\tau\sigma\rho_{C(y)} \subseteq C(y)$.

One solution is demanding that for every pair of distinct variables $x, y \in \mathrm{var}(R) \cap \mathrm{var}(NR)$, $C(x) \neq C(y)$. Then $C(x) \cap C(y) = \emptyset$, so $x\psi \neq y\psi$. This disallows the addition of the eq-atoms in the construction above.

Another solution is to avoid that different variables in a (sub)goal are unified while the (sub)goal is refuted. (That is: to ensure that for every x in a goal, and for every unifier σ in the derivation, either $x\sigma = x$ or $x\sigma$ is a constant.) This condition can be met (for normal derivations) by the demand that variables do not occur more than once in the *head* of a clause. This disallows the addition of the clause $\mathrm{eq}(x, x) \leftarrow$.

In this case such a condition yields $x\psi = x\tau\rho_{C(x)}$ ($x\tau\sigma$ cannot be a constant, since $x\psi$ is a variable). Then $x\tau = x\tau\rho_{C(x)}\tau = x\psi\tau$. Using the condition again (but now w.r.t. τ), we obtain $x = x\psi$ (still, $x\tau$ cannot be a constant). Similarly we obtain $y = y\psi$, so $x\psi \neq y\psi$.

These two solutions give rise to two classes of programs for which the subsumption checks are complete w.r.t. the leftmost selection rule (in the absence of function symbols).

Definition 5.6. Let P be a program. A clause $C = (A \leftarrow NR, R)$ is *chain-restricted svo w.r.t. P* if C is nr-extended svo w.r.t. P, where NR is the nonrecursive part and R is the svo-part of C, and for every pair of distinct variables $x, y \in var(R)$, $C_{NR}(x) \neq C_{NR}(y)$. A program P is *chain-restricted svo* if every clause in P is chain-restricted svo
w.r.t. P.

Definition 5.7. Let P be a program. A clause C is *head-restricted svo w.r.t. P* if C is nr-extended svo w.r.t. P and in the head of C, no variable occurs more than once. A program P is *head-restricted svo* if every clause in P is head-restricted svo w.r.t. P.

Corollary 5.8. *The subsumption and context checks are complete w.r.t. the leftmost selection rule for function-free chain-restricted svo programs.*

Proof. By Subsumption Completeness Theorem 2.22 respectively Context Completeness Theorem 2.26, the Generalization Theorem, Lemmas 5.2, 5.3 and 5.4 and the considerations above, the SVR_L check and the CVR check are complete w.r.t. the leftmost selection rule for function-free chain-restricted svo programs. Since the SVR_L check is the weakest of the subsumption checks and the CVR check is the weakest of the context checks, by the Relative Strength Theorem 2.7, the same holds for the other subsumption and context checks. □

Corollary 5.9. *The subsumption and context checks are complete w.r.t. the leftmost selection rule for function-free head-restricted svo programs.*

Proof. By Subsumption Completeness Theorem 2.22 respectively Context Completeness Theorem 2.26, the Generalization Theorem, Lemma 5.2, 5.3 and 5.4 and the considerations above, the SVR_L check and the CVR check are complete w.r.t. the leftmost selection rule for function-free head-restricted svo programs. Since the SVR_L check is the weakest of the subsumption checks and the CVR check is the weakest of the context checks, by the Relative Strength Theorem 2.7, the same holds for the other subsumption and context checks. □

Finally, we give an example of a function-free head-restricted svo program that does not fall into any other class of programs discussed so far.

Example 5.10. Given a logic program P, it can be interesting that some predicates are defined without the use of recursion. The program *NONREC* characterizes these predicates. First we need an adequate representation of D_P (see Definition 2.12) in *NONREC* (the predicates of P are constants in *NONREC*). We cannot use a representation of the form $\{\mathrm{dep}(p, q) \leftarrow . \mid (p, q) \in D_P\}$, because it fails to express (without the use of negation) that $(p, q) \notin D_P$ for some p and q.

Instead we assume that $\{p_1, \ldots, p_n\}$ is the set of predicates that occur in P and that for every i $(1 \leq i \leq n)$, there is only one ground clause $\mathrm{dep}(p_i, q_1, \ldots, q_n) \leftarrow$ in *NONREC* such that for some m $(0 \leq m \leq n)$: $\{(p_i, q_1), \ldots, (p_i, q_m)\} \subseteq D_P$ and $q_{m+1} = \cdots = q_n = \mathrm{nil}$ (nil is a constant in *NONREC* that is different from $p_1, \ldots, p_n$).

Now we add to *NONREC* the following two clauses:

$$nonrec(\mathrm{nil}) \leftarrow .$$

$$nonrec(x) \leftarrow dep(x, x_1, \ldots, x_n), nonrec(x_1), \ldots, nonrec(x_n).$$

Without loop checking, this program goes into an infinite loop if and only if the predicate p is defined in P by means of recursion. As the program is head-restricted svo and function-free, the subsumption and context checks prune all its infinite loops, thus making the program work properly.

Of course, it is easier to write a restricted program (using the representation $\{dep(p, q) \leftarrow . \mid (p, q) \in D_P\}$ and the transitive closure of *dep*) that succeeds on predicates defined using recursion and fails otherwise. But using this program to define the predicates that do *not* use recursion would require the use of negation again.

The combination of loop checking with negation, as suggested above, is a delicate matter, which is studied in [4].

References

[1] K.R. Apt, Logic programming, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (Elsevier, Amsterdam, 1990) 493-574.

[2] K.R. Apt, R.N. Bol and J.W. Klop, On the safe termination of PROLOG programs, in: G. Levi and M. Martelli, eds., *Proc. 6th Internat. Conf. on Logic Programming* (MIT Press, Cambridge, MA, 1989) 353-368.

[3] Ph. Besnard, On infinite loops in logic programming, Internal Report 488, IRISA, Rennes, 1989.

[4] R.N. Bol, Loop checking and negation, Technical Report CS-R9075, Centre for Mathematics and Computer Science, Amsterdam, 1990. Extended abstract in: J. van Eijck, ed., *Logics in AI*, Lecture Notes in Computer Science, Vol. 478 (Springer, Berlin, 1991) 121-138, to appear in *J. Logic Programming*.

[5] R.N. Bol, K.R. Apt and J.W. Klop, An analysis of loop checking mechanisms for logic programs, *Theoret. Comput. Sci.* **86** (1991) 35-79.

[6] D.R. Brough and A. Walker, Some practical properties of logic programming interpreters, in: *Proc. Internat. Conf. on 5th Generation Computer Systems* (1984) 149-156.

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

[8] A. van Gelder, Efficient loop detection in PROLOG using the tortoise-and-hare technique, *J. Logic Programming* **4** (1987) 23-32.

[9] J.W. Lloyd, *Foundations of Logic Programming* (Springer, Berlin, 2nd edn., 1987).

[10] 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.

[11] A. Martelli and U. Montanari, An efficient unification algorithm, *ACM Trans. Programming Languages and Systems* **4** (1982) 258-282.

[12] J.F. Naughton, One-sided recursion, in: *Proc. 6th ACM Symposium on Principles of Database Systems* (ACM, New York, 1987) 340-348.

[13] D. Poole and R. Goebel, On eliminating loops in PROLOG, *SIGPLAN Notices* **20** (1985) 38-40.

[14] D.E. Smith, M.R. Genesereth and M.L. Ginsberg, Controlling recursive inference, *Artificial Intelligence* **30** (1986) 343-389.

[15] O. Štěpánková and P. Štěpánek, A complete class of restricted logic programs, in: F.R. Drake and J.K. Truss, eds., *Logic Colloquium '86* (North-Holland, Amsterdam, 1988) 319-324.
[16] H. Tamaki and T. Sato, OLD resolution with tabulation, in: G. Goos and J. Hartmanis, eds., *Proc. 3rd Internat. Conf. on Logic Programming*, Lecture Notes in Computer Science, Vol. 225 (Springer, Berlin, 186) 84-98.
[17] L. Vieille, Recursive query processing: the power of logic, *Theoret. Comput. Sci.* **69** (1989) 1-53.