Loop Checking and Negation

Roland N. Bol

Centre for Mathematics and Computer Science
P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.
Phone: (+31) - 20 - 592 4080. E-mail: bol@cwi.nl.

Abstract

In this article we extend the concept of loop checking from positive programs (as described in [ABK]) to locally stratified programs. Such an extension is not straightforward: the introduction of negation requires a (re)consideration of the choice of semantics, the description of a related search space and new soundness and completeness results handling floundering in a satisfactory way. Nevertheless an extension is achieved that allows us to generalize the loop checking mechanisms from positive programs to locally stratified programs, while preserving most soundness and completeness results. The conclusion is that negative literals cannot give rise to loops, and must be simply ignored.
This research was partly supported by Esprit BRA-project 3020 Integration.

1. Introduction

In [ABK] a formal framework is given for loop checking mechanisms that operate on top-down interpreters for positive logic programs. Such loop checks were also studied in [B], [BAK], [BW], [Co], [vG], [KT], [PG], [SGG], [SI] and [V]. However, all these papers except [KT] deal with positive programs only. This article extends the framework of [ABK] to interpreters for general logic programs, i.e. logic programs allowing negative literals in clauses' bodies. Several problems arising in the presence of negation are identified and solved.

First of all, we must choose a semantics for negation. For reasons explained in section 2.1, we restrict our attention to locally stratified programs, for which a clear semantics, namely *perfect model semantics* ([P1]), is available.

Secondly, we need an accurate description of the search space. It appears that for our purposes the standard *SLS-trees* ([P2]) do not present enough detail in the treatment of negative literals. Therefore these SLS-trees are augmented with *justifications*, which show explicitly the construction of a subsidiary SLS-tree of $\leftarrow A$, when $\neg A$ is selected.

A third and major problem (not treated in [KT]) is the occurrence of floundering: when only substitutions are used as computed answers, a non-ground negative literal cannot be answered properly: the derivation is said to *flounder*. Floundering lies between success and failure, making it hard to determine which floundering derivations may be pruned. This problem is solved by considering floundering derivations as *potentially* successful, and giving a *potential* answer substitution. These substitutions 'cover' the semantically correct answers

answers (which cannot be expressed as substitutions), but are possibly more general. A new completeness theorem for SLS-resolution, based on these potential answers, is proposed.

In order to keep the potential answer substitutions as specific as possible, a selection mechanism is proposed that postpones floundering as long as possible. It appears that the restriction to these selection rules allows us to prove a form of the 'independence of the selection rule' property, which is well-known for positive programs.

Once these problems are solved, it is possible to define loop checks for locally stratified programs, their soundness (no potential answer is lost) and completeness (the search space becomes finite). This is done in section 3. In the presence of negation, soundness becomes even more important than it was in the positive case: if a loop check prunes a (potential) success in a subsidiary SLS-tree, then the 'parent' SLS-tree should be extended; this extension might contain unsound answers. It is shown that a top-down SLS-interpreter remains sound and complete when it is augmented with a sound loop check.

Finally, in section 4 it is shown how loop checks for positive programs can be turned into loop checks for locally stratified programs. The main observation is that in locally stratified programs negative literals cannot give rise to a loop. Thus any loop is caused by positive literals and can be detected by a loop check for positive programs; the negative literals are simply removed. It can be shown that this construction preserves the completeness of the loop checks. Soundness is not preserved for every possible loop check (a counterexample using a highly non-typical loop check is given), but for 'reasonable' loop checks, including the ones studied in [BAK] (briefly introduced in the appendix), soundness is preserved.

2. Declarative and procedural semantics of negation

2.1. Motivation

Loop checks are used to prune the search space generated by a top-down interpreter. Therefore, before loop checks can be defined, this search space needs to be described properly. The search space must in turn agree with the intended semantics of the program. In the absence of negation, the choice was obvious: Herbrand models and SLD-trees. However, in the presence of negation, the choice is much less clear.

In the most well-known approach, treated in [L], the intended semantics is derived from the *completion* of a program; the corresponding search space consists of *SLDNF-trees*, obtained when the interpreter is equipped with the *negation as finite failure* rule. Informally this rule states that $\neg A$ may be inferred when an attempt to prove A (again by SLDNF-resolution) fails after a finite number of resolution steps. According to Theorem 16.5 of [L], for positive programs the completion semantics corresponds exactly to finite failure, i.e. $\neg A$ is a logical consequence of the completion of P if and only if $P \cup \{\leftarrow A\}$ fails finitely. Due to the restriction to *finite* failure, this approach is hardly compatible with the use of a loop check. Indeed, the intention of loop checking is to turn infinite (hence failed) paths in the search space into finitely failed paths. Thus if $P \cup \{\leftarrow A\}$ fails finitely due to the use of a loop check, $\neg A$ is not entailed by the completion-semantics. So completion semantics is inappropriate for our purposes.

Numerous alternative semantics have been proposed. Here we adopt an approach of Przymusinski, which is based on *perfect model semantics* ([P1]). Furthermore, we restrict our attention to *locally stratified programs*; it is shown in [P1] that these programs have a unique

perfect Herbrand model (a 'perfect' model is defined as being minimal w.r.t. a certain partial ordering on models, which is a refinement of the usual subset ordering). In [P2] a corresponding search space, called *SLS-trees*, is defined for stratified programs; this definition is generalized here to locally stratified programs. As pointed out by Przymusinski, an SLS-tree represents the search space of a top-down interpreter, equipped with the 'negation as failure' (not necessarily finite failure) rule.

Obviously this rule is in general not effective so, unlike SLDNF-resolution, SLS-resolution cannot be effectively implemented, but only approximated. However, as Przymusinski suggests, loop checks can yield such approximations: 'Suitable loop checking can be added to SLS-resolution without destroying its completeness. For large classes of stratified programs, SLS-resolution with subsumption check will result in finite evaluation trees and therefore can be implemented as a complete and always terminating algorithm. This is the case, in particular, for function-free programs.' ([P2]). One of the contributions of this paper is a substantiation of this claim.

2.2. Basic definitions

Throughout this paper we assume familiarity with the basic concepts and notations of logic programming as described in [L]. For two expressions E and F, we write $E \leq F$ when F is an instance of E. We then say that E is *more general* than F.

DEFINITION 2.1 (Local Stratification).
Let P be a program. P is *locally stratified* if there exists a mapping *stratum* from the set of ground atoms of L_P to the countable ordinals, such that for every clause
$(H \leftarrow A_1,\ldots,A_m,\neg B_1,\ldots,\neg B_n) \in$ ground(P): for $1 \leq i \leq m$, stratum(A_i) $\leq$ stratum(H) and
for $1 \leq i \leq n$, stratum(B_i) $<$ stratum(H). □

Obviously, stratified programs ([ABW]) and programs without negation (*positive* programs) are locally stratified. *From now on, only locally stratified programs shall be considered*, therefore we usually omit the qualification 'locally stratified'. Consequently, we assume that for every considered program a mapping *stratum*, satisfying the above requirements, is available.

DEFINITION 2.2.
Let P be a program. We extend the mapping *stratum* as follows.
1. For an atom A, not necessarily ground,
   stratum(A) = sup { stratum(A_0) | A_0 is a ground instance of A in L_P }.
2. For a negative literal $\neg$A, not necessarily ground,
   stratum($\neg$A) = stratum(A) + 1
3. For a goal G,

$$\text{stratum}(G) = \begin{cases} 0 & \text{if } G = \square, \\ \max \{ \text{stratum}(L_i) \mid 1 \leq i \leq n \} & \text{if } G = \leftarrow L_1,\ldots,L_n. \end{cases}$$

□

A *selection rule* is a rule determining the order in which literals are selected in goals of a derivation. A well-known problem concerning the 'negation as (finite) failure' rule is

floundering: the selection of a non-ground negative literal (cf. [Cl], [L]). We assume that such a selection is avoided whenever possible.

DEFINITION 2.3.
A selection rule is *safe* if it never selects a non-ground negative literal in a goal containing positive and/or ground negative literals. □

Following Przymusinski's presentation for stratified programs in [P2], we now define for a given program P and goal G the SLS-tree of P∪{G}, together with some related notions. The definition uses induction on stratum(G).

DEFINITION 2.4 (SLS-Tree).
Let P be a program and G a goal. Let **R** be a fixed safe selection rule. Assume that SLS-trees have already been defined for goals H such that stratum(H) < stratum(G). We define the *SLS-tree* T *of P∪{G} via R*. (In fact this tree is not uniquely defined, as the choice of the names of auxiliary variables is left free.)

The root node of T is G. For *any* node H in T, its immediate descendents are obtained as follows:

- if H = □, then H has no descendents and is a success leaf.
- if **R** selects a non-ground negative literal in H, then H has no descendents and is a flounder leaf.
- if **R** selects a positive literal L in H, then H has as immediate descendants: for every applicable program clause C in P, a goal K that is obtained by resolving H with (a suitable variant of) C upon the literal L using an idempotent most general unifier θ (such a derivation step is denoted as $H \Rightarrow_{C,\theta} K$).

  If no program clauses are applicable, then H is a failure leaf.
- if **R** selects a ground negative literal L = ¬A in H, then the SLS-tree T' of P∪{←A} via **R** has already been defined.

  (Either some ground instance Bγ of an atom B in G depends negatively on A, therefore stratum(G) ≥ stratum(B) ≥ stratum(Bγ) > stratum(A); or ¬A is an instance of a negative literal in G, so again stratum(G) > stratum(A).)

  T' is called a *side-tree of H* (or, *of T*). We consider three cases:
  - if all leaves of T' are failed, then H has only one immediate descendant, namely the goal K = H - {L}, i.e. the goal H with L removed (such a derivation step is denoted as $H \Rightarrow_{C,\varepsilon} K$).
  - if T' contains a success leaf, then H has no immediate descendants and is a failure leaf.
  - otherwise, H has no immediate descendants and is a flounder leaf.

If T has a success (flounder) leaf then T is *successful* (*floundered*); hence an SLS-tree may be both successful and floundered. T is *failed* if *all* of its leaves are failed (note that a failed SLS-tree may contain infinite branches).

An *SLS-derivation* (*of P∪{G}*) is an initial segment of a branch of an SLS-tree (of P∪{G}). An SLS-derivation ending in a success (flounder) leaf is called *successful* (*floundered*). An SLS-derivation is *failed* if it is infinite or ends in a failure leaf.

A successful SLS-derivation (or *SLS-refutation*) of P∪{G} yields a *computed answer substitution* σ in the same way an SLD-refutation does: whenever in a refutation step a negative literal is selected, such a step does not contribute to the computed answer substitution. Gσ is called the *computed answer* of the derivation.

An SLS-derivation or -tree of P∪{G} is *potentially successful* if it is successful or floundered. The *potential answer substitution* σ of a potentially successful SLS-derivation is again the sequential composition of the mgu's of the derivation (thus the potential answer substitution of a refutation coincides with its computed answer substitution). Its *potential answer* is again Gσ. □

2.3 Soundness and completeness

We need the following soundness and completeness results, which strengthen the results of [Ca], [KT] and [PP]. We omit the proof.

THEOREM 2.5 (Soundness and Completeness of SLS-resolution). *Let P be a program and* $G = \leftarrow L_1,\ldots,L_n$ *a goal. Let* M_P *be the unique perfect Herbrand model of P as defined in [P1]. Let* **R** *be a safe selection rule and* θ *a substitution.*

i) If $G\theta$ *is a computed answer for* $P\cup\{G\}$ *then* $\forall((L_1\wedge\ldots\wedge L_n)\theta)$ *is true in* M_P.

ii) If $P\cup\{G\}$ *has a failed SLS-tree, then* $\neg\exists(L_1\wedge\ldots\wedge L_n)$ *is true in* M_P.

iii) If $\forall((L_1\wedge\ldots\wedge L_n)\theta)$ *is true in* M_P, *then there exists a potentially successful SLS-derivation of* $P\cup\{G\}$ *via* **R** *giving a potential answer* $G\sigma \leq G\theta$.

iv) If $\neg\exists(L_1\wedge\ldots\wedge L_n)$ *is true in* M_P, *then the SLS-tree for* $P\cup\{G\}$ *via* **R** *is not successful.* □

Theorem 2.5 allows us to omit in further considerations the perfect model semantics: in order to show that a loop check respects this semantics it is sufficient to compare pruned SLS-trees with original, unpruned trees. The following example shows why a stronger result than the one presented in [Ca] is needed here.

EXAMPLE 2.6.

Let P = { p(1) ←. (C1),
p(y) ← p(y),¬q(y). (C2),
q(1) ← ¬r(x). (C3) }.

Using the leftmost selection rule yields the SLS-tree of P∪{←p(x)} in Figure 2.1. Since the tree flounders, ordinary completeness results like the one in [Ca] cannot be used. However, a loop check might very well prune the goal ←p(x),¬q(x). Then the pruned tree does not flounder, so it is expected to be complete. Indeed this completeness follows from Theorem 2.5 (as the only potential answer substitution occurring in the tree is {x/1}). □

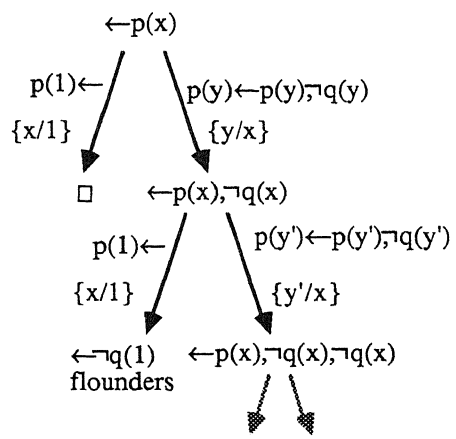

FIGURE 2.1

2.4. A more precise description of the search space

In the positive case, when a program P and a goal G are input to the interpreter, only an SLD-tree of P∪{G} is searched. However, in the presence of negation, not only an SLS-tree of P∪{G} is searched, but also its side-trees, and the side-trees of its side-trees, et cetera. We call such a construct consisting of an SLS-tree and its side-trees (to the required depth) a *justified SLS-tree*. As in Definition 2.4, induction on *stratum* is used.

DEFINITION 2.7 (Justified SLS-Tree).

Let P be a program and G a goal. Let **R** be a fixed safe selection rule. A *justified SLS-tree* T of P∪{G} via **R** consists of an SLS-tree T_{top} of P∪{G} via **R**, which is, for every goal H in T_{top} in which a ground negative literal ¬A is selected, augmented with a justified SLS-tree T' of P∪{←A} via **R**. Such a tree T' is called a *justification of H* (or, *of T*), T_{top} is called the *top level* of T. T is successful (potentially successful, floundered, failed) if T_{top} is successful (potentially successful, floundered, failed). The computed/potential answers of T are those of T_{top}. □

Notice the relationship between a side-tree T of H (an SLS-tree), and a justification J of H (a justified SLS-tree): T is the top level of J. Figure 2.2 shows a justified SLS-tree.

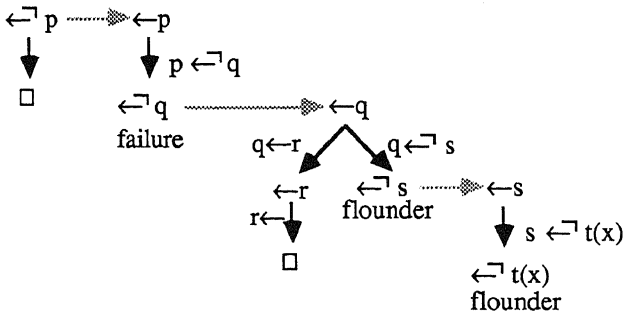

FIGURE 2.2

In order to render potential answers as specific as possible, it is worthwhile to 'postpone' floundering until all non-floundering literals are resolved. This is achieved by considering the class of *deeply safe* justified SLS-trees defined below. The definition uses induction on *stratum* again.

DEFINITION 2.8.

A justified SLS-tree is *deeply safe* if for every flounder leaf ←L_1,...,L_n in it, every literal L_i ($1 \le i \le n$) is a negative literal ¬A_i, and either A_i is non-ground or every deeply safe justified SLS-tree of P∪{←A_i} flounders unsuccessfully. □

In a deeply safe justified SLS-tree, all justifications are also deeply safe (as the definition refers to *every* flounder leaf, not only those at the top level). Using a safe selection rule alone is

not enough to obtain deeply safe trees: a ground negative literal $\neg A$ may be selected in a goal that still contains positive literals; then the side-tree of $\neg A$ may unsuccessfully flounder.

At first, it seems that checking whether 'every deeply safe justified SLS-tree of $P \cup \{\leftarrow A_i\}$ flounders' requires the construction of deeply safe trees of $P \cup \{\leftarrow A_i\}$ via every possible selection rule. The following lemma shows that this is not the case, as for deeply safe trees the independence of the selection rule holds.

LEMMA 2.9 (Independence of the selection rule for deeply safe trees). *Let P be a program and G a goal. Let T_1 and T_2 be deeply safe justified SLS-trees of $P \cup \{G\}$. Then there exists a bijection φ from the potentially successful branches in T_1 to the potentially successful branches in T_2 such that $|B| = |\varphi(B)|$ and the potential answers of B and $\varphi(B)$ are variants. Moreover, B is successful if and only if $\varphi(B)$ is.* □

Therefore a valid method for creating deeply safe justified SLS-trees is to create only one (again deeply safe) justification for a selected negative literal. If this justification flounders unsuccessfully, then the literal is marked as 'floundering' and the interpreter 'backtracks' over this selection (that is, this selection is 'undone', and another literal is selected). Only if all literals in a goal are marked as 'floundering', the goal is a flounder leaf.

The final lemma of this section shows that in deeply safe trees the occurrence of floundering is indeed reduced to a minimum.

LEMMA 2.10. *Let P be a program and G a goal. Let T_1 and T_2 be justified SLS-trees of $P \cup \{G\}$ and let T_1 be deeply safe.*

i) For every computed answer in T_2, T_1 contains a variant of it.

ii) For every potential answer in T_1, T_2 contains a more general potential answer. □

3. Loop checks for locally stratified programs

3.1. Motivation

In this section we give a formal definition of loop checks for locally stratified programs (based on SLS-derivations), following the presentation of loop checks for positive programs in [ABK]. The purpose of augmenting an interpreter with a loop check is to prune the generated search space while retaining soundness and completeness. We define and study those properties of loop checks that are needed to achieve this goal.

Since loop checks can be used to prune every part of a justified SLS-tree, one might define a loop check as a function from justified SLS-trees to justified SLS-trees, directly showing where the trees are changed. However, this would be a very general definition, allowing practically everything. A first restriction we make is that a loop check acts only *within* an SLS-tree, disregarding its justifications and the possibility that this SLS-tree itself may be part of a justification in another SLS-tree. We shall formally call such loop checks for locally stratified programs *one level* loop checks. Nevertheless, we usually omit the qualification 'one level', unless confusion with *positive* loop checks (loop checks for positive programs, as defined in [ABK]) can arise. This restriction leaves the possibility open that loop checks are used to prune more than one tree in a justified SLS-tree.

We restrict the scope of loop checks even more, namely from SLS-trees to SLS-derivations. This in contrast to [KT] and [V], where the complete tree (as far as it is constructed already) is taken into account. Consequently, their methods are more powerful, but also highly dependent on the selection rule used.

As in [ABK] we define for a program P:

- a node in an SLS-tree of P∪{G} (for some goal G) is *pruned* if all its descendants are removed. (Note the terminology: the pruned node itself remains in the tree.)
- by pruning some of its nodes we obtain a pruned version of an SLS-tree.
- whether a node of an SLS-tree is pruned depends only upon its ancestors in the tree, that is on the SLS-derivation from the root to this node.

Therefore, a loop check can be defined as a set of finite SLS-derivations (possibly depending on the program): the derivations that are pruned exactly at their last node. Such a loop check L can be extended in a canonical way to a function f_L^1 from SLS-trees to SLS-trees: f_L^1 prunes in an SLS-tree of P∪{G} the nodes in { H | the SLS-derivation from G to H is in L(P) }. Extending L to a function f_L^* from justified SLS-trees to justified SLS-trees is less straightforward. This subject is discussed in the next section.

In a still more restricted form of a loop check, called simple loop check, the set of pruned derivations is also independent of the program P. This leads us to the following definitions.

3.2. Definitions

DEFINITION 3.1.
Let L be a set of SLS-derivations. *RemSub(L)* = { D ∈ L | L does not contain a proper initial subderivation of D }. L is *subderivation free* if L = RemSub(L). □

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

In the following definition, by a *variant* of a derivation D we mean a derivation D' in which in every derivation step, literals in the same positions are selected and the same program clause respectively side-tree is used. D' may differ from D in the renaming that is applied to the program clauses for reasons of standardizing apart and in the mgu used. Thus any variant of an SLS-derivation is also an SLS-derivation.

DEFINITION 3.2.
A *simple one level loop check* is a computable set L of finite SLS-derivations such that L is closed under variants and subderivation free. □

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

DEFINITION 3.3.
A *one level loop check* is a computable function L from programs to sets of SLS-derivations such that for every program P, L(P) is a simple one level loop check. □

In [ABK], (simple) positive loop checks have been defined in the same way, using SLD-derivations instead of SLS-derivations.

DEFINITION 3.4.
Let L be a loop check. An SLS-derivation D of P∪{G} is *pruned by L* if L(P) contains a initial subderivation D' of D. □

We now formalize how a justified SLS-tree is pruned. To simplify the definition, we assume that only one loop check L is used to prune a justified SLS-tree T: both the top level of T and (recursively) all justifications of T are pruned by L.

A problem arises when L prunes the justification of a goal G to such an extent that (potential) success in it is lost: instead of being a failure (flounder) leaf, G should now obtain a descendant, i.e. the search space of an interpreter with such a loop check *extends* the original search space beyond G. Modelling this additional search space is problematic, as there is no original tree to follow. We avoid this problem temporarily by turning such a leaf G into an *extension leaf*. In this way the pruned tree remains a subtree of the original one. This property can be well exploited in the proof of the soundness and completeness of SLS-resolution with loop checking, where pruned trees are compared with original ones, and Theorem 2.5 is used.

DEFINITION 3.5 (Pruning justified SLS-trees).
Let P be a program and G a goal. Let L be a loop check and let T be a justified SLS-tree of P∪{G}. Then the tree $T_p = f_L^*(T)$, the pruned version of T, is defined as follows.

The root node of T_p is G. For *any* node H in the top level of T_p, the same literal as in T is selected; the immediate descendants of H in T_p are:

- if the SLS-derivation from G to H is pruned by L, then H has no descendants and is a *pruned leaf*.
- otherwise:
  - if a ground negative literal is selected in H, then H has a justification T' in T. The pruned version of T', $T_p' = f_L^*(T')$, is already defined by induction. T_p' is the (pruned) justification of H in T_p. We consider the top level of T_p':
    - if it contains a success leaf, then H has no immediate descendants and is a failure leaf.
    - otherwise, if it contains a flounder leaf, then H has no immediate descendants and is a flounder leaf.
    - otherwise, if it contains an extension leaf or if H has no descendants in T, then H has no immediate descendants in T_p and is an *extension leaf*.
    - otherwise, H has in T_p the same immediate descendant as in T.
  - otherwise H has in T_p the same descendants (or the same leaf-type) as in T.

A pruned justified SLS-tree is *successful* (etc.) if one of its top level leaves is successful (etc.). It is *failed* if all its top level leaves are either failed or pruned. □

EXAMPLE 3.6.
When a loop check pruning the goal ←r is applied to the SLS-tree in Figure 2.2, the tree depicted in Figure 3.1 is obtained. When the goal ←s is also pruned, then the tree of Figure 3.2 is obtained. □

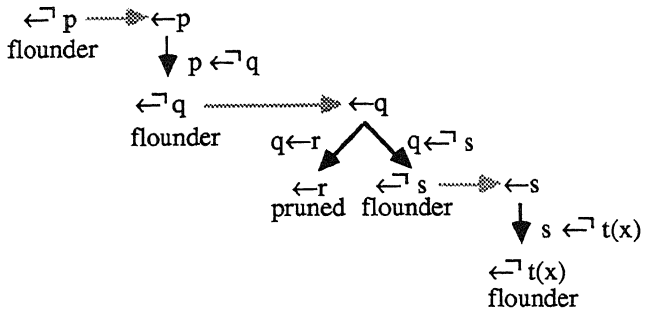

FIGURE 3.1

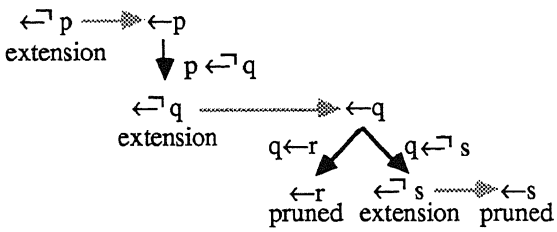

FIGURE 3.2

3.3. Soundness and completeness

In this section a number a properties of one level loop checks is defined. The definitions are only concerned with the effect of applying a loop check on the top level of a justified SLS-tree. In section 3.4 the influence of applying loop checks (satisfying these definitions) on all levels of a justified SLS-tree is studied.

As was pointed out before, using a loop check should not result in losing potential success. In order to retain completeness, an even stronger condition is needed: we may not lose any individual solution. Since Theorem 2.5(iii) involves potential answers, pruning the tree should preserve those successful and floundering branches that indicate (the possibility of) solutions not otherwise found. That is, if the original SLS-tree contains a potentially successful branch (giving some answer), then the pruned tree should contain a potentially successful branch giving a more general answer.

In order to consider only those potential answers that are as specific as possible, only deeply safe justified SLS-trees are taken into account. (Otherwise we would not be allowed to prune a floundering derivation like '$\leftarrow p \Rightarrow_{p \leftarrow p, \neg r} \leftarrow p, \underline{\neg r}$ flounder'.)

DEFINITION 3.7 (Soundness).
Let **R** be a safe selection rule and let L be a loop check.
i) L is *weakly sound* if for every program P and goal G, and potentially successful deeply safe justified SLS-tree T of P∪{G}, $f_L^1(T_{top})$ is potentially successful.
ii) L is *sound* if for every program P and goal G, and deeply safe justified SLS-tree T of P∪{G}: if T contains a potentially successful branch giving a potential answer Gσ, then $f_L^1(T_{top})$ contains a potentially successful branch giving a potential answer Gσ' ≤ Gσ. □

Thus every sound loop check is weakly sound. Moreover, if the initial goal G is ground (which is always the case for side-trees), then the notions of weakly sound and sound coincide.

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

DEFINITION 3.8 (Completeness).
A loop check L is *complete w.r.t. a selection rule **R** for a class of programs* $\mathcal{C}$, if for every program P ∈ $\mathcal{C}$ and goal G in L_P, every infinite SLS-derivation of P∪{G} via **R** is pruned by L. □

We must point out here that in these definitions we have overloaded the terms 'soundness' and 'completeness'. These terms now refer both to loop checks and to interpreters (with or without a loop check). In the next section we study how the soundness and completeness of a loop check affects the soundness and completeness of the interpreter augmented with it.

3.4. Interpreters and loop checks

We now present the conditions ensuring that an SLS-interpreter augmented with a loop check remains sound and complete (in the sense of Theorem 2.5). We need one more definition.

DEFINITION 3.9.
A loop check L is *selection-independent* if for every program P and for every D ∈ L(P), { D' | D' differs from D only in the selection of the literal in its last goal } ⊆ L(P). □

The restriction to selection-independent loop checks is not a severe one. Intuitively, after the creation of a new goal the loop check is performed first. Only when no loop is detected a further resolution step is attempted; to this end a literal is selected. All loop checks in [BAK] (see the appendix) are selection-independent.

Due to the introduction of extension leaves, a pruned justified SLS-tree does generally *not* cover the entire search space for the SLS-interpreter augmented with a loop check. For, whether a node is an extension leaf depends (partly) on the unpruned SLS-tree. This tree is not available for the loop-checked interpreter, so it cannot decide to stop at an extension leaf. Beyond an extension leaf, it might find incorrect answers. Therefore we must first of all ensure the absence of extension leaves in soundness results. The other items correspond to Theorem 2.5.

THEOREM 3.10 (Soundness and Completeness of SLS-resolution with loop checking).
Let P be a program and G a goal. Let **R** *be a safe selection rule and θ a substitution. Let T be a justified SLS-tree of P∪{G} via* **R**. *Let L be a weakly sound selection-independent loop check. Then there exists a justified SLS-tree T' of P∪{G} such that:*
-) $T_P' = f_L^*(T')$ represents the search space for P∪{G} of a top-down SLS-interpreter using **R**, *augmented with the loop check L (i.e. T_P' has no extension leaves and makes all selections according to* **R**, *except for the selections in pruned leaves).*
i) If Gθ is a computed answer in T_P' then Gθ is a computed answer in T'.
ii) If T_P' is failed, then T' is failed.
iii) If L is sound and T' contains a potential answer Gθ, then T_P' contains a potential answer $G\sigma \leq G\theta$.
iv) If T' is not successful, then T_P' is not successful.
PROOF. We omit the full proof; we only outline the construction of T':
1. Replace all justifications J by J' (by induction on *stratum*, J' is derived from J as T' from T).
2. It can be proved that if J is successful/failed, then so is J'. If a floundering justification is replaced by a failed one, then the tree is extended via **R** (still using these 'new' justifications).
3. The goals in this tree that would be pruned by L are marked.
4. The selections made at and beyond these marked goals are replaced by deeply safe ones. Note that this part of the tree is never really constructed by the loop-checked interpreter. □

Thus combining Theorem 3.10(-) and (i)-(iv) with Theorem 2.5(i)-(iv) (applied on T') gives the final soundness and completeness results. However, the (loop-checked) interpreter need not be effective: in general traversing infinite justifications is required. Any real interpreter can only traverse a finite part of a (justified) SLS-tree, and is therefore incomplete.

THEOREM 3.11. *Let P be a program and G a goal in L_P. Let L be a loop check. Let* **R** *be a safe selection rule, let T be a justified SLS-tree of P∪{G} via* **R** *and let $T_P = f_L^*(T)$.*
i) If L is complete w.r.t. **R** *for a class of programs 𝒞 containing P, then T_P is finite.*
ii) If a flounder leaf occurs in T_P, then a flounder leaf occurs in T. □

Applying Theorem 3.11(i) on the tree T' as constructed in Theorem 3.10 shows that using a *complete* loop check (on all levels) ensures that the pruned justified SLS-tree is finite. If also the conditions of Theorem 3.10 are met, then it follows that indeed the search space of the interpreter is finite. In this case the interpreter is *really* sound and complete.

Finally, Theorem 3.11(ii) indicates that the pruned tree can only flounder if somewhere in the original tree (but not necessarily at the top level) floundering occurs. Example 4.10 shows that a stronger result can hardly be expected: if the tree in Figure 4.2 is the side-tree of the goal ←¬p, then most loop checks applied there turn ←¬p from a failure leaf into a flounder leaf.

4. Deriving one level loop checks from positive loop checks

4.1. Definitions

In this section we show how one level loop checks can be derived from positive loop checks. Since a successfully resolved negative literal is simply removed from a goal, negative literals

cannot give rise to loops. (Thanks to the fact that we consider only locally stratified programs, looping 'through negation' cannot occur.) Therefore the basic idea is to remove all negative literals in a derivation. Then an SLD-derivation remains, to which a positive loop check is applied.

NOTATION 4.1.
For every (goal- and program-) clause, program, SLS-derivation and -tree X, X^+ denotes the object obtained from X by removing all negative literals. Thus if X is an SLS-derivation or -tree, then in X^+ every derivation step $G \Rightarrow H$ in which a negative literal is selected in G is deleted, since in this case $G^+ = H^+$. □

Notice that for every SLS-derivation D of $P \cup \{G\}$, D^+ is an SLD-derivation of $P^+ \cup \{G^+\}$. For an SLS-tree T of $P \cup \{G\}$, T^+ is an 'initial segment' of an SLD-tree of $P^+ \cup \{G^+\}$ (due to failure or floundering of a negative selected literal in T, T^+ is not necessarily completed).

In fact the above definition is not completely precise: suppose that in the last goal G of an SLS-derivation D, a negative literal is selected. Then it is not clear which atom is selected in G^+ in D^+. Nevertheless, as the loop checks we are interested in are all selection-independent (Definition 3.9 does also apply to positive loop checks) we do not need to be more precise.

DEFINITION 4.2.
Let L be a positive loop check. *The one level loop check derived from L,*
$O_L = \lambda P.\mathrm{RemSub}(\{ D \mid D \text{ is an SLS-derivation and } D^+ \in L(P^+) \})$. □

The following lemma's establish the required relationships between a positive loop check and the one level loop check derived from it.

LEMMA 4.3. *For every positive loop check L, O_L is a one level loop check.*
Moreover, O_L is simple iff L is simple. □

LEMMA 4.4. *Let L be a positive loop check, D an SLS-derivation and P a program.*
D is pruned by $O_L(P)$ iff D^+ is pruned by $L(P^+)$.
PROOF. D is pruned by $O_L(P)$ iff some initial part of D, $D_{in} \in O_L(P)$, iff some initial part of D^+, $D_{in}^+ \in L(P^+)$, iff D^+ is pruned by $L(P^+)$. □

4.2. Soundness

Unfortunately, as is shown in the following counterexample, it is not the case that a one level loop check derived from a (weakly) sound positive loop check (as defined in [ABK]) is again (weakly) sound.

COUNTEREXAMPLE 4.5.
Let P = {
p ← q(1),q(2). (C1),
q(x) ← ¬r(x). (C2),
q(2) ← q(1). (C3),
r(2) ←. (C4) },

and let G = ←p. P is (locally) stratified and Figure 4.1 shows an SLS-tree T of P∪{G} via the leftmost selection rule (a failure leaf is marked by a box around it).

Let D denote the successful branch in T. Then D^+ is the SLD-derivation $\leftarrow p \Rightarrow_{(C1)} \leftarrow q(1),q(2) \Rightarrow_{q(x)\leftarrow} \leftarrow q(2) \Rightarrow_{(C3)} \leftarrow q(1) \Rightarrow_{q(x)\leftarrow} \square$. Even a *simple* sound loop check L might prune the goal ←q(1) in D^+: it is visible in the second step of D^+ that the clause q(x)← is present in P^+; this clause allows for a shorter way to refute ←q(2) than via (C3) and ←q(1).

Unfortunately, this shortcut fails in the SLS-tree because it introduces the literal ¬r(2) instead of ¬r(1), and ¬r(2) fails. So O_L prunes D, hence O_L is not weakly sound (note that the tree is the top level of a deeply safe justified tree). □

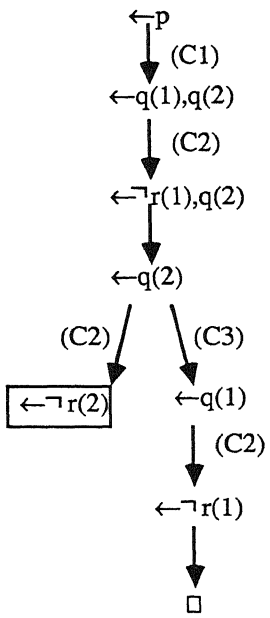

FIGURE 4.1

Although the loop check used in the counterexample formally satisfies the definitions, it is highly non-typical. It appears that more usual (weakly) sound positive loop checks, notably the ones defined in [BAK], derive again (weakly) sound one level loop checks. We omit the proof, which is similar to the soundness proofs for positive loop checks in [BAK]. The loop checks mentioned below are described in the appendix.

THEOREM 4.6 (Soundness of Conversion).

i) The one level loop checks derived from the equality, subsumption and context checks based on goals are weakly sound.

ii) The one level loop checks derived from the equality, subsumption and context checks based on resultants are sound. □

4.3. Completeness

Since some completeness properties of positive loop checks depend on the selection rule used, these selection rules are adapted to the presence of negation.

DEFINITION 4.7.

Let **R** be a selection rule for SLD-derivations.

An *extension of* **R** is a selection rule **R'** for SLS-derivations such that for every SLS-derivation D via **R'**, D^+ is an SLD-derivation via **R**. □

Unlike soundness, completeness carries over from positive to one level loop checks without much difficulty.

THEOREM 4.8 (Completeness of Conversion). *If L is complete w.r.t. a selection rule* **R** *for a class of programs* $\mathcal{C}$*, then* O_L *is complete w.r.t. any safe extension of* **R** *for the class of programs* $\{ P \mid P^+ \in \mathcal{C} \text{ and } L_P = L_{P^+} \}$. □

Notice that the requirement $L_P = L_{P+}$ is just a technicality which can be met easily by adding some non-relevant clauses to P. For example, the result presented in Theorem 5.3(i) is transferred to one level loop checks as follows.

COROLLARY 4.9. *The one level loop checks derived from the equality, subsumption and context checks are complete w.r.t. any safe extension of the leftmost selection rule for locally stratified function-free programs in which in every clause only the rightmost positive literal (if present) may depend on the head of that clause.* □

4.4. Concluding remarks

The Soundness of Conversion Theorem 4.6 and the Completeness of Conversion Theorem 4.8 allow the immediate conversion of all positive loop checks described in [BAK] (see the appendix) and their soundness and completeness results (see [BAK] and [Bol]) to one level loop checks. The following example presents the application of several one level loop checks derived from these positive loop checks .

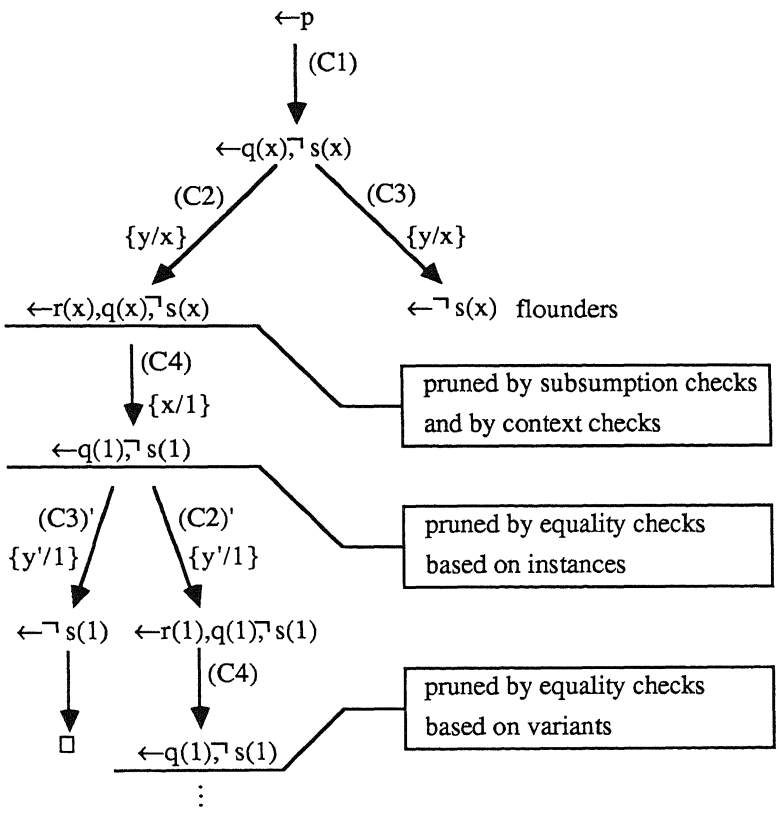

FIGURE 4.2

EXAMPLE 4.10.

Let P = { p ← q(x),¬s(x). (C1),
q(y) ← r(y),q(y). (C2),
q(y) ←. (C3),
r(1) ←. (C4) },

and let G = ←p. In Figure 4.2 an SLS-tree T of P∪{G} via (a safe extension of) the leftmost selection rule is depicted. It is shown where T is pruned by various loop checks.

For every loop check used, potential success is retained in the pruned tree as expected. However, it appears that only the equality checks based on variants retain a successful branch. Obviously the extra instantiation in this branch, which was superfluous in the positive case, serves here to prevent floundering.

Notice that it is not always the case that the equality checks based on variants retain at least one successful branch. Consider the goal ←r and the program {p(x)←; p(1)←p(x); r←p(x),¬q(x) }: applying the second clause instantiates only the negative literal, so the successful derivation is pruned. □

5. Appendix

Here we recall the three groups of simple loop checks that are introduced in [BAK], together with their respective soundness and completeness results.

5.1 Definitions and soundness results

First we present the weakly sound loop checks of each group.

The first group consists of the equality checks. They check whether the current goal G_k is an instance of a previous goal G_i, i.e. if for some substitution τ: $G_k = G_i\tau$. Small variations on this criterion give rise to various loop checks within this group. These variations are notably the two interpretations of '=' that are considered (goals can be treated as lists or as multisets) and the possible addition of the requirement 'τ is a renaming' (in other words: 'G_k is a variant of G_i'). Such variations are also possible in the other groups of loop checks, but as they have not much effect on soundness and completeness, we shall not mention them any more.

The second group consists of the subsumption checks. Their loop checking criterion has the form 'for some substitution τ: $G_k \subseteq G_i\tau$' (or in words: 'G_k is subsumed by an instance of G_i'). Although the replacement of = by $\subseteq$ seems to be yet another small variation, it appears that subsumption checks are really more powerful than equality checks.

The third group consists of the context checks, introduced by Besnard [B]. Their loop checking condition is more complicated: 'For some atom A in G_i, $A\theta_{i+1}\ldots\theta_j$ is selected in G_j to be resolved. As the (direct or indirect) result of resolving $A\theta_{i+1}\ldots\theta_j$, an instance $A\tau$ of A occurs in G_k ($0 \le i \le j < k$). Finally, for every variable x that occurs both inside and outside of A in G_i, $x\theta_{i+1}\ldots\theta_k = x\tau$.'

For all these weakly sound loop checks, a sound counterpart is obtained by adding the condition '$G_0\theta_1\ldots\theta_k = G_0\theta_1\ldots\theta_i\tau$' to the loop checking criterion. For reasons explained in [BAK], the loop checks thus obtained are called 'based on resultants' as opposed to the weakly sound ones, which are 'based on goals'.

Thus the following results were proved in [BAK].

THEOREM 5.1.

i) The equality, subsumption and context checks based on goals are weakly sound.

ii) The equality, subsumption and context checks based on resultants are sound. □

5.2 Completeness results

Due to the undecidability of the halting problem, a weakly sound loop check cannot be complete for all programs. In [BAK] it was shown that a weakly sound *simple* loop check cannot even be complete for all *function-free* programs. Three classes of function-free programs were isolated for which the completeness of (some of) the loop checks mentioned above could be proved. We now present those classes of programs and completeness results.

DEFINITION 5.2.

A program P is *restricted* if for every clause $H \leftarrow A_1,\ldots,A_n$ in P, the definitions of the predicates in $A_1,\ldots,A_{n-1}$ do not depend on the the predicate of H in P. (So recursion is allowed, namely through A_n, but double recursion is not.)

A program P is *non-variable introducing* (*nvi*) if for every clause $H \leftarrow A_1,\ldots,A_n$ in P, every variable that occurs in $A_1,\ldots,A_n$ occurs also in H.

A program P has the *single variable occurrence* property (*is svo*) if for every clause $H \leftarrow A_1,\ldots,A_n$ in P, no variable occurs more than once in $A_1,\ldots,A_n$. □

THEOREM 5.3.

i) All equality, subsumption and context checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.

ii) All subsumption and context checks are complete for function-free nvi programs and for function-free svo programs. □

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.

[ABW] K.R. APT, H. BLAIR and A. WALKER, *Towards a Theory of Declarative Knowledge*, in: Foundations of Deductive Databases and Logic Programming (J. Minker ed.), Morgan Kaufmann Publishers, Los Altos, 1987, 89-148.

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

[BAK] R.N. BOL, K.R. APT and J.W. KLOP, *An Analysis of Loop Checking Mechanisms for Logic Programs*, Technical Report CS-R8942, Centre for Mathematics and Computer Science, Amsterdam; Technical Report TR-89-32, University of Texas at Austin, 1989. To appear in Theoretical Computer Science.

[Bol] R.N. BOL, *Generalizing Completeness Results for Loop Checks*, Technical Report CS-R9025, Centre for Mathematics and Computer Science, Amsterdam, 1990.

[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.

[Ca] L. CAVEDON, *Continuity, Consistency, and Completeness Properties for Logic Programs*, Technical Report 88/33, Dept. of Comp. Sci., University of Melbourne, 1988. To appear in Theoretical Computer Science.

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

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

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

[KT] D.B. KEMP and R.W. TOPOR, *Completeness of a Top-Down Query Evaluation Procedure for Stratified Databases*, in: Proceedings of the Fifth International Conference on Logic Programming, MIT Press, Cambridge Massachusetts, 1988, 178-194.

[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.

[P1] T.C. PRZYMUSINSKI, *On the Declarative Semantics of Deductive Databases and Logic Programs*, in: Foundations of Deductive Databases and Logic Programming (J. Minker ed.), Morgan Kaufmann Publishers, Los Altos, 1987, 193-216.

[P2] T.C. PRZYMUSINSKI, *On the Declarative and Procedural Semantics of Logic Programs*, J. Automated Reasoning 5, 1989, 167-205.

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

[PP] H. PRZYMUSINSKA and T.C. PRZYMUSINSKI, *Weakly Perfect Model Semantics for Logic Programs*, in: Proceedings of the Fifth International Conference on Logic Programming, (R.A Kowalski and K.A. Bowen eds.), MIT Press, Cambridge Massachusetts, 1988, 1106-1120.

[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 69, 1989, 1-53.