LOOP CHECKING AND NEGATION

ROLAND N. BOL

▷ In this paper we extend the concept of loop checking from positive programs (as described in [1]) 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. Note: the material presented in this paper is contained in [5, ch. 5], in which also [1, 4] can be found. ◁

1. INTRODUCTION

In [1], 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 [3, 4, 6, 10, 11, 14, 17, 22–26]. However, all of these papers except [14, 22, 24] deal with positive programs only (the differences between the present paper and these three are discussed in Section 3.4). This paper extends the framework of [1] 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* [19], is available. The recently proposed *stable semantics* [13] and *well-founded semantics* [12, 22] coincide with perfect model semantics for locally stratified programs.

Address correspondence to Roland N. Bol, Technische Universiteit Eindhoven, Faculteit Wiskunde & Informatica HG8.88, Postbus 513, NL 5600 MB Eindhoven, Netherlands.
Received May 1992.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$5.00

Secondly, we need an accurate description of the search space. It appears that for our purposes, the standard *SLS-trees* [20] do not present enough detail in the treatment of negative literals. Therefore, these SLS-trees are augmented with *justifications*, which explicitly show the construction of a subsidiary SLS-tree of $\leftarrow A$, when $\neg A$ is selected.

A third and major problem (not treated in [14, 24]) is the occurrence of floundering: when only substitutions are used as computed answers, a nonground 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 (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 is 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 nontypical loop check is given), but for "reasonable" loop checks, including the ones studied in [4] (briefly introduced in Appendix A), soundness is preserved.

2. DECLARATIVE AND PROCEDURAL SEMANTICS OF NEGATION

2.1. Motivation

Loops 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: least

Herbrand models and SLD-trees. However, in the presence of negation, the choice is much less clear.

In the most well-known approach, introduced in [9] and treated in [16], 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 [16, Theorem 16.5], 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* [19]. Furthermore, we restrict our attention to *locally stratified programs*; it is shown in [19] 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 [20], 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 SLS-resolution cannot be effectively implemented, but only approximated. (The same holds for SLDNF-resolution because one selection rule may result in an infinite failed SLDNF-tree, whereas another may find a finitely failed one.) However, as Przymusinski suggests [20], 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."

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 [16]. For two substitutions σ and τ, we write $\sigma \leq \tau$ when σ is more general than τ, and for two expressions E and F, we write $E \leq F$ when F is an instance of E. We then say that 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 $(H \leftarrow A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n) \in \text{ground}(P)$:

for $1 \leq i \leq m$, $\text{stratum}(A_i) \leq \text{stratum}(H)$ and

for $1 \leq i \leq n$, $\text{stratum}(B_i) < \text{stratum}(H)$.

Obviously, stratified programs [2] 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,
   $\text{stratum}(A) = \sup\{\text{stratum}(A_0) | A_0$ is a ground instance of A in $L_P\}$
   (where sup X denotes the supremum of X).
2. For a negative literal $\neg A$, not necessarily ground,
   $\text{stratum}(\neg A) = \text{stratum}(A) + 1$.
3. For a goal G,

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

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 nonground negative literal (cf. [9, 16]). We assume that such a selection is avoided whenever possible.

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

Following Przymusinski's presentation for stratified programs in [20], we now define for a given program P and goal G the SLS-tree of $P \cup \{G\}$, together with some related notions. The definition uses induction on $\text{stratum}(G)$.

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 $\text{stratum}(H) < \text{stratum}(G)$. We define the *SLS-tree T of $P \cup \{G\}$ 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 descendants are obtained as follows:

- if $H = \square$, then H has no descendants and is a success leaf.
- if **R** selects a nonground negative literal in H, then H has no descendants 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 $L = \neg A$ in H, then the SLS-tree T' of $P \cup \{\leftarrow A\}$ via **R** has already been defined.

  (Either some ground instance $B\gamma$ of an atom B in G depends negatively on A, therefore stratum(G) $\geq$ stratum(B) $\geq$ stratum($B\gamma$) $>$ stratum(A), or $\neg A$ 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.

Definition 2.5. Let T be an SLS-tree. 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* $P \cup \{G\}$) is an initial segment of a branch of an SLS-tree (of $P \cup \{G\}$). 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 $P \cup \{G\}$ 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. $G\sigma$ is called the *computed answer* of the derivation.

An SLS-derivation or -tree of $P \cup \{G\}$ 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 $G\sigma$.

2.3. *Soundness and Completeness*

We need the following soundness and completeness results, which strengthen the results of [7, 14, 18]. Not only does it present the set of computed answers as a "minimal estimate" for the set of correct answers, it also provides a "maximal estimate" for the set of correct answers: the set of potential answers. If floundering does not occur, then the two coincide, of course. But even in some cases with

floundering, the two coincide (cf. Example 2.7), thereby giving a complete description of the set of correct answers which was not achieved by previous completeness results.

Theorem 2.6 (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 [19].[1] *Let* **R** *be a safe selection rule and θ a substitution.*

1. Soundness of positive answers:
   if $G\theta$ is a computed answer for $P \cup \{G\}$, then $\forall((L_1 \wedge \cdots \wedge L_n)\theta)$ is true in M_P.
2. Soundness of negative answers:
   if $P \cup \{G\}$ has a failed SLS-tree, then $\neg\exists(L_1 \wedge \cdots \wedge L_n)$ is true in M_P.
3. Completeness of positive (potential) answers:
   if $\forall((L_1 \wedge \cdots \wedge L_n)\theta)$ is true in M_P, then there exists a potentially successful SLS-derivation of $P \cup \{G\}$ via **R** *giving a potential answer $G\sigma \leq G\theta$.*
4. Completeness of negative answers:
   if $\neg\exists(L_1 \wedge \cdots \wedge L_n)$ is true in M_P, then the SLS-tree for $P \cup \{G\}$ via **R** *is not successful.*

PROOF. 4.) follows immediately from 1.), and 2.) follows immediately from 3.). 1.) and 2.) are proved in [7, Theorem 5.3i)]. 3.) is proved in Appendix B. □

Theorem 2.6 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 [7] is needed here.

Example 2.7. Let

$$\begin{aligned} P = \{ & p(1) \leftarrow && \text{(C1)} \\ & p(y) \leftarrow p(y), \neg q(y) && \text{(C2)} \\ & q(1) \leftarrow \neg r(x) && \text{(C3)}\}. \end{aligned}$$

Using the leftmost selection rule yields the SLS-tree of $P \cup \{\leftarrow p(x)\}$ shown in Figure 1. Since the tree flounders, ordinary completeness results like the one in [7] cannot be used. However, a loop check might very well prune the goal $\leftarrow p(x), \neg q(x)$. Then the pruned tree does not flounder, so it is expected to be complete. Indeed, this completeness follows from Theorem 2.6 (as the only potential answer substitution occurring in the tree is $\{x/1\}$).

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 $P \cup \{G\}$ is searched. However, in the presence of negation, not only an SLS-tree of $P \cup \{G\}$ is searched, but also its side-trees, and the side-trees of its side-trees, etc. 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.

[1] But based on the canonical language of [15] in order to avoid the "universal query problem."

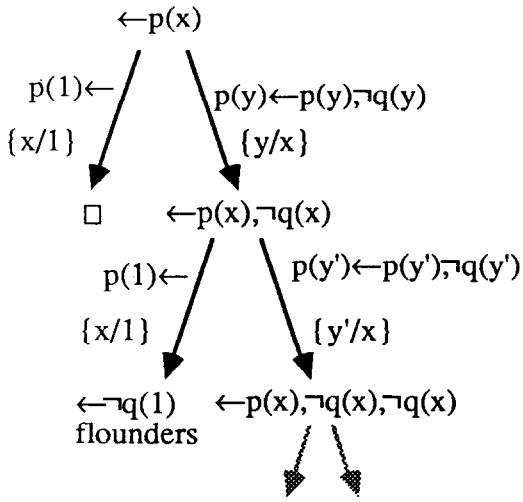

FIGURE 1. A strengthened completeness result is needed.

Definition 2.8 (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 $P \cup \{G\}$ via **R** consists of an SLS-tree T_{top} of $P \cup \{G\}$ via **R**, which is, for every goal H in T_{top} in which a ground negative literal $\neg A$ is selected, augmented with a justified SLS-tree T of $P \cup \{\leftarrow A\}$ 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 shows a justified SLS-tree.

In order to render potential answers as specific as possible, it is worthwhile to "postpone" floundering until all nonfloundering 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.

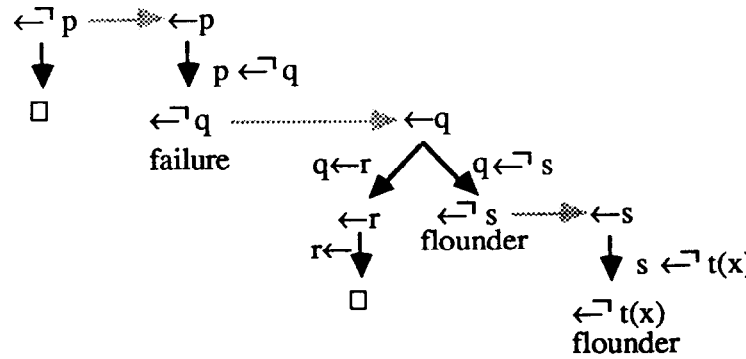

FIGURE 2. A justified SLS-tree.

Definition 2.9. A justified SLS-tree is *deeply safe* if for every flounder leaf $\leftarrow L_1, \ldots, L_n$ in it, every literal L_i $(1 \leq i \leq n)$ is a negative literal $\neg A_i$, and either A_i is nonground or every deeply safe justified SLS-tree of $P \cup \{\leftarrow A_i\}$ 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.

2.5. *Floundering*

In this paper, deeply safe trees are only used as a theoretical construct: Theorem 3.12 implies that the interpreter to which our loop checking mechanism is added can be allowed to use any safe selection rule. Nevertheless, it might prove profitable in practice to construct deeply safe trees, for this reduces the occurrence of floundering to a minimum.

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. (In the lemma, $|B|$ denotes the length of a finite SLS-derivation B.)

Lemma 2.10 (Independence of 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 for each potentially successful branch B of T_1, $|B| = |\varphi(B)|$ and the potential answers of B and $\varphi(B)$ are variants. Moreover, B is successful if and only if $\varphi(B)$ is.

PROOF. Remove all negative literals from T_1 that are never selected (since T_1 is deeply safe, precisely these literals remain in the flounder leaves). The resulting tree is successful; hence, the Switching Lemma (Lemma 3.3 in [14]) can be applied repeatedly. In this way, a successful tree can be obtained in which the selections take place in the same order as in T_2. Now adding the "floundering literals" in their place yields exactly T_2: because T_2 is deeply safe, the added literals are never selected before the flounder leaves.

Notice that induction on *stratum* is needed to show that whether a literal is a "floundering literal" is independent of the selection rule. □

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" is the goal a flounder leaf.

The final lemma of this section shows that the use of deeply safe selections indeed reduces floundering to a minimum, i.e., given a program and a goal, every computed answer that can be obtained is present (modulo variants) in every deeply safe tree. Conversely, if the deeply safe tree has a floundering branch, such a

branch is also present in every other tree (with a more general potential answer, indicating the possibility that the other tree flounders sooner).

Lemma 2.11. 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.

1. *For every computed answer in T_2, T_1 contains a variant of it.*
2. *For every potential answer in T_1, T_2 contains a more general potential answer.*

PROOF. For both claims, we need to consider only the top-level of the trees, as the justifications can be treated by induction on *stratum*.

1. Suppose that T_2 contains a successful branch B. As far as B is concerned (without its justifications), T_2 is deeply safe. In other words, B can be embedded in a deeply safe justified SLS-tree T_3 of $P \cup \{G\}$. Now apply Lemma 2.10 to T_1 and T_3.
2. Suppose that T_1 contains a potentially successful branch B. Consider a deeply safe justified SLS-tree T_3 of $P \cup \{G\}$ that follows the selections of T_2 as long as they are deeply safe. By Lemma 2.10, T_3 contains a potentially successful branch B' of which the potential answer is a variant of the answer of B. T_2 contains either B' or an initial segment of B' that flounders (on a goal in which the selection is not deeply safe). The potential answer of such an initial segment of B' is more general than the potential answer of B' itself. □

The approach to floundering we have sketched here tries to avoid floundering (by using deeply safe trees), and when floundering occurs, it tries to prove that the occurrence is harmless (i.e., that the returned potential answer is less general than some computed answer). In the remaining cases, the potential answer can be used, but no attempt is made to get more information from the floundering goal.

A proposal for trying to get more information is so-called constructive negation, which has been studied both in the setting of SLDNF-resolution [8] and SLS-resolution [21]. This method can provide sound (and in some cases complete) answers to floundering goals.

Both approaches are complementary. Because constructive negation is a rather complicated operation, it makes sense to try to limit its use to those cases where it is really needed. These cases are identified by our approach. In fact, as is shown by Example 4.12, it is almost unavoidable that the application of loop checking turns some successful SLS-trees into floundering ones, a behavior which becomes more acceptable when constructive negation is used.

On a technical level, one might expect a relation between Theorem 2.6(3) and completeness results for constructive negation. However, it appeared hard to point out such a relationship because completeness results for constructive negation necessarily apply to a limited class of programs only.

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 [1]. 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 [1]) 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 [14, 22, 24]; see Section 3.4.)

As in [1], we define for a program P:

- a node in an SLS-tree of $P \cup \{G\}$ (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 $P \cup \{G\}$ the nodes in $\{H | \text{the SLS-derivation from } G \text{ to } H \text{ is in } L(P)\}$. 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.

Initials$(L) = \{D \in L | L$ does not contain a proper initial subderivation of $D\}$. L is *subderivation free* if $L = \text{Initials}(L)$.

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

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 [1], (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 $P \cup \{G\}$ is *pruned by* L if $L(P)$ contains an 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. Modeling 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 2.6 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 $P \cup \{G\}$. 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 descen-

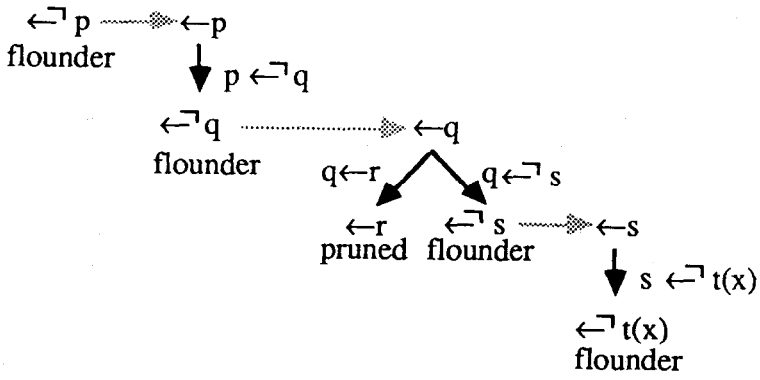

FIGURE 3. Pruning introduced floundering.

dants and is a founder 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 $\leftarrow r$ is applied to the SLS-tree in Figure 2, the tree depicted in Figure 3 is obtained. When the goal $\leftarrow s$ is also pruned, then the tree of Figure 4 is obtained.

3.3. Soundness and Completeness

In this section, a number of 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 2.6(3) involves

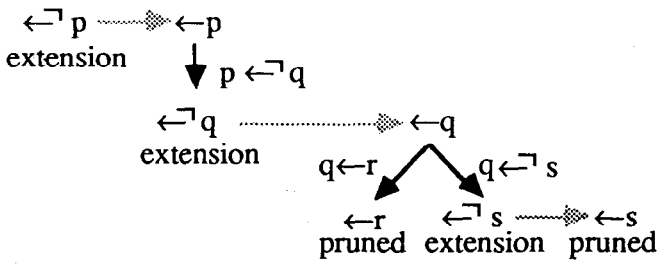

FIGURE 4. Pruning introduced an extension leaf.

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 "$\leftarrow p \Rightarrow_{p \leftarrow p, \neg r} \leftarrow p, \neg r$," where $\neg r$ is selected and its side-tree flounders.)

Definition 3.7 (Soundness). Let **R** be a safe selection rule and let L be a loop check.

1. L is *weakly sound* if for every program P and goal G, and potentially successful deeply safe justified SLS-tree T of $P \cup \{G\}$, $f_L^1(T_{\text{top}})$ is potentially successful.
2. L is *sound* if for every program P and goal G, and deeply safe justified SLS-tree T of $P \cup \{G\}$: if T contains a potentially successful branch giving a potential answer $G\sigma$, then $f_L^1(T_{\text{top}})$ contains a potentially successful branch giving a potential answer $G\sigma' \leq G\sigma$.

The following lemma is an immediate consequence of these definitions.

Lemma 3.8. Every sound loop check is weakly sound.

Moreover, if the initial goal 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 3.9 (Completeness). A loop check L is *complete w.r.t. a selection rule* **R** *for a class of programs* $\mathscr{C}$ if, for every program $P \in \mathscr{C}$ and goal G in L_P, every infinite SLS-derivation of $P \cup \{G\}$ 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 show that under the right conditions, an SLS-interpreter augmented with a loop check remains sound and complete (in the sense of Theorem 2.6). Due to the introduction of extension leaves, a pruned justified SLS-tree generally does *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 ensure the absence of extension leaves in soundness results.

As a first step, we do so for deeply safe justified SLS-trees by comparing their pruned and unpruned versions directly. The enumeration in this lemma links up with Theorem 2.6; its proof can be found in appendix B.

Lemma 3.10. Let P be a program and G a goal. Let **R** *be a safe selection rule and* θ *a substitution. Let T be a deeply safe justified SLS-tree of* $P \cup \{G\}$ *via* **R**. *Let L be a weakly sound loop check and let* $T_p = f_L^*(T)$.

- T_p *represents the search space for* $P \cup \{G\}$ *of a top-down SLS-interpreter using* **R**, *augmented with the loop check L (i.e.,* T_p *has no extension leaves).*

1. *If* $G\theta$ *is a computed answer in* T_p, *then* $G\theta$ *is a computed answer in T.*
2. *If* T_p *is failed, then T is failed.*
3. *If L is sound and T contains a potential answer* $G\theta$, *then* T_p *contains a potential answer* $G\sigma \leq G\theta$.
4. *If T is not successful, then* T_p *is not successful.*

Indeed, combining Lemma 3.10(•) and 1)–4) with 1)–4) of Theorem 2.6 gives the required soundness and completeness results for deeply safe trees. For lifting the requirement that the original tree is deeply safe, we need one more definition.

Definition 3.11. A loop check L is *selection-independent* if for every program P and for every $D \in L(P)$, $\{D' | D'$ differs from D only in the selection of the literal in its last goal$\} \subseteq L(P)$.

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 is a further resolution step attempted; to this end, a literal is selected. All loop checks in [4] (see Appendix A) are selection-independent.

The following theorem shows that it is not really necessary to use deeply safe selections. But, given a justified SLS-tree T constructed via an arbitrary safe selection rule, we cannot obtain the desired soundness and completeness results by comparing T with its pruned version $f_L^*(T)$ directly. Instead, we must construct an intermediate justified SLS-tree T' with the following properties:

1. The unpruned parts of T' and T are equal (except possibly for the selections made at pruned atoms; that is why the restriction to selection-independent loop checks is made).
2. The pruned part of T' is deeply safe.

The pruned part of T' is, of course, never constructed by the interpreter (it is only used for comparison reasons), so from a practical point of view, it is impossible to tell whether $f_L^*(T)$ or $f_L^*(T')$ has been constructed. The desired soundness and completeness results can now be obtained by comparing T' with $f_L^*(T')$.

Notice that the justified SLS-tree T and its relationship with T' are neither needed nor mentioned in the formulation of the theorem below. In Appendix B, a slightly stronger version of this theorem is formulated and proved, in which T reappears.

Theorem 3.12 (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 L be a weakly sound selection-independent loop check. Then there*

exists a justified SLS-tree T' of $P \cup \{G\}$ such that:

- *$T'_p = f_L^*(T')$ represents the search space for $P \cup \{G\}$ 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).*

1. *If $G\theta$ is a computed answer in T'_p then $G\theta$ is a computed answer in T'.*
2. *If T'_p is failed, then T' is failed.*
3. *If L is sound and T' contains a potential answer $G\theta$, then T'_p contains a potential answer $G\sigma \leq G\theta$.*
4. *If T' is not successful, then T'_p is not successful.*

Thus, combining Theorem 3.12(•) and 1)–4) with 1)–4) of Theorem 2.6 (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 3.13. 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 $P \cup \{G\}$ via* **R**, *and let $T_p = f_L^*(T)$.*

1. *If L is complete w.r.t.* **R** *for a class of programs $\mathscr{C}$ containing P, then T_p is finite.*
2. *If a flounder leaf occurs in T_p, then a flounder leaf occurs in T.*

PROOF.

1. Follows immediately from Definition 3.9.
2. Suppose that G is a flounder leaf in T_p, so a negative literal is selected in G. If this negative literal is not ground, then G itself is a flounder leaf in T. Otherwise, let T' denote the justification of G in T, and let $T'_p = f_L^*(T')$. T'_p must be floundered. By induction (on *stratum*), a flounder leaf occurs in T', and hence in T. (Note: this does not imply that T founders!) □

Applying Theorem 3.13(1) on the tree T as constructed in Theorem 3.12 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 3.12 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 3.13(2) indicates that the pruned tree can only flounder if somewhere in the original tree (but not necessarily at the top level) floundering occurs. Example 4.11 shows that a stronger result can hardly be expected: if the tree in Figure 6 is the side-tree of the goal $\leftarrow \neg p$, then most of the loop checks applied there turn $\leftarrow \neg p$ from a failure leaf into a flounder leaf.

3.4. Related Work: Tabulation

The approach in [4] for avoiding infinite derivations for positive programs and its extension to locally stratified programs presented here deliberately stays as close to SLD- (SLS-) resolution as possible: the decision to prune a derivation is based

solely on that derivation. This has the advantage that most of the results obtained in logic programming remain trivially valid, and that the required modifications to the interpreter for implementing loop checking might be relatively small. At the same time, however, this approach excludes more powerful (albeit more complex) techniques.

One such an approach is known under different names, such as memoing or memoization, tabulation [25], and lemma resolution [26]. It can also be found in [23, Algorithms 3.7 and 3.8].

These techniques are essentially the same, although the proposed implementations can differ in details. The main idea, which originates from functional programming, is to store intermediate results, and to look them up when they would normally be recomputed. Apart from the effect on termination, this can improve the efficiency of a program considerably.

In the case of logic programming, an intermediate result consists of the computed answers for an atom. Therefore, it is necessary to use a local selection rule [26]: once an atom is selected, all answers for that *atom* are requested; restrictions imposed by another atom on the answers for the full *goal* are found when (an instance of) this other atom is selected. The part of a derivation between the selection of an atom and the point where the atom is completely resolved can be considered a "local proof" for (an instance of) that atom. Its result is added as a lemma. In [25], only the leftmost selection rule is considered, whereas [26] allows any local selection rule.

The order in which the nodes of an SLD-tree are constructed (visited) is described by the search rule. For ordinary SLD-resolution, a "good" search rule is required for finding all answers, but the shape of the tree itself does not depend on the search rule. When tabulation is used, this is no longer true: only when an atom is encountered for the first time is its SLD-tree constructed; later calls to that atom are resolved by a look-up in the table (lemma resolution).

An important advantage of SLD–AL resolution over SLD-resolution is that for function-free programs, SLD–AL trees are always finite (hence, any search rule is "good"). But in the presence of function symbols, SLD–AL trees can be infinitely branching, in addition to the possibility of having infinite branches.

In [14] (based on [26]) and [24] (based on [25]), the tabulation technique is generalized to stratified programs with SLS-resolution. In contrast to this paper, locally stratified programs are not considered, and the issue of floundering is avoided. Very recently, research has begun on extending and implementing tabulation techniques for general logic programs, starting from a generalization of SLS-resolution that computes the well-founded semantics [22].

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 positive loop checks we are interested in are all selection-independent (Definition 3.11 also does 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,

$$O_L = \lambda P.\ \text{Initials}(\{D \mid D \text{ is an SLS-derivation and } D^+ \in L(P^+)\}).$$

The following lemmas 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.

PROOF. Immediately by the definitions. □

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 [1]) is again (weakly) sound.

Counterexample 4.5. Let

$$\begin{array}{lll} P = \{p & \leftarrow q(1), q(2) & \text{(C1)} \\ \quad q(x) & \leftarrow \neg\, r(x) & \text{(C2)} \\ \quad q(2) & \leftarrow q(1) & \text{(C3)} \\ \quad r(2) & \leftarrow & \text{(C4)}\} \end{array}$$

and let $G = \leftarrow p$.

P is (locally) stratified, and Figure 5 shows an SLS-tree T of $P \cup \{G\}$ via the leftmost selection rule (a failure leaf is marked by a box around it).

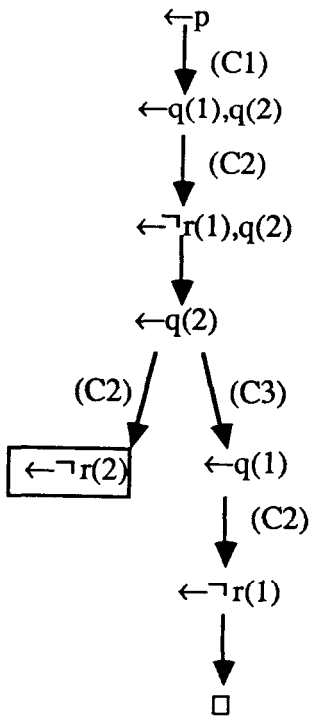

FIGURE 5. Unsound pruning.

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 $\leftarrow q(1)$ in D^+: it is visible in the second step of D^+ that the clause $q(x) \leftarrow$ is present in P^+; this clause allows for a shorter way to refute $\leftarrow q(2)$ than via (C3) and $\leftarrow q(1)$.

Unfortunately, this shortcut fails in the SLS-tree because it introduces the literal $\neg r(2)$ instead of $\neg r(1)$, and $\neg r(2)$ 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).

Although the loop check used in the counterexample formally satisfies the definitions, it is highly nontypical. We shall now show that more usual (weakly) sound positive loop checks, notably the ones defined in [4], derive again (weakly) sound one-level loop checks. To this end, we introduce a soundness condition, which is very similar (also in its proof) to [4, Lemma 4.5].

Lemma 4.6 (Soundness condition). Let L be a one-level loop check. If, for every program P, goal G_0 and potentially successful branch $D = (G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{k-1} \Rightarrow_{\theta_k} G_k \Rightarrow \cdots \Rightarrow_{\theta_m} H)$ $(0 < k \leq m)$ of a deeply safe justified SLS-tree T of $P \cup \{G_0\}$:

[G_k is pruned by L] implies

[for some goal G_i $(0 \leq i < k)$ in D and for some $n < m - i$, there exists a potentially successful branch $G_i \Rightarrow_{\sigma_1} \cdots \Rightarrow_{\sigma_n} H'$ of a deeply safe justified SLS-tree of $P \cup \{G_i\}$],

then L is weakly sound.

Moreover, if also $G_0\theta_1 \cdots \theta_i\sigma_1 \cdots \sigma_n \leq G_0\theta_1 \cdots \theta_k\theta_{k+1} \cdots \theta_m$ *is implied, then L is sound.*

PROOF. First we focus on the weakly sound case. Let P be a program, G_0 a goal, and T a deeply safe justified SLS-tree of $P \cup \{G_0\}$. Suppose that T_{top} contains a potentially successful branch $D = (G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{i-1} \Rightarrow_{\theta_i} G_i \Rightarrow \cdots \Rightarrow G_{k-1} \Rightarrow_{\theta_k} G_k \Rightarrow \cdots \Rightarrow_{\theta_m} H)$ that is pruned by L at G_k. We use here induction on m, i.e., we assume that for every successful branch B in T_{top} shorter than D, $f_L^1(T_{\text{top}})$ contains either B or a potentially successful branch shorter than B.

We prove that $f_L^1(T_{\text{top}})$ contains a potentially successful branch D' that is shorter than D. By assumption, a potentially successful SLS-derivation $D_1 = (G_i \Rightarrow_{\sigma_1} \cdots \Rightarrow_{\sigma_n} H')$ of $P \cup \{G_i\}$ exists. Adding (a properly renamed version of) D_1 to the initial part of D gives the derivation $D_2 = (G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow \cdots \Rightarrow G_{i-1} \Rightarrow_{\theta_i} G_i \Rightarrow_{\tau_1} \cdots \Rightarrow_{\tau_n} H')$. By the independence of the selection rule (Lemma 2.10), T_{top} contains a branch D_3 such that $|D_3| = |D_2|$ and the potential answers of D_3 and D_2 are variants. Since D_3 is shorter than D ($|D_3| = i + n + 1 < i + (m - i) + 1 = m + 1 = |D|$), by the induction hypothesis $f_L^1(T_{\text{top}})$ contains either $D' = D_3$ or a potentially successful branch D' shorter than D_3, which proves the claim.

For the sound case, it remains to prove that $G_0\sigma' \leq G_0\theta_1 \cdots \theta_m$, where σ' is the potential answer substitution of D'. First we strengthen the induction hypothesis: for every potentially successful branch B in T_{top} shorter than D giving a potential answer $G\sigma$, $f_L^1(T_{\text{top}})$ contains either B or a potentially successful branch shorter than B, giving a potential answer $G_0\sigma' \leq G_0\sigma$.

Then either since $D' = D_3$ or by the new induction hypothesis, and since the potential answers of D_3 and D_2 are variants, $G_0\sigma' \leq G_0\theta_1 \cdots \theta_i\tau_1 \cdots \tau_n \leq G_0\theta_1 \cdots \theta_i\sigma_1 \cdots \sigma_n \leq G_0\theta_1 \cdots \theta_m$. □

Indeed, the one-level loop checks derived from the positive loop checks defined in [4] (and informally described in Appendix A) satisfy the above soundness condition. So we can prove that they are (weakly) sound.

Theorem 4.7 (Soundness of conversion).

1) *The one-level loop checks derived from the equality, subsumption, and context checks based on goals are weakly sound.*
2) *The one-level loop checks derived from the equality, subsumption, and context checks based on resultants are sound.*

PROOF (Sketch) . The proofs of Theorems 4.6, 5.7, and 6.6 in [4], in which it is shown that the positive loop checks mentioned satisfy the soundness condition (for the positive case), are straightforwardly generalized to the present case. Every successful SLD-derivation must be replaced by a potentially successful branch of a deeply safe justified SLS-tree. The mgu Lemma, Lifting Lemma, and Independence of the Selection Rule of [16] (used in the positive case) must be replaced by Lemmas B.1, B.2, and 2.10, respectively. □

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 4.8. 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 4.9 (Completeness of conversion). If L is complete w.r.t. a selection rule **R** *for a class of programs $\mathscr{C}$, then O_L is complete w.r.t. any safe extension of* **R** *for the class of programs $\{P \mid P^+ \in \mathscr{C}$ and $L_P = L_{P^+}\}$.*

PROOF. Let P be a program, G a goal in L_P (both possibly containing negative literals) and **R** a selection rule for SLD-derivations. Let **R'** be an arbitrary safe extension of **R**. Let D be an infinite SLS-derivation of $P \cup \{G\}$ via **R'**. Then D^+ is an infinite SLD-derivation of $P^+ \cup \{G^+\}$ via **R**. Let L be a positive loop check that is complete w.r.t. **R** for (a class of programs containing) P^+: for every goal H in L_{P^+}, every infinite SLD-derivation of $P^+ \cup \{H\}$ is pruned by $L(P^+)$. Since G^+ is a goal in $L_P = L_{P^+}$, D^+ is pruned by $L(P^+)$. Hence, by Lemma 4.4, D is pruned by $O_L(P)$. □

Notice that the requirement $L_P = L_{P^+}$ is just a technicality which can be met easily by adding some nonrelevant clauses to P. For example, the result presented in Theorem A.3(1) is transferred to one-level loop checks as follows.

Corollary 4.10. 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 4.7 and the Completeness of Conversion Theorem 4.9 allow the immediate conversion of all positive loop checks described in [4] and their soundness and completeness results (see Appendix A) to one-level loop checks. The following example presents the application of several one-level loop checks derived from these positive loop checks.

Example 4.11. Let

$$\begin{array}{lll} P = \{p & \leftarrow q(x), \neg s(x) & \text{(C1)} \\ \quad q(y) & \leftarrow r(y), q(y) & \text{(C2)} \\ \quad q(y) & \leftarrow & \text{(C3)} \\ \quad r(1) & \leftarrow & \text{(C4)}\} \end{array}$$

and let $G = \leftarrow p$. In Figure 6, an SLS-tree T of $P \cup \{G\}$ 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, the pruned tree is finite. This was to be expected, as P^+ is a restricted program (see Definition A.2). Furthermore, each loop check retains potential success in the pruned tree (they even retain the most general potential answer, since G is ground). However, it appears that only the

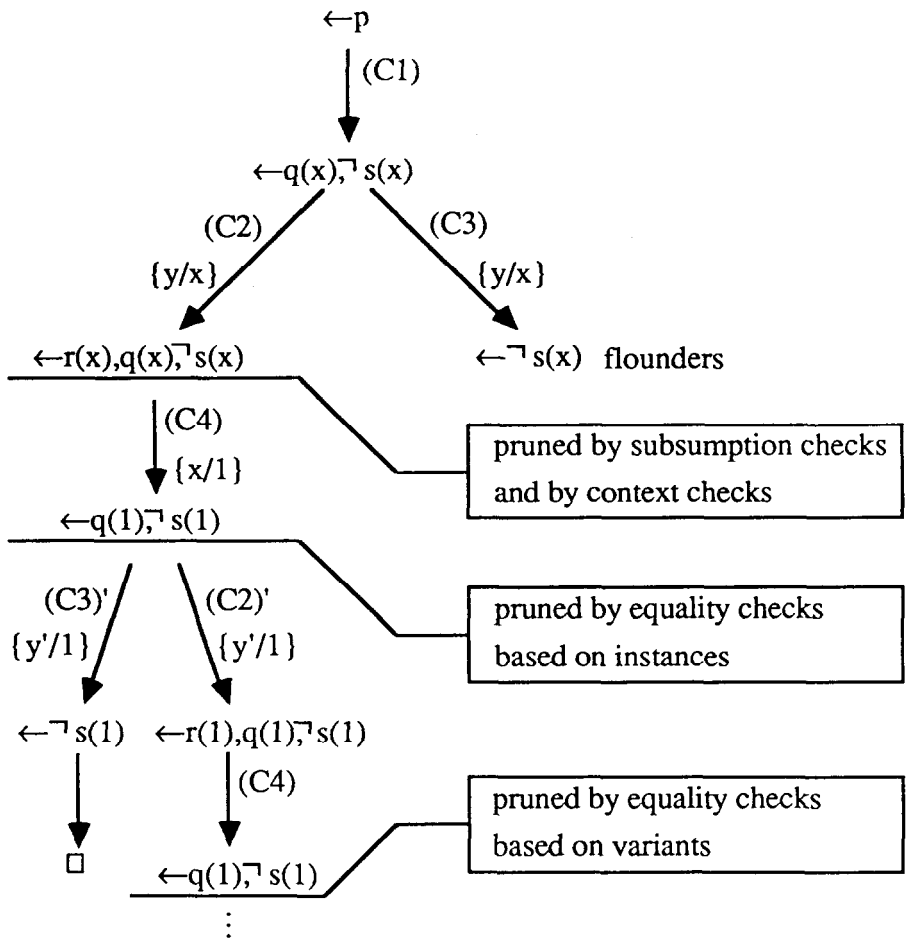

FIGURE 6. An SLS-tree pruned by various loop checks.

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.

But even the equality checks based on variants do not always retain at least one successful branch, as is shown in the following example. As we remarked in Section 2.5, the use of constructive negation can make this behavior more acceptable.

Example 4.12. Consider the program $\{p \leftarrow q(x), \neg s(x).\ q(1) \leftarrow q(y).\ q(y) \leftarrow .\}$ and the goal $\leftarrow p$ (see Figure 7). In order to avoid floundering of $\leftarrow \neg s(x)$, the clause $q(1) \leftarrow q(y)$ must instantiate it to $\neg s(1)$. But the resulting refutation is pruned by all one-level loop checks derived from equality checks.

APPENDIX A. POSITIVE LOOP CHECKS

Here we recall the three groups of simple loop checks that are introduced in [4], together with their respective soundness and completeness results.

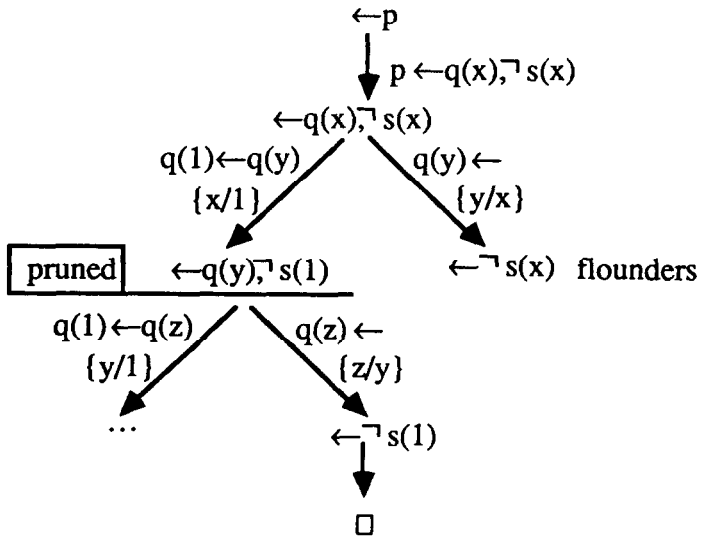

FIGURE 7. Only floundering remains.

A.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 do not have 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 [3]. Their loop checking condition is more complicated: "For some atom A in G_i, $A\theta_{i+1} \cdots \theta_j$ is selected in G_j to be resolved. As the (direct or indirect) result of resolving $A\theta_{i+1} \cdots \theta_j$, 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, $x\theta_{i+1} \cdots \theta_k = x\tau$."

For all of these weakly sound loop checks, a sound counterpart is obtained by adding the condition "$G_0\theta_1 \cdots \theta_k = G_0\theta_1 \cdots \theta_i\tau$" to the loop checking criterion. For reasons explained in [4], 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 [4].

Theorem A.1.

1. *The equality, subsumption, and context checks based on goals are weakly sound.*
2. *The equality, subsumption, and context checks based on resultants are sound.*

A.2. Completeness Results

Due to the undecidability of the halting problem, a weakly sound loop check cannot be complete for all programs. In [4] 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 A.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 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 $A_1, \ldots, A_n$ also occurs 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 A.3.

1. *All equality, subsumption, and context checks are complete w.r.t. the leftmost selection rule for function-free restricted programs.*
2. *All subsumption and context checks are complete for function-free nvi programs and for function-free svo programs.*

APPENDIX B. PROOFS

B.1. Soundness and Completeness of SLS-Resolution

Theorem 2.6 (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 [19]. *Let* **R** *be a safe selection rule and θ a substitution.*

1. *If $G\theta$ is a computed answer for $P \cup \{G\}$, then $\forall((L_1 \wedge \cdots \wedge L_n)\theta)$ is true in M_P.*
2. *If $P \cup \{G\}$ has a failed SLS-tree, then $\neg\exists(L_1 \wedge \cdots \wedge L_n)$ is true in M_P.*
3. *If $\forall((L_1 \wedge \cdots \wedge L_n)\theta)$ is true in M_P, then there exists a potentially successful SLS-derivation of $P \cup \{G\}$ via* **R** *giving a potential answer $G\sigma \leq G\theta$.*
4. *If $\neg\exists(L_1 \wedge \cdots \wedge L_n)$ is true in M_P, then the SLS-tree for $P \cup \{G\}$ via* **R** *is not successful.*

PROOF. 4. follows immediately from 1. and 2. follows immediately from 3. 1. and 2. are proved in [7, Theorem 5.3(1)]. So it remains to prove 3.

We introduce the following terminology.

An SLS-derivation is *unrestricted* if instead of mgu's, arbitrary unifiers are used.

An (unrestricted) SLS-derivation is *grounded* if every goal in it is ground.

An *oracle* SLS-derivation differs from the standard SLS-derivation in the treatment of selected ground negative literals: such a literal $\neg A$ is removed if $A \notin M_P$, and the derivation fails if $A \in M_P$ (and floundering does not occur).

From this, it follows that a grounded (oracle) SLS-derivation never flounders. Now assume that $\forall((L_1 \wedge \cdots \wedge L_n)\theta)$ is true in M_P. It can then be shown (similarly to other completeness proofs, e.g., in [7, 14]) that there exists a grounded oracle SLS-refutation of $P \cup \{G\theta\gamma\}$ via **R**, where $\gamma = \{x_1/a_1, \ldots, x_m/a_m\}$ binds all variables $x_1, \ldots, x_m$ in $G\theta$ to new constants $a_1, \ldots, a_m$. (More precisely, these constants are added to L_P. Notice that P remains locally stratified under this extension of the Herbrand Universe. More importantly, the oracle in the oracle SLS-refutation uses the model M_P w.r.t. the extended Herbrand Universe. The use of the oracle replaces the more usual induction on *stratum* at this point.)

In this grounded oracle SLS-refutation, we can textually replace the constants $a_1, \ldots, a_m$ by $x_1, \ldots, x_m$ again. Thus, we obtain a "derivation" of $P \cup \{G\theta\}$ of which the unifiers do not act on the variables of $G\theta$. However, it is possible that some a_i is replaced by x_i in a selected negative literal, causing this "derivation" to flounder, in which case the rest of the derivation must be discarded. Thus, we obtain a potentially successful unrestricted oracle SLS-derivation of $P \cup \{G\}$ of which the unifiers do not act on the variables of $G\theta$.

Now we supply side-trees for the remaining successful oracle steps (in which a ground negative literal $\neg A$ is selected and removed). As in such a case $A \notin M_P$, from 4) it follows that the constructed side-tree, an SLS-tree of $P \cup \{\leftarrow A\}$ via **R**, is not successful. If it is failed, then we have found the desired side-tree. If it flounders, then again our derivation flounders at this point, and the rest of it is discarded. So we obtain a potentially successful unrestricted SLS-derivation of $P \cup \{G\theta\}$, of which the unifiers do not act on the variables of $G\theta$.

Now we need the following generalizations of the well-known mgu Lemma and Lifting Lemma (see, e.g., [7, Lemmas 5.2 and 5.3]).

Lemma B.1 (mgu Lemma). Let P be a program and G a goal. Suppose that $P \cup \{G\}$ *has a potentially successful unrestricted SLS-derivation using the unifiers* $\theta_1, \ldots, \theta_n$. *Then there exists a potentially successful SLS-derivation of* $P \cup \{G\}$ *using the mgu's* $\theta'_1, \ldots, \theta'_m$, *such that* $G\theta'_1, \ldots, \theta'_m \leq G\theta_1, \ldots, \theta_n$ *and* $m \leq n$.

PROOF. First the construction of the proof of the original mgu Lemma can be applied, disregarding floundering. The resulting "derivation" uses the mgu's $\theta'_1, \ldots, \theta'_n$, and $G\theta'_1, \ldots, \theta'_n \leq G\theta_1, \ldots, \theta_n$. It is a valid SLS-derivation up to the first selection of a nonground negative literal. At this goal (G_m if it exists; if not, then $m = n$), floundering occurs and the rest of the "derivation" is discarded. The result is a potentially successful SLS-derivation with a potential answer $G\theta'_1, \ldots, \theta'_m \leq G\theta'_1, \ldots, \theta'_n \leq G\theta_1, \ldots, \theta_n$; and $m \leq n$. □

Lemma B.2 (Lifting Lemma). Let P be a program, G a goal, and θ a substitution. Suppose that $P \cup \{G\theta\}$ has a potentially successful SLS-derivation using the mgu's $\theta_1, \ldots, \theta_n$. Then there exists a potentially successful SLS-derivation of $P \cup \{G\}$ using the mgu's $\theta'_1, \ldots, \theta'_m$, such that $G\theta'_1, \ldots, \theta'_m \leq G\theta\theta_1, \ldots, \theta_n$ and $m \leq n$.

PROOF. First the construction of the proof of the original Lifting Lemma can be applied, disregarding floundering. The resulting "derivation" uses the mgu's $\theta'_1, \ldots, \theta'_n$, and $G\theta'_1, \ldots, \theta'_n \leq G\theta\theta_1, \ldots, \theta_n$. It is a valid SLS-derivation up to the first selection of a nonground negative literal. At this goal (G_m if it exists; if not, then $m = n$), floundering occurs and the rest of the "derivation" is discarded. The result is a potentially successful SLS-derivation with a potential answer $G\theta'_1, \ldots, \theta'_m \leq G\theta'_1, \ldots, \theta'_n \leq G\theta\theta_1, \ldots, \theta_n$; and $m \leq n$. □

Applying these lemmas to the potentially successful unrestricted SLS-derivation of $P \cup \{G\theta\}$ proves the existence of a potentially successful SLS-derivation of $P \cup \{G\}$, giving a potential answer $G\sigma \leq G\theta$. □

B.2. Soundness and Completeness of SLS-Resolution with Loop Checking

Lemma 3.10. Let P be a program and G a goal. Let **R** *be a safe selection rule and θ a substitution. Let T be a deeply safe justified SLS-tree of $P \cup \{G\}$ via* **R**. *Let L be a weakly sound loop check and let $T_p = f_L^*(T)$.*

- *T_p represents the search space for $P \cup \{G\}$ of a top-down SLS-interpreter using* **R**, *augmented with the loop check L (i.e., T_P has no extension leaves).*

1. *If $G\theta$ is a computed answer in T_p, then $G\theta$ is a computed answer in T.*
2. *If T_p is failed, then T is failed.*
3. *If L is sound and T contains a potential answer $G\theta$, then T_p contains a potential answer $G\sigma \leq G\theta$.*
4. *If T is not successful, then T_p is not successful.*

PROOF

- We prove that T_p does not contain extension leaves. Suppose (in order to obtain a contradiction) that G is an extension leaf in T_p. Then a ground negative literal is selected in G. Let T' be the justification of G in T, and let $T'_p = f_L^*(T')$ be the justification of G in T_p. By induction (on *stratum*), we may assume that T'_p has no extension leaves. So the only case left is that G is a leaf in T, and T'_p is failed. Obviously, G is not a success leaf. So G is a failure leaf or flounder leaf in T. Hence, T' is potentially successful. Since L is weakly sound and T' is deeply safe, we may conclude inductively from 2) that T'_p is potentially successful. Contradiction.

1. and 4.) T_p is a subtree of T.
2. Suppose (in order to obtain a contradiction) that T_p is failed, whereas T is potentially successful. Consider a potentially successful branch B in T. All justifications of B are either failed or floundered. Inductively, by 4.), the pruned justifications are also failed or floundered. Thus, T_p can only be failed if B itself is pruned by L. This holds for every potentially successful branch in T; thus, $f_L^1(T_{\text{top}})$ is failed. However, since L is weakly sound and T

is deeply safe and potentially successful, by Definition 3.7(1), $f_L^1(T_{top})$ must be potentially successful. Contradiction.

3. As 2.), considering a branch only (potentially) successful if its potential answer is more general than $G\theta$. (Notice that if a failed justification of B in T is replaced by a floundering pruned justification in T_p, the potential answer of the remaining part of B in T_p is more general than the potential answer of B.) In this case, Definition 3.7(2) must be used. □

It appears useful for proving Theorem 3.12 to strengthen it a little by making the original tree T reappear in the formulation, and partly stating the relationship between T and T': clauses 5.) and 6.).

Theorem 3.12 (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 $P \cup \{G\}$ via* **R**. *Let L be a weakly sound selection-independent loop check. Then there exists a justified SLS-tree T' of $P \cup \{G\}$ such that:*

- *$T_p' = f_L^*(T')$ represents the search space for $P \cup \{G\}$ 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).*

1. *If $G\theta$ is a computed answer in T_p', then $G\theta$ is a computed answer in T'.*
2. *If T_p' is failed, then T' is failed.*
3. *If L is sound and T' contains a potential answer $G\theta$, then T_p' contains a potential answer $G\sigma \leq G\theta$.*
4. *If T' is not successful, then T_p' is not successful.*
5. *If T is successful, then T' is successful.*
6. *If T is failed, then T' is failed.*

PROOF. First we give a construction of T'.

By induction on *stratum*, we may assume that for every justification J in T, an SLS-tree J' exists that satisfies the above specifications. By replacing in T every justification J by its corresponding J', we obtain a tree T''. If a floundering justification J of a leaf H is replaced by a failed justification J', then H must obtain a descendant in T'' and T'' is expanded beyond H. This expansion takes place via **R**, except that the justifications in the expansion are still the ones inductively derived from the justifications via **R**. By 5.) and 6.), this replacement of justifications cannot give rise to other problems.

For every justification J' in T'', it follows from •) that $f_L^*(J')$ has no extension leaves. Moreover, it follows that $T_p'' = f_L^*(T'')$ has no extension leaves. (Suppose that H is such an extension leaf; then the justification J' of H in T'' must be potentially successful, whereas $f_L^*(J')$ is failed. This contradicts 2.), applied inductively on J').

We obtain the tree T' by expanding T_p'' beyond its pruned leaves, where at those pruned leaves and beyond, selections are made in a deeply safe way (thus, not necessarily via **R**). Notice that differences between T'' and T' do not occur before the selection in a goal where T'' is pruned, so by the assumption that L is selection-independent, it follows that $T_p' = f_L^*(T') = T_p''$ (except possibly in selections in pruned leaves). Now we prove our claims.

- For the justifications in T_p', this is true by induction. As was remarked, the top level of $T_p' = T_p''$ contains no extension leaves. Finally, the top level of T'' (and T_p'') follows **R**, so T_p' does (except possibly in pruned leaves).

1. and 4.) T_p' is a subtree of T'.
2. Suppose that T_p' is failed. Then T_p'' is failed, so apparently no floundering in the justifications of T_p'' reaches its top level. So we may "pretend" that T_p'' is deeply safe, apart from selections in its pruned leaves (i.e., using Lemma 2.11, we could replace every justification of T_p'' by a deeply safe one, without changing its top level). T' is an expansion of T_p'' that is deeply safe in and beyond the pruned leaves of T_p''. Thus, in the same way, we may "pretend" that T is deeply safe. Then by Lemma 3.10(2), T' is failed.
3. First consider the tree T_{ds}, which is obtained from T' by expanding T' in a deeply safe way beyond every flounder leaf that is not deeply safe (either by making another selection or by replacing the justification). Consider a potentially successful branch B in T' that is pruned in T_p'. As B is pruned in T_p'', the tail (the part that is pruned out) of B in T' is already constructed in a deeply safe way. Therefore, B occurs in T_{ds} unexpanded (w.r.t. T').

   By its construction, we may again "pretend" that T_{ds} is deeply safe. Thus, if B yields a potential answer $G\theta$, then, by Lemma 3.10(3) and assuming that L is sound, $f_L^*(T_{ds})$ yields a potential answer $G\sigma' \leq G\theta$. The branch B' giving this answer $G\sigma'$ is either fully present in T' ($\sigma = \sigma'$) or an initial fragment of it is present which flounders (giving a potential answer $G\sigma \leq G\sigma'$). B' cannot be pruned in T_p' because a goal pruned in T_p' is also pruned in $f_L^*(T_{ds})$ (as T_{ds} is an expansion of T', and L is selection-independent).
5. Consider a successful branch B in T. All of its justifications are failed. So from 6), it follows inductively that B is still present in T''. If B is not pruned in T_p'', then it is present in T'. If B is pruned in T_p'', then in T' it is extended beyond the pruned goal in a deeply safe way. By Lemma 2.11(1), this extension is successful.
6. If T is failed, then T has no floundering justifications. So inductively, by 5) and 6), the top levels of T and T'' are identical. T_p'' may contain pruned leaves, but by Lemma 2.11(2), expanding them again in a deeply safe way can only lead to failure again. □

This paper was written at the CWI, Amsterdam, The Netherlands. I thank Prof. K. R. Apt for his assistance during the writing of this paper. Partial support by Esprit BRA-project 3020 Integration is gratefully acknowledged.

REFERENCES

1. Apt, K. R., Bol, R. N., and Klop, J. W., On the Safe Termination of PROLOG Programs, in: G. Levi and M. Martelli (eds.), *Proceedings of the 6th International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1989, pp. 353–368.
2. Apt, K. R., Blair, H., and Walker, A., Towards a Theory of Declarative Knowledge, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1987, pp. 89–148.

3. Besnard, Ph., On Infinite Loops in Logic Programming, Internal Report 488, IRISA, Rennes, France, 1989.
4. Bol, R. N., Apt, K. R., and Klop, J. W., An Analysis of Loop Checking Mechanisms for Logic Programs, *Theoretical Computer Science* 86:35–79 (1991).
5. R. N. Bol, Loop Checking in Logic Programming, Ph.D. dissertation, CWI, Amsterdam, 1991.
6. Brough, D. R. and Walker, A., Some Practical Properties of Logic Programming Interpreters, in: ICOT (eds.), *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984, pp. 149–156.
7. Cavedon, L., Continuity, Consistency, and Completeness Properties for Logic Programs, in: G. Levi and M. Martelli (eds.), *Proceedings of the 6th International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1989, pp. 571–584.
8. Chan, D., Constructive Negation based on the Completed Database, in: R. Kowalski and K. Bowen (eds.), *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1988, pp. 111–125.
9. Clark, K. L., Negation as Failure, in: H. Gallaire and J. Minker (eds.), *Logic and Data Bases*, Plenum, New York, 1978, pp. 293–322.
10. Covington, M. A., Eliminating Unwanted Loops in PROLOG, *SIGPLAN Notices* 20(1):20–26 (1985).
11. Van Gelder, A., Efficient Loop Detection in PROLOG using the Tortoise-and-Hare Technique, *J. Logic Programming* 4:23–31 (1987).
12. Van Gelder, A., Ross, K., and Schlipf, J. S., Unfounded Sets and Well-Founded Semantics for General Logic Programs, in: *Proceedings of the 7th Symposium on Principles of Databases*, ACM SIGACT-SIGMOD, 1988, pp. 221–230.
13. Gelfond, M. and Lifschitz, V., The Stable Model Semantics for Logic Programming, in: R. Kowalski and K. Bowen (eds.), *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1988, pp. 1070–1080.
14. Kemp, D. B. and Topor, R. W., Completeness of a Top-Down Query Evaluation Procedure for Stratified Databases, in: R. Kowalski and K. Bowen (eds.), *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1988, pp. 178–194.
15. Kunen, K., Negation in Logic Programming, *J. Logic Programming* 4:289–308 (1987).
16. Lloyd, J. W., *Foundations of Logic Programming*, 2nd edition, Springer-Verlag, Berlin, 1987.
17. Poole, D. and Goebel, R., On Eliminating Loops in PROLOG, *SIGPLAN Notices* 20(8):38–40 (1985).
18. Przymusinska, H. and Przymusinski, T. C., Weakly Perfect Model Semantics for Logic Programs, in: R. A. Kowalski and K. A. Bowen (eds.), *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1988, pp. 1106–1120.
19. Przymusinski, T. C., On the Declarative Semantics of Deductive Databases and Logic Programs, in: J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 1987, pp. 193–216.
20. Przymusinski, T. C., On the Declarative and Procedural Semantics of Logic Programs, *J. Automated Reasoning* 5:167–205 (1989).
21. Przymusinski, T. C., On Constructive Negation in Logic Programming, in: E. L. Lusk and R. A. Overbeek (eds.), *Proceedings of the North American Conference on Logic Programming*, MIT Press, Cambridge, MA, 1989.
22. T. C. Przymusinski and D. S. Warren, Well Founded Semantics: Theory and Implementation, in preparation.
23. Smith, D. E., Genesereth, M. R., and Ginsberg, M. L., Controlling Recursive Inference, *Artificial Intelligence* 30:343–389 (1986).

24. Seki, H. and Itoh, H., A Query Evaluation Method for Stratified Programs under the Extended CWA, in: R. A. Kowalski and K. A. Bowen (eds.), *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1988, pp. 195–211.
25. Tamaki, H. and Sato, T., OLD Resolution with Tabulation, in: G. Goos and J. Hartmanis (eds.), *Proceedings of the 3rd International Conference on Logic Programming*, LNCS 225, Springer-Verlag, Berlin, 1986, pp. 84–98.
26. Vieille, L., Recursive Query Processing: The Power of Logic, *Theoretical Computer Science* 69:1–53 (1989).