A New Definition of SLDNF-resolution

Krzysztof R. Apt
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
and
Faculty of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Kees Doets
Faculty of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Abstract

We propose a new, "top-down" definition of SLDNF-resolution which retains the spirit of the original definition but avoids the difficulties noted in the literature. We compare it with the "bottom-up" definition of Kunen [Kun89].

1 The problem

The notion of SLD-resolution of Kowalski [Kow74] allows us to resolve only positive literals. As a result it is not adequate to compute with general programs. Clark [Cla79] proposed to incorporate the *negation as finite failure* rule. This leads to an extension of SLD-resolution called *SLDNF-resolution*. The intuition behind it is quite simple: for a ground atom A,

$\neg A$ succeeds iff A finitely fails,
$\neg A$ finitely fails iff A succeeds.

(The restriction to ground atoms was originally introduced to ensure soundness of SLDNF-resolution.) However, this intuition is difficult to formalize. For example, consider the general program $P = \{A \leftarrow A\}$. The query $\neg A$ neither succeeds nor finitely fails, since the query A neither succeeds nor finitely fails. So it is not clear whether there is a resolvent.

The problem is that success and finite failure are not the only possible outcomes of an evaluation: also an unsuccessful tree which is not finitely failed can be generated.

This problem was not properly taken care of in the definition of SLDNF-resolution given in Clark [Cla79] and reproduced in Lloyd [Llo84]. In Lloyd [Llo87] a revised definition of SLDNF-resolution was proposed according to which the SLDNF-trees are constructed "bottom-up" by induction on the number of alternations through negation. Unfortunately, according to this definition for the above mentioned example, or for $P = \{A \leftarrow \neg A\}$ and the query A, no SLDNF-trees or SLDNF-derivations exist.

Kunen [Kun89] avoids this problem by just defining the set of answer substitutions which are computable by the SLDNF-resolution without defining what SLDNF-resolution is. This definition is very simple (see Section 5), and was sufficient for proving completeness results, but it is not useful if one wants to prove results about the SLDNF-derivations themselves, like termination. Note that in both examples mentioned above the top-down interpreter diverges.

These problems were mentioned by Apt and Bezem [AB91, page 352] and Apt and Pedreschi [AP91, pages 267–268]. They were tackled by Martelli and Tricomi [MT92] who proposed a revision of the original definition in which the subsidiary trees used to resolve negative literals are built "inside" the main tree. These authors consider trees whose nodes are formulas more complicated than general goals, which necessitated the introduction of so-called "collapsing cases" to simplify these formulas.

The solution proposed below seems simpler and more intuitive: as in the original definition the subsidiary trees are kept "aside" but their construction is no longer viewed as an atomic step in the resolution process. Instead, they are built in a stepwise "top-down" manner, by constructing their branches in parallel. If during this subsidiary construction divergence arises, the main derivation *is considered* to be infinite. This formalizes the intuitive solution suggested in Apt and Pedreschi [AP91]. In the second part of this note we compare our definition of SLDNF-resolution with those of Lloyd [Llo87], of Martelli and Tricomi [MT92], and of Kunen [Kun89].

Various results concerning the "run time behaviour" behaviour of SLDNF-deri-vations, like termination, absence of floundering, safety of the omission of the occur-check or the groundness of the input positions of the selected literals under some syntactic conditions, can be correctly stated and rigorously proved *only* once an appropriate definition of SLDNF-resolution is available. Some of these properties were studied in the literature and, strictly speaking, the corresponding proofs lacked the formal basis. We found that using the proposed definition of SLDNF-resolution these arguments can be easily justified.

The approach taken here can also be readily used to define correctly several variants of SLDNF-resolution proposed in the literature, for example SLDNFS-resolution of Shepherdson [She89] and the extension of SLD-resolution with so-called constructive negation of Chan [Cha88].

2 A new definition

We start by recalling and introducing a number of auxiliary notions. Below we use ":=" for "is by definition equal to".

Definition 2.1 $Var(E)$ is the set of variables in the expression E.
A *substitution* is a function from variables to terms. ϵ is the identity substitution. We write '$x\alpha$' for the value of the substitution α at the variable x.
The *domain* $Dom(\alpha)$ of α is the set of x's for which $x\alpha \neq x$. (Usually, this is taken to be a finite set.) Its *range* $Ran(\alpha)$ is the set $\bigcup_{x \in Dom(\alpha)} Var(x\alpha)$. (Thus, $Dom(\epsilon) = Ran(\epsilon) = \emptyset$.) The variables form $Dom(\alpha) \cup Ran(\alpha)$ are said to *occur in* α.
If V is a set of variables, then the *restriction* $\alpha|V$ of α to V, is the substitution with domain $V \cap Dom(\alpha)$ which coincides on this domain with α. For an expression E, we write $\alpha|E := \alpha|Var(E)$.
An mgu of two atoms A and B is called *relevant* if every variable occurring in it belongs to $Var(A) \cup Var(B)$. $\square$

A *query* is a finite sequence of literals. (Instead of queries, one usually considers *general goals* which are expressions $\leftarrow C$ where C is a query.) The empty query is denoted by $\square$.

Definition 2.2
(i) We say that C *resolves to* D *via* α *w.r.t.* Σ, or: D (more explicitly, the pair (α, D)) is a *resolvent* of C w.r.t. Σ, notation: $C \stackrel{\alpha}{\Longrightarrow} D$ (Σ), if
either: $\Sigma = (L, R)$, L is (an occurrence of) a positive literal in C, R is a program clause, and for some variant $A \leftarrow E$ (the *input clause*) of R: α is mgu of L and A and $D = C\alpha\{L\alpha/E\alpha\}$ is obtained from $C\alpha$ by replacing $L\alpha$ by $E\alpha$,
or: Σ is (an occurrence of) a negative literal in C, $\alpha = \epsilon$, and $D = C - \{\Sigma\}$ is obtained from C by removing Σ.
(ii) A clause R is called *applicable to* an atom if it has a variant the head of which unifies with the atom. □

Definition 2.3 A (finite or infinite) sequence $C_0 \stackrel{\alpha_1}{\Longrightarrow} \cdots C_n \stackrel{\alpha_{n+1}}{\Longrightarrow} C_{n+1} \cdots$ of resolution steps is a *pseudo derivation* if for every step involving a program clause:

- ("standardisation apart") the input clause employed does not contain a variable from the initial query C_0 or from an input clause used at some earlier step,
- ("relevance") the mgu employed is relevant. □

Intuitively, an SLDNF-derivation is a pseudo derivation in which the deletion of every (ground) negative literal is justified by means of a subsidiary (finitely failed SLDNF-) tree. This brings us to consider special types of trees.

Definition 2.4 A tree is called

- *successful* if it contains a leaf marked as *success*,
- *finitely failed* if it is finite and all its leaves are marked as *failed.* □

In the sequel we consider systems of trees called (for lack of a better name) *complex trees.*

Definition 2.5 A *complex tree* is a system $\mathcal{T} = (\mathcal{T}, T, subs)$ where

- $\mathcal{T}$ is a set of trees,
- T is an element of $\mathcal{T}$ called the *main* tree,
- *subs* is a function assigning to some nodes of trees in $\mathcal{T}$ a ("subsidiary") tree from $\mathcal{T}$.

By a *path* in $\mathcal{T}$ we mean a sequence of nodes $N_0, \ldots, N_i, \ldots$ such that for all i, N_{i+1} is either an immediate descendant of N_i in some tree in $\mathcal{T}$ or the root of the tree $subs(N_i)$. □

Thus a complex tree is a special directed graph with two types of edges — the "usual" ones stemming from the tree structures, and the ones connecting a node with the root of a subsidiary tree. An SLDNF-tree is a special type of complex tree, built as a limit of certain finite complex trees: *pre-SLDNF trees.*

For the rest of this paper, we fix a general program P.

Definition 2.6 A *pre-SLDNF-tree* (relative to P) is a complex tree whose nodes are (possibly marked) queries of (possibly marked) literals. (For queries, there are markers *failed*, *success*, and *floundered*; for literals, we have the marker *selected.*) The function *subs* assigns to nodes containing a marked negative ground literal $\neg A$ a tree in $\mathcal{T}$ with root A. The class of pre-SLDNF-trees is defined inductively.

- For every query C, the complex tree consisting of the main tree which has the single node C is a pre-SLDNF-tree (an *initial* pre-SLDNF tree),

- If $\mathcal{T}$ is a pre-SLDNF-tree, then any *extension* of $\mathcal{T}$ is a pre-SLDNF-tree.

Here, an *extension* of a pre-SLDNF-tree $\mathcal{T}$ is defined by performing the following actions for every non-empty query C which is an unmarked leaf in some tree $T \in \mathcal{T}$:

First, if no literal in C is marked yet as *selected*, mark one as *selected*. Let L be the selected literal of C.

- L is positive.
  - C has no resolvents w.r.t. L and a clause from P.
    Then C is marked as *failed*.
  - C has such resolvents.
    For every clause R from P which is applicable to L, choose one resolvent (α, D) of C w.r.t. L and R and add this as immediate descendant of C in T. These resolvents are chosen in such a way that all branches of T remain pseudo derivations.
- $L = \neg A$ is negative.
  - A is non-ground. Then C is marked as *floundered*.
  - A is ground.
    * $subs(C)$ is undefined.
      Then a new tree T' with the single node A is added to $\mathcal{T}$ and $subs(C)$ is set to T'.
    * $subs(C)$ is defined and successful.
      Then C is marked as *failed*.
    * $subs(C)$ is defined and finitely failed.
      Then the resolvent $(\epsilon, C - \{L\})$ of C is added as the only immediate descendant of C in T.

Additionally, all empty queries are marked as *success*. □

Note that, if no tree in $\mathcal{T}$ has unmarked leaves, then trivially $\mathcal{T}$ is an extension of itself, and the extension process becomes stationary.

Every pre-SLDNF-tree is a tree with two types of edges between possibly marked nodes, so the concepts of *inclusion* between such trees and of *limit* of a growing sequence of such trees have clear meaning.

Definition 2.7

- An *SLDNF-tree* is a limit of a sequence $\mathcal{T}_0, \ldots, \mathcal{T}_i, \ldots$ such that $\mathcal{T}_0$ is an initial pre-SLDNF-tree, and for all i, $\mathcal{T}_{i+1}$ is an extension of $\mathcal{T}_i$.

- *An SLDNF-tree for C* is an SLDNF-tree in which C is the root of the main tree.

- A (pre-)SLDNF-tree is called *successful* (resp. *finitely failed*) if the main tree is successful (resp. *finitely failed*).

- An SLDNF-tree is called *finite* if no infinite paths exist in it (cf. Definition 2.5). $\square$

Next, we define the concept of SLDNF-derivation.

Definition 2.8 A *(pre-) SLDNF-derivation for C* is a branch in the main tree of a (pre-) SLDNF-tree $\mathcal{T}$ for C together with the set of all trees in $\mathcal{T}$ whose roots can be reached from the nodes of this branch. An SLDNF-derivation is called *finite* if all paths of $\mathcal{T}$ fully contained within this branch and these trees are finite. $\square$

Finally, it is clear how to define the notion of a computed answer substitution.

Definition 2.9 Consider a branch in the main tree of a (pre-) SLDNF-tree for C which ends with the empty query. Let $\alpha_1, \ldots, \alpha_n$ be the consecutive substitutions along this branch.

Then the restriction $(\alpha_1 \cdots \alpha_n)|C$ of the composition $\alpha_1 \cdots \alpha_n$ to the variables of C is called a *computed answer substitution* (*c.a.s.* for short) of C. $\square$

Let us illustrate the above definitions by depicting the SLDNF-trees for the two problematic cases considered in the beginning. The edges connecting a node with the root of a subsidiary tree are drawn by dashed lines.

Example 2.10

(i) Consider $P = \{A \leftarrow A\}$ and $C = \neg A$. The only SLDNF-tree has then the following form:

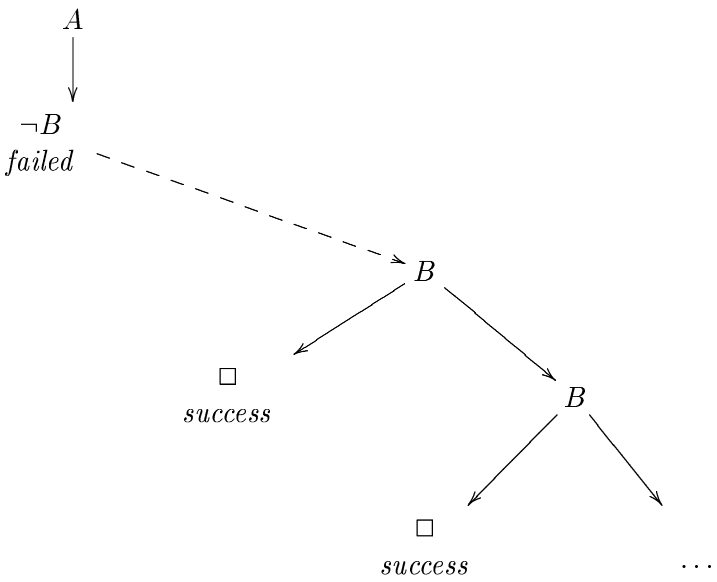
$$\begin{tikzcd}
\neg A \arrow[dr, dashed] & \\
& A \arrow[d] \\
& A \arrow[d] \\
& \cdots
\end{tikzcd}$$

(ii) Consider now $P = \{A \leftarrow \neg A\}$ and $C = A$. Again, there is here only one SLDNF-tree, which is infinite and looks as follows:

$$\begin{tikzcd}
A \arrow[d] & & \\
\neg A \arrow[dr, dashed] & & \\
& A \arrow[d] & \\
& \neg A \arrow[dr, dashed] & \\
& & \cdots
\end{tikzcd}$$

(iii) It is important to realize that according to our definition the construction of a subsidiary tree can go on forever even if the information about its "status" has already been passed to the main tree. The following general program illustrates this point.

Consider $P = \{A \leftarrow \neg B,\ B \leftarrow\ ,\ B \leftarrow B\}$. Then the only SLDNF-tree for A looks as follows:



Here the subsidiary tree with the root B grows forever. However, once an extension of the initial subsidiary tree with the single node B becomes successful, in the next extension the node $\neg B$ is marked as *failed*. Consequently, the SLDNF-tree for A is finitely failed even though it is not finite. $\Box$

Pre-SLDNF-trees may keep growing forever. However, when the resulting SLDNF-tree is successful or finitely failed, this fact becomes apparent after a finite number of steps already. More precisely, we have the following result.

Theorem 2.11

(i) Every pre-SLDNF-tree is finite.

(ii) Every SLDNF-tree is the limit of a unique sequence of pre-SLDNF-trees.

(iii) If the SLDNF-tree $\mathcal{T}$ is the limit of the sequence $\mathcal{T}_0, \ldots, \mathcal{T}_i, \ldots$, then for all τ

(a) $\mathcal{T}$ is successful and yields τ as c.a.s. iff some $\mathcal{T}_i$ is successful and yields τ as c.a.s.,

(b) $\mathcal{T}$ is finitely failed iff some $\mathcal{T}_i$ is finitely failed.

Proof.
(i) Obvious induction.
(ii) The only way in which extensions of a pre-SLDNF tree included in a given SLDNF-tree can become different is by the selection of different literals in non-empty nodes. But this selection is prescribed by the SLDNF-tree given.
(iii) ($\Rightarrow$) A branch of the main tree of $\mathcal{T}$ ending in $\Box$ or a finitely failed main tree of $\mathcal{T}$ consists of finitely many, possibly marked, nodes. Each of these nodes (markings included) belongs to some $\mathcal{T}_i$ and the $\mathcal{T}_i$ with the largest i is the desired pre-SLDNF-tree.
($\Leftarrow$) Each $\mathcal{T}_i$ is contained (markings included) in $\mathcal{T}$. $\Box$

This result allows us to associate with every successful or finitely failed SLDNF-tree $\mathcal{T}$ a natural number, $rank(\mathcal{T}, \tau)$ which is the least i for which the corresponding equivalence in (iii) holds, with $\tau = \epsilon$ when $\mathcal{T}$ is finitely failed.

A notion known to be difficult to define in the case of SLDNF-resolution is that of a selection rule. Intuitively, a selection rule allows us to select a literal in the query which is to be resolved. As pointed out in Shepherdson [She84, page 62] a correct definition of selection rule should take into account the dependence on the already generated nodes, so that for example the "left-most, right-most" selection rule can be defined. This can be easily achieved as follows.

In our definition of an SLDNF-tree, the selection rule is "incorporated" into the construction of an extension — through the selection of literals in the lastly generated nodes.

Clearly, this selection process can be separated from the construction of an extension. Let us drop the selection of literals in the lastly generated nodes from the definition of the pre-SLDNF tree. Then a selection rule is a function defined on pre-SLDNF-trees selecting a literal in every non-empty non-marked leaf.

In this revised set-up an SLDNF-tree is obtained by alternating the process of applying the selection function with the process of extending the pre-SLDNF-tree.

3 Comparison with Lloyd's and Martelli and Tricomi's definitions

Lloyd [Llo87] defines successful SLDNF-derivations and finitely failed SLDNF-trees (in short SLDNF-objects) by simultaneous induction w.r.t. their rank. Informally, a *rank* corresponds to the number of times one passes "through negation" while defining the corresponding SLDNF-object.

Thus in successful SLDNF-derivations of rank 0 and finitely failed SLDNF-trees of rank 0 no negative literals are selected. In turn, a selection of a negative ground literal $\neg A$ in an SLDNF-object of rank $n+1$ succeeds if a finitely failed SLDNF-tree for A of rank n exists, and fails if a successful SLDNF-derivation for A of rank n exists.

Ignoring small points (like the use of goals instead of queries and refutations instead of successful derivations) there are three differences between Lloyd's definition and ours.

First, – as already mentioned in Section 1 – for some programs and queries, like $P = \{A \leftarrow A\}$ and $\neg A$, no SLDNF-derivations and SLDNF-trees according to Lloyd's definition exist.

Secondly, a selection of a ground negative literal $\neg A$ fails if a successful SLDNF-derivation for A exists, whereas in our case if a successful SLDNF-tree for A exists. By replacing such a successful SLDNF-derivation by a successful SLDNF-tree, it is straightforward to show by induction on the rank that if an SLDNF-object exists according to Lloyd's definition, then so it does according to our definition.

Finally, floundering is treated differently. In Lloyd's definition it arises when a query is generated which consists exclusively of non-ground negative literals, as then no literal can be selected in it. In our definition floundering arises as soon as a non-ground negative literal is selected in a query. Clearly, this small difference is of no importance, since Lloyd's notion of floundering can be easily defined in our framework. Our definition of floundering is more appropriate when studying SLDNF-resolution with a fixed selection rule, like Prolog leftmost selection rule.

Martelli and Tricomi [MT92] provided a definition of SLDNF-resolution according to which for every program and query an SLDNF-tree exists. Informally, given a selection rule, according to their definition only one tree is generated and the construction of the subsidiary trees takes place "within" this tree. In particular, selection of a ground negative literal $\neg A$ within a query $\mathbf{K}, \neg A, \mathbf{M}$ yields *one* resolvent, namely $\mathbf{K}, \neg[L_1; \ldots; L_n], \mathbf{M}$. Here ";" is to be interpreted as a disjunction. Thus the syntax of the queries needs to be extended and to properly handle such extended queries so-called "collapsing cases" are needed to simplify them. For example $\mathbf{K}, \neg["fail"; \ldots; "fail"], \mathbf{M}$ is simplified to $\mathbf{K}, \mathbf{M}$. Martelli and Tricomi [MT92] proved that if an SLDNF-object exists according to Lloyd's definition, then so it does according to their definition.

Comparing our definition with that of Martelli and Tricomi we notice the following. Their definition makes it possible to define a *fair* selection rule, according to which in every infinite SLDNF-derivation for every literal appearing in it eventually some further instantiated version of it is selected.

On the other hand, we find that our definition remains more closely to the implementation and also to what probably is the primordial intuitive idea of an SLDNF-tree as a computation in which secondary, tertiary, ... computations can be started from selected negative literals. Also according to our definition the branches of all trees are constructed in parallel, whereas in Martelli and Tricomi's definition they are constructed sequentially. Consequently, different SLDNF-trees in the sense of Martelli and Tricomi are identitified by our definition.

4 Intermezzo on computed answer substitutions

This section proves some technical results about pseudo derivations needed in the last section.

Note that as a consequence of Definition 2.3, in a pseudo derivation $C_0 \stackrel{\alpha_1}{\Longrightarrow} C_1 \stackrel{\alpha_2}{\Longrightarrow} C_2 \cdots$, any variable occurring in α_{n+1} or C_{n+1} occurs either in C_0 or in an input clause used at some step $\leq n$. Also, every subsequence of a pseudo derivation is a pseudo derivation.

Definition 4.1 The variables from $Var(C\alpha) - Var(D)$ are said to be *released* at the resolution step $C \stackrel{\alpha}{\Longrightarrow} D$. □

This notion was introduced in Doets [Doe92]. Its relevance was ilustrated there by showing that the following lemma is responsible for lifting and maximal generality of derivations in the SLD case.

Lemma 4.2 *In a pseudo derivation, no variable released at some step occurs in a query or an mgu of a later step.*

Proof. Assume that $C_0 \stackrel{\alpha_1}{\Longrightarrow} C_1 \cdots$ is a pseudo derivation in which x is released at the first step. Then $x \notin Var(C_1)$ and x occurs in C_0 or in α_1, thus x occurs in C_0 or in the first input clause if the selected literal in C_0 is positive. Therefore, no input clause used in the pseudo derivation $C_1 \stackrel{\alpha_2}{\Longrightarrow} C_2 \cdots$ contains x. Since input clauses are responsible for the introduction of variables in this pseudo derivation, the result follows. □

Lemma 4.3 *If* $C_0 \stackrel{\alpha_1}{\Longrightarrow} \cdots \stackrel{\alpha_n}{\Longrightarrow} C_n$ $(n \geq 2)$ *is a pseudo derivation, then* $Var(C_0\alpha_1) \cap Var(C_1\alpha_2 \cdots \alpha_n) \subseteq Var(C_1)$.

Proof. Assume that $x \in Var(C_0\alpha_1) - Var(C_1)$. That is, x is released at the first step. Clearly, $Var(C_1\alpha_2 \cdots \alpha_n) \subseteq Var(C_1) \cup Ran(\alpha_2) \cup \cdots \cup Ran(\alpha_n)$. By Lemma 4.2, x does not occur at the right-hand side. Therefore, $x \notin Var(C_1\alpha_2 \cdots \alpha_n)$. □

Lemma 4.4 *If* $C_0 \stackrel{\alpha_1}{\Longrightarrow} \cdots \stackrel{\alpha_n}{\Longrightarrow} C_n$ $(n \geq 2)$ *is a pseudo derivation, then* $(\alpha_1((\alpha_2 \cdots \alpha_n)|C_1))|C_0 = (\alpha_1 \cdots \alpha_n)|C_0$.

Proof. If not, a variable $x \in Var(C_0)$ exists such that $x\alpha_1((\alpha_2 \cdots \alpha_n)|C_1) \neq x\alpha_1 \cdots \alpha_n$. Then a variable $y \in Var(x\alpha_1)$ exists such that $y((\alpha_2 \cdots \alpha_n)|C_1) \neq y\alpha_2 \cdots \alpha_n$. Thus, $y \notin Var(C_1)$, and $y((\alpha_2 \cdots \alpha_n)|C_1) = y$. Since $y \in Var(C_0\alpha_1) - Var(C_1)$, y is released at the first step. But then, by Lemma 4.2, $y \notin Dom(\alpha_2) \cup \cdots \cup Dom(\alpha_n)$, hence $y\alpha_2 \cdots \alpha_n = y$. Therefore, $y = y\alpha_2 \cdots \alpha_n \neq y((\alpha_2 \cdots \alpha_n)|C_1) = y$ — a contradiction. □

Corollary 4.5 *Suppose that the main tree of a (pre-) SLDNF-tree for C has a successful branch with a corresponding c.a.s. τ for C. If $C \stackrel{\alpha}{\Longrightarrow} D$ is the first step of this branch, and the rest of it yields the c.a.s. σ for D, then $\tau = (\alpha\sigma)|C$.*

Proof. Apply Lemma 4.4 to this successful branch. □

5 Comparison with Kunen's definition

The difficulty of defining SLDNF-resolution was elegantly circumvented in Kunen [Kun89] where a completeness theorem of SLDNF-resolution for allowed general programs and allowed queries was proved.

In his considerations Kunen [Kun89] dealt only with success and finite failure, which allowed him to define the concepts needed by a remarkably simple "bottom-up" inductive definition avoiding the construction of SLDNF-trees and SLDNF-derivations altogether. This approach is sufficient when dealing with completeness of SLDNF-resolution but cannot be used to reason about properties which inherently refer to SLDNF-trees, like the ones mentioned in the introduction.

We now clarify the relation between our definition of computed answer substitutions and of finite failure and those of Kunen [Kun89]. Let us start by recalling Kunen's definition.

Again, an arbitrary general program P is fixed.

Definition 5.1 The set $\mathbf{F}$ of queries and the set $\mathbf{R}$ of pairs (C, σ) — C a query and σ a substitution for which $Dom(\sigma) \subseteq Var(C)$ —are defined by a simultaneous inductive definition as follows.

0) $\Box \mathbf{R} \epsilon$,

R+) if C resolves to D via α w.r.t. some positive literal of C and a clause from P and $D\mathbf{R}\sigma$, then $C\mathbf{R}(\alpha\sigma)|C$,

R–) if A is a ground atom in $\mathbf{F}$ and $(C, C')\mathbf{R}\sigma$, then $(C, \neg A, C')\mathbf{R}\sigma$,

F+) if L is a positive literal in C and for every clause R from P which is applicable to L there exist α and $D \in \mathbf{F}$ such that $C \stackrel{\alpha}{\Longrightarrow} D$ (L, R), then $C \in \mathbf{F}$,

F–) if A is a ground atom such that $A\mathbf{R}\epsilon$, then $(C, \neg A, C') \in \mathbf{F}$. □

The intention here is that $\mathbf{R}$ is the set of pairs (C, σ) such that σ is a c.a.s. for C and $\mathbf{F}$ is the set of queries C such that there is a finitely failed tree for C.

Kunen's original formulation of F+) *could* be interpreted as stating that if $L \in C$ is positive and *every* resolvent of C w.r.t. L and a clause of P is in $\mathbf{F}$, then $C \in \mathbf{F}$,

but we suspect that this does not change the notions of $\mathbf{R}$ and $\mathbf{F}$; besides, this (unnecessarily) complicates the proof of Theorem 5.3 (case b1) below.

A modification.

The accompanying notion of soundness associated with Kunen's definition is the following one:

- if $C\mathbf{R}\sigma$, then $comp(P) \models C\sigma$, and
- if $C \in \mathbf{F}$, then $comp(P) \models \neg C$.

These implications can be proved simultaneously by a straightforward induction along the clauses of the definition. In fact, soundness still holds if the usual groundness conditions on the atom A in R–) and F–) are left out. (The resulting notion is called *SLDNFE*, for SLDNF *extended.*) However, to get the optimal match between Kunen's notions and ours, we have to change his definition at one point.

The formulation of R+) does *not* ensure that the resulting answer substitutions are most general. For instance, if P consists of the clauses
$Q(x', y') \leftarrow Q(y', y')$,
$Q(x, x) \leftarrow$,
then $\Box\mathbf{R}\epsilon$ (by 5.1.0), $Q(y, y)\mathbf{R}\{y/x\}$ (by R+) and the second clause) and consequently $Q(x, y)\mathbf{R}\{y/x\}$ (by R+), since $Q(x, y)$ resolves to $Q(y, y)$ via $\{x'/x, y'/y\}$ and the first clause. But $\{y/x\}$ is not a c.a.s. for $Q(x, y)$ whereas $\{y/z\}$ is.

Note that the corresponding successful 2-step derivation $Q(x, y) \overset{\{x'/x, y'/y\}}{\Longrightarrow} Q(y, y) \overset{\{y/x\}}{\Longrightarrow} \Box$ is not obtained by properly standardizing apart: the input clause $Q(x, x) \leftarrow$ used at the second step contains a variable used earlier. Also, it is worthwile to mention that this irregularity has no bearing on the class of allowed programs and queries considered in Kunen [Kun89], since the computed answer substitutions are then always grounding.

In order that R+) produces most general answer substitutions, we amend it as follows:

†R+) if C resolves to D via α w.r.t. some positive literal of C and a clause from P, $D\mathbf{R}\sigma$, *and*

$$Var(C\alpha) \cap Var(D\sigma) \subseteq Var(D), \tag{1}$$

then $C\mathbf{R}(\alpha\sigma)|C$.

Note that this condition coincides with the claim of Lemma 4.3. Formulated slightly differently, it says that variables released at the step $C \overset{\alpha}{\Longrightarrow} D$ do not occur in $D\sigma$.

The condition †R+) together with a selection in R+) of input clauses disjoint with the queries can be viewed as a formalization of the standardization apart condition for Kunen's definition.

The following lemma will be needed later.

Lemma 5.2 *If $C \in \mathbf{F}$ and $C \subseteq D$, then $D \in \mathbf{F}$.*

Proof. By a straightforward induction using only clauses F+) and F–) of Definition 5.1. $\Box$

The next theorem uses Kunen's definition as modified above.

Theorem 5.3 *If C is a query, then*

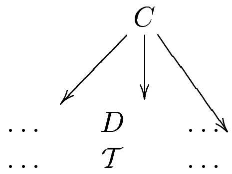
- $C\mathbf{R}\tau$ *iff τ is a c.a.s. for C,*
- $C \in \mathbf{F}$ *iff C has a finitely failed SLDNF-tree.*

Proof.
The left–to–right halves of these equivalences are proved simultaneously by induction along the clauses of the modified Definition 5.1. Below, selected literals in queries are underlined. This part of the proof requires the construction of SLDNF-trees. However, by Theorem 2.11, it suffices to construct pre-SLDNF-trees only. In fact, we shall sometimes only indicate how to construct a relevant *part* of the required pre-SLDNF-tree.

0) $C = \Box$ and $\alpha = \epsilon$.
This case is trivial.

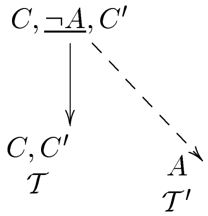
†R+) Suppose that C resolves to D via mgu α w.r.t. some positive literal. Furthermore, assume that $D\mathbf{R}\sigma$, where (cf. the modification) condition (1) holds. We want to show that $(\alpha\sigma)|C$ is a c.a.s. for C.

By induction hypothesis, σ is a c.a.s. for D. That is, the main tree T of an SLDNF-tree $\mathcal{T}$ for D has a branch ending in success and σ is the c.a.s. along this branch. By condition (1), $D\sigma$ does not contain variables from $Var(C\alpha) - Var(D)$. Therefore, we may assume (renaming variables in T if necessary) that T does not involve a variable from $Var(C\alpha) - Var(D)$. But then we can modify the SLDNF-tree by putting C on top of T as a new root, since the resulting branches will be pseudo derivations. This produces *part* of an SLDNF-tree for C, showing C to have the c.a.s. $(\alpha\sigma)|C$ by Corollary 4.5:



R–) Suppose that $A \in \mathbf{F}$ is ground, and $(C, C')\mathbf{R}\sigma$. We want to show that σ is a c.a.s. of $C, \underline{\neg A}, C'$.

By induction hypotheses, there exists a finitely failed SLDNF-tree $\mathcal{T}'$ for A and there is an SLDNF-tree $\mathcal{T}$ for C, C' whose branch yields the c.a.s. σ. Then



is a successful SLDNF-tree for $C, \underline{\neg A}, C'$ whose branch yields the c.a.s. σ.

F+) Suppose that L is a positive literal in C and for every clause R from P which is applicable to L there exist α and $D \in \mathbf{F}$ such that $C \overset{\alpha}{\Longrightarrow} D$ $(L,\, R)$. We want to show that C has a finitely failing SLDNF-tree. Let $D_1, \ldots, D_n \in \mathbf{F}$ be resolvents of C w.r.t. L and, respectively, all clauses $R_1, \ldots, R_n$ of P applicable to L.

By induction hypothesis, choose a finitely failing SLDNF-tree $\mathcal{T}_i$ for every resolvent D_i. Then

$$\begin{tikzcd} & C \arrow[dl] \arrow[d] \arrow[dr] & \\ D_1 & \cdots & D_n \\ \mathcal{T}_1 & \cdots & \mathcal{T}_n \end{tikzcd}$$

is the required finitely failing SLDNF-tree for C. Obviously, we can assume (compare case †R+) that the $\mathcal{T}_i$ are such that the branches of the new main tree will be pseudo derivations.

F–) Suppose that the atom A is ground and $A\mathbf{R}\epsilon$. We want to show that there is a finitely failed SLDNF-tree for $C, \underline{\neg A}, C'$.

By induction hypothesis, there exists a successful SLDNF-tree $\mathcal{T}$ for A. Then

$$\begin{tikzcd} C, \underline{\neg A}, C' & \\ \textit{failed} \arrow[dr, dashed] & \\ & A \\ & \mathcal{T} \end{tikzcd}$$

is a finitely failed SLDNF-tree for $C, \underline{\neg A}, C'$.

The right–to–left halves of the two equivalences are proved simultaneously by induction on $rank(\mathcal{T}, \tau)$ where
a) $\mathcal{T}$ is a successful SLDNF-tree for C with a branch yielding the c.a.s. τ, or
b) $\mathcal{T}$ is a finitely failed SLDNF-tree for C and $\tau = \epsilon$.

$rank(\mathcal{T}, \tau) = 0$.

Then $\mathcal{T}$ is successful (since C is not marked), so $C = \Box$ and $\tau = \epsilon$. Thus $C\mathbf{R}\epsilon$ by clause 0).

$rank(\mathcal{T}, \tau) > 0$.

a1) The selected literal of C is positive.

Let D be the direct descendant of C in $\mathcal{T}$ lying on the branch which yields the c.a.s. τ. D is obtained from C using an mgu α. Let σ be the c.a.s. for D along this branch. By induction hypothesis, $D\mathbf{R}\sigma$. Moreover, by Lemma 4.3 we have $Var(C\alpha) \cap Var(D\sigma) \subseteq Var(D)$. Therefore by clause †R+) we get $C\mathbf{R}(\alpha\sigma)|C$. However, by Corollary 4.5 $\tau = (\alpha\sigma)|C$.

a2) The selected literal of C is negative.

Then $C = D, \underline{\neg A}, D'$ where A is ground and $subs(C)$ fails finitely. But $rank(subs(C), \epsilon) < rank(\mathcal{T}, \tau)$ and A is the root of the main tree of $subs(C)$, so by induction hypothesis $A \in \mathbf{F}$. Moreover, the only direct descendant of C in $\mathcal{T}$ is D, D'. Again by induction hypothesis $(D, D')\mathbf{R}\tau$. Therefore by clause R–) we get $C\mathbf{R}\tau$.

b1) The selected literal of C is positive. By induction hypothesis, all direct descendants of C in $\mathcal{T}$ are in $\mathbf{F}$. Therefore by clause F+) we get $C \in \mathbf{F}$.

b2) The selected literal of C is negative.

Then $C = D, \underline{\neg A}, D'$ where A is ground.

Subcase 1. C is marked as *failed*.

Then $subs(C)$ is successful. But $rank(subs(C), \epsilon) < rank(\mathcal{T}, \tau)$ and A is the root of the main tree of $subs(C)$, so by induction hypothesis $A\mathbf{R}\epsilon$. Therefore by clause F–) we get $C \in \mathbf{F}$.
Subcase 2. C is not marked as *failed*.

$\mathcal{T}$ is finitely failed, so C has a direct descendant. Therefore $subs(C)$ is finitely failed, and D, D' is the only direct descendant of C in $\mathcal{T}$. By induction hypothesis $(D, D') \in \mathbf{F}$. Therefore by Lemma 5.2 we get $C \in \mathbf{F}$. □

Acknowledgements

We thank the referees for helpful remarks on the subject of this paper.

References

[AB91] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 29(3):335–363, 1991.

[AP91] K. R. Apt and D. Pedreschi. Proving termination of general Prolog programs. In T. Ito and A. Meyer, editors, *Proceeding of the International Conference on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 526, pages 265–289, Berlin, 1991. Springer-Verlag.

[Cha88] D. Chan. Constructive negation based on the completed database. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 111–125. The MIT Press, 1988.

[Cla79] K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[Doe92] H. C. Doets. Levationis laus. *Journal of Logic and Computation*, 1992. To appear.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP'74*, pages 569–574. North-Holland, 1974.

[Kun89] K. Kunen. Signed data depedencies in logic programs. *Journal of Logic Programming*, 7:231–246, 1989.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[MT92] M. Martelli and C. Tricomi. A new SLDNF-tree. *Information Processing Letters*, 43(2):57–62, 1992.

[She84] J. C. Shepherdson. Negation as failure: a comparison of Clark's completed data base Reiter's closed world assumption. *Journal of Logic Programming*, 1(1):51–79, 1984.

[She89] J. C. Shepherdson. A sound and complete semantics for a version of negation as failure. *Theoretical Computer Science*, 65(3):343–371, 1989.