SLS-resolution without floundering

Włodzimierz Drabent
IPI PAN, Polish Academy of Sciences
Ordona 21, Pl – 01-237 Warszawa, Poland
wdr@ida.liu.se
and
IDA, Linköping University
S – 581 83 Linköping, Sweden

Abstract

SLS-resolution is an abstract query answering procedure for computing the well-founded semantics of normal programs. It is incomplete due to floundering. We present an extension of SLS-resolution that avoids this problem.

1 Introduction

From the point of view of non-monotonic reasoning, the most suitable semantics for logic programs is the well-founded semantics [5]. It is equivalent to appropriate forms of all four major formalizations of non-monotonic reasoning [15].

The standard abstract query answering mechanism for computing the well-founded semantics for normal programs is SLS-resolution. It was introduced for stratified programs in [14] and generalized for arbitrary programs in [16] and [13]. It is incomplete due to floundering. In this paper we present a generalization of SLS-resolution that is sound and complete.

Methods for computing answers for nonground negated queries are called constructive negation. Our work follows the constructive negation approach presented in [11] for definite programs and the completion semantics. Similar idea was proposed in [19]. The main concept of this approach is as follows. In order to find an answer for $\leftarrow\neg A$, a failed SLD-tree for an instance $\leftarrow A\theta$ of $\leftarrow A$ is built; θ is then an answer for $\leftarrow\neg A$. [11] also shows how to compute such answers: an SLD-tree for $\leftarrow A$ is pruned in order to obtain a finitely failed tree. Pruning means instantiating the tree (thus obtaining an SLD-tree for some $\leftarrow A\sigma$) in such a way that some subtree of the original tree disappears. Removing all the success leaves and infinite branches results in a finitely failed SLD-tree.

In this paper we present a generalization of this approach for normal programs and the well-founded semantics. A proper definition of a failed tree is important here. A straightforward extension of the concept of a failed SLS-tree leads to unsoundness; such a "failed tree" for $\leftarrow A$ does not demonstrate that A is false but only that A is false or undefined.

We introduce a correct definition of a failed tree and present SLSFA-resolution, a query answering method for normal programs and goals. It

is sound and complete with respect to the well-founded semantics. Our method subsumes SLS-resolution as defined in [14] for stratified programs and SLDNF-resolution [8]. Every SLS- (SLDNF-) failed tree is an SLSFA-failed tree and every SLS- (SLDNF-) refutation is an SLSFA-refutation.

Our presentation is informal. Section 2 describes the notation and some preliminary notions. SLSFA-resolution is introduced and explained by means of examples in Section 3. Section 4 contains a formal definition, a soundness and completeness result and some more advanced examples. Section 5 contains conclusions.

We assume that the reader is familiar with the well-founded semantics [5] and with SLS-resolution for stratified programs [14].

2 Preliminaries

We use the standard logic programming terminology and definitions [8]. However, normal programs are just called programs.

Logic programs are written in first order languages that differ only by their sets of predicate symbols and functors (including constants). We do not assume a fixed language for all programs, nor do we define a program's language as that of exactly the functors and predicate symbols occurring in the program. Instead we assume that, for every program under consideration, the set of functors and of predicate symbols of the underlying language $\mathcal{L}$ is known. We will say that $\mathcal{L}$ is (in)finite if its set of functors is (in)finite.

When referring to syntactic objects of $\mathcal{L}$, s, t, u will usually stand for terms, v, x, y variables, a, b, c constants, p, q predicate symbols. Sub- and superscripts may be used if necessary. Overlining will be used to denote a (finite) sequence of objects, e.g. $\overline{x}$ is an abbreviation for $x_1, \ldots, x_n$ for some $n \geq 0$. Symbol $=$ will be used both in $\mathcal{L}$ and in the metalanguage. We take care that this does not lead to ambiguity.

The set of free variables occurring in a syntactic construct (term, formula etc.) F is denoted by FreeVars(F). *Restriction* $F|_S$ of a formula F to a set S of variables is the formula $\exists x_1, \ldots, x_n F$ where $\{x_1, \ldots, x_n\} = \text{FreeVars}(F) \setminus S$.

$WF(P)$ denotes the well-founded model of a program P. We refer to SLS-resolution as defined for stratified programs in [14].

2.1 Constraints

In standard logic programming answers are given in the form of idempotent substitutions. This is not feasible when answers to negative queries are required. Some generalization of the concept of a substitution is needed to conveniently express inequality.

In order not to restrict ourselves to a particular form of answers, we will use arbitrary first order formulae built out of equality and inequality literals. Such formulae will be called *constraints* and denoted by $\theta, \sigma, \delta, \rho$

(possibly with sub- and superscripts). Note that an idempotent substitution $\{x_1/t_1, \ldots, x_n/t_n\}$ corresponds to a constraint $x_1{=}t_1 \wedge \ldots \wedge x_n{=}t_n$. Conjunction of constraints θ and σ will often be denoted by θ,σ or by $\theta\sigma$ (as it plays the role of composition of substitutions).

We are interested only in Herbrand interpretations of the underlying language $\mathcal{L}$ with = interpreted as equality. Equality in the Herbrand universe $\mathcal{U}_\mathcal{L}$ is axiomatized by Clark equality theory (see [8]). If $\mathcal{L}$ is finite then the (weak) domain closure axiom DCA [9],[12] should be added. Informally, the axiom ensures that in the interpretation domain of any model of the theory every object is a value of a non-variable term (under some variable valuation).

By CET we denote the Clark equality theory with added DCA in the case of $\mathcal{L}$ finite. The axiomatization is complete [20]; constraint θ is true in a Herbrand interpretation of $\mathcal{L}$ iff CET $\models \theta$. (As usual, by truth of an open formula θ in an interpretation or in a theory we mean truth of $\forall\theta$.)

A constraint θ is called *satisfiable* iff CET $\models \exists\theta$. θ is *more general* than σ iff CET $\models \sigma \rightarrow \theta$. θ and σ are *equivalent* iff CET $\models \sigma \leftrightarrow \theta$; we will write $\sigma \equiv \theta$. Note that terms t and s are unifiable iff constraint $t{=}s$ is satisfiable.

SLD- (and SLS-) resolution can be in an obvious way converted into a version using constraints instead of substitutions. Instead of applying an m.g.u. to a goal, the corresponding constraint is *added* to a goal. In standard SLD-resolution goal $\leftarrow p(t_1, \ldots, t_n)$ and a clause $p(s_1, \ldots, s_n) \leftarrow B$ resolve into $\leftarrow B\tau$ where τ is a most general idempotent unifier of $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$. In the version with constraints the result is $\leftarrow t_1{=}s_1, \ldots, t_n{=}s_n, B$. Thus we will use goals in the form $\leftarrow \theta, B$ where θ is a satisfiable constraint and B is a sequence of literals.

From the practical point of view it is important to solve constraints, i.e. to transform them into some intelligible form. Many papers are devoted to this subject, we refer to [20], [1], [3], [10] and to the references therein. There exist algorithms that reduce any constraint to an equivalent one in some disjunctive normal form. (Such an algorithm also checks satisfiability; for an unsatisfiable constraint the empty formula **false** is obtained). The normal form may be, for instance, a disjunction of "simple" constraints of the form

$$\exists\, \overline{y}\, (x_1{=}t_1 \wedge \ldots \wedge x_n{=}t_n \wedge \forall ...(v_1{\neq}s_1) \wedge \ldots \wedge \forall ...(v_m{\neq}s_m))$$

where $n, m \geq 0$, $\{x_1/t_1, \ldots, x_n/t_n\}$ is an idempotent substitution, the x_i's do not occur elsewhere in this formula and some (maybe none) variables of the s_i's are universally quantified.

The choice of actual normal form and of a reduction algorithm is an important implementation decision which is outside of the scope of this paper. There is no agreement in the papers on constructive negation on which normal form to use. Our method is independent from this choice, we allow arbitrary constraints. However a restriction can be imposed that every

constraint used in SLSFA-resolution (a computed answer, a fail answer, the constraint in a goal, etc.) is a simple constraint. (Any notion of simple constraints can be applied here. The only requirement is that there exists an algorithm transforming every constraint into an equivalent disjunction of simple constraints). Definitions and theorems of the next sections remain correct with such a restriction.

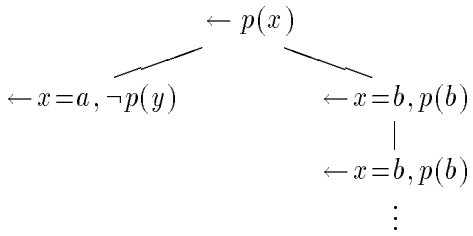
3 Informal presentation

In this section SLSFA-resolution (FA for fail answers) is presented by means of examples. First we introduce our approach using an example for which the notions of an SLS-failed tree and an SLSFA-failed tree coincide. Then we show that generalizing the definition of an SLS-failed tree in a straightforward way leads to unsoundness. We explain the reason for the unsoundness and introduce a notion of a failed tree suitable for our purposes. Then we show how to construct failed trees by pruning.

Example 3.1 Consider a program P

$$p(a) \leftarrow \neg p(x)$$
$$p(b) \leftarrow p(b)$$

and a goal $\leftarrow \neg p(x)$. To find an answer, a failed SLS-tree for some instance of $\leftarrow p(x)$ is constructed. This can be done by pruning the SLS-tree for $\leftarrow p(x)$:



Pruning means adding a constraint to the nodes of the tree in such a way that some subtree of the original tree disappears (and the resulting tree is still an SLS-tree). For example the tree can be pruned at depth 1: applying constraint $x \neq b$ removes the subtree rooted at $\leftarrow x=b, p(b)$, applying $x \neq a$ removes the (single node) subtree rooted at $\leftarrow x=a, \neg p(y)$. As a result we obtain a tree consisting of a single node $\leftarrow x \neq a, x \neq b, p(x)$ which is a failed SLS-tree. Thus $x \neq a, x \neq b$ is a fail answer for $\leftarrow p(x)$ and an answer for $\leftarrow \neg p(x)$.

A more general answer is possible, it is enough to prune the only leaf of the tree. The resulting failed SLS-tree for $\leftarrow x \neq a, p(x)$ consists of a single infinite branch, $x \neq a$ is an answer for $\leftarrow \neg p(x)$. Both answers are sound as $WF(P) \models x \neq a \rightarrow \neg p(x)$.

The ability of answering nonground negative queries makes it possible to avoid floundering. Sequence

$$\leftarrow p(x); \qquad \leftarrow x{=}a, \neg p(y)$$

is a floundering SLS-derivation. Using a previously computed answer $y{\neq}a$ for $\leftarrow\neg p(y)$, this derivation can be extended to an SLSFA-refutation

$$\leftarrow p(x); \qquad \leftarrow x{=}a, \neg p(y); \qquad \leftarrow x{=}a, y{\neq}a.$$

The computed answer of an SLSFA-refutation is obtained similarly as in SLS-resolution. The last goal contains the constraint accumulated in the refutation. The constraint plays the role of the composition of the m.g.u.'s of an SLS-refutation. The answer is the restriction of this constraint to the variables of the initial goal. In our example it is $\exists y(x{=}a, y{\neq}a)$ which is equivalent to $x{=}a$. □

3.1 Unsoundness of a naive solution

The notion of a failed tree in SLS-resolution can be presented as follows. Let us treat goals as equal up to variable renaming. Consider a goal $\leftarrow Q$, a computation rule R and a tree built out of all the derivations for $\leftarrow Q$ via R. The tree is failed if (1) it does not have a success leaf and (2) for every its leaf $\leftarrow \ldots, \underline{\neg A}, \ldots$ with a negative literal selected, A is ground and $\leftarrow A$ succeeds.

The following example shows that using such a definition together with constructive negation leads to unsoundness. We construct a tree satisfying this definition for which Q is not false with respect to the well-founded semantics.

Example 3.2 Let P be the program

$$\begin{array}{lcl} p & \leftarrow & \neg q(x), r(x) \\ q(a) & \leftarrow & \neg q(a) \\ q(b) & & \\ r(a) & & \\ r(b) & & \end{array}$$

In the well-founded semantics of the program, p is undefined as $q(a)$ is undefined.

Assume that a and b are not the only functors of the underlying language. Constraint $x{\neq}a, x{\neq}b$ is a computed answer for $\leftarrow\neg q(x)$ as the (one node) tree for $\leftarrow x{\neq}a, x{\neq}b, q(x)$ is failed. Note that this is a most general answer for $\leftarrow\neg q(x)$ that is correct with respect to the well-founded semantics. Now the tree

$$\begin{array}{c} \leftarrow p \\ | \\ \leftarrow \underline{\neg q(x)}, r(x) \\ | \\ \leftarrow x{\neq}a, x{\neq}b, r(x) \end{array}$$

satisfies the abovementioned definition of a failed tree[1]. Clauses $r(a)$ and $r(b)$ are not applicable to the last goal as it contains $x{\neq}a, x{\neq}b$. Condition (1) is satisfied as the only leaf contains a literal $r(x)$, condition (2) is satisfied trivially. On the other hand, the tree should not be considered failed as p is not false with respect to the well-founded semantics of the program. □

The reason for unsoundness could be informally explained as follows. A tree satisfying the definition above, with the root $\leftarrow Q$, shows that there does not exist any answer for $\leftarrow Q$. Thus $\exists Q$ is *not true* w.r.t. the well-founded semantics. However this does not imply that $\exists Q$ is false, as the underlying logic is three valued.[2] For the purposes of constructive negation we need a notion of a failed tree that demonstrates that a query is neither true nor undefined in the well-founded semantics of the program.

3.2 Correct solution

Now we informally present the concept of SLSFA-failed tree. In comparison with SLS-failed trees, the difference concerns the treatment of goals with a negative literal selected. Consider such a goal $G = \leftarrow \underline{\neg A}, Q$ and assume that x is the only variable occurring in G. To show that $\neg A, Q$ is false we have to prove the following property: Q is false for any x for which A is not true. In SLSFA-resolution we use a slight generalization of this property. Assume that $\delta_1, \ldots, \delta_n$ are some (maybe not all) computed answers for $\leftarrow A$ and let $\delta = \delta_1 \vee \ldots \vee \delta_n$. Then $WF(P) \models \delta \rightarrow A$. Now to show that $WF(P) \models \neg(\neg A, Q)$ it is sufficient to prove that $WF(P) \models \neg\delta \rightarrow \neg Q$. This can be done by constructing a failed tree for $\leftarrow \neg\delta, Q$. Thus $\leftarrow \neg\delta, Q$ can be made the only son of $\leftarrow \underline{\neg A}, Q$ in a failed tree.

Note an important difference. A branch of an SLSFA-failed tree *is not necessarily an SLSFA-derivation.* If $\neg A$ is selected in an SLSFA-derivation then an answer for $\leftarrow \neg A$ is used; if it is selected in an SLSFA-failed tree then the *negation of some answers* for $\leftarrow A$ is used instead.

Example 3.3 Consider the program from the previous example. The tree

$$\begin{array}{c} \leftarrow x{\neq}a, \underline{\neg q(x)}, r(x) \\ \vert \\ \leftarrow x{\neq}a, x{\neq}b, r(x) \end{array}$$

is an SLSFA-failed tree, as $x{=}b$ is an answer for $\leftarrow q(x)$ and no program clause is applicable to goal $\leftarrow x{\neq}a, x{\neq}b, r(x)$.

[1] Strictly speaking, there exist other derivations for $\leftarrow p$ but each of them is an instance of the branch of the tree.

[2] In view of this, the soundness of the concept of SLS-failed tree could be seen as a somehow surprising fact.

Failed trees can be constructed in the following way. An attempt to build a failed tree for $\leftarrow \underline{\neg q(x)}, r(x)$ results in a pre-failed[3] tree

$$\begin{array}{c} \leftarrow \underline{\neg q(x)}, r(x) \\ | \\ \leftarrow x{\neq}b, r(x) \\ | \\ \leftarrow x{=}a \end{array}$$

(Note that the branch of the tree is not an SLSFA-derivation; $x{=}a$ cannot be treated as an answer to $\leftarrow \neg q(x), r(x)$.) By pruning the node $\leftarrow x{\neq}b, r(x)$ (which means applying constraint $\neg(x{\neq}b)$ to the tree) we obtain a failed tree

$$\leftarrow x{=}b, \underline{\neg q(x)}, r(x).$$

The failed tree shown at the beginning of the example is obtained by pruning the leaf $\leftarrow x{=}a$.

There does not exist an SLSFA-failed tree for $\leftarrow p$. The pre-failed tree

$$\begin{array}{c} \leftarrow p \\ | \\ \leftarrow \underline{\neg q(x)}, r(x) \\ | \\ \leftarrow x{\neq}b, r(x) \\ | \\ \leftarrow x{=}a \end{array}$$

cannot be pruned, as its root does not contain any variables. (As in the previous cases, the branch of the tree is not an SLSFA-derivation; existence of a "success" leaf does not imply that p is true). □

Generally, a node $\leftarrow\theta, \ldots$ can be pruned by adding to the nodes of the tree such a constraint ρ that $\rho\theta$ is unsatisfiable. Additionally, the free variables of ρ should occur free in the root of the tree (in order to obtain a correct pre-failed tree). Thus a most general such constraint is $\rho = \neg(\theta|_V)$ where V is the set of the free variables of the root of the tree [11, 4].

It is convenient to allow more than one son of a node $G = \leftarrow\neg A, Q$ in a failed tree. For example consider one answer $\delta = x{\neq}a, x{\neq}b$ for $\leftarrow A$; then $\neg\delta = (x{=}a \vee {=}b)$. Constructing two sons $\leftarrow x{=}a, Q$ and $\leftarrow x{=}b, Q$ of G may be preferable to one son $\leftarrow (x{=}a \vee x{=}b), Q$ if equalities are implemented as substitutions.

This suggests a following condition. For the sake of simplicity assume Prolog computation rule. In a failed tree a node $G = \leftarrow\theta, \underline{\neg A}, Q$ with a negative literal selected has sons $\leftarrow\sigma_1, Q; \ldots; \leftarrow\sigma_m, Q$ (where $m \geq 0$)

[3] A tree satisfying the definition of SLSFA-failed tree except for the condition forbidding "success" nodes will be called a *pre-failed* tree.

provided that there exist $\delta_1, \ldots, \delta_n$ (where $n \geq 0$) that are SLSFA-computed answers for $\leftarrow \theta, A$ such that

$$\theta \rightarrow \delta_1 \vee \cdots \vee \delta_n \vee \sigma_1 \vee \cdots \vee \sigma_m.$$

is true in the Herbrand universe (or, equivalently, in CET).

Actually, this condition is not sufficient to achieve completeness of SLSFA-resolution. It may be necessary to consider infinitely many sons of $\leftarrow \theta, \neg A, Q$ and/or infinitely many answers for $\leftarrow \theta, A$ (see Example 4.10). For the generalized condition see Definition 4.6.

Summarizing this section: Sons of a node $\leftarrow \ldots, \underline{\neg A}, \ldots$ in a failed tree are obtained by negating some answers for $\leftarrow A$. A successor of $\leftarrow \ldots, \underline{\neg A}, \ldots$ in a refutation is obtained by using an answer for $\leftarrow \neg A$ (i.e. a fail answer for $\leftarrow A$).

4 SLSFA-resolution

Here we present a formal definition of SLSFA-resolution. It is followed by examples and a soundness/completeness theorem. We begin with a modification of the concept of a goal. An adjustment is needed due to usage of constraints instead of substitutions.

Definition 4.1 A *goal* is a formula of the form $\neg(\theta \wedge L_1 \wedge \ldots \wedge L_m)$ usually written as

$$\leftarrow \theta, L_1, \ldots, L_m$$

(or just $\leftarrow \theta, \overline{L}$) where θ is a satisfiable constraint and $L_1, \ldots, L_m$ ($m \geq 0$) are literals. We will omit θ if it is (equivalent to) **true**.

Now a formalization of a common notion of a goal with a literal selected.

Definition 4.2 An *s-goal* is a pair of a goal and a literal position $\langle \leftarrow \theta, L_1, \ldots, L_m; i \rangle$ (where $1 \leq i \leq m$ or $m = 0 = i$), usually written as $\leftarrow \theta, L_1, \ldots, L_{i-1}, \underline{L_i}, L_{i+1}, \ldots, L_m$ (or as $\leftarrow \theta, \overline{L}, \underline{L_i}, \overline{L'}$ where $\overline{L} = L_1, \ldots, L_{i-1}$ and $\overline{L'} = L_{i+1}, \ldots, L_m$).

L_i is called the selected literal of the above s-goal (if $i \geq 1$). G is called the goal part of an s-goal $\langle G; i \rangle$. If it does not lead to ambiguity we sometimes do not distinguish between an s-goal and its goal part.

Definition 4.3 Let G be an s-goal $\leftarrow \theta, \overline{L}, \underline{p(t_1, \ldots, t_n)}, \overline{L'}$ and C a clause $p(s_1, \ldots, s_n) \leftarrow \overline{M}$. An s-goal G' is *positively derived* from G using C iff the following holds:

- $\mathrm{FreeVars}(G) \cap \mathrm{FreeVars}(C) = \emptyset$,
- θ' is the constraint $(t_1 = s_1 \wedge \cdots \wedge t_n = s_n)$,

- (the goal part of) G' is $\leftarrow \theta\theta', \overline{L}, \overline{M}, \overline{L'}$.

By the definition of a goal, $\theta\theta'$ above is satisfiable. We will say that a clause C is *applicable* to a goal G if there exists a goal positively derived from G using a variant of C.

The definition of SLSFA-resolution consists of mutually recursive Definitions 4.4, 4.5 and 4.6. To assure correctness of the definition, the concept of ranks is used, as in the definition of SLDNF-resolution [8]. Ranks are ordinal numbers. Refutations are defined in terms of negative derivation steps of the same rank. These are, in turn, defined in terms of failed trees of a lower rank. Failed trees are defined in terms of refutations of a lower rank. The base case is the definitions for rank 0 (of a refutation and a failed tree).

Definition 4.4 Let P be a program and α an ordinal. Assume that the notion of "negatively derived" is defined for ranks $< \alpha$. An *SLSFA-refutation of rank* α is a sequence of s-goals $G_0, \ldots, G_n$ such that G_n is $\leftarrow \theta$ and, for $i = 1, \ldots, n$,

- G_i is positively derived from G_{i-1} using a variant C of a program clause from P such that $\text{FreeVars}(C) \cap \text{FreeVars}(G_0, \ldots, G_{i-1}) = \emptyset$
- or $\alpha > 0$ and G_i is rank α negatively derived from G_{i-1}.

The constraint $\theta|_{\text{FreeVars}(G_0)}$ is a called an SLSFA-*computed answer* for (the goal part of) G_0, of rank α.

Definition 4.5 Let P be a program, $\alpha > 0$ and assume that failed trees of ranks $< \alpha$ are already defined. Let

$$G = \leftarrow \theta, \overline{L}, \underline{\neg A}, \overline{L'}$$

be an s-goal with a negative literal selected. G' is *rank* α *negatively derived* from G if, for some θ',

- $G' = \leftarrow \theta\theta', \overline{L}, \overline{L'}$,
- $\leftarrow \theta\theta', A$ fails and is of rank $< \alpha$,
- $\text{FreeVars}(\theta') \subseteq \text{FreeVars}(A)$;

Constraint $\theta\theta'$ is called a *fail answer* for $\leftarrow \theta, A$.

Definition 4.6 Let P be a program, α an ordinal and G a goal. Assume that SLSFA-refutations of ranks $< \alpha$ are already defined. Then G *fails* and is of *rank* α iff there exists a tree (called rank α *SLSFA-failed tree*) satisfying the following conditions:

1. each node is an s-goal and the goal part of the root node is G;

2. if H is a node in the tree with a positive literal selected then for every clause C of P applicable to H there exists exactly one son of H that is positively derived from H using a variant of C;

3. A node H with a negative literal selected, of the form

$$\leftarrow \theta, \overline{L}, \underline{\neg A}, \overline{L'}$$

has (possibly zero or infinitely many) sons

$$\leftarrow \sigma_1, \overline{L}, \overline{L'};\ \leftarrow \sigma_2, \overline{L}, \overline{L'};\ \ldots$$

provided that there exist (possibly zero or infinitely many) SLSFA-computed answers

$$\delta_1, \delta_2, \ldots$$

of ranks $< \alpha$ for $\leftarrow \theta, A$ such that for every ground substitution τ for FreeVars(θ, A) if $\theta\tau$ is true[4] then some $\delta_i\tau$ or some $\sigma_i\tau$ is true.

4. no node of the tree is of the form $\leftarrow \theta$.

The condition in part 3 of the definition is called safeness condition. A node H satisfying it will be called *correct*. A tree satisfying the definition without part 4 will be called an SLSFA *pre-failed* tree. See [4] and [11] (or Section 3) for ways of obtaining failed trees by pruning pre-failed ones.

When the sets of answers and sons are finite, say n answers and m sons, the safeness condition becomes

$$\text{CET} \models \theta \rightarrow \delta_1 \vee \cdots \vee \delta_n \vee \sigma_1 \vee \cdots \vee \sigma_m.$$

A standard way of computing σ_i's is then converting $\theta\neg\delta_1 \cdots \neg\delta_n$ to a disjunctive normal form $\sigma_1 \vee \cdots \vee \sigma_m$. It is not clear how to compute $\sigma_1, \sigma_2, \ldots$ when the set of δ_i's is infinite. A "brute force" method is to construct a son for every ground substitution τ as above for which $\theta\tau$ is true and every δ_i is false. (The constraint of the son is the constraint corresponding to τ). Usually more general σ_i's are possible (conf. Example 4.10).

A definition of an SLSFA-*derivation* can be obtained from Definition 4.4 by removing the requirements for the form of the last goal and of the finiteness of the sequence.

Note that a refutation (failed goal, failed tree) of rank α is also of any higher rank. A computed answer for $\leftarrow \theta, \overline{L}$ can be represented as $\theta\theta'$ where FreeVars$(\theta') \subseteq$ FreeVars$(\overline{L})$. For other technical properties of derivations and failed trees see [4].

[4] in the Herbrand universe of the underlying language or, equivalently, in CET

4.1 Further examples

We discuss two versions of a standard example: a game with a finite and with an infinite graph. Then we show that infinite ranks and infinite branching are necessary for completeness of SLSFA-resolution.

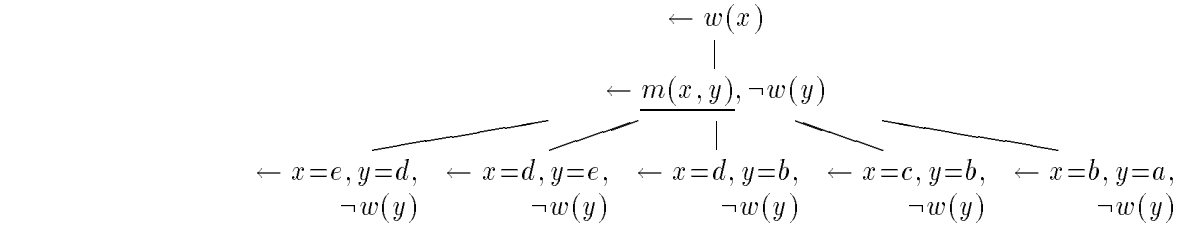
Example 4.7 Remark: for convenience, some constraints in our examples may be replaced by equivalent ones.

Consider a program [6]

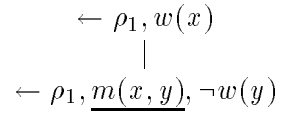
$w(x) \leftarrow m(x,y), \neg w(y)$
$m(e,d)$
$m(d,e)$
$m(d,b)$
$m(c,b)$
$m(b,a)$

In its well-founded semantics $w(b)$ is true, $w(a)$ and $w(c)$ are false and $w(d)$ and $w(e)$ are undefined.

A "top section" of the pre-failed tree for $\leftarrow w(x)$ with Prolog computation rule is



An SLSFA-failed tree can be obtained by pruning the five nodes at depth 3 of the tree. For example, to prune $\leftarrow x{=}e, y{=}d, \neg w(y)$ constraint $\neg\exists_y(x{=}e, y{=}d) \equiv x{\neq}e$ has to be added to the root of the tree. As a result we obtain a failed tree of rank 0:



where $\rho_1 = x{\neq}e, x{\neq}d, x{\neq}c, x{\neq}b$. Under an assumption that a, b, c, d, e are the only functors of $\mathcal{L}$, $\rho_1 \equiv x{=}a$.

Now we can construct a refutation of rank 1:

$$\leftarrow w(x) \quad \leftarrow m(x,y), \underline{\neg w(y)} \quad \leftarrow y{=}a, m(x,y) \quad \leftarrow x{=}b, y{=}a$$

with the computed answer $x{=}b$.

Thus $y{=}b$ is a rank 1 computed answer for $\leftarrow w(y)$. If this answer is used in the pre-failed tree above then the nodes $\leftarrow x{=}d, y{=}b, \neg w(y)$ and

$\leftarrow x{=}c, y{=}b, \neg w(y)$ do not have sons. Hence these nodes do not need to be pruned. Pruning the remaining three nodes at depth 3 results in

$$\begin{array}{c} \leftarrow \rho_2, w(x) \\ | \\ \leftarrow \rho_2, \underline{m(x,y)}, \neg w(y) \\ | \\ \leftarrow x{=}c, y{=}b, \neg w(y) \end{array} \quad \text{where } \rho_2 = x{\neq}e, x{\neq}d, x{\neq}b \equiv x{=}a \vee x{=}c$$

which is a rank 2 SLSFA-failed tree. Note that the fail answer ρ_2 does not give rise to any new answer for $\leftarrow w(x)$ and that ρ_2 is a most general fail answer for $\leftarrow w(x)$. $\square$

Example 4.8 (Previous example with infinite ralation m)

$w(x) \leftarrow m(x,y), \neg w(y)$
$m(f^2(x), f(x))$
$m(g(x), x)$
$m(g(x), g^2(x))$

Assume that f, g and a constant a are the only functors of $\mathcal{L}$. In the well-founded semantics, $w(s)$ is false for s being a or $f^{2i-1}(t)$ where the main functor of t is not f and $i = 1, 2, \ldots$ (conf. the diagram of m below). It is true for s being $g(a)$, $g(f^{2i-1}(t))$ or $f^{2i}(t)$ where t and i are as above. For the remaining terms it is undefined (i.e. for $g(f^{2i}(t))$ where $i > 0$ and t as above and for $g^i(t')$ where $i > 1$ and t' is arbitrary).

$$\begin{array}{ccccccc} \vdots & & \vdots & & \vdots & & \vdots \\ \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow \\ g^2(a) & & g^2(f(t)) & & g^2(f^2(t)) & & g^2(f^3(t)) \\ \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow \\ g(a) & & g(f(t)) & & g(f^2(t)) & & g(f^3(t)) \\ \downarrow & & \downarrow & & \downarrow & & \downarrow \\ a & & f(t) & \longleftarrow & f^2(t) & \longleftarrow & f^3(t) \quad \longleftarrow \cdots \end{array}$$

Similarly as in the previous example we obtain:

- A failed tree of rank 0:

$$\begin{array}{c} \leftarrow \rho, w(x) \\ | \\ \leftarrow \rho, \underline{m(x,y)}, \neg w(y) \end{array}$$

where $\rho = \forall_z x{\neq}f^2(z), \forall_z x{\neq}g(z) \equiv x{=}a \vee \exists_v(x{=}f(v), \forall_z v{\neq}f(z))$.

- Refutations of rank 1 (where $\theta = \exists_v(y{=}f(v), \forall_z v{\neq}f(z))$):

$$\begin{array}{llll} \leftarrow w(x) & \leftarrow m(x,y), \underline{\neg w(y)} & \leftarrow y{=}a, m(x,y) & \leftarrow y{=}a, x{=}g(y) \\ \leftarrow w(x) & \leftarrow m(x,y), \underline{\neg w(y)} & \leftarrow \theta, m(x,y) & \leftarrow \theta, x{=}f^2(x'), y{=}f(x') \\ \leftarrow w(x) & \leftarrow m(x,y), \underline{\neg w(y)} & \leftarrow \theta, m(x,y) & \leftarrow \theta, x{=}g(y) \end{array}$$

with the computed answers $x{=}g(a)$, $\exists_v(x{=}f^2(v), \forall_z v{\neq}f(z))$ and $\exists_v(x{=}gf(v), \forall_z v{\neq}f(z))$ respectively.

• A failed tree of rank $2n$, for $n = 1, 2, \ldots$:

$$\begin{array}{c} \leftarrow \rho_n, w(x) \\ | \\ \leftarrow \rho_n, \underline{m(x,y)}, \neg w(y) \qquad\qquad \text{where } \rho_n = \exists_v(x{=}f^{2n+1}(v), \forall_z v{\neq}f(z)) \\ | \\ \leftarrow \rho_n, x{=}f^2(x'), y{=}f(x'), \neg w(y) \end{array}$$

The leaf of the tree is correct as $\exists_v(y{=}f^{2n}(v), \forall_z v{\neq}f(z))$ is a rank $2n-1$ answer for $\leftarrow w(y)$, see below. Note that not all the lower rank answers need to be used.

• Refutations of rank $2n+1$ ($n = 1, 2, \ldots$):

$$\begin{array}{llll} \leftarrow w(x) & \leftarrow m(x,y), \underline{\neg w(y)} & \leftarrow \rho_n[x/y], m(x,y) & \leftarrow \rho_n[x/y], x{=}f^2(x'), y{=}f(x') \\ \leftarrow w(x) & \leftarrow m(x,y), \underline{\neg w(y)} & \leftarrow \rho_n[x/y], m(x,y) & \leftarrow \rho_n[x/y], x{=}g(y) \end{array}$$

with the computed answers $\exists_v(x{=}f^{2n+2}(v), \forall_z v{\neq}f(z))$ and $\exists_v(x{=}gf^{2n+1}(v), \forall_z v{\neq}f(z))$ respectively.

The rank $2n$ failed tree above ($n = 0, 1, \ldots$) may be constructed by pruning the following pre-failed tree

$$\begin{array}{c} \leftarrow w(x) \\ | \\ \leftarrow \underline{m(x,y)}, \neg w(y) \\ \swarrow \quad | \quad \searrow \\ \leftarrow x{=}f^2(x'), y{=}f(x'), \neg w(y) \quad \leftarrow x{=}g(y), \neg w(y) \quad \leftarrow x{=}g(x'), y{=}g^2(x'), \neg w(y) \\ | \\ \leftarrow x{=}f^2(x'), y{=}f(x'), \neg \delta_n \end{array}$$

where $\delta_0 = \mathbf{false}$ and for $n > 0$ $\delta_n = \exists_v(y{=}f^{2n}(v), \forall_z v{\neq}f(z))$ is an answer for $\leftarrow w(y)$ of rank $2n-1$.

For $n = 0$ pruning the leaves gives constraint ρ. For $n > 0$ in order to prune the first leaf, constraint $\theta_1 = \neg\exists_{x',y}(x{=}f^2(x'), y{=}f(x'), \neg\delta_n)$ should be used. It is equivalent to $\forall_{x'} x{\neq}f^2(x') \vee \exists_{x',y}(x{=}f^2(x'), y{=}f(x'), \delta_n)$ which is equivalent to $\forall_{x'} x{\neq}f^2(x') \vee \rho_n$
(as $\exists_{x',y}(x{=}f^2(x'), y{=}f(x'), \exists_v(y{=}f^{2n}(v), \forall_z v{\neq}f(z)))$ is equivalent to $\exists_v(x{=}f^{2n+1}(v), \forall_z v{\neq}f(z))$ and to ρ_n).

The constraints to prune the second and the third leaf are, respectively, $\theta_2 = \neg\exists_y x{=}g(y)$ and $\theta_3 = \neg\exists_{x',y}(x{=}g(x'), y{=}g^2(x'))$. Both are equivalent to $\forall_z x{\neq}g(z)$. Thus $\theta_1\theta_2\theta_3 \equiv (\forall_{x'} x{\neq}f^2(x') \vee \rho_n), \forall_z x{\neq}g(z) \equiv \rho \vee \rho_n$. $\square$

Example 4.9 (Infinite rank)
Consider a program

$$even(0) \qquad\qquad odd(s(0))$$
$$even(s(x)) \leftarrow \neg even(x) \qquad\qquad odd(s^2(x)) \leftarrow odd(x)$$

For $i = 0, 1, \ldots$, constraint $x{=}s^{2i}(0)$ is a rank $2i$ computed answer for $\leftarrow even(x)$ and there exists an SLSFA-failed tree for $\leftarrow x{=}s^{2i+1}(0), even(x)$ of rank $2i+1$ (see [4] for details).

Assume Prolog computation rule. The failed tree for $\leftarrow \underline{odd(x)}, even(x)$ has an infinite branch with nodes $\leftarrow x{=}s^{2i}(y_i), \underline{odd(y_i)}, even(x)$, $i = 0, 1, \ldots$ and infinitely many finite branches with leaves equivalent to $\leftarrow \ldots, \neg even(s^{2i}(0))$, $i = 0, 1, \ldots$. As above, a successful derivation for $\leftarrow even(s^{2i}(0))$ is of rank $2i$. Thus the rank of the tree is ω. □

Example 4.10 (Infinitely branching tree)
Atom p is false w.r.t. the well-founded semantics of the program

$$p \leftarrow \neg q(x), \neg r(x) \qquad\qquad r(0)$$
$$q(x) \leftarrow \neg r(x) \qquad\qquad r(s(x)) \leftarrow r(x)$$

The answers for $\leftarrow r(x)$ are $\delta_i = x{=}s^i(0)$ for $i = 0, 1, \ldots$. Assume that 0 and s are not the only functors of the underlying language. Then constraints $\delta_i' = \exists_y(x{=}s^i(y), y{\neq}0, \forall_z y{\neq}s(z))$ for $i = 0, 1, \ldots$ are fail answers for $\leftarrow r(x)$ and answers for $\leftarrow q(x)$.

An SLSFA-failed tree for $\leftarrow p$ (of rank 2) has branches

$$\leftarrow p$$
$$\leftarrow \underline{\neg q(x)}, \neg r(x)$$
$$\leftarrow \delta_i, \neg r(x)$$

for $i = 0, 1, \ldots$. The safeness condition is satisfied because for every ground instance $x\tau$ of x some $\delta_i\tau$ or some $\delta_i'\tau$ is true. There does not exist a finitely branching failed tree for $\leftarrow p$. There does not exist a failed tree for $\leftarrow p$ in which a finite set of answers for $\leftarrow q(x)$ (or for $\leftarrow r(x)$) is taken into account. □

4.2 Soundness and completeness

The following theorem formulates soundness, completeness and independence from computation rule for SLSFA-resolution. For a proof see [4].

Theorem 4.11 Let P be a normal program and $\leftarrow \theta, \overline{L}$ be a goal. Let $WF(P)$ be the well-founded (3-valued, Herbrand) model of P.

If δ is an SLSFA-computed answer for $\leftarrow \theta, \overline{L}$ then $WF(P) \models \delta \rightarrow \overline{L}$. If there exists an SLSFA-failed tree for $\leftarrow \theta, \overline{L}$ then $WF(P) \models \neg(\theta, \overline{L})$ (or, equivalently, $WF(P) \models \theta \rightarrow \neg\overline{L}$).

If $WF(P) \models \neg(\theta, \overline{L})$ then for any computation rule there exists an SLSFA-failed tree for $\leftarrow\theta, \overline{L}$. If $WF(P) \models \overline{L}\tau$, where τ is a substitution and $\overline{L}\tau$ is ground, then for any computation rule τ is covered by an SLSFA-computed answer: there exists a computed answer δ for $\leftarrow\theta, \overline{L}$ such that $\delta\tau$ is true in CET provided $\theta\tau$ is true in CET.

5 Conclusions

We presented SLSFA-resolution, a constructive negation approach for the well-founded semantics. It is sound and complete for arbitrary normal programs and goals and for any computation rule. It subsumes SLS-resolution as originally defined in [14] for stratified programs. (Any SLS-refutation is an SLSFA-refutation, the same for failed trees.) In contrast, the other top-down query answering mechanisms for the well-founded semantics [16, 13] are incomplete due to floundering. They are also restricted to computation rules that select a positive literal whenever possible.

We generalized the notion of a failed tree to floundering goals (a straightforward generalization is unsound). An answer to a negated query $\leftarrow\neg A$ is obtained by constructing a failed tree for an instance $\leftarrow\theta, A$ of $\leftarrow A$. The tree can be obtained by pruning a pre-failed tree for $\leftarrow A$. For a discussion of pruning see [4] and [11].

As the well-founded semantics is not computable, SLSFA-resolution is not an effective computational mechanism. What can be implemented is an algorithm that is a sound but incomplete approximation of SLSFA-resolution. A crude approximation is obtained by considering only finite failed trees in which finite numbers of computed answers are used. Such an approximation is called SLDFA-resolution in [4]. It is sound and complete for Clark completion semantics in 3-valued logic [7]. Better approximations are a subject for future research. An obvious hint is to use methods of tabulation [17, 18] for finite representation of infinite trees and methods like those presented in [2] for finite representation of infinite sets of answers.

This work shows that a rather natural generalization of the standard concept of a failed tree provides a sound and complete operational semantics for two declarative semantics for logic programs: the 3-valued completion semantics and the well-founded semantics. The only difference is using finitely failed trees in the first case and infinite ones in the second. The author believes that this confirms the importance and naturalness of these semantics; the first for finite failure and the second for infinite failure.

Acknowledgments

The author wants to thank Teodor Przymusiński for stimulating discussions. This work was partially supported by Swedish Research Council for Engineering Sciences (TFR), grants no. 221-91-331 and NUTEK 90-1676, by the

Polish Academy of Sciences and by University of California (Riverside) Research Grant.

References

[1] D. Chan. Constructive negation based on the completed database. In *Proc. Fifth International Conference and Symposium on Logic Programming,* Seattle, pages 111–125. MIT Press, 1988.

[2] J. Chomicki and T. Imieliński. Finite Representation of Infinite Query Answers. *ACM Transactions on Database Systems*, 1993. To appear.

[3] H. Comon and P. Lescanne. Equational problems and disunification. *J. Symbolic Computation*, 7:371–425, 1989.

[4] W. Drabent. What is failure? An approach to constructive negation. Preliminary version appeared as Technical Report LiTH-IDA-R-91-23, Linköping University, August 1991. Submitted for *Acta Informatica.*

[5] A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:620–650, 1991.

[6] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming,* Seattle, pages 1070–1080. MIT Press, 1988.

[7] K. Kunen. Negation in logic programming. *J. of Logic Programming*, 4:289–308, 1987.

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

[9] J. W. Lloyd and R. W. Topor. A Basis for Deductive Database Systems. *Journal of Logic Programming*, 2(2):93–109, 1985.

[10] M. Maher. Complete axiomatization of the algebras of finite, rational and infinite trees. In *Proc. 3rd Symposium on Logic in Computer Science*, pages 348–357, 1988.

[11] J. Małuszyński and T. Näslund. Fail substitutions for negation as failure. In *Proc. North American Conference on Logic Programming,* Cleveland, pages 461–476. MIT Press, 1989.

[12] P. Mancarella, S. Martini, and D. Pedreschi. Complete logic programs with domain closure axiom. *J. of Logic Programming*, 5(3):263–276, 1988.

[13] T. C. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *Proc. of the Eighth Symposium on Principles of Database Systems*, pages 11–21. ACM SIGACT-SIGMOD, 1989.

[14] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[15] T. C. Przymusinski. Three-valued non-monotonic formalisms and logic programming. In *Proc. of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89), Toronto, Canada*, pages 341–348, 1989.

[16] K. A. Ross. A procedural semantics for well founded negation in logic programs. *J. of Logic Programming*, 13:1–22, 1992.

[17] T. Sato and H. Tamaki. OLD-resolution with tabulation. In *Proceedings of the third International Conference on Logic Programming*, 1986.

[18] H. Seki and H. Itoh. A query evaluation method for stratified programs under the extended CWA. In *Proc. of the Fifth International Conference and Symposium on Logic Programming*, pages 195–211, 1988.

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

[20] J. C. Shepherdson. Language and equality theory in logic programming. Technical Report PM–91–02, School of Mathematics, University of Bristol, 1991.