[To appear in Acta Informatica]

What is Failure?
An Approach to Constructive Negation

Włodzimierz Drabent[*]

March 9, 1994

Abstract

A standard approach to negation in logic programming is negation as failure. Its major drawback is that it cannot produce answer substitutions to negated queries. Approaches to overcoming this limitation are termed constructive negation. This work proposes an approach based on construction of *failed trees* for some instances of a negated query. For this purpose a generalization of the standard notion of a failed tree is needed. We show that a straightforward generalization leads to unsoundness and present a correct one.

The method is applicable to arbitrary normal programs. If finitely failed trees are concerned then its semantics is given by Clark completion in 3-valued logic (and our approach is a proper extension of SLDNF-resolution). If infinite failed trees are allowed then we obtain a method for the well-founded semantics. In both cases soundness and completeness are proved.

1 Introduction

A standard way of dealing with negation in logic programming is negation as failure. Its major drawback is that the only answers it is able to give are "yes" and "no". In other words it cannot produce answer substitutions to negated queries. This results in floundering: for some queries negation as failure is inapplicable.

A generalization that avoids this drawback is presented in this paper. It is based on constructing finitely failed trees. The method is applicable to every normal program. For example, it makes it possible to obtain answers

$x = s(0), x = s^3(0), x = s^5(0), \ldots$

for the goal:

$\leftarrow \neg even(x)$.

and either of the following programs:

$even(0)$
$even(s^2(x)) \leftarrow even(x)$

[*]IPI PAN, Polish Academy of Sciences, Ordona 21, Pl – 01-237 Warszawa, Poland and IDA, Linköping University, S – 581 83 Linköping, Sweden; e-mail: wdr@ida.liu.se.

and

$$\begin{array}{l} even(0) \\ even(s(x)) \leftarrow \neg even(x) \end{array}$$

The basic idea of our method is that in order to answer a query $\leftarrow \neg A$ with respect to a program P a *fail answer* for $P \cup \{\leftarrow A\}$ is found. A fail answer is a substitution θ such that $P \cup \{\leftarrow A\theta\}$ finitely fails. Since $\neg A\theta$ is a logical consequence of the Clark completion comp(P) of the program, θ is given as an answer to $\leftarrow \neg A$. This approach was proposed by Małuszyński and Näslund in [MN89] (and by Shepherdson in [She89]). For definite ("positive") programs the definition of a goal finitely failing is obvious (finite SLD tree with no success leaf) and was used in [MN89]. A straightforward generalization suggested there for normal programs (i.e. programs with negation) is, unexpectedly, incorrect. Also the method of [She89] does not allow recursive usage of constructive negation. This problem is discussed and solved in section 3.

Our treatment of normal goals is similar to that in SLDNF-resolution [Llo87]. (SLDNF-resolution is a standard abstract computational mechanism for normal programs.) We introduce SLDFA-resolution (FA for fail answers) which is a proper extension of SLDNF-resolution. Its semantics is given by Clark completion. SLDFA-resolution is sound with respect to the Clark completion in the standard 2-valued logic. It is sound and complete for the 3-valued completion semantics of Kunen [Kun87]. There are indications that it can be implemented with a reasonable efficiency.

We also present a variant of our method that is sound and complete for the well-founded semantics [GRS91]. Up to our knowledge this is the first constructive negation approach for this semantics[1].

There are two other principal approaches to constructive negation. The first of them is represented by the work of Khabaza [Kha84], Chan [Cha88] and Przymusinski [Prz89a]. In this approach an answer to $\leftarrow \neg A$ is obtained by negating (the disjunction of) the answers to $\leftarrow A$. This approach fails if the number of answers is infinite. It also fails if the search space (eg. SLD-tree) is infinite (except for [Prz89a] where the perfect model semantics is used).

The second principal approach can be understood as using the Clark completion of a program instead of the program itself. The basic concept is, roughly speaking, that in a derivation step for $\leftarrow ..., \underline{\neg p(\bar{t})}, ...$ the completed definition of predicate p is used: literal $\neg p(\bar{t})$ is replaced by the negated right hand side of the completed definition (with an appropriate mgu applied). The resulted formula is then transformed into a form that facilitates a next derivation step. This approach is used in the work of Wallace [Wal87], Lugiez [Lug89], Chan and Stuckey [Cha89, Stu91], Sato and Motoyoshi [SM91] and Plaza [Pla92].

In the second method of Chan [Cha89, Stu91], when $\neg p(\bar{t})$ is selected in a current goal, $\leftarrow p(\bar{t})$ is first treated with what amounts to partial evaluation; the resulting set of clauses constitutes a definition of p whose completion is used in the derivation

[1]As the well-founded semantics is not recursively enumerable, using the word "constructive" may seem strange here. We follow the standard logic programming terminology where "constructive negation" means approaches to overcome the restrictions of negation as failure.

step. Another extension in [Cha89, Stu91] is that not only literals can be selected but also some other negated formulae.

The transformational approach to negation (Barbuti et al. [BMPT90], Sato and Tamaki [ST84]) does not use the program completion explicitly. Instead it introduces new predicates to a program in order to express the negative information. The idea is to construct a definite program that defines both the original and the new predicates. However in many cases the resulting program contains a construct equivalent to universal quantification which causes substantial implementation problems. The approach works only if the functor set of the underlying language (see Section 2.2) is finite. It seems to be impractical unless the set of functors of the language is small. (Auxiliary predicates are also used in the approach of [Pla92]).

All these approaches are strongly related to the completion semantics ([Prz89a] is an exception). In a sense they were constructed "with Clark completion in mind". None of them employs a notion of failure which is fundamental to the standard treatment of negation in logic programming. Our approach is different. It stems from a syntactical concept of a failed tree. As a result this approach is not "bound" to a unique semantics. A natural modification (allowing infinite failed trees) results in a constructive negation method for the well-founded semantics.

The paper is organized as follows. Section 2 contains some preliminary definitions. In section 3, SLDFA-resolution is introduced; the next two sections contain proofs of its soundness and completeness. Section 6 is devoted to computing fail answers by constructing finitely failed SLDFA-trees. It also contains comparisons with other approaches. Section 7 presents SLSFA-resolution, a constructive negation approach for the well-founded semantics.

We assume that the reader is familiar with basics of logic programming, including SLDNF-resolution and Clark completion semantics [Llo87]. To understand Section 5 on completeness of SLDFA-resolution, some familiarity with 3-valued completion semantics [Kun87] is preferable. Familiarity with the well-founded semantics would simplify reading Section 7.

2 Preliminaries

2.1 Notational conventions

When referring to syntactic objects of the underlying first order language(s), s, t, u will usually stand for terms, v, x, y for variables, a, b, c for constants and p, q for predicate symbols. Sub- and superscripts may be used if necessary. A bar 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 integer $n \geq 0$, $p(\overline{t})$ abbreviates $p(t_1, \ldots, t_m)$, where m is the arity of p.

An *equation* is a formula of the form $s = t$. Negation of an equation (a *disequation*) will be written as $s \neq t$. If $\overline{s}$ and $\overline{t}$ are term sequences of the same length n then by $\overline{s} = \overline{t}$ we denote the corresponding conjunction of equations, i.e. $s_1{=}t_1 \wedge \ldots \wedge s_n{=}t_n$. The symbol $=$ is used both as a syntactic symbol in equations and as an equality symbol of the metalanguage. We take care that this does not lead to ambiguity. We assume that $=$ does not occur in programs.

The set of free variables occurring in a syntactic construct (formula, term etc.) F is denoted by $FV(F)$. *Restriction* $F|_S$ of a formula F to a set S of variables is the formula $\exists x_1, \dots, x_n F$ where $\{x_1, \dots, x_n\} = FV(F) \setminus S$. Sometimes we do not distinguish between a sequence and the corresponding set (and for instance write $\overline{x}$ for $\{x_1, \dots, x_n\}$). As usual in logic programming we often use a comma instead of $\wedge$.

We use the standard logic programming terminology and definitions. However, normal programs are just called programs. We prefer term "selection rule" of [Apt90] to "computation rule" of [Llo87] for the function selecting a literal in a goal. The reader is referred to [Llo87] and to [Apt90] for definitions and explanations not present in this paper.

2.2 Semantics

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.

In this paper we are mainly interested in the semantics given by the Clark completion of a program. However in Section 6 the well-founded semantics is considered.

Roughly speaking the Clark completion comp(P) of a program P consists of "iff" counterparts of the clauses of P and some axioms for equality. The "iff" counterparts of the clauses of P can be defined as follows. Let p be a predicate symbol of $\mathcal{L}$, distinct from $=$. Let

$$Q = \{\, p(\overline{t}^i) \leftarrow \overline{L}^i \mid i = 1, \dots, m \,\}$$

be the set of clauses of P that begin with p. (If the i-th clause is unary, we assume that $\overline{L}^i$ is **true**). Let $\overline{y}$ be the variables occuring in Q. The completed definition of p in P is

$$\forall \overline{x} \quad p(\overline{x}) \leftrightarrow \bigvee_{i=1}^{m} \exists \overline{y}\, (\overline{x} = \overline{t}^i \wedge \overline{L}^i)$$

where $\overline{x}$ is a tuple of new variables (of the length equal to the arity of p). If $m = 0$ then the completed definition reduces to $\forall \overline{x}\; p(\overline{x}) \leftrightarrow$**false**.

Completion comp(P)of program P consists of the completed definitions (of the predicate symbols of $\mathcal{L}$) in P and of the axioms of the free equality theory CET.

The axioms consist of equality axioms

$x = x$,
$\overline{x} = \overline{y} \rightarrow f(\overline{x}) = f(\overline{y})$ for each functor f,
$\overline{x} = \overline{y} \rightarrow (p(\overline{x}) \rightarrow p(\overline{y}))$ for each predicate symbol p, including $=$,

and freeness axioms

$f(\overline{x}) = f(\overline{y}) \rightarrow \overline{x} = \overline{y}$ for each functor f,
$f(\overline{x}) \neq g(\overline{y})$ for each pair of distinct functors f, g (including constants),
$x \neq t$ for each variable x and term t such that x occurs in t and x and t syntactically differ.

If the underlying language $\mathcal{L}$ is finite then we add the (weak) domain closure axiom WDCA (sometimes called DCA). 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). The WDCA is defined as follows

$$\forall x(\bigvee_{f \text{ is a functor}} \exists y_1, \ldots, y_{\alpha(f)}(x = f(y_1, \ldots, y_{\alpha(f)})))$$

where $\alpha(f)$ is the arity of f, $\alpha(f) \geq 0$ [MMP88].

2.3 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.

Out of the approaches mentioned in the Introduction only [BMPT90] uses substitutions, but this results in an infinite number of answers in apparently simple cases (e.g. $\{eq(x,x)\leftarrow\} \cup \{\leftarrow\neg eq(x,y)\}$). The other approaches, in addition to substitutions (or equations) use disequations.

In order not to restrict ourselves to a particular form of answers, we will use arbitrary first order formulae built out of equality and disequality literals. Such formulae will be called *constraints* and denoted by $\theta, \sigma, \rho, \delta, \gamma$ (possibly with sub- and superscripts). Note that an idempotent substitution $\{x_1/t_1, \ldots, x_n/t_n\}$ corresponds to a constraint $x_1{=}t_1 \wedge \ldots \wedge x_n{=}t_n$.

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 $\models \sigma \leftrightarrow \theta$. Conjunction of θ and σ will often be denoted by $\theta{,}\sigma$ or by $\theta\sigma$ (as we use it instead of composition of substitutions).

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, see [She91], [CL89], [Mah88], [Cha88] and the references therein. First, CET is a complete theory (for every closed equality formula θ either θ or $\neg\theta$ is a logical consequence of the theory). This holds both for infinite and finite $\mathcal{L}$, due to WDCA added in the latter case. Then, there exist algorithms that reduce any constraint to an equivalent one in some disjunctive normal form. 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, some (maybe none) variables of the s_i's are universally quantified and $\overline{y}$ may contain any variables but x_i's.

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 SLDFA-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 SLDFA-resolution

This section introduces a method of deriving answers for normal queries and normal programs, called SLDFA-resolution. As an important contribution of this section we consider the explanation of the concept of (finite) failure in the context of constructive negation. After some preliminary definitions our motivations and some explanation of the method are given. Then SLDFA-resolution is presented formally followed by some examples. We conclude with discussing some variants of the notion of a finitely failed tree.

Basic notions of SLDFA-resolution are SLDFA-refutation and finitely failed SLDFA-tree. They may be seen as counterparts of the corresponding concepts of SLDNF-resolution [Llo87]. In fact, SLDFA-resolution is a proper extension of SLDNF-resolution. Every SLDNF-refutation is an SLDFA-refutation (with the difference that the latter uses constraints instead of substitutions); the same for finitely failed trees. This holds also for SLDNF-resolution with weak safeness condition (a non ground $\neg A$ fails iff A succeeds with empty substitution, $\neg A$ succeeds iff A fails).

SLDFA-resolution is also an extension of SLDNFS-resolution [She89]. The latter allows selecting non ground negated atoms and using fail answers for them, but only in derivations, not in failed trees. The negated atoms selected in such a tree are ground (or, in an extension of the method, succeed with empty answers).

Similarly to SLDNFS-resolution, SLDFA-resolution does not specify how to construct finitely failed trees. It only defines them. Constructing such trees is discussed later.

3.1 Preliminary definitions

We begin with a modification of the concept of a goal. An adjustment is needed due to usage of constraints instead of substitutions.

Definition 3.1 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 3.2 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 usually do not distinguish between an s-goal and its goal part.

Definition 3.3 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:

- $FV(G) \cap FV(C) = \emptyset$,
- (the goal part of) G' is $\leftarrow \theta\theta', \overline{L}, \overline{M}, \overline{L'}$ where θ' is the constraint $(t_1 = s_1 \wedge \cdots \wedge t_n = s_n)$.

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 (i.e. C with the variables renamed).

3.2 Informal explanation

Before introducing SLDFA-resolution, some rationale has to be given. In this section we discuss the notion of finite failure in the context of constructive negation. First we show unsoundness of a straightforward approach. Then we present an informal explanation of our definition of SLDFA finitely failed trees and conclude with an example. The formal definition is the subject of the next section.

Our approach is based on the notion of a fail answer. In order to construct derivations that may provide computed answers to goals, SLD-resolution is extended by "negative derivation steps". Roughly speaking, goal $\leftarrow \theta, \overline{M}$ is negatively derived from $\leftarrow \underline{\neg A}, \overline{M}$ iff θ is a fail answer for $\leftarrow A$. A fail answer to a goal $\leftarrow A$ is a constraint θ such that $\leftarrow \theta, A$ finitely fails. Thus the notion of finite failure of a goal is crucial in our approach.

It seems that finite failure of a goal G could have been defined as follows. Consider the tree built out of all the derivations for G under some selection rule. Let us call such a tree an *SLDNF*$^+$ tree [MN89]. If the tree is of finite depth and has no success leaves then G finitely fails.

As the following examples show, for normal programs such a definition leads to unsoundness.

Example 3.4 Consider a program $\{\, p \leftarrow \neg q;\ q \leftarrow q \,\}$. The SLDNF$^+$ tree for $\leftarrow p$ consists of one branch $\leftarrow p;\ \leftarrow \neg q$. Node $\leftarrow \neg q$ does not have a son as $\leftarrow q$ does not finitely fail. According to the above-mentioned definition $\leftarrow p$ fails. On the other hand $\leftarrow p$ should not be treated as failed because comp$(\{\, p \leftarrow \neg q;\ q \leftarrow q \,\}) \not\models \neg p$. □

Example 3.5 Let P be the program

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

The following is an SLDNF$^+$ tree for $\leftarrow p$:

$$\begin{tikzcd}
\leftarrow p \arrow[d, dash] \\
\leftarrow \underline{\neg q(x)}, r(x) \arrow[d, dash] \\
\leftarrow x \neq a, r(x)
\end{tikzcd}$$

Goal $\leftarrow x \neq a, r(x)$ is derived from $\leftarrow \underline{\neg q(x)}, r(x)$ since $\leftarrow x \neq a, q(x)$ finitely fails. The branch of the tree subsumes every derivation for $\leftarrow p$ because if $\leftarrow \theta, q(x)$ finitely fails then $x \neq a$ is more general than θ. According to the above-mentioned definition $\leftarrow p$ fails but $\neg p$ is not a logical consequence of comp(P) (as $\neg q(a), r(a)$ is true in some models of comp(P)). $\square$

This unsoundness should not be surprising. There are three possibilities: (1) comp$(P) \models p$, (2) comp$(P) \models \neg p$ and (3) neither (1) nor (2). Under an appropriate completeness assumption, lack of success in a finite depth SLDNF$^+$ tree for $\leftarrow p$ implies that (1) does not hold. But this only means that (2) or (3) holds; (2) is not implied.

In the next section we present a definition of finitely failed SLDFA-trees that gives sound results. The main difference from SLDNF- (and SLDNF$^+$-) failed trees is that the branches of our trees are *not necessarily SLDFA-derivations.* In SLDNF$^+$- and SLDNF-trees the answer(s) for $\leftarrow \neg A$ are used to construct the son(s) of $\leftarrow ..., \underline{\neg A}, ...$. In a failed SLDFA-tree, roughly speaking, negation of some answers for $\leftarrow A$ is used instead. However the two concepts of failed trees are still similar, in fact ours subsumes that of SLDNF-resolution.

Finitely failed SLDFA-trees may be intuitively seen as proofs of their roots w.r.t. the assumed semantics. (Remember that truth of a goal means falsity of its body). They are constructed as follows.

Consider a root G and assume that the selected literal is positive. Then G is treated as in SLDNF-resolution: to prove G it is enough to prove the goals that are positively derived from G. So these goals are made the children of G and failed subtrees for them are constructed.

Now assume that the selected literal is negative, say G is $\leftarrow \theta, \underline{\neg A}, \overline{L}$. The proof is by cases, we show that $\theta \rightarrow (A \vee \neg \overline{L})$. Let $\delta_1, \ldots, \delta_n$ ($n \geq 0$) be some computed answers for $\leftarrow A$. Thus each δ_j implies A (with respect to the completion semantics, i.e. comp$(P) \models \delta_j \rightarrow A$). If $\delta_1, \ldots, \delta_n$ "cover" θ, i.e. $\theta \rightarrow \delta_1 \vee \cdots \vee \delta_n$ (under the underlying equality theory) then we have proven G and no children of G need to be built. Otherwise θ is "split" into $\delta_1, \ldots, \delta_n$ and some $\sigma_1, \ldots, \sigma_m$ (i.e. $\sigma_1, \ldots, \sigma_m$ are found such that $\theta \rightarrow \delta_1 \vee \cdots \vee \delta_n \vee \sigma_1 \vee \cdots \vee \sigma_m$). Now the goals $\leftarrow \sigma_i, \overline{L}$ ($i = 1, \ldots m$) are made the children of G. By building a failed subtree for every such goal it is shown that each σ_i implies $\neg \overline{L}$. This completes the proof of G. Note that an unsound failed SLDNF$^+$-tree (such as those in Examples 3.4, 3.5) corresponds to an erroneous proof where not all the cases are proven due to $\theta \not\rightarrow \delta_1 \vee \cdots \vee \delta_n \vee \sigma_1 \vee \cdots \vee \sigma_m$ for any choice of $n, \delta_1, \ldots, \delta_n$.

Before coming to a formal definition of SLDFA-resolution, an example:

Example 3.6 (SLDFA finitely failed tree, SLDFA-refutation)

Consider a program

$p(a) \leftarrow p(a)$
$p(b)$
$q(b)$.

The following is a finitely failed tree for $\leftarrow \neg p(x), q(x)$:

$$\begin{array}{ll} \leftarrow \underline{\neg p(x)}, q(x) & \\ \quad\quad | & (x{=}b \text{ is a computed answer for } \leftarrow p(x) \text{ and } \sigma_1 = x{\neq}b) \\ \leftarrow x{\neq}b, q(x) & \\ & (\text{no clause is applicable}) \end{array}$$

The tree exemplifies a way of constructing the children of a node with a negative literal selected: to obtain $\sigma_1, \ldots, \sigma_m$ satisfying $\theta \rightarrow \delta_1 \vee \cdots \vee \delta_n \vee \sigma_1 \vee \cdots \vee \sigma_m$, put $m = 1$ and $\sigma_1 = \theta, \neg\delta_1, \ldots, \neg\delta_1$ (here $\theta = \textbf{true}$ and $\delta_1 = x{=}b$). Note that the tree is not an SLDNF$^+$-tree since $x{\neq}b$ is not a fail answer for $\leftarrow p(x)$.

Assume that a and b are not the only functors of the underlying language. The following is a successful SLDFA-derivation (a refutation):

$$\leftarrow \underline{\neg p(x)}, \neg q(x); \quad \leftarrow x{\neq}a, x{\neq}b, \neg q(x); \quad \leftarrow x{\neq}a, x{\neq}b$$

as $x{\neq}a, x{\neq}b$ is a fail answer for $\leftarrow p(x)$ and for $\leftarrow x{\neq}a, x{\neq}b, q(x)$. So $x{\neq}a, x{\neq}b$ is an SLDFA-computed answer for $\leftarrow \neg p(x), \neg q(x)$. $\square$

3.3 Definition

The definition of SLDFA resolution consists of mutually recursive Definitions 3.7, 3.8 and 3.9. To assure correctness of the definition, the concept of ranks is used, as in the definition of SLDNF-resolution [Llo87]. Ranks are natural 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 3.7 Let P be a program and $k \geq 0$. If $k > 0$ then assume that the notion of "negatively derived" is defined for rank k. An *SLDFA-refutation of rank k* 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 $FV(C) \cap FV(G_0, \ldots, G_{i-1}) = \emptyset$
- or $k > 0$ and G_i is rank k negatively derived from G_{i-1}.

The constraint $\theta|_{FV(G_0)}$ is a called a *rank k* SLDFA-*computed answer* for (the goal part of) G_0.

Definition 3.8 Let P be a program, $k > 0$ and assume that finitely failed trees of ranks $< k$ 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 k negatively derived* from G if, for some θ',

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

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

Definition 3.9 Let P be a program, $k \geq 0$ and G be a goal. Assume that SLDFA-refutations of ranks $< k$ are already defined. Then G *finitely fails* and is of *rank k* iff there exists a tree (called *rank k finitely failed SLDFA-tree*) satisfying the following conditions:

1. each node is an s-goal and the goal part of the root node is G;
2. the tree is finite;
3. 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 child of H that is positively derived from H using a variant of C;
4. a node H with a negative literal selected, of the form

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

   has children

   $$\leftarrow\sigma_1, \overline{L}, \overline{L'};\ \ldots;\ \leftarrow\sigma_m, \overline{L}, \overline{L'} \quad \text{where} \quad m \geq 0$$

   provided that there exist

   $$\delta_1, \ldots, \delta_n \quad \text{where} \quad n \geq 0$$

   that are (some of the) SLDFA-computed answers for $\leftarrow\theta, A$ of rank $< k$ such that

   $$\text{CET} \models \theta \rightarrow \delta_1 \vee \cdots \vee \delta_n \vee \sigma_1 \vee \cdots \vee \sigma_m.$$
5. no node of the tree is of the form $\leftarrow\theta$.

The condition in part 4 of the definition is called safeness condition. (Note that it is a generalization of that of SLDNF-resolution). A node H satisfying it will be called *correct*. A tree satisfying the definition without parts 2 and 5 will be called an SLDFA *pre-failed* tree.

3.4 Comments and examples

This section augments the preceding definitions with some explanations and examples. Then we discuss some technical properties of SLDFA-refutations and failed trees.

Note that a refutation with no negative literal selected is of rank 0. A refutation (failed goal, failed tree) of rank k is also of any higher rank.

A definition of an SLDFA-*derivation* can be obtained from Definition 3.7 by removing the requirements for the form of the last goal and of the finiteness of the sequence. A derivation will be called *complete* if it is not a prefix of any other derivation. There exist infinite derivations that do not have a rank. However every finite derivation is of some (finite) rank.

Note that a branch in a rank k failed tree may not be an SLDFA-derivation, due to the children of the nodes with negative literals selected. These children are not necessarily negatively derived from their fathers. The branches are in some sense more general than derivations: if H has children $\leftarrow \sigma_1, ...; \ldots ; \leftarrow \sigma_m, ...$ and, on the other hand, $\leftarrow \sigma, ...$ can be negatively derived from H then $\sigma \rightarrow \sigma_1 \vee \cdots \vee \sigma_m$.

The children of a node $\leftarrow \theta, \ldots, \underline{\neg A}, \ldots$ can be constructed by negating (some) answers $\delta_1, \ldots, \delta_n$ for $\leftarrow \theta, A$. More precisely, it is sufficient to take any $m \geq 0$ and any $\sigma_1, \ldots, \sigma_m$ such that CET $\models (\sigma_1 \vee \ldots \vee \sigma_m) \leftrightarrow \theta \wedge \neg(\delta_1 \vee \ldots \vee \delta_n)$. The children obtained in such a way are in a sense minimally general. (A standard way of computing $m, \sigma_1, \ldots, \sigma_m$ is by transforming $\theta \wedge \neg(\delta_1 \vee \ldots \vee \delta_n)$ to a disjunctive normal form).

As in SLDNF-resolution, finitely failed (pre-failed) tree may be not unique for a given goal. This is due to selecting literals in the goals but also to choosing the answers δ_j's and the constraints σ_i's when a negative literal is selected. Obviously, improper choosing of these answers ("too few answers") may result in nonexistence of a failed subtree for some child $\leftarrow \sigma_i, ...$ due to σ_i being too general. On the other hand, some finite set of answers is sufficient as the completeness theorem shows.

Note that every finitely failed SLDNF- (and SLDNFS-) tree is also an SLDFA-tree of the same rank (modulo using equalities instead of substitutions). The same holds for refutations and derivations.

Example 3.10 It is easy to check that Example 3.6 is compatible with the definition of SLDFA-resolution and that a finitely failed tree for $\leftarrow p$ does not exist for the programs from Examples 3.4 and 3.5. $\square$

Example 3.11 Remark: for convenience, some constraints in our examples may be replaced by equivalent ones.

Consider the program:

$even(0)$
$even(s(x)) \leftarrow \neg even(x).$

Then for $i = 1, 2, \ldots$ and for any constraint θ (including **true**) satisfying the conditions below the tree consisting of a single branch

$$\begin{array}{c} \leftarrow \theta, x{=}s^{2i-1}(0), even(x) \\ | \\ \leftarrow \theta, x{=}s^{2i-1}(0), x{=}s(x'), \neg even(x') \end{array}$$

is a finitely failed tree of rank $2i-1$ and

$$\leftarrow \theta, even(x); \quad \leftarrow \theta, x{=}s(x'), \neg even(x'); \quad \leftarrow \theta, x{=}s(x'), x'{=}s^{2i-1}(0)$$

is a refutation of rank $2i$ with the computed answer $\theta, x{=}s^{2i}(0)$. The conditions on θ are as follows. For proper standardization apart it is required that $x' \notin FV(\theta)$. For the satisfiability of the constraints in the goals, $\theta, x{=}s^{2i-1}(0)$ (respectively $\theta, x{=}s^{2i}(0)$) has to be satisfiable. $\square$

Example 3.12 Consider the following program
$r \leftarrow \neg p(x), \neg q(x).$
$p(x) \leftarrow p(x).$
$p(a).$
$q(a) \leftarrow q(a).$
$q(x) \leftarrow \neg s(x).$
$s(a).$
and assume that a is not the only functor of the underlying language. With respect to comp(P), $p(x)$ is true for $x = a$ and $q(x)$ for $x \neq a$; r is false.

The following is a finitely failed tree for $\leftarrow r$ of rank 2:

$$\begin{array}{cl}
\leftarrow r & \\
| & \\
\leftarrow \underline{\neg p(x)}, \neg q(x) & \\
| & \quad (\leftarrow p(x) \text{ succeeds with } x = a) \\
\leftarrow x \neq a, \neg q(x). &
\end{array}$$

The last node does not have children since $x{\neq}a$ is a computed answer for $\leftarrow x{\neq}a, q(x)$ obtained from the following refutation of rank 1: $\leftarrow x{\neq}a, q(x)$; $\leftarrow x{\neq}a, \neg s(x)$; $\leftarrow x{\neq}a$. $\square$

Now we mention some, rather technical, properties of SLDFA-resolution. Note that if $\leftarrow \sigma, \overline{M}$ is a goal in a derivation beginning with $\leftarrow \theta, \overline{L}$ then σ is of the form $\theta\theta'$ where $FV(\theta) \cap FV(\theta', \overline{M}) \subseteq FV(\overline{L})$. Any computed answer for $\leftarrow \theta, \overline{L}$ is of the form $(\theta\theta')|_V$ (where $V = FV(\leftarrow \theta, \overline{L})$) which is equivalent to $\theta(\theta'|_V)$ and to $\theta(\theta'|_{FV(\overline{L})})$.

Let $x \notin FV(\overline{L})$. It can be proven that $\theta\delta$ is (equivalent to) a computed answer for $\leftarrow \theta, \overline{L}$ iff $(\exists x\theta)\delta$ is (equivalent to) a computed answer for $\leftarrow (\exists x\theta), \overline{L}$ and that $\leftarrow \theta, \overline{L}$ finitely fails iff $\leftarrow (\exists x\theta), \overline{L}$ finitely fails. This makes it possible to use simpler constraints at lower ranks: if $\neg A$ is selected in $\leftarrow \theta, \ldots$ then it is sufficient to compute fail answers and answers for $\leftarrow \theta|_{FV(A)}, A$ instead of $\leftarrow \theta, A$. For instance, in the case of the derivation in Example 3.11 this means computing a fail answer for $\leftarrow even(x')$ instead of $\leftarrow \theta, x{=}s(x'), even(x')$.

The following two properties are also useful in simplifying the constraints in derivations and (pre-) failed trees.

Consider a node $\leftarrow \sigma, \overline{M}$ of an SLDFA pre-failed tree and a variable x occurring free in σ and not occurring in $\overline{M}$. By the previous property, $\leftarrow \sigma, \overline{M}$ can be replaced in the tree by $\leftarrow \exists x\sigma, \overline{M}$ together with the corresponding replacement for its descendant nodes ($\leftarrow \sigma', \overline{M'}$ by $\leftarrow \exists x\sigma', \overline{M'}$.) Obviously, the obtained tree is finite and without a “success” node iff the original tree is finitely failed. (Usually the obtained tree is not an SLDFA pre-failed tree but its modified subtree is).

For example, the pre-failed tree for $\{\, p(s(x)) {\leftarrow} p(x) \,\} \cup \{\, {\leftarrow} p(x) \,\}$ has nodes ${\leftarrow}\, x{=}s(x_1), \ldots, x_{i-1}{=}s(x_i), p(x_i)$, for $i > 0$. Multiple application of the last property makes it possible to replace them by ${\leftarrow}\, x{=}s^i(x_i), p(x_i)$ respectively.

Similarly, let ${\leftarrow}\, \theta\theta', \overline{M}$ be a goal in a refutation for ${\leftarrow}\, \theta, \overline{L}$ with the last goal ${\leftarrow} \theta\theta'\theta''$. Let $x \in FV(\theta\theta')$ and $x \notin FV(\theta, \overline{L}) \cup FV(\overline{M})$. Then ${\leftarrow} \theta\theta', \overline{M}$ can be replaced by ${\leftarrow}\, \exists x(\theta\theta'), \overline{M}$ (together with the corresponding replacement of its successors) without changing the answer. (The new answer is $(\exists x(\theta\theta')\theta'')|_{FV(\theta,\overline{L})}$ which is equivalent to $(\theta\theta'\theta'')|_{FV(\theta,\overline{L})}$.)

SLDFA-resolution also works for goals (and clauses) in the style of [Cha89] where not only atoms can be negated but also formulae of the form $\exists\overline{x}(\sigma, \overline{M})$. Extension (along the lines of [Llo87]) to clause (and goal) bodies being arbitrary first order formulae seems obvious.

3.5 Variants of SLDFA-resolution

Here we present two variants of the notion of finitely failed tree.

1. It is possible to introduce a sound notion of a failed tree where branches *are* derivations. However the safeness condition (from Def. 3.9.4) is still necessary for soundness. So the corresponding definition is obtained from Definition 3.9 by adding a requirement that a child of a node with a negative literal selected is negatively derived from its father. This variant is incomplete; for the program from Example 3.12 there does not exist such a finitely failed tree. The version from Definition 3.9 is also simpler because the failed trees refer only to refutations of a lower rank and not to failed trees.

2. The children of a node H with a negative literal selected in Definition 3.9 may be of the form
$$\leftarrow \sigma_j, \overline{L}, \neg A, \overline{L'}$$
($\neg A$ is not removed) for $j = 1, \ldots, m$. This modification may simplify constructing failed trees. If a tree from Definition 3.9 is built top-down then all the necessary answers for ${\leftarrow}\, \theta, A$ are to be found before building the children of ${\leftarrow}\theta, ..., \underline{\neg A}, ...$. This modification makes it possible to delay computing some of these answers.

   Note that with this modification part 4 of Definition 3.9 may be simplified by using only one answer δ_1 for ${\leftarrow}\, \theta, A$ $(n = 1)$.

Soundness of these variants follows from the proof of soundness of SLDFA-resolution, see below (with obvious minor modifications for case 2).

4 Soundness of SLDFA resolution

In this section we prove soundness of SLDFA-resolution. We begin with a lemma that is an extension of Lemma 15.3 of [Llo87].

Lemma 4.1 Let P be a program and G an s-goal with a positive literal selected. Let $\{G_1, \ldots, G_n\}$, where $n \geq 0$, be the set of goals positively derived from G. Then

$$\text{comp}(P) \models G \leftrightarrow \forall \overline{x}(G_1 \wedge \ldots \wedge G_n).$$

where $\overline{x} = FV(G_1 \wedge \ldots \wedge G_n) \setminus FV(G)$.

(Hence $\text{comp}(P) \models \forall G \leftrightarrow \forall(G_1 \wedge \ldots \wedge G_n)$. If $n = 0$ then $\text{comp}(P) \models G$).

PROOF

Without lack of generality we assume that the first atom, say $p(\overline{s})$, is selected in G. Let the set Q of clauses of P that begin with p be as in section 2.2. We may assume that the variables of Q are renamed in the same way as they had been renamed to derive $G_1, \ldots, G_n$ from G.

Let G be $\leftarrow \theta, \underline{p(\overline{s})}, \overline{M}$ and $\overline{y}$ be the variables of Q. Then, by the definition of the completed definition of p in P, G is equivalent (with respect to comp(P)) to $\leftarrow [\theta \wedge (\bigvee_{i=1}^{m} \exists \overline{y}(\overline{s}{=}\overline{t}^i, \overline{L}^i)) \wedge \overline{M}]$ and to

$$\forall \overline{y} \bigwedge_{i=1}^{m} \leftarrow \theta, \overline{s}{=}\overline{t}^i, \overline{L}^i, \overline{M}.$$

as variables $\overline{y}$ are distinct from those occurring in G. By removing from the conjunction every element in which $\theta, \overline{s}{=}\overline{t}^i$ is unsatisfiable we obtain an equivalent formula $\forall \overline{y}(G_1 \wedge \ldots \wedge G_n)$. Removing from $\overline{y}$ the variables that do not occur in $G_1, \ldots, G_n$ results in $\forall \overline{x}(G_1 \wedge \ldots \wedge G_n)$. □

Theorem 4.2 (Soundness of SLDFA resolution)

Let P be a program and $G = \leftarrow \theta, \overline{L}$ a goal.

1° If δ is an SLDFA computed answer for G then

$$\text{comp}(P) \models \delta \rightarrow \overline{L}$$

(and also $\text{comp}(P) \models \delta \rightarrow \theta, \overline{L}$).

2° If G finitely fails then

$$\text{comp}(P) \models G$$

(or, equivalently, $\text{comp}(P) \models \theta \rightarrow \neg \overline{L}$).

PROOF

Induction on the rank. Let $k \geq 0$. If $k \neq 0$ then assume that 1° holds for every G and δ obtained from a refutation of rank $< k$ and that 2° holds for every G of rank $< k$. We show that 1° and 2° hold for rank k.

To simplify the presentation of the proof we write the selected literal as the first in the goal. The necessary generalization is trivial.

To prove 1° consider a rank k refutation for G with an answer δ. Its last goal is then $\leftarrow \delta'$, where δ is δ' restricted to the free variables of G. Let

$$\leftarrow \sigma, p(\overline{t}), \overline{H}$$

(where H is a possibly empty sequence of literals) be an element of the refutation with a positive literal selected. Then there exists a variant $p(\bar{u}) \leftarrow \overline{B}$ of a clause of P such that $\leftarrow \sigma, \bar{u}{=}\bar{t}, \overline{B}, \overline{H}$ is the next goal in the refutation. Note that $P \models \overline{B} \rightarrow p(\bar{u})$ and CET $\models \bar{u}{=}\bar{t}, p(\bar{u}) \rightarrow p(\bar{t})$. Hence

$$\mathrm{comp}(P) \models (\sigma, \bar{u}{=}\bar{t}, \overline{B}, \overline{H}) \rightarrow (\sigma, p(\bar{t}), \overline{H}) \tag{1}$$

Consider a "negative SLDFA step"; let

$$\leftarrow \sigma, \neg A, \overline{H}$$

be an element of the refutation with a negative literal selected. Then the next goal in the refutation is $\leftarrow \sigma\rho, \overline{H}$ and $\leftarrow \sigma\rho, A$ finitely fails and is of rank $< k$. Hence by the inductive assumption on 2° for lower ranks $\mathrm{comp}(P) \models \sigma\rho \rightarrow \neg A$. Thus

$$\mathrm{comp}(P) \models \sigma\rho, \overline{H} \rightarrow \sigma, \neg A, \overline{H} \tag{2}$$

From (1) and (2) it follows that if $\leftarrow Q_{i+1}$ is (positively or rank k negatively) derived from $\leftarrow Q_i$ then $\mathrm{comp}(P) \models Q_{i+1} \rightarrow Q_i$. Hence by simple induction on the length of the refutation we obtain $\mathrm{comp}(P) \models \delta' \rightarrow \theta, \overline{L}$ and

$$\mathrm{comp}(P) \models \delta \rightarrow \theta, \overline{L}$$

since δ is δ' with existential quantification of variables that do not occur in $\theta, \overline{L}$. Hence 1° holds for refutations of rank k, QED(1°).

2° will be proved by induction on the depth of the tree which is referred to in the definition of finite failure. Let $l \geq 0$. If $l > 0$ then assume that 2° holds for every finitely failed H of rank k such that there exists a finitely failed SLDFA tree for H of depth $< l$.

Consider a finitely failed SLDFA tree for G of depth l. There are two possibilities.

1. A positive literal is selected in G. By lemma 4.1:

   If G has no children then $\mathrm{comp}(P) \models G$, QED.

   If G has children $G_1, \ldots, G_n$ then $\mathrm{comp}(P) \models \forall G \leftrightarrow \forall(G_1 \wedge \ldots \wedge G_n)$. By the inductive assumption for smaller depths, for every $i = 1, \ldots, n$, $\mathrm{comp}(P) \models \forall G_i$. Hence $\mathrm{comp}(P) \models G$, QED.

2. A negative literal is selected in G. Let $G = \leftarrow \theta, \neg A, \overline{H}$. There exist bindings $\delta_1, \ldots, \delta_m, \sigma_1, \ldots, \sigma_n$ where $m, n \geq 0$ such that

   (a)
   $$\mathrm{CET} \models \theta \rightarrow \delta_1 \vee \ldots \vee \delta_m \vee \sigma_1 \vee \ldots \vee \sigma_n \tag{3}$$

   (b) G has n children $\leftarrow \sigma_j, \overline{H}$, $j = 1, \ldots, n$,

   (c) for every $j = 1, \ldots, m$, δ_j is an answer for $\leftarrow \theta, A$ obtained from a refutation of rank $< k$.

By 2c and the inductive assumption on 1° for lower ranks, for every $j = 1, \ldots, m$, $\text{comp}(P) \models \delta_j \rightarrow A$. Thus

$$\text{comp}(P) \models \delta_j \rightarrow \neg(\neg A, \overline{H}) \tag{4}$$

By 2b and the inductive assumption on 2° for trees of lower depth, for every $j = 1, \ldots, n$, $\text{comp}(P) \models \sigma_j \rightarrow \neg(\overline{H})$ Thus

$$\text{comp}(P) \models \sigma_j \rightarrow \neg(\neg A, \overline{H}) \tag{5}$$

Now, by (3), (4), (5),

$$\text{comp}(P) \models \theta \rightarrow \neg(\neg A, \overline{H})$$

which proves 2° for goals of rank k and trees of depth l, QED.

□

5 Completeness of SLDFA-resolution

It is a well-known fact that SLDNF-resolution is incomplete. Introducing SLDFA-resolution removes the problem of floundering. However, floundering is not the only reason for incompleteness w.r.t. Clark completion. Possible inconsistency of the program completion and, using the terminology of [Cav88], the problem with excluded middle remain. These two problems do not occur with strict programs. For arbitrary programs they can be solved by using three-valued logic or by introducing another form of completion, the strict completion [DM91] (called double completion in [Wal93]).

In this section we use 3-valued completion semantics introduced by Kunen [Kun87]. The meaning of a program is given by logical consequences of its completion in a 3-valued logic [Fit85]. The third truth value, added to **t** and **f**, is **u** ("undefined" or "unknown"). The logical connectives (but $\leftrightarrow$) are interpreted as in Kleene's logic. (Their truth functions are extensions of those of the standard logic, obtained by setting $\neg\mathbf{u}$ to be **u**, $\mathbf{u} \vee \mathbf{t}$ and $\mathbf{t} \vee \mathbf{u}$ to be **t**, $\mathbf{u} \vee \mathbf{f}$, $\mathbf{u} \vee \mathbf{u}$ and $\mathbf{f} \vee \mathbf{u}$ to be **u**. Conjunction may be defined by $\alpha \wedge \beta := \neg(\neg\alpha \vee \neg\beta)$ and implication by $\alpha \rightarrow \beta := \neg\alpha \vee \beta$). $F \leftrightarrow F'$ is true if both F and F' have the same truth value, otherwise it is false. The existential quantifier is treated as an infinite disjunction ($\exists F$ is **t** in an interpretation if F is **t** in this interpretation for some variable valuation; it is **f** if for all variable valuations F is **f**; otherwise it is **u**). Similarly the universal quantifier is treated as an infinite conjunction. The notions of a model and of logical consequence (denoted by $\models_3$) are obvious modification of those of the standard logic. However it is assumed that $=$ is always 2-valued (i.e. never takes value **u**).

We prove completeness of SLDFA-resolution and its independence from selection rule. Selection rule is a function selecting a literal in a goal. For generality we assume that the selected literal in a goal in a derivation (branch of a pre-failed tree) is a function of the preceding part of the derivation (branch) [Apt90]. A selection rule is *fair* iff in any infinite branch of a pre-failed tree every literal is eventually selected. (Obviously a fairness requirement is not needed for derivations).

In the case of SLD-resolution the completeness property states that if θ is a correct answer substitution then there exists a computed answer that is more general that θ. This does not hold for SLDFA-resolution. (Neither it holds for SLD-resolution when the completion semantics with WDCA is employed [MMP88]). Let P be $\{\, p(a)\leftarrow;\ p(x)\leftarrow\neg q(x);\ q(a)\leftarrow\,\}$. Then comp$(P) \models p(x)$ but **true** is not a computed answer to $\leftarrow p(x)$. (The answers are $x{=}a$ and $x{\neq}a$). Hence for our purposes a completeness property has to be formulated in another way. A correct answer is to be "covered" by a finite set of computed answers.

Theorem 5.1 (Completeness of SLDFA-resolution and independence from selection rule)

Let P be a program, $G = \leftarrow\theta, \overline{L}$ a goal. Then for any fair selection rule

- if comp$(P) \models_3 G$ then G finitely fails and
- if there exists δ such that comp$(P) \models_3 \delta \to \theta, \overline{L}$ then there exist SLDFA-computed answers $\delta_1, \ldots, \delta_n$ for G such that CET $\models \delta \to \delta_1, \vee \ldots \vee \delta_n$.

The proof is preceded by a technical lemma. In what follows the prefix SLDFA- will be omitted.

Lemma 5.2 Assume that $\leftarrow\theta, \overline{L}$ finitely fails and is of rank k. If CET $\models \sigma \to \theta$ then $\leftarrow\sigma, \overline{L}$ finitely fails and is of rank k.

Assume that δ is a computed answer for $\leftarrow\theta, \overline{L}$. For any constraint σ, if $\sigma\delta$ is satisfiable then $\sigma\delta$ is (equivalent to) a computed answer for $\leftarrow\sigma\theta, \overline{L}$.

PROOF

Without loss of generality it may be assumed that any variable occurring in a failed tree (respectively refutation) for $\leftarrow\theta, \overline{L}$ and not occurring in $\leftarrow\theta, \overline{L}$ does not occur in σ. (Otherwise the variables in the tree (refutation) may be renamed).

By induction on the rank we obtain the following. Adding σ to every node of a finitely failed tree for $\leftarrow\theta, \overline{L}$ results in a finitely failed tree with the root $\leftarrow\sigma\theta, \overline{L}$ (some nodes may have been removed as the conjunction of σ and the constraint in the node may be unsatisfiable). Adding σ to every goal of the derivation that gives the answer δ results in a derivation that gives an answer equivalent to $\sigma\delta$. □

From the proof it follows that the same literals are selected in any pair of corresponding goals of the corresponding failed trees for $\leftarrow\theta, \overline{L}$ and $\leftarrow\sigma, \overline{L}$ (respectively of the corresponding refutations for $\leftarrow\theta, \overline{L}$ and $\leftarrow\sigma\theta, \overline{L}$).

PROOF (of the theorem)

We use a characterization of logical consequences of comp(P) given by Shepherdson [She91]. For any formula F without $\leftrightarrow$ and for any natural number n a pair of constraints $\mathbf{T}_n(F)$ and $\mathbf{F}_n(F)$ is defined. The definition is by induction on n and on the structure of F. If F is an equation $s{=}t$ then $\mathbf{T}_n(F)$ is $s{=}t$ and $\mathbf{F}_n(F)$ is $s{\neq}t$. Let F be an atom $p(\overline{s})$. Let $p(\overline{t}^i) \leftarrow \overline{M}^i$ $(i = 1, \ldots, l)$ be the clauses of P with the heads unifiable with $p(\overline{s})$ (and with the variables standardized apart). Both $\mathbf{T}_0(F)$ and $\mathbf{F}_0(F)$ are **false** and

$$\begin{array}{lll} \mathbf{T}_{n+1}(p(\overline{s})) & \text{is} & \bigvee_{i=1}^{l} \exists \overline{y}^i (\overline{s}{=}\overline{t}^i \wedge \mathbf{T}_n(\overline{M}^i)) \\ \mathbf{F}_{n+1}(p(\overline{s})) & \text{is} & \bigwedge_{i=1}^{l} \forall \overline{y}^i (\overline{s}{=}\overline{t}^i \rightarrow \mathbf{F}_n(\overline{M}^i)) \end{array}$$

where $\overline{y}^i$ are the free variables of $p(\overline{t}^i) \leftarrow \overline{M}^i$.

For non atomic formulae $\mathbf{T}_n(F)$ and $\mathbf{F}_n(F)$ are defined in an obvious way:

$$\begin{array}{rclrcl} \mathbf{T}_n(\neg F) & = & \mathbf{F}_n(F) & \mathbf{F}_n(\neg F) & = & \mathbf{T}_n(F) \\ \mathbf{T}_n(F \wedge G) & = & \mathbf{T}_n(F) \wedge \mathbf{T}_n(G) & \mathbf{F}_n(F \wedge G) & = & \mathbf{F}_n(F) \vee \mathbf{F}_n(G) \\ \mathbf{T}_n(F \vee G) & = & \mathbf{T}_n(F) \vee \mathbf{T}_n(G) & \mathbf{F}_n(F \vee G) & = & \mathbf{F}_n(F) \wedge \mathbf{F}_n(G) \\ \mathbf{T}_n(F \rightarrow G) & = & \mathbf{F}_n(F) \vee \mathbf{T}_n(G) & \mathbf{F}_n(F \rightarrow G) & = & \mathbf{T}_n(F) \wedge \mathbf{F}_n(G) \\ \mathbf{T}_n(\forall x F) & = & \forall x \mathbf{T}_n(F) & \mathbf{F}_n(\forall x F) & = & \exists x \mathbf{F}_n(F) \end{array}$$

Note that for a constraint θ, $\mathbf{T}_n(\theta)$ and $\mathbf{F}_n(\theta)$ are just (equivalent to) θ and $\neg\theta$ respectively. If F is a closed formula so are $\mathbf{T}_n(F)$ and $\mathbf{F}_n(F)$.

The characterization of Kunen semantics is given by the following property established by Theorem 6 and Lemma 4.1 of [She91]. For a closed F

$$\begin{array}{rcll} \text{comp}(P) \models_3 F & \text{iff} & \text{CET} \models \mathbf{T}_n(F) & \text{for some } n \\ \text{comp}(P) \models_3 \neg F & \text{iff} & \text{CET} \models \mathbf{F}_n(F) & \text{for some } n \end{array}$$

(Lemma 4.1 provides a characterization of the 3-valued immediate consequence operator [Fit85] in terms of $\mathbf{T}_n$ and $\mathbf{F}_n$. Theorem 6 relates the immediate consequence operator and 3-valued logical consequences of comp(P); it is a generalization of Theorem 6.3 of [Kun87] for languages other than those with infinitely many function symbols of all arities.)

As $\mathbf{T}_n(\forall(\leftarrow\theta, \overline{L}))$ is $\forall(\theta \rightarrow \mathbf{F}_n(\overline{L}))$, comp$(P) \models_3 \leftarrow\theta, \overline{L}$ implies that $\mathbf{F}_n(\overline{L})$ is more general than θ for some n. Thus, by Lemma 5.2 to prove the first clause of the theorem it is sufficient to show that
(1) for any fair selection rule, n and $\overline{L}$ the goal $\leftarrow\mathbf{F}_n(\overline{L}), \overline{L}$ finitely fails (or $\mathbf{F}_n(\overline{L})$ is unsatisfiable).

As $\mathbf{T}_n(\forall(\delta \rightarrow \theta, \overline{L}))$ is $\forall(\delta \rightarrow \theta, \mathbf{T}_n(\overline{L}))$, to prove the second clause of the theorem it is sufficient to prove that
(2) for any fair selection rule, n and $\overline{L}$ there exist computed answers $\delta_1, \ldots, \delta_k$ ($k \geq 0$) for $\leftarrow\overline{L}$ such that CET $\models \mathbf{T}_n(\overline{L}) \rightarrow \delta_1 \vee \ldots \vee \delta_k$ (because $\theta\delta_i$, if satisfiable, is a computed answer for $\leftarrow\theta, \overline{L}$ by Lemma 5.2, $i = 1, \ldots, k$).

Now we introduce some auxiliary notation.

$$\begin{array}{rcl} \mathbf{F}_{(n_1,\ldots,n_m)}(L_1, \ldots, L_m) & = & \mathbf{F}_{n_1}(L_1) \vee \cdots \vee \mathbf{F}_{n_m}(L_m) \\ \mathbf{T}_{(n_1,\ldots,n_m)}(L_1, \ldots, L_m) & = & \mathbf{T}_{n_1}(L_1) \wedge \cdots \wedge \mathbf{T}_{n_m}(L_m) \end{array}$$

Obviously $\mathbf{F}_n(\overline{L}) = \mathbf{F}_{(n,\ldots,n)}(\overline{L})$ and $\mathbf{T}_n(\overline{L}) = \mathbf{T}_{(n,\ldots,n)}(\overline{L})$.

To prove (1) and (2) we prove a pair of stronger properties. For any fair selection rule, for any $\overline{L}$ and for any nonempty sequence of natural numbers $\overline{n}$ of the same length as $\overline{L}$
(1') $G' = \leftarrow \mathbf{F}_{\overline{n}}(\overline{L}), \overline{L}, \overline{M}$ finitely fails for any $\overline{M}$ (or $\mathbf{F}_{\overline{n}}(\overline{L})$ is unsatisfiable)
and
(2') There exist computed answers $\delta_1, \ldots, \delta_k$ for $\leftarrow \overline{L}$ such that CET $\models \mathbf{T}_{\overline{n}}(\overline{L}) \rightarrow \delta_1 \vee \ldots \vee \delta_k$.

Notice that (1') implies that for any fair selection rule $\leftarrow \mathbf{F}_{\overline{n}}(\overline{L}), \overline{N}$ finitely fails for any permutation $\overline{N}$ of $\overline{L}, \overline{M}$.

The proof is by induction on $\overline{n}$ with respect to a kind of multiset ordering defined as follows [Kun89]. Let pre-ordering $\leq$ be the least transitive and reflexive relation such that $\overline{m} \leq \overline{n}$ whenever $\overline{m}$ is a subsequence of a permutation of $\overline{n}$, or $\overline{m}$ results from replacing some component n_i of $\overline{n}$ by an arbitrary finite sequence of numbers less than n_i. The induction uses the corresponding strict partial ordering $<$; so $\overline{m} < \overline{n}$ iff $\overline{m} \leq \overline{n}$ and $\overline{m}$ is not a permutation of $\overline{n}$.

The base case, $\overline{n} = (0)$, is trivial as $\mathbf{F}_{(0)}(\overline{L})$ and $\mathbf{T}_{(0)}(\overline{L})$ are **false**. Consider some $\overline{n}$ and assume that (1') and (2') hold for any $\overline{m} < \overline{n}$. Let $\overline{n} = (n_1, \ldots, n_m)$, $\overline{n}' = (n_2, \ldots, n_m)$, $\overline{L} = L_1, \ldots, L_m$ and $\overline{L'} = L_2, \ldots, L_m$.

We prove (2') first. To simplify the presentation of the proof we assume that L_1 is selected in $\leftarrow\overline{L}$. For any other selected literal the proof is identical. For $n_1 = 0$ $\mathbf{T}_{\overline{n}}(\overline{L})$ is false and (2') obviously holds. Let $n_1 > 0$. There are two cases.

1. $L_1 = p(\overline{s})$. Let $p(\overline{t}^i) \leftarrow \overline{M}^i$ ($i = 1, \ldots, l$) be the clauses of P with the heads unifiable with $p(\overline{s})$ (and with the variables standardized apart). For $i = 1, \ldots, l$, goal $\leftarrow \overline{s}{=}\overline{t}^i, \overline{M}^i, \overline{L}'$ is positively derived from $\leftarrow\overline{L}$. By the inductive assumption, for any fair selection rule there are answers $\delta^i_1, \ldots, \delta^i_{k_i}$ for $\leftarrow \overline{M}^i, \overline{L}'$ such that CET $\models \mathbf{T}_{\overline{m}}(\overline{M}^i, \overline{L}') \rightarrow \delta^i_1 \vee \ldots \vee \delta^i_{k_i}$ where $\overline{m} = (n_1 - 1, \ldots, n_1 - 1, n_2, \ldots, n_m)$.

   By Lemma 5.2, for $j = 1, \ldots, k_i$ constraint $\overline{s}{=}\overline{t}^i, \delta^i_j$ is a computed answer for $\leftarrow \overline{s}{=}\overline{t}^i, \overline{M}^i, \overline{L}'$. We may assume that it is obtained from a refutation that uses clause variants which do not contain any variable from $\overline{L}$. Adding $\leftarrow\overline{L}$ in front of the refutation results in a refutation for $\leftarrow\overline{L}$ with the computed answer $(\overline{s}{=}\overline{t}^i, \delta^i_j)|_{FV(\overline{L})}$.

   By the definition, $\mathbf{T}_{n_1}(p(\overline{s}))$ is $\bigvee_{i=1}^{l} (\overline{s} = \overline{t}^i \wedge \mathbf{T}_{n_1 - 1}(\overline{M}^i))|_{FV(\overline{s})}$. Thus $\mathbf{T}_{\overline{n}}(\overline{L})$ is (equivalent to) $\bigvee_{i=1}^{l} (\overline{s} = \overline{t}^i \wedge \mathbf{T}_{\overline{m}}(\overline{M}^i, \overline{L}'))|_{FV(\overline{L})}$. Hence, $\mathbf{T}_{\overline{n}}(\overline{L})$ implies $\bigvee_{i=1}^{l} (\overline{s}{=}\overline{t}^i \wedge (\delta^i_1 \vee \ldots \vee \delta^i_{k_i}))|_{FV(\overline{L})}$ which is equivalent to a disjunction of computed answers for $\leftarrow\overline{L}$, namely to $\bigvee_{i=1}^{l} \bigvee_{j=1}^{k_i} (\overline{s}{=}\overline{t}^i, \delta^i_j)|_{FV(\overline{L})}$.

2. L_1 is a negative literal $\neg A$. So $\mathbf{T}_{\overline{n}}(\overline{L})$ is $\mathbf{F}_{n_1}(A) \wedge \mathbf{T}_{\overline{n}'}(\overline{L}')$. By the inductive assumption (1') $\leftarrow \mathbf{F}_{n_1}(A), A$ fails and $\leftarrow \mathbf{F}_{n_1}(A), \overline{L}'$ is negatively derived from $\leftarrow\overline{L}$. By the inductive assumption (2') for $\overline{L}'$ and $\overline{n}'$, in a way similar as above we obtain computed answers for $\overline{L}$ satisfying (2').

Now we prove (1'). First we construct a tree T which is a "top part" of a pre-failed tree for G'. Each node of T is of the form $\leftarrow \mathbf{F}_{\overline{n}}(\overline{L}), \rho, \overline{L}, \overline{N}$. If its selected literal is from $\overline{L}$ then the node is a leaf, otherwise it has (zero or more) children built according to Definition 3.9. T is finite as the selection rule is fair.

We show that T can be extended to a finitely failed tree for G' by building a finitely failed tree for every leaf of T with a selected literal from $\overline{L}$. By Lemma 5.2 it is sufficient to prove that for any fair selection rule and any $\overline{N}$ there exists a finitely failed tree for $H = \leftarrow \mathbf{F}_{\overline{n}}(\overline{L}), \overline{L}, \overline{N}$ provided a literal of $\overline{L}$ is selected in H.

To simplify technical details, we can assume as before that the selected literal of $\overline{L}$ is L_1. Consider first the case of $\mathbf{F}_{n_1}(L_1)$ being unsatisfiable. Then $\mathbf{F}_{\overline{n}}(\overline{L})$ is $\mathbf{F}_{\overline{n}'}(\overline{L}')$ and H fails by the inductive assumption. So we can assume that $\mathbf{F}_{n_1}(L_1)$ is satisfiable (hence $n_1 > 0$) and consider two cases.

1. Let $L_1 = p(\overline{s})$. Let us denote the matching clauses of P as previously. The only goals that can be positively derived from H are those of $H_i = \leftarrow \mathbf{F}_{\overline{n}}(\overline{L}), \overline{s}{=}\overline{t}^i, \overline{M}^i, \overline{L}', \overline{N}$ $(i = 1, \ldots, l)$ whose constraints are satisfiable.

   By definition $\mathbf{F}_{n_1}(p(\overline{s}))$ is $\bigwedge_{i=1}^{l} \forall \overline{y}^i (\overline{s}{=}\overline{t}^i \rightarrow \mathbf{F}_{n_1-1}(\overline{M}^i))$ where $\overline{y}^i$ are the free variables of $p(\overline{t}^i) \leftarrow \overline{M}^i$. Thus $\mathbf{F}_{n_1}(p(\overline{s})), \overline{s}{=}\overline{t}^i$ implies $\mathbf{F}_{n_1-1}(\overline{M}^i)$. So $\mathbf{F}_{\overline{n}}(\overline{L}), \overline{s}{=}\overline{t}^i$ implies $\mathbf{F}_{n_1-1}(\overline{M}^i) \vee \mathbf{F}_{\overline{n}'}(\overline{L}')$ which is $\mathbf{F}_{\overline{m}}(\overline{M}^i, \overline{L}')$ where $\overline{m} = (n_1{-}1, \ldots, n_1{-}1, n_2, \ldots, n_m) < \overline{n}$.

   As for any fair selection rule $\leftarrow \mathbf{F}_{\overline{m}}(\overline{M}^i, \overline{L}'), \overline{M}^i, \overline{L}', \overline{N}$ finitely fails (by the inductive assumption), there exists a finitely failed tree for H_i (by Lemma 5.2, for any fair selection rule). Thus there exists a finitely failed tree for H (and for the selection rule under consideration).

2. Let $L_1 = \neg A$. $\mathbf{F}_{n_1}(\neg A)$ is by definition $\mathbf{T}_{n_1}(A)$. Thus $\mathbf{F}_{\overline{n}}(\overline{L})$ is $\mathbf{T}_{n_1}(A) \vee \mathbf{F}_{\overline{n}'}(\overline{L}')$. A failed tree for H can be built in the following way. H has a single child $\leftarrow \mathbf{F}_{\overline{n}'}(\overline{L}'), \overline{L}', \overline{N}$ and the rest of the tree is a finitely failed tree that exists by inductive assumption (1') for $\overline{L}'$ and $\overline{n}'$. The safeness condition for H is satisfied by inductive assumption (2') for A and (n_1) and by Lemma 5.2. □

From the proof it follows that it is sufficient to use failed trees in which a node with a negative literal selected has at most one child. The completeness theorem also holds when a restriction is imposed that the sons of a node $\leftarrow \theta, \ldots, \underline{\neg A}, \ldots$ are computed by negating some answers to $\leftarrow \theta, \underline{A}$, as described in Section 3.4. To show this one needs an obvious modification of the last paragraph of the proof. Also the second variant of SLDFA-resolution from Section 3.5 is complete. We omit a proof which can be based on transforming any SLDFA-failed tree into a failed tree of the variant.

In another paper [Dra93a] we show that from the completeness of SLDFA-resolution it follows that SLDNF-resolution is complete w.r.t. 3-valued completion semantics for non floundering queries (under appropriate fairness assumptions).

It remains to prove that SLDFA-resolution is sound w.r.t. 3-valued completion semantics. It could be done by checking that the proofs of Theorem 4.2 and Lemma 4.1 hold in 3-valued logic. A simpler proof can be constructed using the equivalence of 3-valued completion semantics and strict completion semantics ([Dra93a], this equivalence is implicit in [Wal93]). Soundness of SLDFA-resolution w.r.t. the strict completion semantics follows easily from Theorem 4.2. We outline the proof.

Strict completion of P is the Clark completion of a program P' obtained from P by renaming (some occurrences of) predicate symbols. Similar renaming of a failed tree (refutation) for P and $\leftarrow Q$ results in a a failed tree (refutation) for P' and $\leftarrow Q'$ (where $\leftarrow Q'$ is the renamed $\leftarrow Q$). By Theorem 4.2 comp$(P') \models \neg Q'$ (respectively comp$(P') \models \delta \rightarrow Q'$ where δ is the computed answer, the same for both refutations). This means, by definition, that the strict completion semantics of P entails $\neg Q$ (respectively $\delta \rightarrow Q$).

6 Computing fail answers

SLDFA-resolution as presented above does not specify how to construct finitely failed SLDFA-trees to compute fail answers. In this section we informally describe a method of constructing such trees. The method stems from [MN89]. The basic idea is to build a pre-failed tree for a given goal G. If the tree is not finitely failed, due to an infinite branch or a "success" leaf, then it is pruned by adding an appropriate constraint to G. For the equality theory with WDCA, constructing finitely failed trees for definite programs was discussed in [MN89] and for normal programs in [Näs90] (with failed trees defined as in Section 3.5, version 1).

We begin with defining some concepts. Then we discuss pruning a node of a tree and constructing a finitely failed tree by pruning a set of nodes. Then the search for fail answers is briefly discussed. A comparison with the methods of [Cha88] and [Cha89, Stu91] ends the section.

In what follows we assume a fixed program P. By a successful branch of a pre-failed tree we mean a branch that is finite and ends at a node of the form $\leftarrow \theta$. Remember that in general θ is not related to a correct answer for the root of the tree.

A *cross-section* of a pre-failed tree is any finite set S of nodes of the tree such that every successful or infinite branch has a node in S.

By *instantiating* a pre-failed tree T with the root $\leftarrow \theta, \overline{L}$ by a constraint σ we mean changing every node $\leftarrow \theta', \overline{M}$ into $\leftarrow \theta'\sigma, \overline{M}$ if $\theta'\sigma$ is satisfiable or removing the node otherwise; for technical reasons we require that $\theta\sigma$ is satisfiable. If $FV(\sigma) \cap FV(T) \subseteq FV(\theta, \overline{L})$ (the variables "introduced" in the tree do not occur free in σ) then the obtained tree is still a pre-failed tree. Indeed, by Lemma 5.2 the safeness condition holds for the nodes with a negative literal selected; the rest of the proof is obvious. Thus if the instantiated pre-failed tree is finite and has no successful branches then the tree is a finitely failed SLDFA-tree provided that $FV(\sigma) \cap FV(T) \subseteq FV(\theta, \overline{L})$.

To construct a finitely failed tree from a pre-failed tree T it is sufficient to choose a cross-section $\{\leftarrow \theta_1, ...; \ldots; \leftarrow \theta_n, ...\}$ of T and find a constraint σ (satisfying the above condition) such that instantiating T with σ removes the nodes of the cross-section from T. The remaining tree is finitely failed by the definition of the cross-section. (Obviously, the whole pre-failed tree need not to be constructed, it is enough to build the part "between the root and the cross-section"). As we are interested in fail answers satisfying Definition 3.8, from now on we require that if σ is used to instantiate a tree with the root $\leftarrow \theta, A$ then $FV(\sigma) \subseteq FV(A)$.

We begin with pruning a single node, i.e. finding a σ such that instantiating the tree with σ results in removing a given node from a pre-failed tree.

Consider such a tree with the root $\leftarrow\theta, A$ and its node $\leftarrow\theta', \overline{L}$. Pruning boils down to instantiating the tree with a constraint σ such that $\theta'\sigma$ is not satisfiable. This condition is equivalent to CET $\models \sigma \rightarrow \neg(\theta'|_{FV(A)})$ (since $FV(\sigma) \subseteq FV(A)$). So a most general σ satisfying this is (equivalent to) $\sigma_{\rm pr} = \neg(\theta'|_{FV(A)})$.

If $\sigma_{\rm pr}\theta$ is unsatisfiable then the required σ does not exist and the node $\leftarrow\theta', \overline{L}$ cannot be pruned. Otherwise instantiating the tree with $\sigma_{\rm pr}$ (or with a σ less general than $\sigma_{\rm pr}$) results in removing $\leftarrow\theta', \overline{L}$ and the subtree rooted at this node from the tree. Some other nodes may be removed too.

Example 6.1 Consider the following pre-failed tree.

$$\begin{tikzcd}
\leftarrow p(x) \arrow[d, dash] \\
\leftarrow x{=}y, q(y) \arrow[d, dash] \\
\leftarrow x{=}y, y{=}f(z), r(z) \arrow[d, dash] \\
\leftarrow x{=}y, y{=}f(z)
\end{tikzcd}$$

The second node of this tree cannot be pruned. A most general constraint pruning the third (or the fourth) node is $\neg\exists y, z(x{=}y, y{=}f(z))$ which is equivalent to $\neg\exists z(x{=}f(z))$. $\square$

Now we are ready to discuss computing fail answers. Consider a pre-failed tree with the root $\leftarrow\theta, A$. In order to prune all the nodes of a cross-section $\{\leftarrow\theta_1, ...; \ldots; \leftarrow\theta_n, ...\}$ (thus obtaining a finitely failed tree) the tree has to be instantiated with a constraint σ such that CET $\models \sigma \rightarrow \sigma^1_{\rm pr} \wedge \ldots \wedge \sigma^n_{\rm pr}$ where $\sigma^i_{\rm pr}$ is $\neg(\theta_i|_{FV(A)})$ for $i = 1, \ldots, n$. A most general σ satisfying this is (equivalent to) $\sigma_{\rm pr} = \sigma^1_{\rm pr} \wedge \ldots \wedge \sigma^n_{\rm pr}$. If $\sigma_{\rm pr}\theta$ is unsatisfiable then the required σ does not exist and the cross-section cannot be pruned. Otherwise instantiating the tree with $\sigma_{\rm pr}$ results in a finitely failed tree. Constraint $\theta\sigma_{\rm pr}$, equal to

$$\theta \wedge \neg(\theta_1|_{FV(A)}) \wedge \ldots \wedge \neg(\theta_n|_{FV(A)}),$$

is a fail answer for $\leftarrow\theta, A$. Usually it is convenient to transform such a fail answer to some disjunctive normal form $\gamma_1 \vee \ldots \vee \gamma_m$ and treat $\gamma_1, \ldots, \gamma_m$ as separate fail answers. As explained in Section 2.3, the actual methods of transforming constraints are outside the scope of this paper.

Example 6.2 The following is a pre-failed tree (of rank 1)

$$\begin{tikzcd}
 & \leftarrow \mathit{even}(x) \arrow[dl, dash] \arrow[dr, dash] & \\
\leftarrow x{=}0 & & \leftarrow x{=}s(x_1), \neg \mathit{even}(x_1) \arrow[d, dash, "(\leftarrow \mathit{even}(x_1) \text{ succeeds with } x_1 = 0)"] \\
 & & \leftarrow x{=}s(x_1), x_1{\neq}0
\end{tikzcd}$$

for the program:

$\mathit{even}(0).$
$\mathit{even}(s(x)) \leftarrow \neg \mathit{even}(x).$

The set $\{\leftarrow x{=}0;\ \leftarrow x{=}s(x_1), x_1{\neq}0\}$ is a cross-section of the tree. Constraint $\sigma^1_{\rm pr} = x{\neq}0$ prunes $\leftarrow x{=}0$. Constraint $\sigma^2_{\rm pr} = \neg\exists x_1(x{=}s(x_1), x_1{\neq}0)$ prunes $\leftarrow x{=}s(x_1), x_1{\neq}0$. Constraint $\sigma^1_{\rm pr} \wedge \sigma^2_{\rm pr}$ is a fail answer for $\leftarrow even(x)$.

Under CET, $\sigma^2_{\rm pr}$ is equivalent to $x{=}s(0) \vee \forall x_1(x{\neq}s(x_1))$. The fail answer above is equivalent to $x{=}s(0) \vee x{\neq}0, \forall x_1(x{\neq}s(x_1))$. It is often convenient to treat the elements of such a disjunction as separate fail answers.

If s and 0 are the only functors of the underlying language then, due to WDCA, $\sigma^2_{\rm pr}$ is equivalent to $x{=}s(0) \vee x{=}0$ and the fail answer above is equivalent to $x{=}s(0)$. □

The described method of computing fail answers is not deterministic. First, the pre-failed tree is not unique for a given goal, even with a fixed selection rule. For every node $\leftarrow\theta', ...$ with a $\neg B$ selected, choosing different sets of computed answers for $\leftarrow B$ results in different trees.

The obtained fail answer depends on the choice of a pre-failed tree and on the choice of a cross-section. If S and S' are cross-sections of the pre-failed tree such that S is "above" S' (there is a node from S in every path between the root and some node from S') then the fail answer $\theta\sigma'_{\rm pr}$ obtained by using S' is more general than that obtained from S.

To justify this, let $S = \{\leftarrow\theta_1, ...; \ldots; \leftarrow\theta_n, ...\}$ and $S' = \{\leftarrow\theta'_1, ...; \ldots; \leftarrow\theta'_m, ...\}$ be cross-sections of a pre-failed tree for $\leftarrow\theta, A$ such that S is "above" S'. We have to assume that if $\leftarrow\theta'', ...$ is a son of a node $\leftarrow\theta', ...$ with a negative literal selected then $\theta'' \rightarrow \theta'$. (For example this holds for the way of constructing the sons of $\leftarrow\theta', ...$ described in Section 3.4). The fail answers corresponding to S and S' are respectively $\theta\sigma_{\rm pr}$ and $\theta\sigma'_{\rm pr}$ where $\sigma_{\rm pr} = \neg(\theta_1|_{FV(A)}) \wedge \ldots \wedge \neg(\theta_n|_{FV(A)})$ and $\sigma'_{\rm pr} = \neg(\theta'_1|_{FV(A)}) \wedge \ldots \wedge \neg(\theta'_m|_{FV(A)})$. For every $j = 1, \ldots, m$ there exists i, $1 \leq i \leq n$, such that $\leftarrow\theta'_j, ...$ is a descendant of $\leftarrow\theta_i, \ldots$. Hence $\theta'_j \rightarrow \theta_i$ and $\neg(\theta_i|_{FV(A)}) \rightarrow \neg(\theta'_j|_{FV(A)})$. Thus $\sigma_{\rm pr} \rightarrow \sigma'_{\rm pr}$.

Example 6.3 Consider the program

$even(0)$

$even(s^2(x)) \leftarrow even(x)$

and the pre-failed tree (of rank 0)

$$\begin{tikzcd}
 & \leftarrow even(x) \arrow[dl, no head] \arrow[dr, no head] & & & \\
\leftarrow x{=}0 & & \leftarrow x{=}s^2(x_1), even(x_1) \arrow[dl, no head] \arrow[dr, no head] & & \\
 & \leftarrow x{=}s^2(0) & & \leftarrow x{=}s^4(x_2), even(x_2) \arrow[dl, no head] \arrow[dr, no head] & \\
 & & \ldots & & \ldots
\end{tikzcd}$$

(the constraints are simplified as explained at the end of Section 3.4).

Cross-section $S_1 = \{\leftarrow x{=}0;\ \leftarrow x{=}s^2(x_1), even(x_1)\}$ gives a fail answer $\sigma_1 = x{\neq}0, \forall x_1(x{\neq}s^2(x_1))$. Cross-section $S_2 = \{\leftarrow x{=}0; \leftarrow x{=}s^2(0); \leftarrow x{=}s^4(x_2), even(x_2)\}$ results in $\sigma_2 = x{\neq}0, x{\neq}s^2(0), \forall x_2(x{\neq}s^4(x_2))$. Further cross-sections result in $x{\neq}0, x{\neq}s^2(0), \ldots, x{\neq}s^{2n-2}(0), \forall x_n(x{\neq}s^{2n}(x_n))$. □

Similarly, let tree T' be obtained from a pre-failed tree T by using some additional computed answers (for $\leftarrow \theta, B$, where $\neg B$ is selected in some node $\leftarrow \theta', \ldots$ of the tree). If S and S' are "corresponding" cross-sections of T and T' then the fail answer obtained from S' is more general than that obtained from S.

Obviously, in general it is impossible to choose a "lowest" cross-section in a tree (since the pre-failed tree may be infinite). It may also be impossible to use all the relevant computed answers while building a pre-failed tree (since there may be infinitely many such answers associated with a single node). So after obtaining a fail answer from a cross-section of a pre-failed tree, it may be necessary to modify the tree and/or choose a "lower" cross-section in order to obtain a more general fail answer. The need for modifying the tree can be avoided by using the alternative definition of (pre-) failed trees of Section 3.5, version 2.

The fail answers obtained in this way overlap, any fail answer covers the previous ones. If required, mutually excluding answers can be obtained by adding the negation of the previous answers to the current one, obtaining $\theta\sigma'_{\rm pr}(\theta_1 \vee \ldots \vee \theta_n)|_{FV(A)}$ instead of $\theta\sigma'_{\rm pr}$ (where $\theta_1, \ldots, \theta_n$ are the constraints of the previous cross-section and $\theta\sigma'_{\rm pr}$ is the fail answer obtained from the current cross-section as described above).

Example 6.4 Applying this modification to the previous example results in mutually excluding fail answers (equivalent to) $x{\neq}0, \forall z(x{\neq}s^2(z))$; $\exists y(x{=}s^2(y)), x{\neq}s^2(0), \forall z(x{\neq}s^4(z)); \ldots; \exists y(x{=}s^{2n}(y)), x{\neq}s^{2n}(0), \forall z(x{\neq}s^{2n+2}(z)); \ldots$.

Note that if 0 and s are the only functors then, under WDCA, these answers are equivalent to $x{=}s(0); \ldots; x{=}s^{2n+1}(0); \ldots$.

In this example the mutually excluding fail answers are simpler than the overlapping ones obtained previously. This is not a general rule. Removing clause $even(0)$ from the program leads to an opposite situation. The answers computed as previously are $\forall z(x{\neq}s^{2n}(z))$ for $n = 1, 2, \ldots$. The non overlapping answers are $\exists y(x{=}s^{2n-2}(y)), \forall z(x{\neq}s^{2n}(z))$. $\Box$

6.1 Comparison with the methods of Chan

After having described a way of computing fail answers we are ready for comparisons with other approaches. We compare our method with that of [Cha88] and that of [Cha89, Stu91]. They are representative for the first and the second principal approach referred to in Section 1 and they seem to be the most important among the methods mentioned there.

First we show how our approach subsumes the first method of Chan (SLD-CNF-resolution of [Cha88]). In SLD-CNF-resolution, negation is treated by negating (the disjunction of) all the answers to the corresponding non-negated query. The number of answers has to be finite; soundness requires that the corresponding SLD-CNF-tree is finite. It is easy to show by induction w.r.t. ranks that every SLD-CNF-tree is also a pre-failed tree (where for each node $\leftarrow \theta' \ldots, \underline{\neg B}, \ldots$ all the answers for $\leftarrow \theta', B$ are used to compute the sons of the node). Obtaining a fail answer from the cross-section containing the success nodes of the tree is equivalent to negating the conjunction of the computed answers given by the tree.

Summarizing, the first method of Chan is an instance of our approach. It is obtained when (1) all the answers for $\leftarrow \theta, A$ are used in constructing the sons of

$\leftarrow \theta, ..., \underline{\neg A}, ...$ in a pre-failed tree and (2) only the "lowest" cross-section of a pre-failed tree is used (if it exists, it contains the leaves of the form $\leftarrow \delta$). Our method is stronger as there is no requirement on finiteness of the trees.

Now we describe briefly and discuss the second method of Chan [Cha89, Stu91]. In that method, if a negated subgoal $\neg A$ is selected in a goal $H = \leftarrow \theta, ..., \underline{\neg A}, ...$ then a "lower level" computation begins and a cross-section $S = \{\leftarrow \theta_1, \overline{L_1}; \ldots; \leftarrow \theta_n, \overline{L_n}\}$ of an SLD-tree for $\leftarrow \theta, A$ is chosen[2]. No other cross-sections of this tree are used. The cross-section is converted into a set of "answers" for $\leftarrow \theta, \neg A$ as described below.

The difference between the methods may be informally described as follows. In our method many cross-sections of the tree may be selected and only the constraints of such cross-section are used to compute answers. In [Cha89, Stu91] only one cross-section is selected but whole goal bodies of the cross-section are used. (The methods of [Wal87], [Lug89], [SM91] and [Pla92] are similar, however they always select the cross-section at depth 1). This results in "answers" that may contain (negative) subgoals to be resolved by further derivation steps.

Consider the chosen cross-section S. Let W be $FV(\theta, A)$. From Lemma 4.1 it follows that, with respect to comp(P), $(\theta, \neg A)$ is equivalent to $\theta \wedge \neg((\theta_1, \overline{L_1})|_W) \wedge \ldots \wedge \neg((\theta_n, \overline{L_n})|_W)$. Now $\neg((\theta_i, \overline{L_i})|_W)$ is equivalent to $F_i^0 \vee F_i^1$ where $F_i^0 = \neg(\theta_i|_W)$ and $F_i^1 = (\theta_i|_W) \wedge \neg((\theta_i, \overline{L_i})|_W)$. Thus $\theta, \neg A$ is equivalent to

$$\theta \wedge \bigvee_{\langle j_1, \ldots, j_n \rangle \in \{0,1\}^n} F_1^{j_1} \wedge \ldots \wedge F_n^{j_n}.$$

Every $F_1^{j_1} \wedge \ldots \wedge F_n^{j_n}$ can be used to replace $\neg A$ in goal H in order to obtain its successor H' in a derivation[3].

Building (a part of) an SLD-tree and choosing a cross-section is the same as in our approach (because an SLD-tree is a pre-failed tree). The answer $\theta \wedge F_1^0 \wedge \ldots \wedge F_n^0$ is equivalent to the fail answer obtained in our approach. The difference between restricting to W that occurs in F_i^j and restricting to $FV(A)$ used in our method is inessential because $\theta \wedge \neg((\theta_i, \overline{L_i})|_W)$ and $\theta \wedge \neg((\theta_i, \overline{L_i})|_{FV(A)})$ are equivalent, also for $\overline{L_i}$ being empty. (This can be shown using the fact that θ_i is $\theta\theta_i'$ where $FV(\theta) \cap FV(\theta_i', \overline{L_i}) \subseteq FV(A)$. Then, for X being W or $FV(A)$, it holds that $\neg((\theta_i, \overline{L_i})|_X)$ is equivalent to $\neg\theta|_X \vee \neg((\theta_i', \overline{L_i})|_X)$. Also $(\theta_i', \overline{L_i})|_{FV(A)}$ and $(\theta_i', \overline{L_i})|_W$ are equivalent.)

All the other "answers" $\theta \wedge F_1^{j_1} \wedge \ldots \wedge F_n^{j_n}$ (where $j_i = 1$ for some i) are not constraints, they contain other literals than (in)equality.

If $\neg((\theta_i, \overline{L_i})|_W)$ is selected later on in the computation then it is treated as $\neg A$ above. This corresponds, in our method, to pruning a cross-section S_1 in the subtree rooted at $\leftarrow \theta_i, \overline{L_i}$ (in the same SLD-tree for $\leftarrow \theta, A$). Due to fundamental differences between the two methods it is difficult to establish a formal relationship between the (nodes of the) trees constructed by them.

[2]or for some $\leftarrow \theta', A$ where $\theta \rightarrow \theta'$. [Stu91] allows negative literals to be selected in the tree.

[3]In [Cha89] these formulae are transformed into some disjunctive normal form and the elements of the disjunctions are used. The transformation is done by converting each $F_i^{j_i}$ into a certain disjunctive form and applying distributivity.

Example 6.5 Consider a program

$$\begin{aligned}
&p(f(x), y) \leftarrow q(x)\\
&p(x, g(y)) \leftarrow r(y)\\
&q(a)\\
&r(b).
\end{aligned}$$

We show how goal $\leftarrow\neg p(x, y)$ is treated by the second method of Chan [Cha89]. First a cross-section is selected in the SLD-tree

$$\begin{tikzcd}
& \leftarrow p(x,y) \arrow[dl, dash] \arrow[dr, dash] & \\
\leftarrow x{=}f(x'), q(x') \arrow[d, dash] & & \leftarrow y{=}g(y'), r(y') \arrow[d, dash] \\
\leftarrow x{=}f(a) & & \leftarrow y{=}g(b)
\end{tikzcd}$$

Assume that it is $S = \{\leftarrow x{=}f(x'), q(x');\ \ \leftarrow y{=}g(y'), r(y')\}$. $\neg p(x, y)$ is equivalent to $\neg\exists x'(x{=}f(x'), q(x')) \wedge \neg\exists y'(y{=}g(y'), r(y'))$. The first element of the conjunction is equivalent to $\neg\exists x'(x{=}f(x')) \vee \exists x'(x{=}f(x'), \neg q(x'))$, the second element is transformed analogically.

Let $\rho = \neg\exists x'(x{=}f(x'))$ and $\sigma = \neg\exists y'(y{=}g(y'))$. Now $\neg p(x, y)$ is equivalent to $\rho, \sigma \ \vee\ \exists x'(x{=}f(x'), \neg q(x')), \sigma \ \vee\ \rho, \exists y'(y{=}g(y'), \neg r(y')) \ \vee\ \exists x', y'(x{=}f(x'), \neg q(x'), y{=}g(y'), \neg r(y'))$.

From the SLD-trees for $\leftarrow q(x')$ and for $\leftarrow r(y')$ we obtain in a similar way that $\neg q(x')$ is equivalent to $x'{\neq}a$ and $\neg r(y')$ to $y'{\neq}b$. This leads to four top level refutations represented in the following tree

$$\begin{tikzcd}
& \leftarrow\neg p(x,y) \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
\leftarrow\rho, \sigma & \leftarrow x{=}f(x'), \sigma, \underline{\neg q(x')} \arrow[d, dash] & \leftarrow\rho, y{=}g(y'), \underline{\neg r(y')} \arrow[d, dash] & \leftarrow x{=}f(x'), y{=}g(y'), \underline{\neg q(x')}, \neg r(y') \arrow[d, dash] \\
& \leftarrow x{=}f(x'), x'{\neq}a, \sigma & \leftarrow\rho, y{=}g(y'), y'{\neq}b & \leftarrow x{=}f(x'), y{=}g(y'), x'{\neq}a, \underline{\neg r(y')} \arrow[d, dash] \\
& & & \leftarrow x{=}f(x'), y{=}g(y'), x'{\neq}a, y'{\neq}b
\end{tikzcd}$$

with the computed answers ρ, σ; $\exists x'(x{=}f(x'), x'{\neq}a, \sigma)$; $\exists y'(\rho, y{=}g(y'), y'{\neq}b)$ and $\exists x', y'(x{=}f(x'), y{=}g(y'), x'{\neq}a, y'{\neq}b)$ respectively. Note that $\neg q(x')$ and $\neg r(y')$ are selected twice in the derivations.

In our method the first of these answers is obtained by using the same cross-section. The remaining three answers prune lower cross-sections (eg. $\exists x'(x{=}f(x'), x'{\neq}a, \sigma)$ prunes $\{\leftarrow x{=}f(a);\ \leftarrow y{=}g(y'), r(y')\}$). The cross-section at depth 2 results in answer $x{\neq}f(a), y{\neq}g(b)$ which covers all the four answers above.

In our method only one tree (the SLD-tree for $\leftarrow p(x, y)$ with 5 nodes) has to be constructed. In the other method we have the tree of the derivations depicted above (9 nodes) plus the "top sections" of SLD-trees for $\leftarrow p(x, y)$, $\leftarrow q(x')$ and $\leftarrow r(y')$. The resulted answers are less general. (However, both methods coincide when the lowest cross-section is chosen instead of S.)
□

Example 6.6 Such efficiency difference between the methods does not occur for the program and goal from Example 6.3. For simplicity assume that the method of [Cha89] always selects the cross-section at depth 1. This leads to refutations $\leftarrow\neg \mathit{even}(x)$; $\leftarrow x{\neq}0, \forall x_1(x{\neq}s^2(x_1))$ and

$$\begin{aligned}\leftarrow\neg \mathit{even}(x);\ \leftarrow x{=}s^2(x_1), \neg \mathit{even}(x_1);\ \ldots;\ \leftarrow x{=}s^{2n}(x_n), \neg \mathit{even}(x_n);\\ \leftarrow x{=}s^{2n}(x_n), x_n{\neq}0, \forall x_{n+1}(x_n{\neq}s^2(x_{n+1}))\end{aligned}$$

for $n = 1, 2, \ldots$. Their computed answers are the same as the mutually excluding answers in Example 6.4 (and simpler than those obtained in Example 6.3). The constructed trees of depth 1 are identical to the elementary subtrees of the pre-failed tree used by our method (conf. Example 6.3). Thus the search spaces are of the same size, in the sense that for any answer the search to obtain it is of the same size in both methods. The author does not know any example for which the search space is smaller in the method of [Cha89]. □

Example 6.5 suggests that our method may be more efficient, at least for some programs and some choices of cross-sections done in the other method. This can be explained as follows.

Consider the goal H, the tree and the n element cross-section S as above. In the method of [Cha89, Stu91] H is rewritten into 2^n successor goals. (This number may be smaller as some constraints may be unsatisfiable). This creates, in a sense, 2^n search spaces. In contrast, the corresponding number in our method is n (as for every goal in the cross-section one subtree of the pre-failed tree has to be built).

Each F_i^1 occurs in 2^{n-1} successor goals, this usually gives rise to repeated computations. There are no such repeated occurrences of subgoals in our method. The constraints in F_i^0 and in F_i^1 are mutually excluding, this leads to many less general answers. In our method there usually are fewer and more general answers due to using cross-sections of a single tree.

This argument suggests that our method may have practical advantages. The search space is smaller, at least in some cases, and the answers are more general. All the goals are normal goals, the only formulae to be reduced to normal form are constraints. Work on designing an implementation and computer experiments should provide better understanding of these issues.

7 Extension for the well-founded semantics

The well-founded semantics [GRS91] (for a definition see also [Prz90], [BNN91], [AB94] or Section 7.1) is often considered as the most appropriate semantics for logic programs from the point of view of non-monotonic reasoning and knowledge representation [Prz91]. A constructive negation approach for the well-founded semantics can be obtained by a simple extension of SLDFA-resolution. It is enough to remove the finiteness requirement from the definition of finitely failed trees. However, to achieve completeness it is necessary to allow infinitely many sons for a node with a negative literal selected. Additionally, the ranks should be allowed to be ordinal numbers because an infinite failed tree may refer to infinitely many refutations and the finite upper bound of their ranks may not exist.

Definition 7.1 (SLSFA-resolution).

The definition of SLDFA-resolution (Definitions 3.7, 3.8 and 3.9) with the following modifications:

- The rank is a countable ordinal α instead of a natural number k.
- Condition 2 (finiteness of the failed tree) in Definition 3.9 is removed.
- A node H with a negative literal selected, of the form
$$\leftarrow \theta, \overline{L}, \underline{\neg A}, \overline{L'}$$
has (possibly infinitely many) sons
$$\leftarrow \sigma_1, \overline{L}, \overline{L'};\ \leftarrow \sigma_2, \overline{L}, \overline{L'};\ \ldots$$
provided that there exist (possibly 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 $FV(\theta, A)$ if $\theta\tau$ is true in CET then some $\delta_i\tau$ or some $\sigma_i\tau$ is true in CET.
- "Finitely failed" is replaced by "failed" and "SLDFA" by "SLSFA" everywhere in the definitions.

Obviously SLSFA-resolution subsumes SLDFA-resolution. It also subsumes SLS-resolution, up to replacing substitutions by equations. (SLS-resolution can be seen as a natural extension of SLDNF-resolution that permits infinite failed trees. The reader is referred to [Prz89b] or to [AB94] for a definition.) An SLS-tree [Prz89b] that is failed is also an SLSFA-failed tree. A successful branch of an SLS-tree is also an SLSFA-refutation.

Example 7.2 Consider a program

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

(and assume that a and b are not the only functors of the underlying language). In the well-founded semantics of the program, $p(a)$ is true, $p(b)$ is undefined and $p(x)$ is false for x distinct from a and from b. Constraint $x{\neq}a, x{\neq}b$ is a fail answer for $\leftarrow p(x)$ since the following is an infinite SLSFA-failed tree:

$$\begin{array}{c} \leftarrow x{\neq}a, x{\neq}b, p(x) \\ | \\ \leftarrow x{\neq}a, x{\neq}b, p(x) \\ | \\ \cdots \end{array}$$

Constraint $x{=}a$ is a computed answer for $\leftarrow p(x)$ since

$\leftarrow p(x);\quad \leftarrow x{=}a, \neg p(x');\quad \leftarrow x{=}a, x'{\neq}a, x'{\neq}b$

is a refutation (of rank 1). For goal $\leftarrow p(b)$ neither a refutation nor a failed tree exists, as a refutation (a failed tree) for $p(b)$ of rank α exists only if a failed tree (a refutation) for $p(b)$ exists for some rank $< \alpha$. $\square$

Further examples are given in [Dra93b]. The following examples show the need for infinite ranks and for infinitely branching failed trees.

Example 7.3 Consider a program

$$\begin{array}{ll} even(0) & odd(s(0)) \\ even(s(x)) \leftarrow \neg even(x) & odd(s^2(x)) \leftarrow odd(x) \end{array}$$

Assume Prolog selection rule. The failed tree for $\leftarrow odd(x), even(x)$ has an infinite branch with nodes $\leftarrow x{=}s^{2i}(y_i), odd(y_i), even(x)$, $i = 0, 1, \ldots$ and infinitely many finite branches with leaves equivalent to $\leftarrow ..., \neg even(s^{2i}(0))$, $i = 0, 1, \ldots$. As in Example 3.11, a successful derivation for $\leftarrow even(s^{2i}(0))$ is of rank $2i$. Thus the rank of the tree is ω. □

Example 7.4 Atom p is false w.r.t. the well-founded semantics of the program

$$\begin{array}{ll} p \leftarrow \neg q(x), \neg r(x) & r(0) \\ q(x) \leftarrow \neg r(x) & r(s(x)) \leftarrow r(x) \end{array}$$

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

$$\begin{array}{l} \leftarrow p \\ \leftarrow \underline{\neg q(x)}, \neg r(x) \\ \leftarrow \delta_i, \neg r(x) \end{array}$$

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. □

SLSFA-resolution is sound and complete in the following sense.

Theorem 7.5 (Soundness, completeness and independence from the selection rule)
Let P be a normal program and $WF(P)$ its well-founded model.

If δ is an SLSFA-computed answer for a goal $\leftarrow \theta, \overline{L}$ then $WF(P) \models_3 \delta \rightarrow \overline{L}$. If there exists an SLSFA-failed tree for a goal G then $WF(P) \models_3 G$.

If G is a goal such that $WF(P) \models_3 G$ then for any selection rule there exists an SLSFA-failed tree for G. If $WF(P) \models_3 \overline{L}\tau$, where τ is a substitution, $\overline{L}\tau$ is ground and $\theta\tau$ is true in CET, then for any selection rule there exists an SLSFA-computed answer δ for $\leftarrow \theta, \overline{L}$ such that $\delta\tau$ is true in CET.

The proof is presented in the following section.

Obviously SLSFA-resolution cannot be implemented, as the well-founded semantics is uncomputable. What is possible is an algorithm that is a sound but incomplete approximation of SLSFA-resolution. The approach to constructing failed trees described in the previous section is in principle applicable here. A definition of

(SLSFA-) pre-failed tree is obtained from that of SLSFA-failed tree by removing the requirement 5 from Definitions 7.1, 3.9 thus allowing "success" leaves. A cross-section of a pre-failed tree is a set S of nodes such that every successful branch has a node in S. Pruning a single node and computing a fail answer by pruning a finite cross-section are like discussed previously.

Example 7.6 The failed tree from Example 7.2 can be obtained by pruning the pre-failed tree

$$\begin{tikzcd}[arrows=dash]
 & \leftarrow p(x) \arrow[dl] \arrow[d] \arrow[dr] & \\
\leftarrow x{=}a, \neg p(x') \arrow[d] & \leftarrow p(x) \arrow[dl, shorten >=2em] \arrow[d] \arrow[dr, shorten >=2em] & \leftarrow x{=}b, \neg p(b) \arrow[d] \\
\leftarrow x{=}a & \dots & \leftarrow x{=}b
\end{tikzcd}$$

of rank 0, using the cross-section containing all the leaves of the tree. There are infinitely many leaves but the set of their labels $\{ \leftarrow x{=}a;\ \leftarrow x{=}b \}$ is finite. $\Box$

This approach cannot deal with pruning infinite cross-sections (infinite formulae may result) and with computing sons of a node $\leftarrow ..., \underline{\neg A}, ...$ if infinitely many answers for $\leftarrow A$ are involved. A subject for future work is developing techniques to finitely represent (some) infinite pre-failed trees, cross-sections and sets of constraints in order to strengthen this approach. Methods of tabulation [ST86, SI88, BD93] and of finite representing of infinite sets of answers [CI93] are relevant here.

7.1 Proof of Theorem 7.5

We begin with a characterization of the well-founded semantics. Then we prove lemmas that imply soundness, completeness and independence from the selection rule of SLSFA-resolution (Theorem 7.5).

Consider a language $\mathcal{L}$, the corresponding Herbrand base $\mathcal{H}$ and a program P. By a (3-valued) interpretation we mean a set I of literals, $I \subseteq \mathcal{H} \cup \neg\mathcal{H}$ (where $\neg\mathcal{H} = \{\neg A \mid A \in \mathcal{H}\}$) such that if $A \in I$ then $\neg A \notin I$. A ground atom A is true in I ($I \models_3 A$) iff $A \in I$, it is false in I iff $\neg A \in I$, otherwise A is undefined in I.

For the purpose of our work we use the following characterization of the well-founded semantics. The well-founded model $WF(P)$ of P is the least (with respect to $\subseteq$) fixpoint of a mapping Ψ_P (which maps interpretations into interpretations). Ψ_P is defined below. There exists a countable ordinal β such that $WF(P) = \Psi_P^{\beta}(\emptyset)$. (Powers of Ψ_P are defined in a usual way: $\Psi_P^{\alpha+1}(I) = \Psi_P(\Psi_P^{\alpha}(I))$ and, for a limit ordinal α, $\Psi_P^{\alpha}(I) = \bigcup_{\gamma<\alpha} \Psi_P^{\gamma}(I)$.)

Our characterization of the well-founded semantics is just a variant of characterizations by iterated fixed point [BNN91, Prz90]. It was proposed independently in [FD92], for a correctness proof the reader is referred there.

Definition 7.7 (mapping Ψ_P)

For every predicate symbol p we will treat $\neg p$ as a new distinct predicate symbol. A normal program can thus be treated as a definite program over Herbrand base $\mathcal{H} \cup \neg\mathcal{H}$.

Let I be an interpretation. We define two ground, possibly infinite, definite programs $P/_t I$ and $P/_{tu} I$.

$P/_t I$ is the ground instantiation of P together with ground unary clauses that show which negative literals are true in I:

$$P/_t I = \mathit{ground}(P) \cup \{\neg A \mid \neg A \in I\}$$

$P/_{tu} I$ is similar but all the negative literals that are true or undefined in I are made true here:

$$P/_{tu} I = \mathit{ground}(P) \cup \{\neg A \in \neg\mathcal{H} \mid A \notin I\}$$

Now $P/_t I$ is used to determine which atoms are true in $\Psi_P(I)$ and $P/_{tu} I$ is used to determine which are false (by determining which are true or undefined). Ψ_P is defined as

$$\Psi_P(I) = (\mathcal{M}_{P/_t I} \cap \mathcal{H}) \cup \neg(\mathcal{H} \setminus \mathcal{M}_{P/_{tu} I})$$

where $\mathcal{M}_Q$ is the least 2-valued Herbrand model of a positive program Q; the standard definitions are used here (an interpretation is a subset of Herbrand base $\mathcal{H} \cup \neg\mathcal{H}$, etc.).

Note that $P/_t I \subseteq P/_{tu} I$, $\mathcal{M}_{P/_t I} \subseteq \mathcal{M}_{P/_{tu} I}$, $\Psi_P(I)$ is an interpretation, Ψ_P is monotone, $\Psi_P^\beta(\emptyset) \subseteq \Psi_P^\alpha(\emptyset)$ for $\beta \leq \alpha$.

To formulate our lemmas, we need a notion of a ground instance of a goal.

Definition 7.8 Let $\leftarrow\theta, \overline{L}$ be a goal and τ a substitution. If $CET \models \theta\tau$ and $\overline{L}\tau$ is ground then $\leftarrow\overline{L}\tau$ is called a *ground instance* of $\leftarrow\theta, \overline{L}$. A ground instance of a formula $\theta, \overline{L}$ and of a goal sequence is defined analogically (the same τ is used for all the goals of the sequence).

Lemma 7.9 (Soundness of SLSFA-resolution)

1. If δ is an SLSFA-computed answer for $\leftarrow\theta, \overline{L}$ obtained from a refutation of rank α and if $\overline{L'}$ is a ground instance of $\delta, \overline{L}$ then
$$\Psi_P^{\alpha+1}(\emptyset) \models_3 \overline{L'}$$
2. If $\leftarrow\overline{L'}$ is a ground instance of a root of an SLSFA failed tree of rank α then
$$\Psi_P^{\alpha+1}(\emptyset) \models_3 \neg\overline{L'}$$

PROOF
The proof is by transfinite induction. Assume that the lemma holds for any $\beta < \alpha$.

1. By part 2 of the inductive assumption, every ground instance $\leftarrow\overline{L_0}; \ldots; \leftarrow\overline{L_k}$ of an SLSFA-refutation of rank α is an SLD-refutation w.r.t. $P/_t I$ where $I = \Psi_P^{\beta+1}(\emptyset)$ and $\beta < \alpha$. By soundness of SLD-resolution, $\mathcal{M}_{P/_t I} \models \overline{L_0}$. In other words, if L is a literal of $\overline{L_0}$ then $L \in \mathcal{M}_{P/_t I}$.

   Thus $L \in \Psi_P(I)$ (for a positive L it is obvious; if $\neg A \in \mathcal{M}_{P/_t I}$ then $\neg A \in P/_t I$ and $\neg A \in I \subseteq \Psi_P(I)$).

   On the other hand $\Psi_P(I) = \Psi_P^{\beta+2}(\emptyset) \subseteq \Psi_P^{\alpha+1}(\emptyset)$. We have shown that $\Psi_P^{\alpha+1}(\emptyset) \models_3 \overline{L_0}$.

2. Assume that part 2 of the lemma does not hold. Let $I = \Psi_P^\alpha(\emptyset)$. Let $\leftarrow \overline{L'}$ be a ground instance of a root of a rank α failed tree T. Assume that $\Psi_P(I) \not\models_3 \neg\overline{L'}$. In other words, every literal of $\overline{L'}$ is true or undefined in $\Psi_P(I)$.

   We show that every such literal is a member of $\mathcal{M}_{P/_{tu}I}$. For positive literals it follows immediately from the definition of Ψ_P. If the literal is negative, say $\neg A$, then A is false or undefined in $\Psi_P(I)$ and $A \notin \Psi_P(I)$. As $I \subseteq \Psi_P(I)$, $A \notin I$ and $\neg A$ is a unary clause in $P/_{tu}I$.

   By completeness of SLD-resolution, for the selection rule used in the failed tree T, there exists an SLD-refutation for $\leftarrow \overline{L'}$ and definite program $P/_{tu}I$. Simple induction shows that every goal of the refutation is an instance of a node in the tree. Thus there is a node of the form $\leftarrow \theta$. Contradiction. □

We prove the completeness of SLSFA-resolution for atomic queries. The extension to general queries of the form $\leftarrow \theta, \overline{L}$ is obvious (by adding a clause $\mathbf{g}(\overline{x}) \leftarrow \overline{L}$, where $\mathbf{g}$ is a new symbol, to the program and using $\leftarrow \theta, \mathbf{g}(\overline{x})$ instead of $\leftarrow \theta, \overline{L}$).

Lemma 7.10 (Completeness of SLSFA-resolution) Let A be an atom. Assume an arbitrary selection rule.

1. Let A' be a ground instance of θ, A such that $\Psi_P^{\alpha+1}(\emptyset) \models A'$ for some ordinal α. Then there exists an SLSFA-computed answer δ for $\leftarrow \theta, A$ such that A' is an instance of δ, A. The answer is obtained from an SLSFA-derivation of rank α.

2. Let $\leftarrow \theta, A$ be a goal such that, for some α, $\Psi_P^{\alpha+1}(\emptyset) \models \neg A'$ for any ground instance $\leftarrow A'$ of the goal. Then there exists a rank α SLSFA-failed tree for this goal.

PROOF

By induction on α. Assume that the lemma holds for any ordinal less than α.

Let $\Psi_P^{\alpha+1}(\emptyset) \models A'$. By the completeness of the SLD-resolution there exists an SLD-refutation R for $\leftarrow A'$ and for program $P/_tI$, where $I = \Psi_P^\alpha(\emptyset)$. In other words R is an SLD-refutation for program *ground*$(P) \cup \{\neg A_1, \ldots, \neg A_l\}$ where $\neg A_1, \ldots, \neg A_l$ are those unary clauses of $P/_tI$ that are used in R and do not belong to *ground*(P). Note that $\neg A_1, \ldots, \neg A_l$ are true in I.

Let R_1 be R with the first goal replaced by $\leftarrow A$. R_1 is, using the terminology of [Llo87], an unrestricted SLD-refutation for program $P \cup \{\neg A_1, \ldots, \neg A_l\}$. By Mgu Lemma of [Llo87], there exists an SLD-refutation R_2 for $\leftarrow A$ and the same program, with the computed answer substitution τ such that A' is an instance of $A\tau$. An obvious transformation of R_2 (that amounts to replacing unifiers by the corresponding constraints) results in a rank 0 SLSFA-refutation R_3 for the same goal and program (still treated as a definite program). The computed answer δ of R_3 is such that A' is an instance of δ, A.

By the inductive assumption, there exist SLSFA-failed trees of rank $< \alpha$ for program P and goals $\leftarrow A_1, \ldots, \leftarrow A_l$. Thus R_3 is a rank α SLSFA-refutation for P and for $\leftarrow A$. Adding θ to every goal of R_3 results in a rank α SLSFA-refutation for P and for $\leftarrow\theta, A$ with the answer $\theta\delta$ satisfying the requirements of the lemma. This completes the induction step for the first part of the lemma.

Consider a goal $\leftarrow\theta, A$ such that for every its ground instance $\leftarrow A'$, $\Psi_P^{\alpha+1}(\emptyset) \models \neg A'$. We construct a rank α pre-failed SLSFA-tree for $\leftarrow\theta, A$. For every its node with a positive literal selected the sons are constructed in a standard way. Let $H = \leftarrow\theta', \overline{M}, \underline{\neg B}, \overline{M'}$ be a node with a negative literal selected. Consider the ground instances $B\tau$ of θ', B. Let $\overline{x}$ be the variables of B. The sons of H are those goals of the form $\leftarrow\theta'(\overline{x}{=}\overline{x}\tau), \overline{M}, \overline{M'}$ for which $\Psi_P^{\alpha}(\emptyset) \not\models B\tau$.

Note that if $\Psi_P^{\alpha}(\emptyset) \models B\tau$ then, by the inductive assumption, $B\tau$ is a ground instance of δ, B for some SLSFA-computed answer δ for $\leftarrow\theta', B$ obtained from a derivation of rank $< \alpha$. Thus the tree is a pre-failed tree.

Now we show that the tree is a SLSFA-failed tree. Consider any ground instance of any branch of the tree. It is an SLD-derivation for program $ground(P) \cup \{\neg B \in \neg\mathcal{H} \mid B \notin \Psi_P^{\alpha}(\emptyset)\} = P/_{tu}I$ where $I = \Psi_P^{\alpha}(\emptyset)$. Suppose that the derivation is successful and that $\leftarrow A'$ is its first goal. Then by the soundness of SLD-resolution, $A' \in \mathcal{M}_{P/_{tu}I}$. From $\Psi_P^{\alpha+1}(\emptyset) \models \neg A'$ it follows that $A' \notin \mathcal{M}_{P/_{tu}I}$. Contradiction. Thus the pre-failed tree does not have a successful branch and it is an SLSFA-failed tree. □

8 Conclusion

Negation as failure has been a standard, although rather restricted, way of dealing with negation in logic programming. In this paper we generalize the notion of failure in order to obtain a constructive negation approach. The generalization is nontrivial as a straightforward approach is incorrect. We introduce SLDFA-resolution which is a method of obtaining answers for normal programs and normal goals, based on building failed trees. SLDFA-resolution is sound and complete with respect to the 3-valued completion semantics, for any fair selection rule.

Our method is a generalization of that of [MN89]. It subsumes SLDNF-resolution and SLDNFS-resolution [She89]. It also subsumes the method of [Cha88]. It neither subsumes nor is subsumed by the methods of [Wal87], [Lug89], [Cha89, Stu91], [SM91] and [Pla92]. Comparison with [Cha89] suggests that our method may be competitive from the efficiency point of view.

A natural extension — allowing infinite failed trees — results in SLSFA-resolution that is sound and complete for the well-founded semantics. Up to our knowledge this is the first constructive negation approach for this semantics.

In another paper [Dra93a] we present an interesting corollary of the completeness of SLDFA-resolution. Namely we prove the completeness of SLDNF-resolution for 3-valued completion semantics for arbitrary programs and queries that do not flounder under appropriate fairness conditions. We expect that SLDFA-resolution should be easily extensible to constraint logic programming by simply allowing general constraints instead of only equality constraints. Teusink [Teu93] applies SLDFA-resolution to abductive logic programming.

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 difference is in 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, respectively for finite and infinite failure.

Acknowledgments

The author wants to thank Teodor Przymusiński for stimulating discussions. Roland Bol, Laurent Fribourg, Feliks Kluźniak, Jan Małuszyński and anonymous referees suggested improvements in the presentation. Torbjörn Näslund co-operated at an early stage of this work.

This work was partially supported by the Swedish Natural Science Research Council (NFR) as a participation in ESPRIT Project BRA 3020 "Integration", 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

[AB94] K. Apt and R. Bol. Logic programming and negation: A survey. Technical Report CS-R9402, CWI Amsterdam, 1994. To appear in *Journal of Logic Programming.*

[Apt90] K. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 10, pages 493–574. Elsevier Science Publishers B.V., 1990.

[BD93] R. N. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 199–219. The MIT Press, 1993.

[BMPT90] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *J. of Logic Programming*, 8(3):201–228, 1990.

[BNN91] S. Bonnier, U. Nilsson, and T. Näslund. A simple fixed point characterization of three-valued stable model semantics. *Information Processing Letters*, 40:73–78, 1991.

[Cav88] L. Cavedon. On completeness of SLDNF resolution. Master's thesis, Dept. of Computer Science, University of Melbourne, 1988. Technical Report 88/17.

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

[Cha89] D. Chan. An extension of constructive negation and its application in coroutining. In E. L. Lusk and R. A. Overbeek, editors, *Proc. North American Conference on Logic Programming,* Cleveland, pages 477–493. MIT Press, 1989.

[CI93] J. Chomicki and T. Imieliński. Finite Representation of Infinite Query Answers. *ACM Transactions on Database Systems*, 1993. To appear.

[CL89] H. Comon and P. Lescanne. Equational problems and disunification. *J. Symbolic Computation*, 7:371–425, 1989.

[DM91] W. Drabent and M. Martelli. Strict completion of logic programs. *New Generation Computing*, 9:69–79, June 1991.

[Dra93a] W. Drabent. On completeness of SLDNF-resolution. Submitted. Modified version of Technical Report LiTH-IDA-R-93-38, Linköping University, 1993.

[Dra93b] W. Drabent. SLS-resolution without floundering. In L. M. Pereira and A. Nerode, editors, *Proc. 2nd International Workshop on Logic Programming and Non-Monotonic Reasoning*, pages 82–98. MIT Press, June 1993.

[FD92] G. Ferrand and P. Deransart. Proof method of partial correctness and weak completeness for normal logic programs. In K. Apt, editor, *Joint International Conference and Symposium on Logic Programming.* MIT Press, 1992.

[Fit85] M. Fitting. A Kripke-Kleene semantics for logic programs. *J. of Logic Programming*, 2(4):295–312, 1985.

[GRS91] A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:620–650, 1991.

[Kha84] T. Khabaza. Negation as failure and parallelism. In *Proc. IEEE Conf. on Logic Programming*, pages 70–75. IEEE Press, 1984.

[Kun87] K. Kunen. Negation in logic programming. *J. of Logic Programming*, 4:289–308, 1987.

[Kun89] K. Kunen. Signed data dependencies in logic programs. *J. of Logic Programming*, 7(3):231–245, November 1989.

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

[Lug89] D. Lugiez. A deduction procedure for first order programs. In G. Levi and M. Martelli, editors, *Proc. of Sixth International Conf. on Logic Programming,* Lisbon, pages 585–599. MIT Press, 1989.

[Mah88] 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.

[MMP88] P. Mancarella, S. Martini, and D. Pedreschi. Complete logic programs with domain closure axiom. *J. of Logic Programming*, 5(3):263–276, 1988.

[MN89] J. Małuszyński and T. Näslund. Fail substitutions for negation as failure. In E. L. Lusk and R. A. Overbeek, editors, *Proc. North American Conference on Logic Programming,* Cleveland, pages 461–476. MIT Press, 1989.

[Näs90] T. Näslund. Sldfa-resolution. Licentiate thesis, Department of Computer and Information Science, Linköping University, 1990.

[Pla92] J. A. Plaza. Fully declarative logic programming. In *Programming Language Implementation and Logic Programming, Proceedings 1992*, pages 415–427. Springer-Verlag, 1992. LNCS 631.

[Prz89a] T. C. Przymusinski. On constructive negation in logic programming. In *Proc. North American Conference on Logic Programming,* Addendum. MIT Press, 1989.

[Prz89b] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Prz90] T. C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundamenta Informaticae*, XIII(4):445–463, 1990.

[Prz91] T. C. Przymusinski. Well-founded completions of logic programs. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming,* Paris, pages 726–741. MIT Press, 1991.

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

[She91] J. C. Shepherdson. Language and equality theory in logic programming. Technical Report PM–91–02, School of Mathematics, University of Bristol, 1991. Newer version of PM–88–08.

[SI88] H. Seki and H. Itoh. A query evaluation method for stratified programs under the extended CWA. In R. A. Kowalski and Bowen K. A., editors, *Proc. of the Fifth International Conference and Symposium on Logic Programming*, pages 195–211. The MIT Press, 1988.

[SM91] T. Sato and F. Motoyoshi. A complete top-down interpreter for first order logic programs. In *Logic Programming, Proc. of the 1991 International Symposium*, pages 35–53. MIT Press, 1991.

[ST84] T. Sato and H. Tamaki. Transformational logic program synthesis. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 195–201, 1984.

[ST86] T. Sato and H. Tamaki. OLD-resolution with tabulation. In E. Shapiro, editor, *Proceedings of the third International Conference on Logic Programming*, pages 84–98. Springer-Verlag, 1986. LNCS 225.

[Stu91] P. Stuckey. Constructive negation for constraint logic programming. In *6th Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, July 1991.

[Teu93] Frank Teusink. Using SLDFA-resolution with abductive logic programs. In *ILPS'93 post-conference workshop "Logic Programming with Incomplete Information"*, 1993.

[Wal87] M. Wallace. Negation by constraints: A sound and efficient implementation of negation in deductive databases. In *Proc. 1987 Symposium on Logic Programming,* San Francisco, pages 253–263. IEEE Computer Society Press, 1987.

[Wal93] M. Wallace. Tight, consistent and computable completions for unrestricted logic programs. *J. of Logic Programming*, 15(3):243–274, 1993.