Regular Search Spaces and Constructive Negation

ALBERTO MOMIGLIANO, *Department of Philosophy, Carnegie Mellon University, 15213 Pittsburgh PA, USA.*
E-mail: am4e@cmu.edu

MARIO ORNAGHI, *Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano, Via Comelico 39/41, Milano, Italy.*
E-mail: ornaghi@imiucca.csi.unimi.it

Abstract

The aim of this paper is to show the fruitfulness and fecundity of the authors' proof-theoretic analysis of logic programming (both for definite and normal programs). It is based on a simple logical framework that goes under the name of *regular search spaces*. The challenge faced here is to give a treatment in proof-theoretic terms of the issue of negation, which has been one of the toughest problems that has plagued logic programming from its very beginning. While negation-as-failure (*NF*) has been overwhelmingly the more widespread answer, its intrinsic limitations have made it a rather unsatisfactory solution. In the present paper it is first contended that the notion of *regularity* offers a better understanding of the traditional theory of *NF*, and second a firm yet very simple and natural basis for a form of *constructive negation*, in the sense of Chan, Stuckey and Harland. A version of constructive negation is presented, based on the notion of *regular splitting*, a transformation technique where the failure axiom(s) of a predicate occurring negatively in a program are split into new clauses according to a covering of the underlying signature.

Keywords: Logic programming, proof-theory, negation-as-failure, constructive negation, search spaces.

1 Introduction

The aim of this paper is to show the fruitfulness and fecundity of the proof-theoretic analysis of logic programming developed in [33] (both for definite and normal programs). It is based on a simple logical framework that goes under the name of *regular search spaces*. Here many logics can be expressed and, provided they are shown to be *regular*, i.e. they satisfy some elementary closure properties, they are then guaranteed to enjoy the very features that make pure Prolog a feasible and successful implementation of a computational logic, namely the Horn fragment.

Some of the tools that we are going to use are the concept of *axiom application rule* (AAR), which can be seen as an abstraction of an inference step of a logic programming interpreter (or more generally as the atomic nucleus of rule-based systems) and of *most general proof-tree* (*mgpt*), which is the analogue of a *SLD*-derivation. Mgpts are in fact based on the notion of *AAR*, which easily generalizes to various definitions of clauses and goals. Finally, in the background, all is connected by the notion of *regular search space*, which plays the role of a Prolog-like search space.

The challenge that we face here is to give a treatment in our proof-theoretic terms of the issue of negation, which has been one of the toughest problem that has plagued logic programming from its very beginning. While negation-as-failure (*NF*) [10] has been overwhelmingly the more widespread answer (see Section 6), its intrinsic limitations have made it a rather unsatisfactory solution. One of the more questionable features is that it is incapable of providing logically justified answers to open queries, consequently restricting negation to be just a test, rather than a logical operator.

In the present paper we will first contend that the notion of regularity offers a better understanding of the traditional theory of *NF* and second a firm, yet very simple and natural basis for a form of *constructive negation*, in the sense of [8, 49, 18], a trend of studies recently pursued aiming to remedy some of *NF*'s maladies. In particular, we shall be concerned with the transformational approach initiated in [5], also known as *intensional* negation [26, 30, 7].

We will introduce an axiom-application rule F which allows one to interpret *SLDNF*-trees as search-trees for F-proofs of negative goals. Due to this analysis of *NF*, the soundness issue related to the *safeness* condition on the selection function is shown to originate from (an analogue of) the usual proviso on parameters of the $\exists$-left and $\forall$-right rule in the sequent calculus [38]. More importantly, the analysis clarifies the intrinsic incompleteness of *NF* due, to a great extent, to the fact that the F rule gives rise to a non-regular search space.

We contend that the incapability of answering negative open queries is due to the non-regularity of *SLDNF*-search trees. We will show that regularity can be achieved through a *splitting* of the given program, obtaining in this way a regular system, where it is possible to answer negative open queries. Splitting is a simple transformation technique where the failure axiom(s) of the predicate definition occurring negatively in the source program are split into new clauses according to a covering [17, 30] of the underlying signature and then executed in an opportune inference system. This is similar to the method in [5] and ancestors (see Section 6 for a comparison).

We want to stress at this point that our interest lies *mainly* in showing the versatility and the adaptability of our approach—having digested a few simple initial definitions—rather than keeping up with front-line research on negation in logic programming: in particular we shall deal only marginally with new developments in the field about constructive negation, completion, answers sets and completeness of *NF* (see for example [44, 6] and the recent survey [3]). Similarly, our review of related work is meant to help the reader to *situate and compare* our approach to the aforementioned issues, rather than detailed discussions about its edge over other proposals.

This paper, therefore, has in part a pedagogical vein. Our long-term intent is to show that many issues in the theory of logic programming, starting from *NF*, which are well-known but not necessarily well-*understood*, have a very natural, elegant, concise and stimulating proof-theoretic reading. Besides, the latter can provide the researcher with useful tools that are indeed relevant to current research (see, for example, Stärk's thesis [47] and following papers). Note that there are several points where we move from our reconstruction of folk material to the presentation of new ideas and connections.

The paper is organized as follows: in Section 2 we review the proof-theory of logic programming, from the abstract point of view formulated in [33] down to a reconstruction of *SLD*-resolution. Section 3 formulates the theory of Regular *AAR*-Systems. Section 4 deals with the proof-theory of *NF* and the related soundness and completeness problem. Section 5 proposes regular splitting as a solution to the problem of evaluating open negative queries. Eventually we give a version of the method in the positive fragment of pure Prolog and in Section 6 we compare our approach with other treatments of negation in logic programming. In the Appendix we detail the proof of the Finite Failure Theorem (4.4), stated in Section 4.

2 Systems based on axiom-application rules

Our view of an abstract[1] logic programming system is that of an idealized interpreter endowed with *rules*—the inference mechanism—that apply *axioms*—the program—starting from a goal and searching for a proof. We formulate this approach in all its generality and we exemplify it with some systems that are related to $SLD(NF)$-resolution. We assume some familiarity with logic programming [27, 2] and basic proof theory [38, 42, 16].

2.1 *AAR-systems*

An AAR-system is a triple $\langle \mathcal{G}, \mathcal{A}, \mathcal{R} \rangle$ where:

1. $\mathcal{G}$ is a set of admissible goals. In this paper those will be literals, but more general forms could be used (see [33]). Goals are therefore distinct from Prolog goals, denoted as usual by the sequence $\leftarrow L_1, \ldots, L_n$.
2. $\mathcal{A}$ is a set of admissible axioms. For example, that could be (the universal closure of) definite or normal clauses, the completed definitions of the predicates of a program, or more general kinds of axioms (see [33]).
3. $\mathcal{R}$ is a set of axiom application rules. A rule $R \in \mathcal{R}$ is any relation from goals and axioms to sequences of goals, including the empty sequence Λ, i.e. $R \subseteq (\mathcal{G} \times \mathcal{A}) \times \mathcal{G}^*$.

We shall say that $G_1; \ldots; G_n \in R(G, Ax)$ for those sequences of goals $G_1; \ldots; G_n$ such that $\langle\langle G, Ax\rangle, G_1; \ldots; G_n\rangle \in R$ (namely $R(G, Ax)$ is a set of sequences of goals). Using semicolons to separate goals as well as axioms, we will draw this as

$$\frac{G_1; \cdots; G_n; \quad Ax}{G}(R).$$

When $\Lambda \in R(G, Ax)$, we call Ax a *fact* and we write

$$\frac{Ax}{G}(R).$$

A program $\mathcal{P}$ is a set of axioms from $\mathcal{A}$.

NOTATION 2.1
If A is an atom, $neg(A) = \neg A$; otherwise $neg(\neg A) = A$. Capital letters will denote logical variables, while lower case will be reserved for terms, quantified variables and ***eigenvariables*** [38]. Recall that the latter (often called *parameters*) are variables whose only possible substitutions are capture-freeing renaming. $\forall(M)$ ($\exists(M)$) will denote the universal (existential) closure of M, whose free variables are bound by universal (existential) quantifiers. For the sake of mental hygiene, we will tacitly confuse terms with *tuples of terms*: for example $Q.y$ will either denote a (possibly empty) list of quantified variables, or a single occurrence binding the variable y. It will be apparent from the context which is intended.

We now introduce, as examples, the systems we will be more interested in.

[1]The notion of *abstract* logic programming language was introduced some years ago by Miller *et al.* [31] as one satisfying some constructive provability conditions, interpreted as search instructions; in this respect the two notions are complementary, since basic in our approach is the accent on *regularity* as an analysis of the conditions for completeness in abstract search spaces.

- The P-system contains a single rule P: the admissible goals are literals and it applies axioms of the form $\forall(\forall y(L_1 \wedge \cdots \wedge L_n) \rightarrow M)$, where M, $L_1, \ldots, L_n$ are literals and y may appear in L_i but not in M. Note that the presence of universal quantifiers in the body of clauses is a mild extension of the Horn format in the direction of Harrop formulae [31]. If $n = 0$, then the axiom is $\forall(M)$. The rule P is defined as follows. For every substitution θ renaming y with eigenvariables:

$$\theta L_1; \ldots; \theta L_n \in \mathrm{P}(\theta M, \forall(\forall y(L_1 \wedge \cdots \wedge L_n) \rightarrow M)).$$

When we need to differentiate in the P-system applications of rules to positive and negative goals, we shall use the obvious notation P^+, P^-.

An application of P with positive conclusion is, for example:

$$\frac{\neg sum(v,v,X); \quad \forall x(\forall z \neg sum(z,z,x) \rightarrow odd(x))}{odd(X)}\,(\mathrm{P}^+)$$

where an eigenvariable v has been introduced for z.

The P rule can be shown to be admissible in minimal logic: its instances are derivable in natural deduction, along the following lines, where the vertical dots allude to a closing branch for the assumption $\neg sum(v,v,X)$: observe the interplay among the different type of variables:
– the eigenvariable v has uniformly substituted z
– X is a logical variable
– x is universally quantified.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \neg sum(v,v,X)\end{array}}{\forall z.\neg sum(z,z,X)}\,\forall\text{-}I \qquad \dfrac{\forall x(\forall z \neg sum(z,z,x) \rightarrow odd(x))}{\forall z \neg sum(z,z,X) \rightarrow odd(X)}\,\forall\text{-}E}{odd(X)}\,\rightarrow\text{-}E$$

- The F-system. We allow rules in which some proper (i.e. non-logical) axiom is implicitly used. In Clark's equality theory [10] we can derive the following *failure rule* F to apply *failure axioms* of the form:

$$Fax(p): \quad \forall x(p(x) \rightarrow \exists y((x = t_1 \wedge L_1) \vee \cdots \vee (x = t_n \wedge L_n))),$$

related, as we will see, to the only-if part of the completion axiom for a predicate definition $p(\ldots)$ [27]:[2]

$$neg(\sigma_1 L_{i_1}); \ldots; neg(\sigma_k L_{i_k}) \in \mathrm{F}(\neg p(a), Fax(p)),$$

where a is a term which unifies only with $t_{i_1}, \ldots, t_{i_k}$ with idempotent mgus $\sigma_1, \ldots \sigma_k$. Moreover, if $\sigma_h L_{i_h}$ contains, modulo renaming, some of the existentially quantified y, those variables must be *new* (w.r.t. a) eigenvariables.

[2] It is possible to present an alternative multiple goals characterization of *SLDNF*-resolution [33]. This *AAR*-system, based on the standard notion of completion, can be enriched by other rules, for instance connected to model elimination [48], that cannot be formulated in the other (weaker) system.

Examples of applications of F are, given the standard program for the *member* predicate and its failure axiom *Fax(member)*:

$$\begin{array}{lcl} member(X, X.XS). & & \\ member(X, Y.YS) & :- & member(X, YS). \\ \forall x, xs(member(x, xs) & \rightarrow & \exists y, ys, z((x = y \land xs = y.ys) \lor \\ & & (x = z \land xs = y.ys \land member(z, ys))) \end{array}$$

$$\frac{Fax(member)}{\neg member(X, nil)}(\mathrm{F}) \qquad \frac{\neg member(1, W) \quad Fax(member)}{\neg member(1, 2.W)}(\mathrm{F})$$

In the first case unification fails, hence $\Lambda \in \mathrm{F}(\neg member(X, nil), Fax(member))$. In the second case the mgu $[z/1, y/2, ys/W]$ yields:

$$\neg member(1, W) \in \mathrm{F}(\neg member(1, 2.W), Fax(member)),$$

where no eigenvariable occurs in $\neg member(1, W)$, since z, y and ys have been replaced by the mgu. See Example 3.22 for a different situation.

- The PF-system contains, guess what, the rules P and F.

The set of proof-trees $\mathcal{T}(\mathcal{G}, \mathcal{A}, \mathcal{R})$ of an *AAR*-system $\langle \mathcal{G}, \mathcal{A}, \mathcal{R} \rangle$ is inductively defined below, with $\Pi :: G$ as our linear notation for a proof-tree Π with root G:

DEFINITION 2.2
Every $G \in \mathcal{G}$ is a proof-tree. If $\Pi_1 :: G_1, \ldots, \Pi_n :: G_n$ are proof-trees and $G_1; \ldots; G_n \in R(G, Ax)$, then the following is also a proof-tree:

$$\frac{\begin{array}{cc}\Pi_1 & \Pi_n \\ G_1; \ldots G_n; & Ax\end{array}}{G}(R)$$

We say that a goal is an *assumption* of a proof-tree if it is a minor premiss in some leaf. The *axioms* of a proof-tree are those appearing as major premisses. The root of a proof-tree is called its *consequence.*

EXAMPLE 2.3
To illustrate the latter notion, we give a F-proof-tree with consequence $\neg member(1, [2, 3])$, assumption $\neg member(1, nil)$ and two occurrences of the axiom $Fax(member)$.

$$\frac{\dfrac{\neg member(1, nil) \quad Fax(member)}{\neg member(1, [3])}(F) \qquad Fax(member)}{\neg member(1, [2, 3])}(F)$$

DEFINITION 2.4
A proof-tree is a *proof* of G iff G is its consequence and it has no assumption. Otherwise it is a called a *partial* proof-tree. If the axioms of a proof-tree belong to a program $\mathcal{P} \subseteq \mathcal{A}$, we say that it is a proof-tree with axioms from $\mathcal{P}$. The *height* of a proof-tree is the height of its longest branch.

2.2 *P-system and SLD-resolution*

To a definite clause C we associate an axiom $Ax(C)$ and to a program P the set $Ax(P)$ of the axioms which correspond to its clauses, in the obvious way. For example, let us consider the program SUM for computing the sum, containing the following clauses s_1, s_2:

$$\begin{array}{ll} s_1 : & sum(X, 0, X). \\ s_2 : & sum(X, s(Y), s(Z)) : -sum(X, Y, Z). \end{array}$$

The associated axioms *Ax(SUM)* are:

$$\begin{array}{ll} Ax(s_1) : & \forall x(sum(x, 0, x)) \\ Ax(s_2) : & \forall x, y, z(sum(x, y, z) \rightarrow sum(x, s(y), s(z))). \end{array}$$

SLD-derivations corresponds to the inferences in the P-system, with the following restrictions: no universal quantifier occurs in the body of a clause and no negative literal is involved. The set of proof-trees of the P-system is closed under substitution and the application of a substitution to a proof-tree Π is denoted by $\theta\Pi$.

According to Theorem 2.6 we associate *continuations* of proof-trees to *SLD*-steps.

DEFINITION 2.5 (Continuation)
Let $Ax(C) = \forall(A_1 \wedge \ldots \wedge A_n \rightarrow B)$ be an axiom corresponding to a clause C and

$$\begin{array}{c} \ldots H \ldots \\ \Pi \end{array}$$

be a proof-tree with an assumption H s.t. $\theta B = \theta H$, for some substitution θ. The *continuation* of Π *selecting* H and *applying* $Ax(C)$ w.r.t. θ is the proof-tree:

$$\begin{array}{c} \dfrac{\theta A_1; \cdots ; \theta A_n \quad Ax(C)}{\ldots \theta H \ldots} \\ \theta\Pi \end{array}$$

THEOREM 2.6 (Soundness and completeness of the P-system)
Let P be a definite program and A an atom:

(a) If there is a *SLD*-refutation for $P \cup \{\leftarrow A\}$ with answer substitution δ, then there is a proof $\Pi :: \delta A$ in the P-system, using only axioms from $Ax(P)$.

(b) If there is a proof $\Pi :: \theta A$ in the P-system with axioms from $Ax(P)$, then there is a *SLD*-refutation of $P \cup \{\leftarrow A\}$ and the answer substitution δ is such that $\theta = \sigma\delta$, for a suitable σ.

PROOF. Point (b) follows from the validity of the P-system w.r.t. classical first order logic ([27], Theorem 8.6).

Point (a) can be proved as follows. Let $G_0, G_1, \ldots, G_m$ (with clauses $C_1, \ldots, C_m$ and substitutions[3] $\theta_1, \ldots, \theta_m$) be a refutation, with $G_0 = \leftarrow A$. For $0 \leq i \leq m$, let δ_i be the composition of the $\theta_1, \ldots, \theta_i$ and $\leftarrow A_{i_1}, \ldots, A_{i_{h_i}}$ be the atoms in the goal G_i; starting with $i = 0$, we associate to $G_0, \ldots, G_i$ a proof-tree $\Pi_i :: \delta_i A$ with assumptions $A_{i_1}, \ldots, A_{i_{h_i}}$, as follows:

[3]Note that the proof does not rely on $\theta_1, \ldots, \theta_m$ being mgus.

Step 0. Associate A to G_0.

Step i+1. Let $\Pi :: \delta_i A$ be the proof-tree associated to $G_0, \ldots, G_i$ at *step i* and let G_{i+1} be obtained applying C_{i+1} to the selected atom A_{n_i}, with *mgu* θ_{i+1}; build the continuation of $\Pi_i :: \delta_i A$ selecting A_{n_i} and applying $Ax(C_{i+1})$ w.r.t. θ_{i+1}.

The last goal G_m is empty, hence the last proof-tree $\Pi_m :: \delta_m A$ has no assumptions, i.e. it is a proof, and δ_m is the answer substitution. ∎

3 Regular AAR-systems

Now we approach the analysis of AAR-systems in an abstract setting, from the point of view of proof-search. This is quintessential to understand the intrinsic properties of logic programming and to evaluate any departure from *SLD*-resolution as its kernel. In particular, this section is a fundamental preliminary to our proof-theoretic treatment of NF inasmuch as it introduces the key notion of *similarity* and *regularity* together with the central results of the theory.

The search-space of an AAR-system can be organized as a search-tree, where nodes are (partial) proof-trees and arcs (search steps) are continuations (see Definition 2.5 and Theorem 2.6). A leaf that contains a proof is a *success node*, and a leaf that contains a partial proof-tree is a *failure node*. *Finite failure* can be characterized as a property of the set of failure nodes.

In general, search in the complete tree is intractable. One of the problems is computing the right substitutions. It can be dealt with in the following way.

For the P-system, the subsumption ordering [23] on first-order terms can be lifted to proof-trees [19], and *most general proof trees* (mgpts) are defined as the maximal elements w.r.t. this ordering. In 'good' systems search can be pursued only in the subspace of the mgpt, through *most general continuations* (mgcs); mgcs correspond to *SLD*-steps in logic programming systems. The completeness of the search in the subspace of the mgpt depends on a property of the search space, that we call *regularity*. The idea of regularity, in its more general setting, can be outlined as follows.

A subspace is obtained through a suitable equivalence relation among proof-trees, i.e. it is built by an appropriate quotientation of the (entire) search space. Regularity is a property of the equivalence classes. It ensures that a regular subspace is *success-complete*, that is for every successful path from a goal G to a proof Π in the complete search space, the subspace contains at least one path from the equivalence class of G to the one of Π.

Thus regularity entails that a search strategy working on representatives of the equivalence classes will not miss success nodes. Note that success-completeness deals with the completeness of a search strategy w.r.t. a given proof system, not with the one of the proof system w.r.t. some logic; the latter is to be studied by different (traditional) means.

3.1 Systems closed under substitution

Properties of substitutions, or more properly of instantiations, will turn out to be central in our treatment. Hence we have to restrict to sets $\mathcal{G}$ of goals for which a notion of substitution as an answer/result of a computation makes sense. Assuming that the application of a substitution to a goal is well-defined, it is clear how to extend it to trees. We suppose, as well, that axioms and rules are not affected by substitutions.

Note that, under the more general version that we are developing, Π may belong to $\mathcal{T}(\mathcal{G}, \mathcal{A}, \mathcal{R})$, while $\theta\Pi$ does not. To ensure this, we introduce the following:

DEFINITION 3.1
We say that an AAR-system $\langle \mathcal{G}, \mathcal{A}, \mathcal{R}\rangle$ is *closed under substitution* if, for all θ, $G \in \mathcal{G}$ entails $\theta G \in \mathcal{G}$ and, for all $R \in \mathcal{R}$, $Ax \in \mathcal{A}$ and $G \in \mathcal{G}$, $R(\theta G, Ax) = \theta R(G, Ax)$.

As mentioned, the P-system is closed under substitution. As far as the $F(P)$-system is concerned, the situation is more complicated as discussed in subsection 3.5.

One easily sees that, if $\langle \mathcal{G}, \mathcal{A}, \mathcal{R}\rangle$ is closed under substitution, so is $\mathcal{T}(\mathcal{G}, \mathcal{A}, \mathcal{R})$, i.e. $\Pi \in \mathcal{T}(\mathcal{G}, \mathcal{A}, \mathcal{R})$ entails $\theta\Pi \in \mathcal{T}(\mathcal{G}, \mathcal{A}, \mathcal{R})$.

This allows us to introduce the following *pre-ordering* (intuitively to be read as 'Π_1 is less general or more instantiated than Π_2') and *equivalence* relation among proof-trees.

DEFINITION 3.2 (Subsumption ordering for proof-trees)

- $\Pi_1 \leq \Pi_2$ iff there is a θ such that $\Pi_1 = \theta\Pi_2$.
- $\Pi_1 \equiv \Pi_2$ iff $\Pi_1 \leq \Pi_2$ and $\Pi_2 \leq \Pi_1$.

Note that the induced equivalence relation on proof-trees corresponds to identity of trees modulo renaming of variables.

We will be mainly interested in most general proof-trees, defined as follows.

DEFINITION 3.3 (Mgpt)
Π^* is a *most general proof-tree* if it is a maximal element w.r.t. $\leq$, that is, for every Π, $\Pi^* \leq \Pi$ entails $\Pi \equiv \Pi^*$.

3.2 *Search spaces for AAR-systems*

Now, let us consider how we could approach the following *search problem* in a Prolog-like way, where (finite) sets of axioms are programs and the desired outcome of the computation are answer substitutions.

> Let $\mathcal{P}$ be a program and $G \in \mathcal{G}$ a goal: search for a proof $\Pi :: \theta G$ with axioms from $\mathcal{P}$, for some substitution θ.

If a proof (i.e. a proof-tree *without assumptions*) $\Pi :: \theta G$ exists, we say that θ is an answer substitution for G w.r.t. $\mathcal{P}$. If, on the other hand, every proof-tree $\Pi :: \theta G$ has assumptions, we say that the goal G fails w.r.t. $\mathcal{P}$.

First of all, we characterize our *complete* search space through the following notion of *one-step continuation*, which generalizes Definition 2.5.

DEFINITION 3.4
Given $G \in \mathcal{G}$, $Ax \in \mathcal{P}$ and $R \in \mathcal{R}$, we say that Ax can be *applied to G by R using θ* iff $G_1;\ldots;G_n \in R(\theta G, Ax)$. Given a proof Π with at least one assumption G, the above application gives rise to a *one-step continuation*, as follows:

$$\begin{array}{c} \dfrac{G_1;\cdots;G_n \quad Ax}{\ldots\ \theta G \ldots}(R) \\ \theta\Pi \end{array}$$

Iteration yields many-step continuations. There is a more abstract alternative characterization (illustrated in Figure 1):

DEFINITION 3.5
Call Π' an *initial subtree* of Π iff Π' is a subtree of Π and they have the same root. Then Π_2 is a *continuation* of Π_1, denoted $\Pi_1 \preceq \Pi_2$, iff there is an initial subtree Π_3 of Π_2, s.t. $\Pi_3 \leq \Pi_1$.

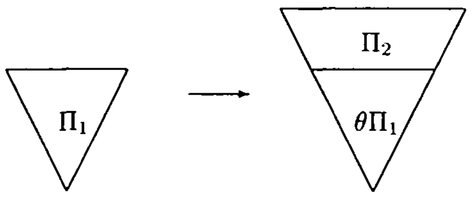

FIG. 1. Alternative characterization of continuation

Note that $\Pi_1 \leq \Pi_2$ implies $\Pi_1 \preceq \Pi_2$. In this case we will speak of the *trivial* continuation.

PROPOSITION 3.6
For the non-trivial case, $\Pi_1 \preceq \Pi_2$ iff Π_2 is a many-step continuation of Π_1.

One immediately sees that G has an answer substitution θ w.r.t. a program $\mathcal{P}$ iff there is a continuation $\Pi :: \theta G$ of the 0-height proof-tree G, such that Π is a proof. Then our search problem can be restated as follows:

Let $\mathcal{T}(\mathcal{G}, \mathcal{A}, \mathcal{R})$ be the set of proof-trees of a fixed AAR-system. Let $\mathcal{T}(\mathcal{P}) \subseteq \mathcal{T}(\mathcal{G}, \mathcal{A}, \mathcal{R})$ be the (sub)set of the proof-trees with axioms from $\mathcal{P}$ and $\mathcal{T}(\mathcal{P}, G) \subseteq \mathcal{T}(\mathcal{P})$ be the (sub)set of the continuations of G. The $\preceq$ relation is easily seen as a pre-ordering on each of those sets. As hinted above, to obtain a partial ordering we have to take the quotient $\mathcal{T}(\mathcal{P})/\equiv$ (i.e. consider proof-trees modulo variable renaming). Finally, take the po-set (that through standard duplications can be treated as a tree with root G):

$$\langle \mathcal{T}(\mathcal{P}, G)/\equiv, \preceq \rangle.$$

The leafs are (equivalences classes) of proof-trees which have no continuation; in particular, a *success node* is a leaf containing a proof. Otherwise, they are *failure nodes*. In particular, $\langle \mathcal{T}(\mathcal{P}, G)/\equiv, \preceq \rangle$ is *failed* iff every leaf is a failure node (see Section 3.4 for more on that).

The po-set $\langle \mathcal{T}(\mathcal{P}, G)/\equiv, \preceq \rangle$ is the complete search space we mentioned above and it is the starting point for our analysis of regularity. It contains *all* the proof-trees (modulo renaming) and our search problem corresponds to the search of success nodes in such a tree.

For every node $[\Pi]$, where square brackets denote the equivalence class witnessed by Π, $[\Pi']$ is a (non-trivial) child of $[\Pi]$ iff the former is a one-step continuation of the latter.

According to Definition 3.4, a one-step continuation is parametrized by a 4-tuple $\langle G, Ax, R, \theta \rangle$. Therefore sequences $\langle \langle G_0, Ax_0, R_0, \theta_0 \rangle \rangle, \ldots, \langle G_n, Ax_n, R_n, \theta_n \rangle \rangle$ correspond to non-trivial paths in the tree. Thus, in general, we may have to backtrack on four *dimensions* (choices of $\langle G_k, Ax_k, R_k, \theta_k \rangle$). Moreover, due to the presence of substitutions, even using a finite set of axioms and rules, a node may have infinitely many children.

Consequently it is desirable to eliminate at least the need to backtrack on substitutions. This is what is achieved by first-order resolution, thanks to the existence of most general unifiers. Moreover, *SLD*-resolution enjoys the independence of the selection function [27]. In our model this is reflected by Proposition 3.14.

It could be expected that the elimination of some dimension of backtracking might cause success-incompleteness of a search strategy. *SLD*-resolution is success-complete, but, as we will see, it becomes success-incomplete when constructive negation[4] is considered.

[4]Constructive in the sense of being provable in a constructive logic from the completion—this only partially coincides with the use of the term in the literature [8, 9, 49].

To (re)achieve success-completeness, we have to accept, in a first approximation, the re-introduction of the dreadful dimension of backtracking on substitutions. As shown in [33], the possibility of using a success-complete resolution method analogous to *SLD*-resolution depends on the property of regularity of the search space $\langle \mathcal{T}(\mathcal{P}, G)/\equiv, \preceq\rangle$. The treatment is based on the notions of *similarity*. In the next subsection we recall the main definitions and results. More details and proofs can be found in [33].

3.3 Regular search spaces

In our model the possibility of using a resolution-like method corresponds to the computation of *most general continuations*, among the (possibly infinite) *similar continuations*, where similarity is a suitable equivalence relation among proof-trees.

To informally motivate the notion of similarity, let us consider a path in the search tree $\langle \mathcal{T}(\mathcal{P}, G_0)/\equiv, \preceq\rangle$:

$$\langle\langle G_0, Ax_0, R_0, \theta_0\rangle, \ldots, \langle G_n, Ax_n, R_n, \theta_n\rangle\rangle.$$

Let us call *similar* two paths determined by the same sequence of axioms and rules, but possibly by different sequences of substitutions. Two proof-trees are *similar* if they can be obtained by similar paths.

Now, suppose that every set of similar proof-trees contains a most general proof-tree subsuming the others: it is apparent that a Prolog-like (idealized) interpreter will preferably compute on this one, forget about the others and in particular avoid backtracking on the selection of substitutions. The problem is to achieve *success-completeness*, that is no success node should be lost in this way. To study success-completeness in its generality, it is convenient to formulate similarity in a more abstract way, as a *structural* property of proof-trees:

DEFINITION 3.7
An *axiom/rule-occurrence* in a proof-tree Π is a triple $\langle p, Ax, R\rangle$ such that p is a path in Π from the root to a node containing an axiom Ax applied by a rule R. We say that two proof-trees Π_1, Π_2 are *similar*, written $\Pi_1 \sim \Pi_2$, if they have the same (non-empty) set of axiom/rule-occurrences.

Note that in the previous definition, substitutions do not play any role, as expected: two proof trees are similar if and only if they can be obtained through similar paths. One easily sees that $\sim$ is an equivalence relation; the corresponding equivalence classes, denoted by $[\Pi]_\sim$, will be called *similarity classes*.

We use similarity to curtail the subspace $\langle \mathcal{T}(\mathcal{P}, G)/\sim, \preceq\rangle$, where the continuation relation $\preceq$ has been lifted to similarity classes as follows:

$$[\Pi_1]_\sim \preceq [\Pi_2]_\sim \text{ iff there are } \Pi_1 \in [\Pi_1]_\sim, \Pi_2 \in [\Pi_2]_\sim \text{ s.t. } \Pi_1 \preceq \Pi_2. \qquad (3.1)$$

It is apparent that any path computed by an interpreter that does not perform backtracking on substitutions corresponds to a path in this subtree. Thus our quotientation is adequate to study the behaviour of interpreters of this kind.

An interpreter works on proof-trees, not on equivalence classes. Hence it chooses suitable representatives of the equivalence classes, and different choices correspond to different search strategies. Since (3.1) does not require that every representative Π of $[\Pi_1]_\sim$ has a continuation in $[\Pi_2]_\sim$, a complete search strategy has to choose a 'good representative' of $[\Pi_1]_\sim$, i.e. a $\Pi \in [\Pi_1]_\sim$ that has a 'good representative' of $[\Pi_2]_\sim$ as a continuation.

Thus the first condition that our subspace must satisfy is that good representatives exist. Moreover, since an interpreter will compute only on the latter, we also require that our intuition of 'representatives' of all the proof-trees belonging to their similarity classes is met.

Now we claim that regularity, which is the basis for the existence of most general proof-trees among similar trees, ensures both the above conditions. Regularity is defined as follows:[5]

DEFINITION 3.8 (Regular search space)
A set $\mathcal{S}$ of proof-trees is a *regular search space* iff, for every similar $\Pi_1, \Pi_2 \in \mathcal{S}$, there is a $\Pi \in \mathcal{S}$ such that $\Pi_1 \leq \Pi$ and $\Pi_2 \leq \Pi$.

Note that regularity is parameterized by the notion of similarity we have chosen to deal with. The one presented here is the simplest and corresponds to the eliminability of backtracking on substitutions in Prolog-like languages. Regularity can be made more interesting, for example introducing versions of similarity that take into account permutability of rules, i.e. for richer fragments, like hereditary Harrop formulae, where all rules are permutable and therefore it is possible to restrict to a good representative, namely uniform proofs [31], so that there is no backtracking on rule application.

Coming back to our analysis of regularity, let us consider any regular search space $\mathcal{S}$. For example, $\mathcal{S}$ could be the set of proof-trees of a program $\mathcal{P}$, $\mathcal{T}(\mathcal{P})$, or its subset $\mathcal{T}(\mathcal{P}, G)$, in the P- system, as we will see in the next subsection. The possibility to avoid backtracking on substitutions is connected to the following propositions (for more details and proofs see [33]).

PROPOSITION 3.9
$\mathcal{S}$ is a regular search space iff every similarity class $[\Pi]_\sim$ contains a proof-tree Π^* such that, for every $\Pi' \in [\Pi]_\sim$, $\Pi' \leq \Pi^*$, i.e. Π^* is a mgpt representing the former class.

One easily proves that, for two mgpts $\Pi_1^*, \Pi_2^* \in [\Pi]_\sim$, $\Pi_1^* \equiv \Pi_2^*$; therefore every similarity class contains a mgpt, which is unique up to renaming. This mgpt represents all the proof-trees of the class, in the sense that it subsumes them. Moreover, it represents a good choice for an interpreter, due to the following proposition.

PROPOSITION 3.10
If Π_1 is a mgpt, then, for every Π_2 such that $\Pi_1 \sim \Pi_2$, if there is a Π s.t. $\Pi_2 \preceq \Pi$, then $\Pi_1 \preceq \Pi$.

PROOF. Let Π be a continuation of Π_2. Then $\theta\Pi_2$ is an initial subtree of Π. Since Π_1 is a mgpt similar to Π_2, there exists a substitution σ s.t. $\Pi_2 = \sigma\Pi_1$. Hence $(\theta\sigma)\Pi_1$ is an initial subtree of Π, i.e. Π is a continuation of Π_1. ∎

As a corollary we obtain:

PROPOSITION 3.11
Let Π_1, Π_2 be mgpts. Then $[\Pi_1]_\sim \preceq [\Pi_2]_\sim$ iff $\Pi_1 \preceq \Pi_2$.

PROOF. The right-to-left direction is obvious. Conversely, let $[\Pi_1]_\sim \preceq [\Pi_2]_\sim$; then there are $\Pi \in [\Pi_1]_\sim$, $\Pi' \in [\Pi_2]_\sim$ s.t. $\Pi \preceq \Pi'$. By Proposition 3.10, $\Pi_1 \preceq \Pi'$ and $\Pi' = \theta\Pi_2$ (since the latter is a mgpt). Therefore $\theta\Pi_2$ contains an initial subtree $\theta\Pi^* \leq \Pi_1$, and then Π^* is an initial subtree of Π_2 similar to Π_1. Since Π_1 is a mgpt, $\Pi^* \leq \Pi_1$, i.e. $\Pi_1 \preceq \Pi_2$. ∎

[5] Regularity is connected to *generalization* or *anti-unification*, independently introduced by Reynolds and Plotkin (see [23] and references therein) in the lattice of (first-order) terms. This has been further explored in [19]. Under the propositions-as-types interpretation, proof-trees are proof λ-terms. [37] presents [anti]unification algorithms in the Calculus of Constructions, although restricted to *higher-order patterns*. From their unary unification problem [32], the existence of a mgpt is guaranteed.

Thus mgpts can be chosen as good representatives, and we can model our subspace as follows. Let $\mathcal{S}$ be a regular search space and $\mathcal{T}(\mathcal{S}, G)$ the set of the proof-trees $\Pi :: \theta G \in \mathcal{S}$; define $Gen(\mathcal{S})$ and $Gen(\mathcal{S}, G)$ to be the corresponding sets of mgpts. By Proposition 3.11, the subspace $\langle \mathcal{T}(\mathcal{S}, \mathcal{G}) / \sim, \preceq \rangle$ is isomorphic to $\langle Gen(\mathcal{S},G) / \equiv, \preceq \rangle$ and we can therefore operate on the latter. To analyse the properties of this subspace, and to understand the underlying geometry, we introduce the notion of *canonical continuation* of a proof-tree.

DEFINITION 3.12
A continuation Π^* of a proof-tree Π is a *most general continuation* (*mgc*) if, for every other continuation Δ similar to Π^*, $\Delta \leq \Pi^*$. A continuation is *canonical* iff it is a one-step mgc.

PROPOSITION 3.13
If Π is a mgpt, then its mgcs are mgpts. In particular, its canonical continuations are mgpts.

By the above proposition $\langle Gen(\mathcal{S}, \mathcal{G}) / \equiv, \preceq \rangle$ can be built using only canonical continuations, thus avoiding even the problem to choose substitutions. Moreover, we can prove:

PROPOSITION 3.14
Let Π be a mgpt of $\mathcal{S}$ and H be an assumption of Π. If there is a proof Δ that is a mgc of Π, then there is a canonical continuation Π' of Π selecting H such that Δ is a mgc of Π'.

PROOF. Suppose the contrary, that there is a proof-tree Δ that is a mgc of Π, but there are no canonical continuation Π' of Π selecting H such that Δ is a mgc of Π'. But there is an initial subtree $\tilde{\Delta}$ of Δ that is a one-step continuation of Π selecting H. Hence $\tilde{\Delta}$ is similar to the canonical continuation Π^* of Π selecting H. By Proposition 3.13, Π^* is a mgpt and, by Proposition 3.10, it continues in Δ, absurdum. ∎

Proposition 3.14 shows that, by using canonical (i.e. most general) continuations, during the search the selection of the assumption H may be completely non-deterministic. Therefore, we can further reduce the search space by using selection functions, which associate to every proof-tree one of its assumptions.

DEFINITION 3.15
A *selection function* is a mapping $F : \mathcal{T}(\mathcal{S}) / \equiv \longrightarrow \mathcal{G}$. An *F-search tree* is a subtree of $\langle Gen(\mathcal{S}, \mathcal{G}) / \equiv, \preceq \rangle$ such that, for every node $[\Pi]$, its children are the canonical continuations selecting the assumption $F([\Pi])$. Moreover, $\preceq_F$ will denote the subset of the continuation relation such that the selected assumption in the continuation step is chosen by F.

It is clear that $\preceq_F$ is still a partial order and $\langle \mathcal{T}(\mathcal{S}, \mathcal{G}) / \sim, \preceq_{\mathcal{F}} \rangle$ is a subtree of $\langle \mathcal{T}(\mathcal{S}, \mathcal{G}) / \sim, \preceq \rangle$.

COROLLARY 3.16
For every selection function F, $\langle Gen(\mathcal{S}, \mathcal{G}) / \equiv, \preceq_{\mathcal{F}} \rangle$ is success-complete.[6]

This is a second reason, beyond eliminating backtracking on substitutions, for stressing the relevance of regularity in logic programming.

Now we say that an *AAR*-system is *regular* iff the set of its proof-trees is a regular search space. As one can easily see, the regularity of an *AAR*-system implies the regularity of the subspaces $\mathcal{T}(\mathcal{P})$ and $\mathcal{T}(\mathcal{P}, \mathcal{G})$. In $\mathcal{T}(\mathcal{P}, \mathcal{G})$, we can avoid backtracking on substitutions and search only for most general proof-trees. Therefore in an *AAR*-system we can use essentially the same search strategy adopted for *SLD*-resolution and the same main results hold.

[6]From now on, we suppress mention to the renaming quotientation.

The problem is then how to compute canonical continuations. We say that there is a ***resolution method*** when canonical continuations can be computed depending on the selected assumption and not on the whole proof-tree.

DEFINITION 3.17 (Resolution method)
A partial function $Res(G, Ax, R)$ is a ***resolution method*** iff:

- $Res(G, Ax, R)$ is defined iff Ax can be applied to G by R.
- $Res(G, Ax, R) = [\langle \theta, G_1; \ldots; G_n \rangle]$, where $G_1; \ldots; G_n \in R(\theta G, Ax)$ and the corresponding continuations are canonical.[7]

If the search space is regular, then for every R, Ax and every proof-tree with selected assumption G, either there is canonical continuation ($Res(G, Ax, R)$ is defined) or no continuation exists ($Res(G, Ax, R)$ is not defined). Moreover, for the same R, Ax and selected G, any two canonical continuations are equivalent; therefore $Res(G, Ax, R)$ is defined as an operator computing equivalence classes. This operator imports all the search properties of pure Prolog, in particular the independence of the selection function with respect to success-completeness.

3.4 *Finite failure and* AAR-*systems*

There is a natural proof-theoretic characterization of finite failure in an *AAR*-system: call an assumption *failed* with respect to a program $\mathcal{P}$, if no axiom of $\mathcal{P}$ can be applied to it, and call a proof-tree *k-failed* iff it contains at least one failed assumption which occurs in a branch at height less or equal to k.

DEFINITION 3.18 (k-failure)
A search-tree $\langle \mathcal{T}(\mathcal{P}, G), \preceq \rangle$ is *k-failed* if every leaf is k-failed. It is *finitely* failed if there is a k such that it is k-failed.

A proof-theoretic analysis of the abstract idea of ***negation as failure*** (*NF*) can be based on the above characterization of finite failure. For the sake of simplicity, we consider here only the P-system, but it should be obvious how this treatment could apply to *AAR*-systems in general, provided that they are regular. Our aim is to correlate k-failed search-trees to the usual notion of *finitely failed SLD*-trees (as defined, for example, in [27]). To achieve that (Proposition 3.21) we reconsider selection functions.

First we note that, as a consequence of success-completeness (Corollary 3.16), for every selection function F a complete search-tree $\langle \mathcal{T}(\mathcal{P}, G), \preceq \rangle$ is failed iff so is the corresponding F-search tree $\langle Gen(\mathcal{P}, G), \preceq_F \rangle$. This is refined in Proposition 3.20 with respect to k-failure. Moreover, the next Proposition (3.19) shows that F-search trees are finite, provided that the selection function is *fair*. As usual, the fairness of F guarantees that in any path of the corresponding F-search tree every open assumption is eventually selected.

PROPOSITION 3.19
If the search tree $\langle \mathcal{T}(\mathcal{P}, G), \preceq \rangle$ of a (finite) program $\mathcal{P}$ is finitely failed, then every fair F-search subtree $\langle Gen(\mathcal{P}, G), \preceq_F \rangle$ is finite.

PROOF. Let F be a fair selection function. Since there is a k such that $\langle \mathcal{T}(\mathcal{P}, G), \preceq \rangle$ is k-failed, fairness guarantees that every path in the subtree $\langle Gen(\mathcal{P}, G), \preceq_F \rangle$ is finite. Moreover the latter is finitely branching, since $\mathcal{P}$ is finite. ∎

[7] Assuming the usual standardization apart.

PROPOSITION 3.20
Let F be a selection function. If the F-search tree $\langle Gen(\mathcal{P}, G), \preceq_F \rangle$ is k-failed, then so is the corresponding complete tree $\langle \mathcal{T}(\mathcal{P}, G), \preceq \rangle$.

PROOF. We prove a more general statement, namely: for a proof-tree Π, if the F-search tree of its continuations $\langle Gen(\mathcal{P}, \Pi), \preceq_F \rangle$ is k-failed, then so is the complete tree $\langle \mathcal{T}(\Pi, \mathcal{P}), \preceq \rangle$. The proof is by induction on the height of $\langle Gen(\mathcal{P}, \Pi), \preceq_F \rangle$.

- *Basis.* $\langle Gen(\mathcal{P}, \Pi), \preceq_F \rangle$ has height 0, i.e. the selected assumption $F(\Pi)$ is k-failed. Our assert holds because (an instance of) this assumption belongs to every continuation of Π.
- *Step.* $\langle Gen(\mathcal{P}, \Pi), \preceq_F \rangle$ has height $i + 1$. Let $\Pi_1, \dots, \Pi_n$ be the one-step mgc of Π. For $1 \leq i \leq n$, by the inductive hypothesis on $\langle Gen(\mathcal{P}, \Pi_i), \preceq_F \rangle$, the complete space $\langle \mathcal{T}(\mathcal{P}, \Pi_i), \preceq \rangle$ is k-failed. Since every non-trivial continuation of Π belongs to some $\langle \mathcal{T}(\mathcal{P}, \Pi_i), \preceq \rangle$, we get our assert.

■

Therefore k-failure can be finitely discovered, by using fair selection rules. Remark that the latter two propositions hold for any regular *AAR*-system.

For definite programs, we can relate k-failed search trees to finitely failed *SLD*-trees.

PROPOSITION 3.21 (k-failure)
Let P be a definite program, F a fair selection function and A an atom. The *SLD*-tree for $P \cup \{\leftarrow A\}$ is finitely failed iff there is a k such that $\langle \mathcal{T}(P, G), \preceq \rangle$ is k-failed.

PROOF. For every selection function F and every atom A, the *SLD*-tree with root $\leftarrow A$ corresponds to a F-search tree $\langle Gen(P, G), \preceq_F \rangle$ such that, for every node $\leftarrow B_1, \dots, B_n$ of the *SLD*-tree, the corresponding node in the F-search tree is a class $[\Pi]$, where Π has assumptions $B_1, \dots, B_n$. The proof is similar to that for Theorem 2.6. Conversely, one easily sees that, renaming apart, the assumptions of the proof-trees of an F-search tree originate the required *SLD*-tree.

By the above correspondence, we have that the *SLD*-tree for $P \cup \{\leftarrow A\}$ is finitely failed iff $\langle Gen(P, G), \preceq_F \rangle$ is k-failed. Then our assert follows from Proposition 3.20. ■

3.5 Examples

Now we analyse regularity and the existence of a resolution method for the systems of the previous sections and others. For the P-systems one can prove that regularity holds (an instructive proof can be found in [33]). As a consequence, for every goal G and program P, the set of the proof-trees $\mathcal{T}(P, G)$ is a regular search space and we can use F-search trees to find proofs of θG. Moreover, the resolution method is defined as follows.

Let $\forall(\forall y(L_1 \wedge \cdots \wedge L_n) \rightarrow B)$ be an axiom for the P-system and A be a goal. If the unification algorithm computes an mgu θ of B and A, then

$$Res(A, \forall(\forall y(L_1 \wedge \ldots \wedge L_n) \rightarrow B), \mathrm{P}) = [\langle \theta, \theta L_1; \ldots; \theta L_n \rangle]$$

otherwise *Res* is undefined. Again note the complete analogy with *SLD*-resolution. Observe that no substitution is attempted on the variable(s) y and that, in order to obtain most general continuations, θ must replace by new names (w.r.t. the current proof-tree) the (possible) variables of $L_1 \dots L_n$ that do not occur in B.

The F and PF-systems are non-regular systems.

EXAMPLE 3.22
Consider the following proof-trees Π_1, Π_2, Π_3, where $Fax(odd)$ is

$$\forall x(odd(x) \rightarrow \exists y(x = s(0) \wedge \mathit{true} \ \vee \ x = s(s(y)) \wedge odd(y)))$$

$$\frac{Fax(odd)}{\neg odd(0)}(F) \qquad \frac{\neg odd(W) \quad Fax(odd)}{\neg odd(s(s(W)))}(\mathrm{F}) \qquad \frac{\neg \mathit{true} \quad \neg odd(v) \quad Fax(odd)}{\neg odd(W)}(\mathrm{F})$$

indeed, they are similar,[8] but there is no proof-tree Π such that $\Pi_i \leq \Pi$ for $i = 1, 2, 3$.

As far as the last proof-tree is concerned, note first, that among the examples it is the only one not closed under substitution. Second, its derivation is:

- $\neg \mathit{true}$ is generated since W and $s(0)$ unify;
- $\neg odd(v)$ is generated, since W unifies with $s(s(y))$ which is not instantiated by the mgu $W/s(s(y))$ and v is the eigenvariable renaming the existentially quantified y.

In the absence of regularity, the notions of mgpt and of canonical continuation do not behave as desired. This means that we cannot find a success-complete resolution method unless we admit backtracking on substitutions. In particular *Res* has to compute many candidate substitutions and goal sequences: similar results (with different style) are contained in [45, 29]. Thus strategies like *SLDNF*-resolution cannot be success-complete, as we will discuss in the next section.

On the other hand, the latter is not the only case where backtracking on substitutions is requested by a condition of non-regularity. Consider the case of higher-order Horn clauses [31]. It is well known that the unification problem is in general infinitary [46]. And consider the following case, taken from [35]:

$$\begin{array}{ll} Ax1: & \mathit{mapfun}\ F\ \mathit{nil}\ \mathit{nil}. \\ Ax2: & \mathit{mapfun}\ F\ X.XS\ (FX).YS \leftarrow \mathit{mapfun}\ F\ XS\ YS. \end{array}$$

Suppose you are evaluating the goal:

$$\mathit{mapfun}\ F\ [1,1]\ [(g\ 1\ 1), (g\ 1\ 2)].$$

In this case the unification problem is finite, but not unary; F can be assigned to $\lambda x.gxx$, $\lambda x.gx1$, $\lambda x.g1x$, $\lambda x.g11$, leading to configurations like this one, which applies the substitution $F/\lambda x.gxx$:

$$\frac{\mathit{mapfun}\ F\ [1]\ [(g\ 1\ 2)] \quad Ax2}{\mathit{mapfun}\ F\ [1,1]\ [(g\ 1\ 1), (g\ 1\ 2)]}(\mathrm{P})$$

Unfortunately, only the third unification will succeed for the whole goal. No mgpt exists and thus the interpreter will have to backtrack (potentially for an infinite amount of trials) during search.[9] Hence a formulation of *Res* can be infinitary and look something like:

> If Ax can be applied to G by R, then $Res(G, Ax, R) = \{[\langle\theta_j, \mathbf{G}_j\rangle]\}$, for $\theta_j \in J$, where J is a complete set of unifiers [46] and $\mathbf{G}_j$ is the sequence of goals corresponding to θ_j.

[8] Note that the F rule can be formulated in a way such that the PF-system is closed under substitutions, still it is not regular. This is also the case for higher-order Horn clauses, addressed below, where closure but not regularity is guaranteed.

[9] If, on the other hand, we label the rules with the indication of the relevant substitution, we recover the regularity of the space, but at the cost of allowing an infinite number of rules.

4 Proof-theoretic analysis of NF

We introduce Clark's explanation of *NF*, that is we characterize it as provability from the completion [10]. We use a weakening (in a sense detailed below) of Clark's completion, together with the Domain Closure Axiom (*DCA*) [30]. Then we give our proof-theoretic reading of the problems of *SLDNF*-resolution w.r.t. soundness and completeness, namely enlightening the role of regularity for the latter (point (b) in 4.2)). In Section 5, we introduce the idea of *constructive negation via regular splitting* to overcome some of these problems and we relate it with the already developed notion of *intensional negation* [5].

4.1 *F-systems and* SLDNF*-resolution*

Here we relate finite failure to proofs in the F_{DCA}-system, that is in the F-system enriched by the P-rule restricted to applying suitable instances of the Domain Closure Axiom.

DCA depends on the signature Σ of the underlying language. It essentially says that every element of the domain can be represented by a ground term of Σ, i.e.

$$(DCA): \quad \forall x(\bigvee_{f \in \Sigma} \exists \vec{y}(x = f(\vec{y}))).$$

The following schema is admissible w.r.t. DCA, as shown, for example, in [30], where it is called *proof by case analysis*:

DEFINITION 4.1 (Covering)
Let $||L||$ be the set of ground instances of a literal L. A *Σ-covering* for L is a set $\sigma_1 L, \ldots, \sigma_n L$ such that $||\sigma_1 L|| \cup \cdots \cup ||\sigma_n L|| \supseteq ||L||$. The instance of the DCA-schema corresponding to a Σ-covering for L is

$$\forall(\forall y(\sigma_1 L \wedge \ldots \wedge \sigma_n L) \to L)$$

where y are the new variables introduced by $\sigma_1, L \ldots, \sigma_n L$ (we assume that the range of every σ_i contains only new variables that do not occur in L or in the range of every other substitution).

In the following, we will stipulate Σ to be finite, being the signature of the underlying program, rather than an infinite universal language as in [21, 3]. Moreover we will extend the notion of covering to arbitrary formulae.

To use the F-system, we associate to a program its failure axioms, which will be applied in the F-system. The starting point is the only-if part of the completion of a predicate $p(\ldots)$:

$$\forall x(p(x) \to \bigvee_{i=1}^{n} \exists y_i(x = t_i \wedge \bigwedge_{k=1}^{h_i} L_{i,k})).$$

For every $k_1, \ldots, k_n$ such that $1 \leq k_i \leq h_i$ we infer

$$Fax_{k_1,\ldots,k_n}(p): \quad \forall x(p(x) \to \exists y \bigvee_{i=1}^{n} (x = t_i \wedge L_{i,k_i})).$$

By convention, the failure axiom of a unit clause introduces the constant *true*, hence $h_i \geq 1$ is always fulfilled. Observe that these axioms contain *exactly* a singleton literal in every disjunct of the consequent. For a program P, $Fax(P)$ will indicate the set of its failure axioms in the latter sense, called *weak completion axioms*. The cardinality of $Fax_{k_1,\ldots,k_n}(p)$ is $\prod_{i=1}^{n} h_i$, where n is the number of clauses and h_i the number of literals in each clause of P with head $p(\ldots)$.

EXAMPLE 4.2
The only-if part of t (for *times*), the usual program for computing the product, is:

$$\begin{aligned}\forall(t(a,b,c) \to\ & \exists x(a = x \wedge b = 0 \wedge c = 0) \vee \\ & \exists x,y,z,w(a = x \wedge b = s(y) \wedge c = z \wedge t(x,y,w) \wedge sum(w,x,z)))\end{aligned} \tag{4.1}$$

From (4.1) we derive the failure axioms:

$$\begin{aligned}\forall(t(a,b,c) \to \exists x,y,z,w\ & ((a = x \wedge b = 0 \wedge c = 0 \wedge true) \vee \\ & (a = x \wedge b = s(y) \wedge c = z \wedge t(x,y,w))). \\ \forall(t(a,b,c) \to \exists x,y,z,w\ & ((a = x \wedge b = 0 \wedge c = 0 \wedge true) \vee \\ & (a = x \wedge b = s(y) \wedge c = z \wedge sum(w,x,z))).\end{aligned}$$

Note that, in general, the conjunction of the $Fax_{k_1,\ldots,k_n}(p)$ is notably weaker and does not imply the only-if part of $Comp(p)$. This is due to the shared binding of local variables, i.e. those which appears in the body but not in the head of a clause—like w in the former example. In the case of shared local variables, a factorization of failure axioms is needed. For example:

EXAMPLE 4.3
By DCA, the right-hand side of (4.1) is equivalent to the following factorization:

$$\begin{aligned}& \exists x(a = x \wedge b = 0 \wedge c = 0) \vee \\ & \exists x,y,z(a = x \wedge b = s(y) \wedge c = z \wedge t(x,y,0) \wedge sum(0,x,z)) \vee \\ & \exists x,y,z,j(a = x \wedge b = s(y) \wedge c = z \wedge t(x,y,s(j)) \wedge sum(s(j),x,z)),\end{aligned}$$

which corresponds to the fact that w is covered by $0, s(j)$. We can derive four failure axioms from this factorization, parallel to the possible choices between $t(x,y,0)$, $sum(0,x,z)$ in the second row and $t(x,y,s(j)), sum(s(j),x,z)$ (in the third one). For example, one of these axioms is:

$$\begin{aligned}Fax1:\ \forall(t(a,b,c) \to \exists x,y,z,j\ & ((a = x \wedge b = 0 \wedge c = 0 \wedge true) \vee \\ & (a = x \wedge b = s(y) \wedge c = z \wedge sum(0,x,z)) \vee \\ & (a = x \wedge b = s(y) \wedge c = z \wedge t(x,y,s(j))).\end{aligned}$$

Notice that $\neg sum(0,0,s(v)); \neg t(0,0,s(j)) \in \mathrm{F}(\neg t(0,s(0),s(v)), Fax1)$. If one looks at the finitely failed *SLD*-tree for $\{t(0,s(0),s(v))\}$, one can recognize that such finite failure is reduced to the finite failure of $sum(0,0,s(v))$ and $\neg t(0,0,s(j))$.

The key issue here is *regularity*, which is independent from the problems due to shared local variables. Therefore, we will assume that local variables are restricted to occur only in a single literal in the body of a clause, similarly to condition *(d)* in Barbuti's definition of *flat* programs [5]. This can always be achieved by a simple *folding* step that can be demonstrated to be semantics preserving. Note that under this hypothesis the weak completion coincides with Clark's completion.

Moreover we will assume, for the sake of the following two Theorems (4.4 and 4.5) that the heads of clauses are *unrestricted*, or *left-linear*, i.e. no variable occurs therein more than once.[10] There are other alternatives, however, as sketched in Example 5.7.

THEOREM 4.4 (Finite failure)
Given a definite program P and an atom A, if $P \cup \{\leftarrow A\}$ has a finitely failed *SLD*-tree, then there is a proof $\Pi :: \neg A$ in the F_{DCA}-system applying only axioms from $Fax(P)$.

The proof requires some additional machinery and is detailed in the Appendix. Now we approach the relation between finite failure and proofs in the PF_{DCA}-system. To every normal program P we associate the set $WComp(P) = Ax(P) \cup Fax(P)$, where $Fax(P)$ has been straightforwardly extended to deal with literals.

THEOREM 4.5 (PF_{DCA}/WComp(P) adequacy)
Let P be a normal program and L a literal. If there is a finitely failed *SLDNF*-tree for $P \cup \{\leftarrow L\}$, then there is a PF_{DCA}-proof $\Pi :: neg(L)$ with axioms from $WComp(P)$. If there is a *SLDNF*-refutation of $P \cup \{\leftarrow L\}$ with answer substitution δ, then there is a PF_{DCA}-proof $\Pi :: \delta L$, with axioms from $WComp(P)$.

PROOF. By induction on the rank of the finitely failed *SLDNF*-tree, as defined in [27], or of the *SLDNF*-refutation.

- *Basis.* The basis for refutation is analogous to Theorem 2.6 and the basis for finitely failed trees to Theorem 4.4 (see Remark A.17 in the Appendix).
- *Step for refutation.* We proceed as in Theorem 2.6, by *Step 0* and *Step i+1*, provided that the selected literal L_{n_i} is positive. Suppose in *Step i+1* $L_{n_i} = \neg A$. In this case, there is a finitely failed tree of rank k for $P \cup \{\leftarrow A\}$. By inductive hypothesis we have a PF_{DCA}-proof $\Pi :: \neg A$.
- *Step for finitely failed trees.* Let Φ be a finitely failed *SLDNF*-tree of rank $k+1$, with root $\leftarrow L$. Before translating Φ into a proof in the PF_{DCA}-system, we delete all the nodes $\leftarrow L_1, \ldots, L_i, \ldots, L_m$ where the selected literal is a negated atom $L_i = \neg B$ and there is a finitely failed tree (of rank k) for $P \cup \{\leftarrow B\}$. Note that this step is an implicit application of weakening. More precisely, we delete the descendant $\leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m$, of $\leftarrow L_1, \ldots, L_i, \ldots, L_m$ and we add the (suitable instances of) L_i in all the nodes under the deleted one. One easily sees that we can iterate this process until we obtain a *SLDNF*-tree Φ^* such that in every intermediate node the selected literal is an atom.
  Φ^* is such that L is the root and $A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m$ are the literals selected in the leafs. $A_1, \ldots, A_n$ have a finitely failed *SLDNF*-tree of rank $\leq k$, and $B_1, \ldots, B_m$ have a *SLDNF*-refutation of rank $\leq k$. Therefore, by the inductive hypothesis, we have PF_{DCA}-proofs $\Pi_1 :: \neg A_1, \ldots, \Pi_n :: \neg A_n, \Delta_1 :: B_1, \ldots, \Delta_m :: B_m$ and by Remark A.17, there is a PF_{DCA}-proof $\Pi :: neg(L)$.

∎

REMARK 4.6
The proof enlightens the proviso on open literals. While no problem arises from $A_1, \ldots, A_n$, were $B_1, \ldots, B_m$ not ground, by inductive hypothesis we would get proofs $\Delta_1 :: \theta_1 B_1, \ldots, \Delta_m :: \theta_m B_m$, and this would correspond to unsound substitutions of eigenvariables of Φ^*.

[10] It is, nevertheless, well known that every clause with restricted head can be left-linearized by introducing new variables and constraining them by a new predicate, say eq, axiomatized by $eq(X, X)$ [48, 5].

To conclude, a few observations are in order.

- DCA is an axiom-schema, and therefore cannot be properly applied by an AAR. It could be substituted by the collection of all its instances, but this would give rise to infinitely many axioms. A working alternative is to introduce a systematic mechanism to generate deeper and deeper instances of the schema. This solution is similar to the herbrand procedure of [5]. We will not develop this issue here. Our aim is to underline the *role of regularity*, which is an independent cause of the incompleteness of *SLDNF*-resolution.
- As is technically shown in the Appendix, DCA is not needed for two classes of programs.
  1. Krom programs [20]. In this case the body of a non-unit clause contains just one literal. In our following examples, we will consider mainly Krom programs.
  2. Left-linear programs without local variables. In this case, if $\neg A$ (with A open) is a logical consequence of the completion, then there is a covering of $\neg A$ such that its elements have a PF-proof with axioms from the weak completion.

4.2 *On (un)soundness and (in)completeness of NF*

SLDNF-resolution has a non-logical behaviour if open negative goals are selected. In our model, we can distinguish different causes for that.

(a) *Soundness* problems: during the computation (logically unsound) substitutions on eigenvariables may occur.

(b) *Success-completeness* problems: given an open goal G, *SLDNF*-resolution fails to return an answer, even if there is a PF-proof $\Pi :: \theta G$: the culprit can be found in the lack of regularity, which lies at the basis of the success-incompleteness of NF.

(c) *Incompleteness* of the PF_{DCA}-system: there are goals θG such that $Comp(P) \models \theta G$ in classical logic but no PF_{DCA}-proof $\Pi :: \theta G$ exists. For example, consider the well known example: $p \leftarrow q, p \leftarrow \neg q, q \leftarrow q$. No PF_{DCA}-proof of p exists, even if it is a logical consequence of the completion. We will not discuss this issue further. Note that point (b) and (c) are independent: if the PF_{DCA}- system were complete w.r.t. classical logic, yet not regular, the success-incompleteness issue would not be solved.

As far as (a) is concerned, this corresponds to the fact that a non-legal substitution on the eigenvariables may be introduced in some continuation step, as shown by the following example, taken from [27].

$$\begin{aligned} Ax1: &\quad p :- \neg q(X) \\ Ax2: &\quad q(a). \end{aligned}$$

In standard Prolog, using an unsound selection function, the goal $\leftarrow \neg p$ succeeds (since $\leftarrow p$ fails), although it is not a logical consequence of the completion of the program. In our model, safeness (i.e. soundness) is enforced not by an external condition on the selection function, but by the usual proof-theoretic proviso on eigenvariables, i.e. that they cannot be instantiated by substitutions, as the following proof-tree shows. Once we have obtained the goal $q(u)$, with eigenvariable u, we cannot continue our proof-tree in a sound way; so we do not obtain any proof of $\neg p$.[11]

[11] Note that in a more detailed representation of the *SLDNF*-success tree, that node and the branch ending with it would have been labelled with something like 'floundering', while in our approach this is just a failing branch.

$$\frac{q(u) \quad p \to \exists x \neg q(x)}{\neg p}(\text{F}).$$

This shows that we have a natural way to distinguish proofs of negated goals (where such a proof has no assumptions) from proof-trees with assumptions that cannot be continued. The latter corresponds to unprovability.

Last we remark that, since failure rules do not introduce unifying substitutions, dangerous substitutions can arise only if a positive assumption is selected in a continuation step and the related unification modifies the eigenvariables of some failure rule; but, as shown in the proof of Theorem 4.5 and Remark 4.6, this cannot happen if no open negated formula is selected in a continuation. This is particularly true for definite programs and open negative queries where no soundness problem can arise. Moreover, if we ensure that no instantiation link arises on eigenvariables, then we may safely select open negated goals, as is well known [36, 45], under the name of $ESLDNF$- resolution.

With respect to (b), there are PF-proofs instantiating open negated goals which are ignored by standard *SLDNF*-resolution. We show how this is related to the *non-regularity* of the PF-system.

Let us consider for example the following (non-stratified) program $EVEN$:

$$\begin{aligned} Ax1: &\quad even(0). \\ Ax2: &\quad even(s(X)) :- \neg even(X). \end{aligned}$$

$Ax(EVEN)$ contains the obvious success axioms and $Fax(EVEN)$ contains one failure axiom $Fax(even)$:

$$\forall x(even(x) \to \exists u(x = 0 \wedge true \vee x = s(u) \wedge \neg even(u))).$$

If we start from $\neg even(X)$, *SLDNF*-resolution yields a finitely failed (not safe) tree and no solution is found, even if the PF-system contains infinitely many proofs with axioms from $WComp(EVEN)$ and consequence $\neg even(\ldots)$. Success-incompleteness of *SLDNF*-resolution is due to non-regularity, as Figure 2 shows.

Ovals contain similarity classes. Since $\neg even(X_0)$ is a mgpt, we can use it to generate all canonical continuations of the root class. All the continuations are similar (only Fax can be applied), and we get the descendant class sc_2. The latter is non-regular: it contains two maximal, yet incompatible proof-trees, π_1 and π_2. Since the identical substitution is more general than $X_0/s(X_1)$, *SLDNF*-resolution selects π_1, which contains the unprovable assumption $\neg true$, and fails. On the other hand, π_2 has two descendants, and this shows that *SLDNF*-resolution is *success-incomplete*. Concerning sc_3, sc_4, the inscribed proof-trees are mgpts. sc_3 is a success node, while sc_4 can be continued. Since the unique assumption that can be selected for a continuation is $\neg even(X_2)$, sc_4 behaves as the root, i.e. it has one non-regular descendant class.

When a similarity class contains a finite set of maximal proof-trees, backtracking on them provides success-completeness.

In the next subsection we sketch a partial solution to these problems: each failure axiom is split into a set of negative axioms, and backtracking on substitutions is replaced by backtracking on the negative split axioms, applied by the regular rule P^-.

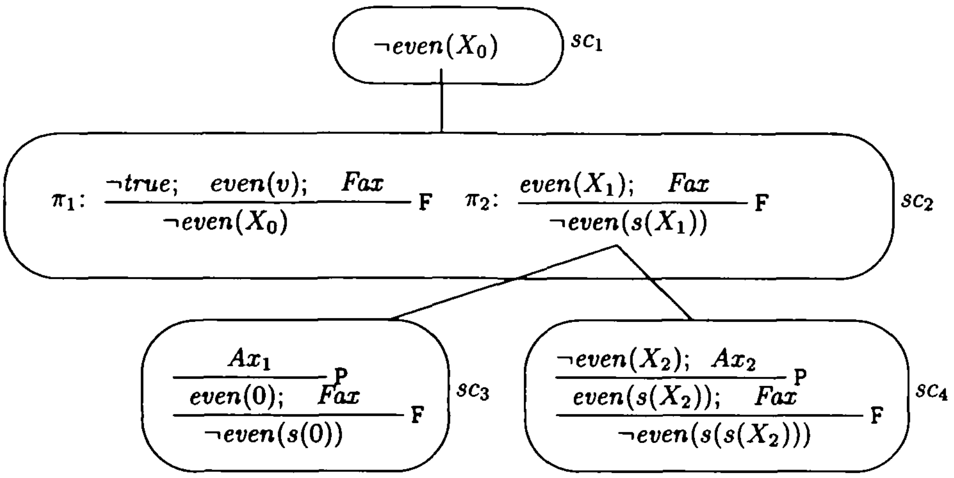
$\neg even(X_0)$
sc_1
π_1: $\dfrac{\neg true;\quad even(v);\quad Fax}{\neg even(X_0)}$ F
π_2: $\dfrac{even(X_1);\quad Fax}{\neg even(s(X_1))}$ F
sc_2
$\dfrac{\dfrac{Ax_1}{even(0);\quad Fax}\,P}{\neg even(s(0))}$ F
sc_3
$\dfrac{\dfrac{\neg even(X_2);\quad Ax_2}{even(s(X_2));\quad Fax}\,P}{\neg even(s(s(X_2)))}$ F
sc_4

FIG. 2. Search space for $\neg even(X_0)$

5 Regular splitting

Let us consider a failure axiom $Fax_{k_1,\ldots,k_n}(p)$

$$\forall x(p(x) \rightarrow \exists y \bigvee_{i=1}^{n} (x = t_i \wedge L_{i,k_i})).$$

For the sake of simplicity, we will omit $k_1, \ldots, k_n$ and therefore the double index and write $Fax(p)$.

The F rule has the following behaviour. Let $p(a)$ be an open instance of $p(x)$ such that:

$$neg(\sigma_1 L_{i_1}); \ldots; neg(\sigma_k L_{i_k}) \in \mathrm{F}(\neg p(a), Fax(p)).$$

Let θ be a substitution; in general, the sequence $neg(\theta\sigma_1 L_{i_1}); \ldots; neg(\theta\sigma_k L_{i_k})$ does not belong to $\mathrm{F}(\theta\neg p(a), Fax(p))$, but the latter may contain a shorter and possibly empty sequence: this destroys both closure under substitution and regularity.

We say that $p(a)$ is a *substitution-closed instance* (s.c. instance) w.r.t. $Fax(p)$ when the above incident does not occur, i.e. $p(a)$ is sufficiently instantiated in order to yield the closure under substitution of $\mathrm{F}(\neg p(a), Fax(p))$. A trivial case is when $p(a)$ is already ground. In the general case the following holds.

REMARK 5.1
$p(a)$ is a *s.c. instance* w.r.t. $Fax(p)$ if there are k substitutions $\sigma_1, \ldots, \sigma_k$ s.t. $a = \sigma_1 t_{i_1}, \ldots, a = \sigma_k t_{i_k}$, where $t_1, \ldots, t_n$ are the terms involved in $Fax(p)$ and a unifies only with $t_{i_1}, \ldots, t_{i_k}$.

Even considering only s.c. instances, regularity would not hold, since the existence of an upper bound of similar trees is not guaranteed. Therefore we split $Fax(P)$ into axioms that can be applied by the regular system P, as follows.

If $p(a)$ is a s.c. instance w.r.t. $Fax(p)$, we build by contraposition the negative axiom $Nax(p(a))$, where the outermost quantifier binds the open variables in $p(a)$, while the possibly empty string of bindings $\forall y$ originates from the local variables:

$$\forall(\forall y \bigwedge_{j=1}^{k} \neg(\sigma_j L_{i_j}) \to \neg p(a)).$$

In the limiting case where $h = 0$ (i.e. a does not unify with any t_j), we have that $Nax(p(a))$ is $\forall(\neg p(a))$.

REMARK 5.2
For every s.c. instance $\theta p(a)$ w.r.t. $Fax(p)$:

$$\mathrm{F}(\neg\theta p(a), Fax(p)) = \mathrm{P}^-(\neg\theta p(a), Nax(p(a))).$$

By the above considerations, the behaviour of the non-regular rule F applied to $Fax(p)$ coincides with the one of the regular rule P^- applied to $Nax(p(a))$, for every s.c. instance $p(a)$ w.r.t. $Fax(p)$. Thus we will call $Nax(p(a))$ a ***regular instance*** of $Fax(p)$.

DEFINITION 5.3 (Regular splitting)
A *regular splitting* of a failure axiom $Fax(p)$ over a signature Σ is a set $\{Nax(p(a_1)), \ldots, Nax(p(a_n))\}$ of regular instances of $Fax(p)$, such that $p(a_1), \ldots, p(a_n)$ is a Σ-covering of $p(x)$.

By $Nax(p)$ we will indicate a regular splitting of $Fax(p)$ and by $Nax(P)$ a collection of regular splittings for every predicate occurring in a program P.

In the usage of a regular splitting $Nax(P)$ of a program P and its failure axioms $Fax(P)$, we will consider *only* proof-trees with terms built out of Σ. Ways of parameterizing (constructive) negation to other domains have been investigated in [26, 49] and could be applied to our technique. Nevertheless, this is beyond the spirit of the paper. Furthermore, the right framework for regular splitting (as well as for intensional negation) is a many-sorted one, as already hinted in [41].

We need the following substitution lemma, whose easy proof is omitted.

LEMMA 5.4
In the PF_{DCA}-system for every proof $\Pi :: L$ of and substitution θ, there is a proof $\Pi^* :: \theta L$ with height less than or equal to $\Pi :: L$.

In particular, the proof $\Pi^* :: \theta L$ may introduce eigenvariables in the root. This fact will be understood in the proof of Theorem 5.5. Moreover, for a failure axiom $Fax(p)$, the ***regular heads of*** $Fax(p)$ will be the instances $p(a_1), \ldots, p(a_n)$ such that $\neg p(a_i)$ is the head of a regular instance of $Nax(p)$.

THEOREM 5.5
Let P be a normal program and let us assume that, for every failure axiom we have a corresponding regular splitting over Σ. The following holds:

(a) for every proof $\Pi :: G$ in the PF_{DCA}-system with axioms from $Ax(P) \cup Fax(P)$ there is a proof $\Delta :: G$ in the P_{DCA}-system with axioms from $Ax(P) \cup Nax(P)$;
(b) for every proof $\Delta :: G$ in the P_{DCA}-system with axioms from $Ax(P) \cup Nax(P)$, there is a proof $\Pi :: G$ in the PF_{DCA}-system with axioms from $Ax(P) \cup Fax(P)$.

PROOF. (b) is proved by an easy induction on $\Delta :: G$, using Remark 5.2. (a) is proved by induction on the height of $\Pi :: G$.

Basis. If $\Pi :: G$ is an application of an axiom $\forall x L$, it is already in the format of the P-system. If the axiom is $\forall x(p(x) \rightarrow \exists y(x = t_1 \wedge L_1 \vee \ldots \vee x = t_n \wedge L_n))$, then G is $\neg p(t)$ and t does not unify with $t_1, \ldots, t_n$. $p(t)$ is covered by the regular heads $p(a_1), \ldots, p(a_k)$ of $Fax(p)$. Let a_{i_j} $(1 \leq j \leq m)$ be the heads such that the $\mu_j = mgu(a_{i_j}, t)$ is defined, and let $b_j = \mu_j t = \mu_j a_{i_j}$. Then $\{p(b_j)\}$ is a covering of $p(t)$ and none of the a_{i_j} unifies with $t_1, \ldots, t_n$ (follows from the regularity of the splitting). Then each $\neg p(b_j)$ has an immediate proof in the P_{DCA}-system with axiom $Nax(p(a_{i_j}))$, and this yield our final proof by an application of the DCA-instance $\forall(\forall y(\neg p(b_1) \wedge \ldots \wedge \neg p(b_m)) \rightarrow \neg p(t))$, where the proofs of $\neg p(b_1), \ldots, \neg p(b_m)$ introduce the suitable eigenvariables.

Step. If the axiom applied at the root of $\Pi :: G$ is a DCA-instance, or an axiom from $Ax(P)$, our thesis is an immediate consequence of the inductive hypothesis. Otherwise it is $\forall x(p(x) \rightarrow \exists y(x = t_1 \wedge L_1 \vee \ldots \vee x = t_n \wedge L_n))$, G is $\neg p(t)$, and t unifies with $t_{i_1}, \ldots, t_{i_h}$. As in the inductive basis, we can build a covering $p(b_1), \ldots, p(b_m)$ of $p(t)$, where for $1 \leq j \leq m$, $b_i = \mu_i i_j = \mu_i t$. By Lemma 5.4, there are PF_{DCA}-proofs $\Pi_1^* :: \neg p(b_1), \ldots, \Pi_m^* :: \neg p(b_m)$ of height less than or equal to Π. By the inductive hypothesis, for every immediate subproof $\Pi' :: L$ of Π_i^* there is a corresponding P_{DCA}-proof $\Pi'' :: L$, and hence (by Remark 5.2) we can construct a P_{DCA}-proof of $\neg p(b_i)$. We can now proceed as in the basis, by applying a suitable instance of the DCA-schema. ∎

The translatability of PF_{DCA}-proofs into P_{DCA}-proofs is independent from the issue of completion consistency [3] with respect to classical (or even intuitionistic) logic. Indeed, the logic of the PF_{DCA}-system is so weak that no collapse due to the presence of inconsistent premisses is possible. One reason is the lack of the intuitionistic rule for negation. The other is the non-availability of hypothetical judgments that could nevertheless force the collapse of the negative fragment of the calculus. On the other hand completion consistency with respect to three-valued logic [21] can be ensured by exploiting the three-valued completeness of the PF_{DCA}-system, as briefly addressed in the Conclusions.

Let us come back to our starting problem of regularizing programs through splitting. In the following we will restrict discussion to the case where local variables are not involved and we will consider the equivalence with respect to *ground answers*. Therefore we will need a milder version of Theorem 5.5, where for every PF-proof $\Pi :: L$ with axioms from $Ax(P) \cup Fax(P)$ we get P-proofs of a finite covering of L, with axioms from $Ax(P) \cup Nax(P)$, i.e. we will not use DCA. Since the P-system is regular, $Ax(P) \cup Nax(P)$ allows one to search for answers of open negative goals, basically using the computation model of *SLD*-resolution.

In [22] and later in [5] (more targeted to logic programs) algorithms to compute the (relative) complement of first-order terms are described (see also [11]). Those algorithms can be adapted to compute regular splitting, as informally described in the examples below.

EXAMPLE 5.6

Let us consider the program $EVEN$ (4.2) over the signature $\Sigma_{EVEN} = \{0, s\}$ of natural numbers. The failure axiom $\forall x(even(x) \rightarrow \exists u(x = 0 \wedge true \vee x = s(u) \wedge \neg even(u)))$ can be split as follows.

First we consider $x = 0$. Since $even(0)$ is already ground, we can build by contraposition the corresponding negative axiom or regular instance:

$$Nax(even(0)) : \neg true \rightarrow \neg even(0).$$

The terms not covered by the above regular instance are

$$\|v\| \setminus \|0\| = \|s(W)\|$$

where $\setminus$ represents set difference among terms as in [22]. Since $even(s(W))$ is a s.c. instance w.r.t. $Fax(even)$, we may obtain the regular instance:

$$Nax(even(s(W))) : even(W) \rightarrow \neg even(s(W)).$$

Now all terms are covered. Hence the regular splitting contains $Nax(even(0))$ and $Nax(even(s(W)))$.

EXAMPLE 5.7
Consider the program SUM (2.2). Its failure axiom is;

$$\begin{aligned}\forall a,b,c(sum(a,b,c) \rightarrow \exists x,y,z \quad & (a = x \wedge b = 0 \wedge c = x \wedge true \vee \\ & a = x \wedge b = s(y) \wedge c = s(z) \wedge sum(x,y,z))).\end{aligned}$$

The case $a = x, b = s(y), c = s(z)$ covers the set $\|(x, s(y), s(z))\|$. We obtain the regular instance:

$$\forall x,y,z(\neg sum(x,y,z) \rightarrow \neg sum(x,s(y),s(z))).$$

The non-covered set of terms is now $\|(u,0,v)\| \cup \|(u,v,0)\|$. Splitting $\|(u,v,0)\|$ in $\|(0,v,0)\| \cup \|(s(u),v,0)\|$ we obtain the regular instances:

$$\begin{aligned}&\forall v(\neg true \rightarrow \neg sum(0,v,0))\\ &\forall u,v(\neg sum(s(u),v,0)).\end{aligned}$$

Now it remains to find the regular instances covering $\|(u,0,v)\|$. We obtain $\forall(\neg true \rightarrow \neg sum(u,0,u))$ and the non-covered set is $\|(u,0,v)\| \setminus \|(u,0,u)\|$. This difference has no finite representation. The standard solution is to left-linearize the source program: an alternative could be to find a recursive representation[12] of $\|(u,0,v)\| \setminus \|(u,0,u)\|$, namely (stretching the notation) $\|(s(u),0,0),(0,0,s(u)),(s(u),0,s(v)) \leftarrow (u,0,v)\|$. This yields the regular instances:

$$\begin{aligned}&\forall u \neg sum(s(u),0,0)\\ &\forall u \neg sum(0,0,s(u))\\ &\forall u,v(\neg sum(u,0,v) \rightarrow \neg sum(s(u),0,s(v))).\end{aligned}$$

Summarizing, we obtain the following clauses:

$$\begin{aligned}&\forall u,v,w(\neg sum(u,v,w) \rightarrow \neg sum(u,s(v),s(w)))\\ &\forall u,v(\neg sum(u,0,v) \rightarrow \neg sum(s(u),0,s(v)))\\ &\forall u,v \neg sum(s(u),v,0)\\ &\forall u(\neg sum(s(u),0,0))\\ &\forall u(\neg sum(0,0,s(u))).\end{aligned}$$

Note that we have discarded implications with *true* as consequent, since they can only lead to the unsatisfiable goal through the P^- rule. Apart from the elimination of redundancies, other optimizations are possible; for example localizing subsumption cases (in the former example the fourth clause is an instance of the third) or applying partial evaluation. In general,

[12]This is connected to the notion of regular tree languages.

post-transformation analysis of the program will be recommended, if only to recover the operational behaviour of the old program. This obviously depends on pragmatic considerations motivated by the actual behaviour of the Prolog engine, namely its sensitiveness to the ordering of literals and clauses. This is beyond the aims of this paper and is treated elsewhere in the literature.

5.1 *The positive method*

Now we show how to adapt the former technique to the pure Prolog inference machinery, by a transformation of normal programs into *equivalent* definite programs based on regular splitting. This is very close to the idea of *intensional negation* [5]. See Section 6 for other transformational approaches. Note that, as soon as the machinery of *AAR*-systems is set—machinery which is not specific to this application—the reconstruction of intensional negation through regular splitting is almost trivial.

We assume an enriched signature where, for every predicate symbol p, we have the additional symbol p^c, that we call the complement of p. Let us consider the translation C:

$$C(\neg p(t)) = p^c(t) \qquad \text{and} \qquad C(p(t)) = p(t).$$

For every clause c and axiom $Ax(c)$:

$$C(Ax(\forall(L_1 \wedge \cdots \wedge L_h \rightarrow A))) = \forall(C(L_1) \wedge \cdots \wedge C(L_h) \rightarrow A).$$

For every axiom $Nax(p(a))$

$$C(Nax(\forall(\forall y(L_1 \wedge \ldots \wedge L_n) \rightarrow \neg p(a)))) = \forall(\forall y(C(L_1)) \wedge \cdots \wedge C(L_n)) \rightarrow p^c(a)).$$

Let P be a normal program and $Nax(P)$ the negative axioms, obtained by a regular splitting of $Fax(P)$ over a signature Σ. We have that using the above translation C, we can compute a bijective map from the proof-trees of the P-system with axioms from $Ax(P) \cup Nax(P)$ to the proof-trees of the P-system with axioms from $C(Ax(P) \cup Nax(P))$. We obtain:

PROPOSITION 5.8
Given a normal program P, whenever $C(Ax(P) \cup Nax(P))$ is a set of Horn axioms, it is a definite program equivalent (via translation C) to $Ax(P) \cup Nax(P)$.

Therefore $C(Ax(P) \cup Nax(P))$ can be run using standard *SLD*-resolution.

EXAMPLE 5.9
Considering Example 5.6, $C(Nax(even(0)))$ is *false* $\rightarrow$ $even^c(0)$ and $C(Nax(even(s(w))))$ is $\forall w(even(w) \rightarrow even^c(s(w)))$. Analogously for $C(Ax(EVEN))$. From the Horn axioms obtained this way, we extract the program (where the redundant $even^c(0) \leftarrow false$ is omitted):

$$\begin{array}{lll} even(0). & & \\ even(s(X)) & \leftarrow & even^c(X) \\ even^c(s(X)) & \leftarrow & even(X). \end{array}$$

PROPOSITION 5.10
Let P be a definite program and $Nax(P)$ be a regular splitting of $Fax(P)$. Then, if $C(Nax(P))$ are Horn axioms, the corresponding program is the complement of P in the sense of [5].

PROOF. Since $C(Nax(P))$ are Horn axioms, there are no local variables. If $P \cup \{\leftarrow A\}$ finitely fails, by Corollary A.12 (in the Appendix), there are $\|\rho_1 A\| \cup \cdots \cup \|\rho_n A\| \supseteq \|A\|$ such that each $\|\rho_i A\|$ has an F-proof with axioms from $Fax(P)$. By the aforementioned mild version of Theorem 5.5, each $\|\rho_i A\|$ has a P-provable covering with axioms from $Nax(P)$. Therefore there is a finite covering of A^c that is P-provable with axioms from $C(Nax(P))$. ∎

EXAMPLE 5.11
Consider the $\leq$ relation defined as follows

$$\begin{array}{ll} Ax1: & 0 \leq X \\ Ax2: & s(X) \leq s(Y) \leftarrow X \leq Y. \end{array}$$

A regular splitting of the failure axioms is:

$$\begin{array}{lcl} Nax(0 \leq X) & : & \forall x(\neg true \rightarrow \neg 0 \leq x) \\ Nax(s(X) \leq s(Y)) & : & \forall x, y(\neg x \leq y \rightarrow \neg s(x) \leq s(y)) \\ Nax(s(X) \leq 0) & : & \forall x(\neg s(x) \leq 0). \end{array}$$

From $C(Nax(s(X) \leq s(Y)), Nax(s(X) \leq 0))$ (omitting the redundant first axiom) we extract:

$$\begin{array}{l} s(X) \leq^c 0. \\ s(X) \leq^c s(Y) \leftarrow X \leq^c Y \end{array}$$

which is the complement of the program for $\leq$.

We conclude with an example of regular instance with a non-Horn translation. $\forall x(even(x) \rightarrow \exists y.sum(y, y, x))$ is translated into $\forall x(\forall y.sum^c(y, y, x) \rightarrow even^c(x))$. A possibility of accommodating the above axiom in the Horn setting would be to consider y as a new constant, thus obeying the proviso that eigenvariables cannot be substituted. However, even if a regular splitting might give rise to a regular search space and translate the original program into a program executable by *SLD*-resolution, we do not achieve, in this case, the possibility of answering negative open goals, due to the presence of eigenvariables, and DCA is needed (see Theorem 5.5). This corresponds to the well-known problem of the treatment of local variables during complementation, which requires an extension of standard *SLD*-resolution (see following section).

6 Related work

6.1 Constructive negation

Constructive negation is an attempt to devise methods capable of providing logically justified answers to non-ground negative queries, in analogy with the witnessing property of constructive logics [50]. Formally, for a suitable derivability relation, this property holds when, if $\vdash \exists x p(x)$, then there is a term t s.t. $\vdash p(t)$. Accordingly, it is suggested that from $\vdash \exists x \neg p(x)$ the same property infers a term t s.t. $\vdash \neg p(t)$. We can roughly distinguish two approaches:

i. Program transformation: [41, 13, 5].
ii. Negation by constraints: [51] for Datalog programs, [8, 9] for Prolog, extended to CLP in [49]; fail substitutions: [45, 29].

Historically, the original attempt to deal with negation was simply to try to avoid the floundering phenomenon: given that the latter is in general undecidable, one possibility is to try to make sure that when a negative literal is called it has already been grounded: there are basically three possibilities:

1. Satisfy the syntactical, although very restrictive, conditions on allowed computations [43], which essentially reduces evaluation to *ground* evaluation.
2. Try to achieve grounding by delaying, as in *[M]NU-Prolog* or *Sicstus*, where a goal may be declared to be 'frozen' and is evaluated only when it reaches a sufficient degree of instantiation. This is obviously only a partial solution, since at run-time there is no guarantee to eventually ground the problematic query. A further improvement is offered by the computation rules of *IC-Prolog* (see [36], for a comprehensive analysis and references).
3. Covering the open negative query with a generator of values for the relevant variables [13].

6.1.1 Static approaches

If we are dealing with datalog programs, i.e. with finite Herbrand Universe U_P, the naive approach would be to instantiate all the rules with potentially troublesome goals with terms from U_P [1], say through propagation in every negative literal in the program. This is clearly infeasible, since it may result in an intractable number of rules, proportional to all the permutations of arguments over U_P; besides, this may return many undesired (untypable) answers.

A sophistication of this idea can be found in [13]: the proposal is to automatically infer a 'type', seen as a set of facts, for the problematic variables and transform the original program into one where grounding is ensured by the coverage from those types. Then useless answers originating from general instantiation would be excluded by the typing discipline. Although it can be shown that the new program is equivalent to the old one, this cannot be extended to full Prolog: function symbols make the type infinite and non-ground facts undermine the instantiation capability of the type.

The transformational approach has a fairly long history—see [36] for a survey of the early 1980s. The idea is to implement negation using inequalities, so that the complement of any predicate occurring negatively in the original program P is synthesized in order to obtain an equivalent definite program P^c (in the sense of $F_P = M_{P^c}$). This was first proposed in [41]. In this seminal paper it is shown how to massage the completion so that the derived program mimics failed computation of the original one. The technique consists essentially in taking the contrapositive of the completion, putting the right-hand side in disjunctive normal form and eliminating the universal quantified inequalities by introducing a 'non-unifiable' predicate. From the simplification of those constraints and by regarding the negated predicate as a new name the new definition is obtained.

Two interrelated problems are involved in this simple transformation:

1. We need a way to solve the aforementioned inequalities: this was first suggested in [22] in the general context of explicit representation of generalization and refined in [24]; there it is shown that the *uncover* algorithm may simplify those constraints into canonical forms but also that this simplification is not achievable in general. Another version, specifically calibrated to logic programming and constructive negation, is given in [17].

2. The presence of local variables is problematic, as they turn out to be universally quantified in the target program and therefore outside the scope of *SLD*-resolution.

There have been several partial solutions to the above problems: [41] proposes to use a more general fold/unfold transformation system. In [5] it is proposed to compute the set-theoretic complement of the terms in the negative predicate, compiling away the inequalities. See [7] for a more updated presentation of intensional negation.

In the light of the analysis in [24], it is worthwhile comparing the two approaches: the Sato and Tamaki format [41] requires locally stratified possibly non-left-linear many-sorted programs with no local variables. There is no explicit mechanism to handle inequalities, yet a sketch of the adequacy of the procedure is offered. The Barbuti *et al.* format [5] demands flat (non-stratified programs) with local variables provided they are not shared in the body.

The latter yield non-Horn programs with extensional universal quantification, whose operational interpretation is refined from the generate-and-test approach [5] to the idea of *proof by case analysis* [30] . Left-linearization reintroduces inequalities giving more impulse to the move to CLP languages, as proposed in [26].

It can be shown that Barbuti's and Sato's transformations are equivalent w.r.t. success for left-linear programs. Hence, our proof-theoretical reconstruction of intensional negation will work for the Sato transformational approach as well.

6.1.2 Dynamic approaches

Chan [8] is acknowledged to be the inventor of the term 'constructive' negation; his approach can be roughly characterized as a mix of *NF* with a constraints-solving attitude. In essence it consists in taking a negative goal, calling the positive version and negating the answer obtained. As in the CLP family, unification and disunification are kept explicit and returned as solutions. The key observation is that if G is a goal and $A_1, \ldots, A_n$ the answer substitutions (treated as atoms), $G \leftrightarrow A_1 \vee \ldots \vee A_n$ is a logical consequence of the completed database. Therefore the answer to $\neg G$ is the negation of the answers to G. Of course, there is some work to do in keeping the (in)equalities tidy (normalized answers) and there are some obvious problems when dealing with computations that have infinite answers, fixed in [9] by quantifying over the answer substitutions. No proof of completeness is offered. A very neat generalization to constraint logic programming over arbitrary structures is offered in [49]; it turns out to be sound and complete w.r.t. the three-valued models of the completion. A further generalization is presented in [12]. Constructive negation has been also studied in the context of disjunctive logic programming [28] and of functional logic programming [34], where it extends the narrowing-based procedural semantics.

$(E)SLDNF - S$ [45]: The finite failure case in the definition of *SLDNF*-resolution is modified as follows: given the goal $\leftarrow (\Gamma, \neg A)$, then it has a child $\leftarrow \theta\Gamma$, if there is a finitely failed-tree for $\leftarrow \theta A$, where $dom(\theta) \subseteq vars(A)$. So *NF* instantiates under success, i.e. negative goals directly return substitutions: given P and G the aim is to look for a (fail) substitution θ s.t. $P \cup \{\theta G\}$ has a finitely failed tree; then by the soundness of the *NF* rule $\forall\theta\neg G$ is a consequence of $comp(P)$ and thus θ is an answer for the query $\leftarrow \neg G$. This seems very costly, since it allows a full-fledged dimension to substitutions.

This is refined in [29], where it is shown how to avoid generating all possible substitutions in lieu of a maximal general fail substitution. Moreover, an improvement w.r.t. Chan's work lies in the feature of always including some positive bindings for the variable in the negated goal. If the *SLD*-tree is infinite, the method enumerates the set of fail substitutions.

6.2 Non-failure driven negation

Over recent years, ways of incorporating other more logical forms of negation than NF have appeared. Since most of the time this gives back full non-clausal-logic, most of them are catalogued as ATPs. In all these accounts, negative information has to be provided explicitly and specific rules are offered to deal with that. Sometimes it is possible to mix OWA and CWA predicates safely. Without the pretense of being exhaustive, we still have to quote some of the most known proposals: for a more detailed account and bibliography, let us refer to [40].

- *N(Q)Prolog* [14] and *Negation as Inconsistency* [15]: the former is a complete implementation of positive intuitionistic logic. By defining disjunction classically and allowing a restart rule, Gabbay shows it to be complete for full classical logic as well. The latter evaluates a query against an ordered pair $\langle P, N\rangle$, where P is a Horn program and N a set of queries that are required *not* to succeed; this is logically equivalent to adding to the program the negation of all the members of N, and permits importing negative facts and rules. Both systems have a very awkward first-order version.
- Stickel's $PTTP$ [48] supplements *SLD*-resolution with the model elimination rule to offer a complete method for full clausal logic. This entails keeping track of the ancestors of the goal, losing one of the key feature of Prolog, namely input resolution.
- Loveland's $nHProlog$ [40] incorporating case analysis in *SLD*-resolution demands each time the invocation of a restart rule, until the stack of unsolved (disjunctive) heads is empty. Without requiring contrapositives as *PTTP*, it simulates case analysis with different runs of essentially the Prolog engine. Unfortunately naive *nH-Prolog* is incomplete and the new versions (Progressive nH and Inheritance nH) have a less natural and convincing description.
- Another close relative goes under the name of *disjunctive logic programming* (see [39] and references therein). It aims to deal with full clausal logic by extending the machinery for Horn Logic. In particular the procedural semantic, called SLO-resolution, extends the *SLD*-step by having the selection function choose a clause in the current goal (a sequence of clauses) and trying to find a clause whose head subsumes it; if successful, the goal is incremented by the body, without deleting the selected clause.

7 Conclusions

We are in the process of fulfilling the promise made in [33], where we claimed that quite a lot of topics in the logic programming literature would benefit from a proof-theoretic reading. We have shown here that from this perspective much of the mystery of NF disappears or at least it is brought back to standard issues in basic proof-theory. Moreover this is more evidence of the fruitfulness and explicative power of the notion of *regularity* as an abstract characterization of *SLD*-resolution. Indeed we have indicated how constructive negation can be seen as a side effect of regularizing normal programs.

Once the basic theory is digested, the reconstruction of the core of intensional negation (say until [5]) has been accomplished with very little effort. The splitting method for restricted terms is based on Lassez's *uncover* algorithm for the relative complement problem. A well-known and elegant approach to the foundation of *NF* is to address it in the CLP-framework [49, 25]. Nevertheless we believe it is interesting to try to stretch the traditional tools of logic programming, therefore gaining a more in-depth understanding of their limits. In particular, it is worthwhile to investigate how far the sophistication of the splitting method, for example in the direction of a recursive representation of the covering of unrestricted terms, can bring us.

As we have mentioned several times, our main interest is to test the framework of regular search spaces with a relevant application as (constructive) negation. The paper is not meant to try to present original results in the subfield of negation in logic programming, but to formulate a logical revision of a possibly obsolete version of constructive negation in simple terms. At the same time, we think that some new issues have manifested.

It is easy to show the completeness of the PF-system w.r.t. the three-valued semantics of the (weak) completion, although limited to the case where eigenvariables are not involved. We also believe that some new insights can be gained concerning the issue of fail substitutions, which is related to the DCA-schema. Moreover, the need for the latter in complete logic programs [30] has been further highlighted.

In conclusion: while it is clear that the first steps for a new framework are the formalizations of more or less well-known problems in the field, we hope to show that the theory of regular search spaces can be fruitfully used in front-line subjects such as more general program transformation techniques.

Acknowledgements

We would like to thank Roy Dyckhoff, Frank Pfenning, Robert Stärk and two anonymous referees for helpful comments on various drafts of this paper.

References

[1] K. A. Apt, H. A. Blair and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed. pp. 89–148, Morgan Kaufmann, Los Altos, CA, 1988.

[2] K. A. Apt. Logic programming. In *Handbook of Theoretical Computer Science*, J. Leuween, ed. Elsevier, Amsterdam, 1990.

[3] K. A. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, **19**, 9–71, 1994.

[4] R. Barbuti, P. Mancarella, D. Pedreschi and F. Turini. Intensional negation of logic programs. In *Proceedings of Tapsoft'87*, Vol. 259 of *Lecture Notes in Computer Science*, pp. 96–110. Springer-Verlag, Berlin, 1987.

[5] R. Barbuti, P. Mancarella, D. Pedreschi and F. Turini. A transformational approach to negation in logic programming. *Journal of Logic Programming*, **8**, 201–228, 1990.

[6] N. Bidoit. Negation in rule-based database languages: a survey. *Theoretical Computer Science*, **8**, 3–83, 1991

[7] F. Bruscoli, F. Levi, G. Levi and M. C. Meo. Intensional negation in constraint logic programming. In *Proceedings of GULP93*, D. Saccà, ed. pp. 359–373. Università della Calabria, 1993.

[8] D. Chan. Constructive negation based on the completed database. In *Logic Programming. Proceedings of the Fifth International Conference and Symposium*, R. Kowalski and K. Bowen, eds. pp. 111–125. The MIT Press, Cambridge, MA, 1988.

[9] D. Chan. An extension of constructive negation and its application in coroutining. In *Logic Programming. Proceedings of the 1989 North American Conference*, E. Lusk and R. Overbeek, eds. pp. 477–493. The MIT Press, Cambridge, MA, 1989.

[10] K. L. Clark. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, eds. pp. 293–322. Plenum Press, New York, 1978.

[11] H. Comon. Disunification: a survey. In *Computational Logic*, J.-L. Lassez and G. Plotkin, eds. pp. 322–359. The MIT Press, Cambridge, MA, 1991.

[12] W. Drabent. What is failure? An approach to constructive negation. *Acta Informatica*, **32**, 27–59, 1995.

[13] N. Foo, A. Rao, A. Taylor and A. Walker. Deduced relevant types and constructive negation. In *Logic Programming. Proceedings of the Fifth International Conference and Symposium*, R. Kowalski and K Bowen, eds. pp 126–139. The MIT Press, Cambridge, MA, 1988.

[14] D. M. Gabbay and U. Reyle. N-Prolog: an extension of prolog with hypothetical implications: 1. *Journal of Logic Programming*, **1**, 319–355, 1984.

[15] D.M. Gabbay and M. Sergot. Negation as inconsistency: 1. *Journal of Logic Programming*, **3**, 1–36, 1986.

[16] J. H. Gallier. *Logic for Computer Science. Foundations of Automatic Theorem Proving*. John Wiley, New York, 1987.

[17] J. Harland. *On Hereditary Harrop Formulae as a Basis for Logic Programming* PhD Thesis, Edinburgh, 1991.

[18] J. Harland. A clausal form of the completion of logic programs. In *Logic Programming. Proceedings of the Eighth International Conference*, K. Furukawa, ed. pp. 711–725. The MIT Press, Cambridge, MA, 1991.

[19] R. S Kemp and G. A. Ringwood. Reynolds and Heyting models of logic programs. In *Proceedings of the post-Conference Workshop on Proof-Theoretical Extension of Logic Programming (PTELP'94)*, A. Momigliano and M. Ornaghi, eds. pp. 99–108. Università di Milano, Milano, 1994.

[20] M. R. Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Zeit. math. Logik*, **13**, 15–20, 1967.

[21] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, **4**, 289–308, 1987.

[22] J.-L Lassez and K. Marriot. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, **3**, 301–317, 1987.

[23] J.-L Lassez, M. J. Maher and K. Marriot. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed. pp. 587–626. Morgan Kaufmann, Los Altos, CA, 1988.

[24] J.-L. Lassez, M. J. Maher and K. Marriot. Elimination of negation in term algebras. In *Mathematical Foundations of Computer Science*. A. Tarlecki, ed. pp. 1–16. Springer-Verlag, Berlin, 1991.

[25] J.-L. Lassez and M. J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, **20**, 503–581, 1994.

[26] F. Levi. *Constructive Negation and Universal Quantification*. MS Thesis, Università di Pisa, 1992.

[27] J. W. Lloyd. *Foundations of Logic Programming*. Second Extended Edition, Springer-Verlag, Berlin, 1987.

[28] J. Lobo. On constructive negation in disjunctive databases. In *Logic Programming. Proceedings of the 1990 North American Conference*, S. Debray and M. Hermenegildo, eds. pp. 694–705. The MIT Press, Cambridge, MA, 1990.

[29] J. Maluzinski and T. Naslund. Fail substitutions for negation as failure. In *Logic Programming Proceedings of the 1989 North American Conference*, E. Lusk and R. Overbeek, eds. pp. 461–476. The MIT Press, Cambridge, MA, 1989.

[30] P. Mancarella, S. Martini and D. Pedreschi. Complete logic programs with domain closure axiom. *Journal of Logic Programming*, **5**, 263–276, 1988.

[31] D. Miller, G. Nadathur, F. Pfenning and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, **51**, 125–157, 1991.

[32] D. Miller. A logical programming language with lambda-abstraction, function variables and simple unification. In *Extensions of Logic Programming*, P. Schröder-Heister, ed. pp. 253–281. Vol. 475 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, 1991.

[33] A. Momigliano and M. Ornaghi. Regular search spaces as a foundation of logic programming. In *Extensions of Logic Programming*, R. Dyckhoff, ed. pp. 222–254. Vol 798 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, 1994.

[34] J. J. Moreno Navarro. Extending constructive negation for partial functions in lazy functional-logic languages. In *Extensions of Logic Programming*, R. Dyckhoff, H. Herre and P. Schröder-Heister, eds. pp. 213–228. Vol. 1055 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, 1996.

[35] G. Nadathur and D. Miller. An overview of λProlog. In *Logic Programming. Proceedings of the Fifth International Conference and Symposium*, R. Kowalski and K. Bowen, eds. pp. 810–827. The MIT Press, Cambridge, MA, 1988.

[36] L Naish. *Negation and Control in Prolog*. Vol. 239 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1985.

[37] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science*. pp. 74–85, 1991.

[38] D. Prawitz. *Natural Deduction*. Almqvist and Wiksell, Stockholm, 1965.

[39] A. Rajasekar, J. Minker and J. Lobo. *Foundations of Disjunctive Logic Programs*. The MIT Press, Cambridge, MA, 1992.

[40] D. W. Reed and W. D. Loveland. A comparison of three Prolog extensions. *Journal of Logic Programming*, **12**, 25–50, 1992.

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

[42] H. Schwitchtenberg. Some applications of cut-elimination. In *Handbook of Mathematical Logic*, J. Barwise, ed. pp. 867–896. North-Holland, 1977.

[43] J. C. Shepherdson. Negation as failure (II). *Journal of Logic Programming*, 3, 185–202, 1985.

[44] J. C. Shepherdson. Negation in logic programming. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed. pp. 19–88. Morgan Kaufmann, Los Altos, CA, 1988.

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

[46] W Snyder. *A Proof-Theory for General Unification*. Birkhäuser, Boston, MA, 1992.

[47] R. Stärk. *The Proof-Theory of Logic Programs with Negation*. PhD Thesis, University of Bern, 1992.

[48] M. Stickel. A Prolog technology theorem prover: a new exposition and implementation in Prolog. 464 SRI International, Technical Note, 46, 4, Menlo Park, CA, 1989.

[49] P. Stuckey. Constructive negation for constraint logic programming. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science*. pp. 328–339. IEEE Computer Society Press, 1991.

[50] A. S. Troesltra and D. van Dalen. *Constructivism in Mathematics*. Vol 1–2. North-Holland, Amsterdam, 1988.

[51] M. Wallace. Negation by constraints: a sound and efficient implementation of negation in deductive databases. In *Proceedings of the IEEE International Symposium on Logic Programming*. pp. 253–263. IEEE Computer Society Press, 1987.

Appendix

A Proof of the finite failure theorem

We now detail the proof of the finite failure theorem (Theorem 4.4). To achieve that, we need to introduce quite a number of definitions and properties of partial (see below) and finitely-failed *SLD*-trees, before we may embark on the proof of some crucial lemmata. The theorem follows as a corollary of Theorem A.14, that generalizes the left-linearity requirement with the existence of a regular partition of $Fax(P)$.

DEFINITION A.1
A *partial SLD*-tree for $P \cup \{G\}$ is any initial subtree Φ_P of a *SLD*-tree for $P \cup \{G\}$. Let O be (the goal in) a leaf of Φ_P and A be the corresponding selected atom If A is not failed w.r.t. P, then we say that A and O are *open* in Φ_P.

DEFINITION A.2
A selection function is *local* if at any stage the selected atom belongs to the most recently introduced ones. A *SLD*-tree has *local selection* if it is built via a local selection function.

We will denote with $\Phi_P(G \mid \mathbf{H})$ a partial (possibly finitely-failed) *SLD*-tree for $P \cup \{G\}$ with open selected atoms $\mathbf{H} = H_1, \ldots, H_n$. When inferible from the context we will omit the subscript $_P$.

DEFINITION A.3
Let $\leftarrow \mathbf{M}, A, \mathbf{N}$ be a goal, and $A' \leftarrow \mathbf{B}$ a variant of a clause: if $A = \sigma A'$, then $\leftarrow \mathbf{M}, \sigma\mathbf{B}, \mathbf{N}$ is a resolvent. In this case we say that $\leftarrow \mathbf{M}, \sigma\mathbf{B}, \mathbf{N}$ is a *plain* resolvent of $\leftarrow \mathbf{M}, A, \mathbf{N}$. A *SLD*-tree is *plain* if, for every non-terminal node G, the children of G are plain resolvents of G. Moreover we say that Φ *does not introduce local variables* if the applied clauses do not have local variables.

We also need the following extension of the subsumption ordering:

DEFINITION A.4
$\mathbf{H} \leq \mathbf{H}'$ iff $\forall H \in \mathbf{H}\, \exists H' \in \mathbf{H}' : H \leq H'$.

PROPOSITION A.5 (Substitution)
Let $\Phi_P(G \mid \mathbf{H})$ be a partial *SLD*-tree and σ a substitution. Then there is a partial *SLD*-tree $\Phi^{\sigma}_P(\sigma G \mid \mathbf{H}^{\sigma})$ such that:

1. If A is selected in G, then σA is selected in σG, $\mathbf{H}^{\sigma} \leq \mathbf{H}$, and for every open goal O^{σ} of Φ^{σ}_P, there is an open goal O of Φ_P such that $O^{\sigma} = \sigma O$.

2. If Φ_P has local selection, then so does Φ^{σ}_P.

PROOF. By induction on the height of Φ.

- *Basis*. We have the following cases:
  - $\Phi(G \mid \emptyset)$ contains just the goal G and the selected atom A is failed: then $\Phi^{\sigma}(\sigma G \mid \emptyset)$ contains just σG, with (failed) selected atom σA;
  - $\Phi(G \mid A)$ contains just G and the selected atom A is not failed; then Φ^{σ} is either $\Phi^{\sigma}(\sigma G \mid \sigma A)$ (if σA is not failed), or $\Phi^{\sigma}(\sigma G \mid \emptyset)$ (if σA is failed).
- *Step*. $\Phi(G \mid \mathbf{H})$ has immediate subtrees $\Phi_1(G_1 \mid \mathbf{H}_1), \ldots, \Phi_n(G_n \mid \mathbf{H}_n)$, where $G_1, \ldots, G_n$ are the resolvents of G w.r.t. the selected A. The resolvents of σG with respect to σA are $\delta_{i_1} G_{i_1}, \ldots, \delta_{i_m} G_{i_m}$. By the inductive hypothesis, for $1 \leq j \leq m$ there are $\Phi^{\delta_{i_j}}_{i_j}(\delta_{i_j} G_{i_j} \mid \mathbf{H}^{\delta_{i_j}}_{i_j})$ such that 1 and 2 are satisfied (w.r.t. $\Phi_{i_j}(G_{i_j} \mid \mathbf{H}_{i_j})$). Then Φ^{σ} is the *SLD*-tree that has root σG, selected atom σA, and immediate subtrees $\Phi^{\delta_{i_1}}_{i_1}(\delta_{i_1} G_{i_1} \mid \mathbf{H}^{\delta_{i_1}}_{i_1}), \ldots, \Phi^{\delta_{i_1}}_{i_1}(\delta_{i_1} G_{i_1} \mid \mathbf{H}^{\delta_{i_1}}_{i_1})$. One can easily check that Φ^{σ} satisfies 1. Point 2 follows from the fact that the selected atoms in the roots of the subtrees of $\Phi^{\delta_{i_j}}_{i_j}$ are preserved.

∎

We will say that Φ^{σ} has been obtained from Φ by *substitution*. In the following, Φ^{σ} will be the *SLD*-tree that is built up recursively, according to the above proof. For plain trees we have:

PROPOSITION A.6
Let $\Phi_P(G \mid \mathbf{H})$ be a plain *SLD*-tree, and σ be a substitution. Then $\Phi^{\sigma}_P(\sigma G \mid \mathbf{H}^{\sigma})$ coincides with the tree that is obtained by the application of σ to every nodes of Φ and $\mathbf{H}^{\sigma} \subseteq \sigma\mathbf{H}$.

PROOF. By induction on the height of Φ.

- *Basis*. The proof is as in Proposition A.5.
- *Step*. $\Phi(G \mid \mathbf{H})$ has immediate subtrees $\Phi_1(G_1 \mid \mathbf{H}_1), \ldots, \Phi_n(G_n \mid \mathbf{H}_n)$. Since $G_1, \ldots, G_n$ are plain resolvents, the resolvents of σG are $\sigma G_1, \ldots, \sigma G_n$. Then the proof follows from the inductive hypothesis applied to the immediate subtrees.

∎

In the next proposition, we will use the following *weakening operation*. Let $\Phi_P(G \mid \mathbf{H})$ be a partial *SLD*-tree, and G a subgoal of G'. Then one can build a partial *SLD*-tree $\Phi^{w}_P(G' \mid \mathbf{H}')$ by adding the (appropriate instances of the) new atoms of G' to all the nodes of Φ. It is clear that

- $\mathbf{H}' = \mathbf{H}$
- If Φ has local selection, then does Φ^{w}
- If Φ is plain, so is Φ^{w}.

PROPOSITION A.7
Let $\Phi_P(G \mid \mathbf{H})$ be a plain partial *SLD*-tree. Then there is a plain partial *SLD*-tree with local selection $\Phi'_P(H \mid \mathbf{H}')$, where $H \in G$ and $\mathbf{H}' \subseteq \mathbf{H}$.

PROOF. By induction on the height of Φ. The basis is obvious. For the step, let G be $\leftarrow \mathbf{M}, A, \mathbf{N}$, and let $A_1 \leftarrow \mathbf{B}_1, \ldots A_n \leftarrow \mathbf{B}_n$ be the parent clauses such that $A = \sigma_1 A_1, \ldots, A = \sigma_n A_n$ (being Φ plain). Let $\Phi_i(\mathbf{M}, \sigma_i \mathbf{B}_i, \mathbf{N} \mid \mathbf{H}_i)$ be the immediate subtrees. By the inductive hypothesis we obtain n plain $\Phi'_i(H_i \mid \mathbf{H}'_i)$ with $\mathbf{H}'_i \subseteq \mathbf{H}_i$. For the sake of readability, we consider the case $n = 2$. If $H_1 \in \mathbf{M}, \mathbf{N}$, we take Φ'_1 as the desired tree, else, if $H_2 \in \mathbf{M}, \mathbf{N}$, we take Φ'_2 as the desired tree. If none of the above holds, then $H_1 \in \sigma\mathbf{B}_1$ and $H_2 \in \sigma\mathbf{B}_2$, and one can easily build the plain tree $\Phi'(A \mid \mathbf{H}'_1, \mathbf{H}'_2)$ with subtrees obtained by a suitable weakening on $\Phi'_i(H_i \mid \mathbf{H}'_i)$. ∎

Now we can start to prove our theorem. The proof follows from some lemmata that link partial *SLD*-trees to F-proofs.

LEMMA A.8
Let $\Phi_P(G \mid H_1, \ldots, H_n)$ be a partial *SLD*-tree with local selection and A be selected in G: then there is a proof of $\neg A$ in the F-system with assumptions $\neg H_1, \ldots, \neg H_n$, that uses only axioms from $Fax(P)$.

PROOF. By induction on the height of the partial *SLD*-tree.

- *Basis*. Φ has height 0: we have two cases.

$$\begin{tikzcd}[column sep=small] & \leftarrow A_1,\dots,A_s,\dots,A_m \arrow[dl, dash] \arrow[dr, dash] & \\ \leftarrow \sigma_1 \mathbf{M}_{i_1} & \cdot\cdot & \leftarrow \sigma_h \mathbf{M}_{i_h} \\ \Phi_1 & & \Phi_h \end{tikzcd} \qquad \frac{\begin{array}{c}\cdots\cdots\\ \Pi_1\\ \neg\sigma_1 B_{i_1,k_{i_1}}\end{array} \quad \cdots \quad \begin{array}{c}\cdots\cdots\\ \Pi_h\\ \neg\sigma_h B_{i_h,k_{i_h}}\end{array};\quad Fax_{k_1,\dots,k_n}(p)}{\neg A_s}$$

FIG. 3. From partial *SLD*-trees to F-proof-trees

- Φ is failed. Then no clause head of P unifies with the selected $A_s = p(a)$. Hence $\Lambda \in \mathrm{F}(\neg A_s, Fax_{k_1,\dots,k_n}(p))$, where $Fax_{k_1,\dots,k_n}(p)$ is any failure axiom of $Fax(p)$, and we have a F-proof of $\neg A_s$.
- Φ is not failed. Then we have the trivial proof $\neg A_s$ from assumption $\neg A_s$.

- *Step.* Φ has the form indicated in the left-hand side of Figure 3, where $A_s = p(a)$ is the initial selected atom, and for $1 \le j \le h$, $\sigma_j B_{i_j,k_{i_j}}$ is the atom selected in the immediate descendant $\sigma_j \mathbf{M}_{i_j}$. Note that, by the property of local selection, the former has been just introduced and does not belong to $\leftarrow A_1,\dots,A_{s-1},A_{s+1},\dots,A_m$. Thus there is a partial tree $\Phi_j(\sigma_j B_{i_j,k_{i_j}} \mid \mathbf{H}_j)$ with local selection. Now, $\forall x(p(x) \to \bigvee_{i=1}^{n} \exists y_i(x = t_i \wedge \mathbf{M}_i))$ is the only-if definition of $p(\dots)$, and for $1 \le j \le h$, a unifies with t_{i_j} with mgu σ_j. Therefore there is a failure axiom

$$Fax_{k_1,\dots,k_n}(p) = \forall x(p(x) \to \exists y \bigvee_{i=1}^{n}(x = t_i \wedge B_{i,k_i})).$$

By the inductive hypothesis there is a F-proof $\Pi_j :: \neg\sigma_j B_{i_j,k_{i_j}}$ with assumptions $\neg\mathbf{H}_j$, as suggested by the dots at the top of the right-hand side of the figure. Moreover, since t_{i_j} unifies with a with mgu σ_j

$$\neg\sigma_1 B_{i_1,k_{i_1}};\dots;\neg\sigma_h B_{i_h,k_{i_h}} \in \mathrm{F}(\neg p(a), Fax_{k_1,\dots,k_n}(p)).$$

Hence we can build the trees on the right and with one application of the F rule derive $\neg A_s$. Note that here no problem with eigenvariables arises.

∎

DEFINITION A.9
A set **A** of atoms is *F-refuted* by a program P if there are F-proofs $\Pi_1 :: \neg\sigma_1 H_1,\dots,\Pi_k :: \neg\sigma_k H_k$ with axioms from $Fax(P)$ such that $\{H_1,\dots,H_k\} \subseteq \mathbf{A}$ and $\|\sigma_1\mathbf{A}\|,\dots,\|\sigma_k\mathbf{A}\|$ is a covering of $\|\mathbf{A}\|$.

LEMMA A.10
Let $\Phi_P(G \mid \mathbf{H})$ be a plain *SLD*-tree which does not introduce local variables. If every open goal O of Φ_P is F-refuted by P, so is the root G.

PROOF. Let $O_1,\dots,O_k$ be the open goals of Φ. For $1 \le i \le k$, let $\Pi_j^i :: \neg\sigma_j^i H_j^i$ (with $1 \le j \le m_i$) be the F-proofs that F-refute O_i. Consider the substitutions

$$\sigma_{\bar{j}_k} = glb(\sigma_{j_1}^1,\dots,\sigma_{j_k}^k)$$

and let $\Phi_{\bar{j}_k}(G \mid H_{j_1}^1,\dots,H_{j_k}^k)$ be the tree equal to Φ, but selecting $H_{j_1}^1,\dots,H_{j_k}^k$ in the open goals. By Proposition A.7, there is $\Phi_{loc}(A \mid H_{j_1}^1,\dots,H_{j_k}^k)$ with local selection function, such that $A \in G$. We can thus apply $\sigma_{\bar{j}_k}$ to Φ_{loc} and obtain $\Phi_{loc}^{\sigma_{\bar{j}_k}}(\sigma_{\bar{j}_k} A \mid \mathbf{H}^{\sigma_{\bar{j}_k}})$ s.t. $\mathbf{H}^{\sigma_{\bar{j}_k}} \subseteq \{\sigma_{\bar{j}_k} H_{j_1}^1,\dots,\sigma_{\bar{j}_k} H_{j_k}^k\}$ (Proposition A.6). By Lemma A.8 there is a proof $\Pi :: \neg\sigma_{\bar{j}_k} A$ with assumptions belonging to $\neg\mathbf{H}^{\sigma_{\bar{j}_k}}$, and the latter have F-proofs with axioms from $Fax(P)$ (indeed, in the F-system, an instance of a provable formula is provable by the same axioms). Now we have to prove that that the set of all the $\|\sigma_{\bar{j}_k} G\|$ is a covering of $\|G\|$. This follows from the fact that, for $1 \le i \le k$, $\|\sigma_1^i O_i\|,\dots,\|\sigma_{m_i}^i O_i\|$ is a covering of the open goal $\|O_i\|$, and the variables of the open goals are a subset of the ones of G, as Φ is plain and does not introduce local variables. ∎

LEMMA A.11 (Factorization)
Let $\Phi_P(G \mid \mathbf{H})$ be a partial *SLD*-tree that does not introduce local variables and assume that every failure axiom $Fax(p)$ of P has a regular splitting. Then there is a factorization $\Phi_1(\rho_1 G \mid \mathbf{H}_1)\dots,\Phi_n(\rho_n G \mid \mathbf{H}_n)$ such that:

(1) for $1 \leq j \leq n$, each open goal of Φ_j is an instance of some open goal of Φ, $\mathbf{H}_j \leq \mathbf{H}$ and Φ_j is a plain *SLD*-tree which does not introduce local variables;

(2) $\|\rho_1 G\|, \ldots, \|\rho_n G\|$ is a covering of $\|G\|$.

PROOF. By induction on the height of Φ. The basis is obvious. The step goes on as follows. Let G be $\leftarrow \mathbf{M}, p(t), \mathbf{N}$, and let $\neg p(a_1), \ldots, \neg p(a_n)$ be the regular heads of $Fax(p)$ such that $\mu_j = mgu(p(t), p(a_j))$ is defined. Then $\|p(\mu_1 t)\|, \ldots, \|p(\mu_n t)\|$ is a covering of of $\|p(t)\|$. Therefore $\|\mu_1 G\|, \ldots, \|\mu_n G\|$ is a covering of $\|G\|$, and we can construct a first factorization $\Phi^{\mu_1}(\mu_1 G \mid \mathbf{H}^{\mu_1}) \ldots \Phi^{\mu_n}(\mu_n G \mid \mathbf{H}^{\mu_n})$, by the substitution operation. Then we further factorize each $\Phi^{\mu_j}(\mu_j G \mid \mathbf{H}^{\mu_j})$ as follows.

Let $\Phi^j_1(G^j_1 \mid \mathbf{H}^j_1), \ldots, \Phi^j_{m_j}(G^j_{m_j} \mid \mathbf{H}^j_{m_j})$ be the immediate subtrees of $\Phi^{\mu_j}(\mu_j G \mid \mathbf{H}^{\mu_j})$. By the inductive hypothesis, each Φ^j_h, $1 \leq h \leq m_j$, can be factorized into plain trees $\Phi^j_{h,1}(\rho^j_{h,1} G^j_h \mid \mathbf{H}^j_{h,1}), \ldots, \Phi^j_{h,k^j_h}(\rho^j_{h,k^j_h} G^j_h \mid \mathbf{H}^j_{h,k^j_h})$ that satisfy (1), (2) (w.r.t Φ^j_h). In particular, $\|\rho^j_{h,1} G^j_h\|, \ldots, \|\rho^j_{h,k^j_h} G^j_h\|$ is a covering of $\|G^j_h\|$. Now, let us consider the combinations such that $\rho^j_{\bar{t}_{m_j}} = glb(\rho^j_{1,t_1}, \ldots, \rho^j_{m_j,t_{m_j}})$ is defined. Then, for every h such that $1 \leq h \leq m_j$ there is a σ^j_h such that $\rho^j_{\bar{t}_{m_j}} = \sigma^j_h \rho^j_{h,t_h}$, and, by the substitution operation, we can build the tree:

$$\Phi^{j\sigma^j_h}_{h,t_h}(\rho^j_{\bar{t}_{m_j}} G^j_h \mid \mathbf{H}^{j\sigma^j_h}_{h,t_h}).$$

Notice that $\mu_j G = \mu_j \mathbf{M}, p(\mu_j t), \mu_j \mathbf{N}$, where $p(\mu_j t) = p(\mu_j a_j)$ and $p(a_j)$ is a regular instance. Therefore $\rho^j_{\bar{t}_{m_j}} G^j_1, \ldots, \rho^j_{\bar{t}_{m_j}} G^j_{m_j}$ are the plain resolvents of $\rho^j_{\bar{t}_{m_j}} \mu_j G$, and we can build the *SLD*-tree:

$$\Phi^j_{\bar{t}_{m_j}}(\rho^j_{\bar{t}_{m_j}} \mu_j G \mid \mathbf{H}^{j\sigma^j_1}_{1,t_1}, \ldots, \mathbf{H}^{j\sigma^j_{m_j}}_{m_j,t_{m_j}})$$

which has $\Phi^{j\sigma^j_1}_{1,t_1}, \ldots, \Phi^{j\sigma^j_{m_j}}_{m_j,t_{m_j}}$ as the immediate subtrees. One can check that every $\Phi^j_{\bar{t}_{m_j}}$ satisfies (1).

Now we prove (2). Since (for $1 \leq j \leq n$) $\mu_j G$ has plain children and no eigenvariable is introduced, the variables of the children $G^j_1, \ldots, G^j_{m_j}$ are contained in the ones of $\mu_j G$. By inductive hypothesis, for $1 \leq h \leq m_j$, $\|\rho^j_{h,1} G^j_h\|, \ldots, \|\rho^j_{h,k^j_h} G^j_h\|$ is a covering of $\|G^j_h\|$. Then one can show that the collection of the $\|\rho^j_{\bar{t}_{m_j}} \mu_j G\|$ (w.r.t. the possible choices of $t_1, \ldots, t_{m_j}$) is a covering of $\|\mu_j G\|$. Therefore, the entire collection (w.r.t. the possible choices of $j, t_1, \ldots, t_{m_j}$) is a covering of $\|G\|$. ∎

COROLLARY A.12
Let P be a program which admits a regular splitting and let $\Phi_P(G \mid \mathbf{H})$ be a partial *SLD*-tree that does not introduce local variables. If every open goal O of Φ_P is F-refuted by P, so is the root G.

PROOF. By Lemma A.11, we can factorize Φ_P. Since the open goals of the *SLD*-trees of the factorization are instances of open goals of Φ_P, the former are F-refuted by P. Then we can apply Lemma A.10 to every plain element of the factorization. ∎

In order to treat the case of local variables, we need stronger hypotheses. First of all, DCA is required. Moreover, we will assume that the program admits a *deterministic splitting*, namely a splitting where $\|p(a_i)\| \cap \|p(a_j)\| = \emptyset$, for $i \neq j$, and we will deal with *disjoint coverings*, where a covering $\|\rho_1 G\|, \ldots, \|\rho_n G\|$ is disjoint if any two distinct elements of it are disjoint. Starting from Definition A.9, everything can be restated by using F_{DCA}-proofs instead of F-proofs, disjoint coverings, instead of coverings, and deterministic splittings instead of regular splittings. Indeed, from now on, we will use appropriately modified versions, in the above sense, of previous results in the Appendix.

LEMMA A.13
Let P be a program that admits a deterministic splitting and let $\Phi_P(G \mid \mathbf{H})$ be a finitely failed *SLD*-tree for $P \cup \{G\}$. If the open goals of Φ_P are F_{DCA}-refuted by P, then so is G.

PROOF. By induction on the number of clauses that introduce local variables in Φ.

Basis. The proof follows from Corollary A.12.

Step. For the sake of readability, we assume that P contains only one clause $p(\tau) \leftarrow K$ for its definition, where K introduces the local variables v. Φ_P can be decomposed into a partial *SLD*-tree $\Phi_0(G \mid p(t_1), \ldots, p(t_n))$ that does not introduce local variables and n finitely failed *SLD*-trees $\Phi_j(G_j \mid \mathbf{H}_j)$, where, for $1 \leq j \leq n$, O_j is the open goal of Φ_0 selecting $p(t_j)$, and the root of Φ_j is the resolvent of O_j with parent clause $p(\tau) \leftarrow K$.

By Lemma A.11, there is a factorization of Φ_0 into plain *SLD*-trees $\Phi^1(\rho_1 G \mid \mathbf{H}^1), \ldots, \Phi^n(\rho_n G \mid \mathbf{H}^n)$. Let O be an open goal of Φ^j: we further factorize Φ^j w.r.t. O, as follows. O is of the form $\leftarrow \mathbf{M}, p(t), \mathbf{N}$, where $p(t)$ unifies with $p(\tau)$. A disjoint covering of $\|p(t)\|$ is $\|p(\mu_1 a_1)\|, \ldots, \|p(\mu_h a_h)\|$, where $p(a_1), \ldots, p(a_h)$ are the regular heads of $Fax(p)$ such that $\mu_i = mgu(p(t), p(a_i))$ is defined. By the substitution operation, we build the *SLD*-trees $\Phi^{j\mu_1}(\mu_1 \rho_j G \mid \mathbf{H}^{j\mu_1}), \ldots, \Phi^{j\mu_h}(\mu_h \rho_j G \mid \mathbf{H}^{j\mu_h})$. For $1 \leq i \leq h$, $\Phi^{j\mu_i}$ contains the open atom $p(\mu_i a_i)$ in the place of $p(t)$, since $\mu_i t = \mu_i a_i$ and Φ^j is plain (by Proposition A.6). Notice that $\|\mu_1 \rho_j G\|, \ldots, \|\mu_h \rho_j G\|$ is a disjoint covering of $\|\rho_j G\|$, as the variables of $\rho_j G$ contain those of $p(t)$ (indeed Φ^j is plain and does not introduce local variables). Now we can further factorize each *SLD*-tree of $\{\Phi^{j\mu_i}(\mu_i \rho_j G \mid \mathbf{H}^{j\mu_i})\}$ w.r.t. another open goal, and go on in this way, until we obtain a final factorization of Φ^j, where each open atom is an instance of some regular head of $Fax(p)$.

If we put the final factorizations of $\Phi^1, \ldots, \Phi^n$ together, we get a factorization $\Phi'_1(\delta_1 G \mid \mathbf{H}'_1), \ldots, \Phi'_z(\delta_z G \mid \mathbf{H}'_z)$ of $\Phi_0(G \mid \mathbf{H}_0)$. Since at each factorization step we have built a disjoint covering of an element of a previous disjoint covering, $\|\delta_1 G\|, \ldots, \|\delta_z G\|$ is a disjoint covering of $\|G\|$. Therefore, if we can show that the open goals of each Φ'_j (with $1 \leq j \leq z$) are F_{DCA}-refuted by P, we can apply (the modified version of) Corollary A.12 to each Φ'_j, and we are done.

Let δ_j $(1 \leq j \leq z)$ be be a substitution of the above factorization. The tree $\Phi_P^{\delta_j}$ (obtained by substitution from Φ_P) can be decomposed into an initial subtree $\Phi'(\delta_j G \mid p(b_1), \ldots, p(b_r))$ and r subtrees $\Psi_1(Q_1 \mid \mathbf{H}_1^{\delta_j}), \ldots, \Psi_r(Q_r \mid \mathbf{H}_r^{\delta_j})$ such that:

- $\Phi'(\delta_j G \mid p(b_1), \ldots, p(b_r))$ coincides with $\Phi'_j(\delta_j G \mid \mathbf{H}'_j)$;
- for $1 \leq i \leq r$, the goal containing $p(b_i)$ is of the form $\leftarrow \mathbf{M}_i, p(b_i), \mathbf{N}_i$ and Q_i is its plain child, of the form $\leftarrow \mathbf{M}_i, \theta_i K, \mathbf{N}_i$ ($p(b_i)$ is of the form $\theta_i p(\tau)$, since it is an instance of some regular head of $Fax(p)$).

For $1 \leq i \leq r$, the open goals of Ψ_i are instances of open goals of Φ_P, hence they are F_{DCA}-refuted by P. Since the substitution operation does not increase the height of a tree, we can apply the inductive hypothesis to the subtrees $\Psi_1, \ldots, \Psi_r$. Therefore, for $1 \leq i \leq r$, the set $\mathbf{M}_i, \theta_i K, \mathbf{N}_i$ is F_{DCA}-refuted by P. Let $\theta_i K$ be $K'(x, y)$, where y are the suitable renaming of the local variables, and let

$$\neg\beta_1 C_1, \ldots, \neg\beta_u C_u, \neg\gamma_1 K'(x, y), \ldots, \neg\gamma_s K'(x, y)$$

be the F_{DCA}-proved formulae, where $C_1, \ldots, C_u$ belong to $\mathbf{M}_i, \mathbf{N}_i$. Since we have a disjoint covering and y are not in the domain of $\beta_1, \ldots, \beta_u$, for $1 \leq m \leq s$, $K'(\gamma_m(x), y)$ must be covered by $\gamma_1 K'(x, y), \ldots, \gamma_s K'(x, y)$. Since the latter have F_{DCA}-proofs, by an application of the DCA-instance $\forall(\forall w(\neg\gamma_1 K'(x, y) \wedge \cdots \wedge \neg\gamma_s K'(x, y)) \rightarrow \neg K'(\gamma_m(x), y)$ we get an F_{DCA}-proof of $\neg K'(\gamma_m(x), y)$. Since $\neg K'(\gamma_m(x), y) \in \mathrm{F}(\neg\gamma_m p(b_i), Fax(p))$, we get a F_{DCA}-proof of $\neg\gamma_m p(b_i)$. Therefore $\neg\beta_1 C_1, \ldots, \neg\beta_u C_u, \neg\gamma_1|_x p(b_i), \ldots, \neg\gamma_s|_x p(b_i)$ are F_{DCA}-proved formulae and the involved substitutions give rise to a disjoint covering of $\|O_i\|$, that is the open goal containing $p(b_i)$ is F_{DCA}-refuted by P. Since this holds for $1 \leq i \leq r$, the proof is concluded. ∎

Now we can prove our main theorem.

THEOREM A.14 (Finite failure 1)
If for a program P which admits a deterministic splitting for $Fax(P)$ and for an atom A, $P \cup \{\leftarrow A\}$ has a finitely failed *SLD*-tree, then there is a proof $\Pi :: \neg A$ in the F_{DCA}-system applying only axioms from $Fax(P)$.

PROOF. By hypothesis there is a *SLD*-tree $\Phi_P(A \mid \emptyset)$. By Lemma A.13, we have a (disjoint) covering $\|\sigma_1 A\|, \ldots, \|\sigma_k A\|$ of $\|A\|$ such that there is a proof $\Pi_j :: \neg\sigma_j A$, for $1 \leq j \leq k$. An application of the DCA-instance $\forall(\neg\sigma_1 A \wedge \ldots \wedge \neg\sigma_k A \rightarrow \neg A)$ yields our proof. ∎

THEOREM A.15 (Finite failure 2)
If for a left-linear program P and an atom A, $P \cup \{\leftarrow A\}$ has a finitely failed *SLD*-tree, then there is a proof $\Pi :: \neg A$ in the F_{DCA}-system applying only axioms from $Fax(P)$.

PROOF. Since P is left-linear, then we can build a regular splitting for $Fax(P)$. ∎

REMARK A.16
There are cases where DCA is not needed. By Lemma A.8, this happens for Krom programs. Indeed in a Krom program non-local selection is not possible. If the clauses of P do not contain local variables, DCA is not needed in a weaker sense. In this case, by Corollary A.12, we have that if $P \cup \{\leftarrow A\}$ has a finitely failed *SLD*-tree, then there are F-proofs $\Pi_1 :: \neg\sigma_1 A, \ldots, \Pi_n :: \neg\sigma_n A$ such that $\|\sigma_1 A\|, \ldots, \|\sigma_n A\|$ is a covering of $\|A\|$.

REMARK A.17
We have taken into account only definite programs and *SLD*-trees. The treatment easily extends to normal programs and *SLDNF*-trees, under the following hypotheses:

- Only atoms are selected in the intermediate nodes of a *SLDNF*-tree.
- A leaf is open if the selected literal is a non-failed atom or a negated atom.
- Instead of using the F_{DCA}-system, we use the PF_{DCA}-system. Moreover, we extend Definition A.9 to sets of literals in the obvious way (A is refuted by $\Pi :: \neg A$ and $\neg A$ by $\Pi :: A$).

In particular, we can apply the extension Lemma A.13 to the *SLDNF*-tree $\Phi^*(L \mid A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m)$ considered in the proof of Theorem 4.5. Indeed, left-linearity entails the existence of a regular partition. Note that, in Theorems A.10 and 4.5, left-linearity can be replaced by the existence of a deterministic splitting.

Received 3 January 1995