The Notion of Floundering for SLDNF-Resolution Revisited

Jean-Hugues Réty

INRIA - Rocquencourt, B.P. 105, 78153 Le Chesnay cedex, France.
E-mail: **jean-hugues.rety@inria.fr**

Abstract. SLDNF-resolution is the standard operational semantics for normal programs. Its safeness condition induces a notion of floundering which may be considered as a programming error. But this notion of floundering seems too weak: a program and a goal may flounder despite the existence of an SLDNF-tree and, in such a case, one computation may flounder while another one may not. We propose a novel stronger notion of floundering – called *true floundering* – which depends only on the existence of an (extended) SLDNF-tree and not on a particular strategy. This notion formalises the idea that the existence of a floundered goal is irrelevant if with another strategy, it would have been possible to continue the computation without floundering. We also introduce a new operational semantics called *extended SLDNF-resolution*. Extended SLDNF-resolution may be efficiently implemented. Moreover, with a result from Drabent ([Dra94]), extended SLDNF-resolution is complete wrt Kunen semantics for non true floundering programs and goals, which corresponds to a large class of normal programs with a declarative characterization.

1 Introduction

SLDNF-resolution is an extension of SLD-resolution which incorporates the negation as failure rule proposed by Clark ([Cla79]). Its soundness needs the application of a safeness condition (a negative literal in a goal cannot be selected until it is not ground) which involves a notion of floundering.

Current implementations consider floundering as an error. So one problem is to avoid floundering, i.e. to build programs that do not flounder. Research has mainly taken two approaches: the first one is to define (syntactic) classes of floundering-free programs and, the second one is to design analysis tools for testing floundering-freeness. Let us cite for instance [BM90] and [MSD90]. We think that the problems encountered mainly come from the notion of floundering itself which is not satisfactory.

Floundering is a computational notion and a real problem is that it depends on the selection rule and on the clause choice rule (see the examples in Sect. 4). One computation for a program and a goal may flounder while another one may not. This is obviously very unexpected: it makes difficult to statically state whether a program may flounder or not. Moreover we will show that in some cases, when a floundering occurs, it is possible to continue the computation (with a refined notion of computation) and be complete wrt Kunen semantics ([Kun87]). Indeed a program and a goal may flounder although an SLDNF-tree exists. The notion of floundering seems to be too weak. So it is worth visiting it again.

The literature usually defines floundering in the following way: Let P be a normal program and G a normal goal, The evaluation of $P \bigcup \{G\}$ flounders if a goal which contains only non-ground negative literals is reached. This is strongly operational and actually, is related to computation. There is a real lack of a declarative notion of floundering (i.e. abstracted from computation).

According to the standard definition of SLDNF-resolution ([Llo87]), for some programs and goals, no SLDNF-tree or SLDNF-derivation exists. There are three reasons: Firstly, SLDNF-trees being built in a "bottom-up" way with ranks which are natural numbers, infinite sequences of negative calls cannot be taken into account. Secondly, for a program like $\{p \leftarrow p\}$ and a goal $\leftarrow \neg p$ (p being ground), there exists neither an SLDNF-refutation nor a finitely failed SLDNF-tree for $\leftarrow p$. The third reason is floundering: no declarative notion of floundering is given and, e.g. for the goal $\leftarrow \neg p(X)$ and some program, no SLDNF-tree exists. These problems are mentioned in [AB91] and [AP91] and some re-definitions of SLDNF-resolution have been proposed ([AD92], [MT92], [MN90]) but these papers are mainly interested in solving the first two problems.

This paper proposes a stronger notion of floundering called *true floundering* which corresponds to the idea that the existence of a floundered goal is irrelevant if, with another strategy, it would have been possible to continue the computation without floundering. We prove that true-floundering is independent of the selection rule. We prove a more general result of which a particular consequence is that, for SLDNF-resolution, successes are independent of the selection rule. Moreover, true floundering can be easily and efficiently implemented.

Another approach to the problem of floundering is in [Dra94]. Drabent remarks that the notion of floundering is too general. He gives a new notion of floundering - called serious floundering. We compare true floundering and serious floundering in Sect. 5. A consequence of the main result of [Dra94] is that, for an implementation of SLDNF-resolution with true floundering, a computation which terminates and does not truly flounder is complete wrt Kunen semantics. We consider that the absence of true floundering is a desirable property of normal programs.

Section 2 defines true floundering and extends SLDNF-resolution. Section 3 shows the independence from the selection rule of extended SLDNF-resolution. Section 4 is devoted to implementation issues: we study the computational problems that arise from floundering and we discuss the implementation of true floundering.

2 True Floundering

We define a notion of floundering that we call *true floundering*. We choose to base our work on the definition of SLDNF-resolution of [Llo87]. We do not recall the definitions of SLDNF-refutation and finitely failed SLDNF-tree ([Llo87]).

We first define the preliminary notion of *fully floundered SLDNF-tree*. Intuitively, a fully floundered SLDNF-tree has no success leaf node and at least one floundering leaf node.

Definition 1. Let P be a normal program and G a normal goal. A *fully floundered SLDNF-tree of rank 0* for $P \bigcup \{G\}$ is a tree satisfying the following:

1. Each node of the tree is a non-empty normal goal.
2. The root node is G.
3. Only positive literals are selected at nodes in the tree.
4. Let $\leftarrow L_1, ..., L_m, ..., L_p$ be a non-leaf node in the tree and suppose that L_m is an atom and it is selected. Then for each program clause (variant) $A \leftarrow M_1, ..., M_q$ such that L_m and A are unifiable with mgu θ, this node has a child $\leftarrow (L_1, ..., L_{m-1}, M_1, ..., M_q, L_{m+1}, ..., L_p)\theta$.
5. Let $\leftarrow L_1, ..., L_p$ be a leaf node in the tree, then either
   (a) each L_i is a non-ground negative literal, or
   (b) an atom L_m is selected and there is no program clause (variant) in P whose head unifies with L_m.
6. There is at least one leaf node satisfying 5.(a).

Definition 2. Let P be a normal program and G a normal goal. A *fully floundered SLDNF-tree of rank k+1* for $P \bigcup \{G\}$ is a tree satisfying the following:

1. Each node of the tree is a non-empty normal goal.
2. The root node is G.
3. Let $\leftarrow L_1, ..., L_m, ..., L_p$ be a non-leaf node in the tree and suppose L_m is selected. Then either
   (a) L_m is an atom and, for each program clause (variant) $A \leftarrow M_1, ..., M_q$ such that L_m and A are unifiable with mgu θ, this node has a child $\leftarrow (L_1, ..., L_{m-1}, M_1, ..., M_q, L_{m+1}, ..., L_p)\theta$, or
   (b) L_m is a ground negative literal $\neg A$ and there is finitely failed SLDNF-tree of rank k for $P \bigcup \{\leftarrow A_m\}$, in which case the only child is $\leftarrow L_1, ..., L_{m-1}, L_{m+1}, ..., L_p$.
4. Let $\leftarrow L_1, ..., L_p$ be a leaf node in the tree, then either
   (a) each L_i is a non-ground negative literal, or a ground negative literal $L_i = \neg A_i$ satisfying: there is a fully floundered SLDNF-tree of rank k for $P \bigcup \{\leftarrow A_i\}$, or
   (b) an atom L_m is selected and there is no program clause (variant) in P whose head unifies with L_m, or
   (c) a ground negative literal $L_m = \neg A_m$ is selected and there is an SLDNF-refutation of rank k of $P \bigcup \{\leftarrow A_m\}$.
5. There is at least one leaf node satisfying 4.(a).

Definition 3. Let P be a normal program and G a normal goal. A *fully floundered SLDNF-tree* for $P \bigcup \{G\}$ is a fully floundered SLDNF-tree of rank k for $P \bigcup \{G\}$, for some k.

The notion of SLDNF-tree is extended in order to deal with ground negative literals for which there exists a fully floundered SLDNF-tree. This is simply achieved by adding the case 4.(c) to the definition of SLDNF-tree ([Llo87]).
The notion of SLDNF-derivation may be extended in the same way.

Definition 4. Let P be a normal program and G a normal goal. An *extended SLDNF-tree* of $P \bigcup \{G\}$ is a tree satisfying the following:

1. Each node of the tree is a (possibly empty) normal goal.
2. The root node is G.
3. Let $\leftarrow L_1, ..., L_m, ..., L_p$, $(p \geq 1)$ be a non-leaf node in the tree and suppose that L_m is selected. Then either
   (a) L_m is an atom and, for each program clause (variant) $A \leftarrow M_1, ..., M_q$ such that L_m and A are unifiable with mgu θ, the node has a child $\leftarrow (L_1, ..., L_{m-1}, M_1, ..., M_q, L_{m+1}, ..., L_p)\theta$, or
   (b) L_m is a ground negative literal $\neg A_m$ and there is finitely failed SLDNF-tree for $P \bigcup \{\leftarrow A_m\}$, in which case the only child is $\leftarrow L_1, ..., L_{m-1}, L_{m+1}, ..., L_p$.
4. Let $\leftarrow L_1, ..., L_m, ..., L_p$, $(p \geq 1)$ be a leaf node in the tree. Then either
   (a) L_m is an atom, L_m is selected and there is no program clause (variant) in P whose head unifies with L_m, or
   (b) L_m is a ground negative literal $\neg A_m$, L_m is selected and there is an SLDNF-refutation of $P \bigcup \{\leftarrow A_m\}$, or
   (c) each L_i is either a non-ground negative literal or a ground negative literal $L_i = \neg A_i$ satisfying: there is a fully floundered SLDNF-tree for $P \bigcup \{\leftarrow A_i\}$.
5. Nodes which are the empty goal have no children.

A leaf node which satisfy 4.(c) is said to be *true floundering*. A leaf node which satisfy 5. (resp. 4.(a) or 4.(b)) is said to be *success* (resp. *failure*).

Definition 5. An extended SLDNF-tree is said to be *truly floundered* if it has (at least) one true floundering leaf node.

3 Independence from the Selection Rule

Independence from the selection rule is an expected property for a non-deterministic algorithm. This independence problem, despite its relevance, is not solved in general for SLDNF-resolution.

Since we are interested in an extended definition of SLNDF-resolution including a notion of floundering, a further difficulty arises from this floundering. It may be related to the difficulties encountered in SLD-resolution with delay ([Nai93],[BFR93]) or concurrent (constraint) logic programming ([CF+93]).

These papers study the independence from the selection rule (from the scheduling for [CF+93]) and have to deal with a notion of deadlock which is close to floundering. However the problem here is complicated by the presence of subsidiary derivations.

The *computed answer* of a non-failure leaf node of an extended SLDNF-tree is the composition of the substitutions used in the branch.

Proposition 6. *Let P be a normal program and G a normal goal. Let T and T' be two extended SLDNF-trees for $P \bigcup \{G\}$.*
If there exists a non-failure leaf node B (possibly empty) of T with computed answer σ, $\langle C_1, ..., C_n \rangle$ being the sequence of the variants of program clauses used in the branch. Then there exists a non-failure leaf node B' of T' with computed answer σ', $\langle C'_1, ..., C'_n \rangle$ being the sequence of the variants of program clauses used in the branch, satisfying:
σ' is a variant of σ and,
B' is a variant of B and,
$\forall i \in \{1, ..., n\}$ there exists a variant $\tilde{C}_i$ of C_i such that the sequence $\langle C'_1, ..., C'_n \rangle$ is a permutation of the sequence $\langle \tilde{C}_1, ..., \tilde{C}_n \rangle$.

The proof is based on the grammatical view of logic programming ([DM93]) which provides a general framework to prove static properties of logic programs. It is an extention of the proof of the main result of [BFR93] which states the independence from the selection rule for SLD-resolution with delay.

We give a very precise formulation of this independence in term of skeletons of partial proof trees. In order to deal with negation we extend the notion of skeleton to incorporate negative leaf nodes. We define a particular set of skeletons (a skeleton is an object which "sums up" the clause choices of a derivation) which is intrinsic to the program. Then the result is proven by the existence of a bijection between the set of non-failure leaf nodes of an extended SLDNF-tree and this set of skeletons. We don't give the proof here. The interested reader may refer to the appendix.

We now give the definition of true floundering for a program and a goal:

Definition 7. Let P be a normal program and G a normal goal. $P \bigcup \{G\}$ *truly flounders* if there is a truly floundered extended SLDNF-tree for $P \bigcup \{G\}$.

Remark. One may argue that (for a program and a goal) true floundering is irrelevant if there is a full success leaf node (its computed answer does not instantiate the goal). Definition 7 may be modified in the following way: $P \bigcup \{G\}$ *truly flounders* if there is a truly floundered extended SLDNF-tree with no full success leaf node for $P \bigcup \{G\}$.

Remark. Another improvement may be made by taking into account non-defined predicates and slightly weakening the safety condition. Indeed if a predicate p is non-defined in a program then a call such as $\neg p(X)$ (where X is not ground) may be considered successful.

4 Implementation Issues

There is a important problem of incompleteness between SLDNF-resolution and implementations. The reasons are infinite computations (this may be related to the case of SLD resolution but it is here complicated by the problem of infinite sequences of negative calls) and floundering.

In this section we give a refined notion of computation - called *extended computation* - which calculates an extended SLDNF-tree when it terminates.

We first define what a computation is and discuss the problems that arise from floundering. Then we define extended computation.

In [Llo87] a computation is defined as an attempt to build an SLDNF-derivation. For our purposes, we may say that a computation for SLDNF-resolution behaves like a computation for SLD-resolution as long as an atom is selected. When a ground negative literal $\neg A$ is selected in a goal H, a computation for A is started. If it eventually fails (a finitely failed SLDNF-tree being built) then H has one child with the literal $\neg A$ removed. If it eventually succeeds (an SLDNF-refutation for A being built) then H has failed.
A computation flounders if a goal with only non-ground negative literals is reached.

Floundering being considered as an error, actual implementations of SLDNF-resolution stop a computation as soon as a floundering occurs. The following examples show that it would be worth continuing the computation.

Example 1. Let's consider the normal program $P1$:
$s(X) \leftarrow \neg p(a), q.$
$p(a) \leftarrow \neg r(X).$

and the goal $G1$:
$\leftarrow s(X).$

Figure 1 shows two computations for $P1 \bigcup \{G1\}$ under two different selection rules. The selection of $\neg p(a)$ (computation C_1) in the goal $\neg p(a), q$ leads to a floundering in the subsidiary computation. The selection of q (computation C_2) leads to a failure and computation C_2 does not flounder.

There exists an SLDNF-tree for $P1 \bigcup \{G1\}$ but computation C_1 does not calculate it. One may remark that a correct answer could have been obtained by going back to the node $\neg p(a), q$, selecting q and then continuing the computation. Such a computation would then be correct and complete despite the existence of a floundering.

Example 2. Let's consider the normal program $P2$:
$p(a) \leftarrow \neg r(X).$
$p(a).$

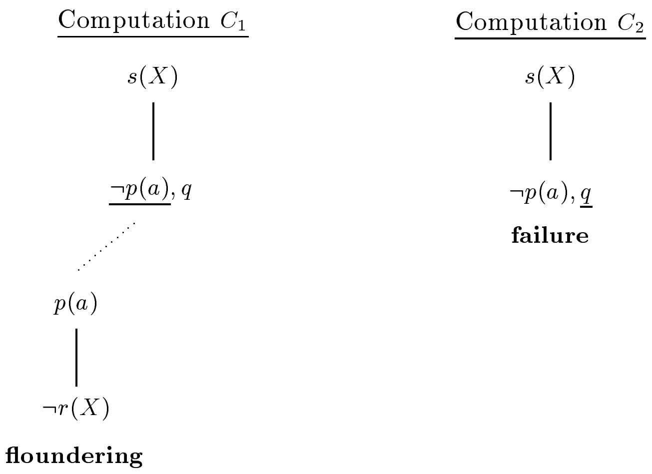

Fig. 1. Some computations for example 1

and the goal $G2$:
$\leftarrow \neg p(a)$.

Figure 2 shows two computations for $P2 \bigcup \{G2\}$. The choice of the first clause in the definition of p leads to a floundering (computation C_3) while the choice of the second clause leads to a failure and computation C_4 does not flounder.

There exists an SLDNF-tree for $P2 \bigcup \{G2\}$ but computation C_3 does not calculate it. A correct answer could have been obtained by backtracking and continuing the (subsidiary) computation for $\neg p(a)$. Then, an SLDNF-refutation being built for $p(a)$, one could have conclude to a failure for $P2 \bigcup \{G2\}$ despite the existence of a floundering. Such a computation would be correct and complete.

These examples show that in some cases it is possible to continue a computation (with possibly a different selection rule, with possibly a different clause choice rule) and obtain all the answers despite a floundering.

We propose a computation to apply the following rules:

1. Only select in a goal either an atom or a ground negative literal *non-marked as floundered*.
2. When a goal with only non-ground negative literals or ground negative literals marked as floundered is reached, backtrack to the previous choice point and continue the computation.
3. When a computation for a negative call $\neg A$ in a goal H builds a (finite) tree without success but with at least one leaf node with only non-ground negative literals or ground negative literals marked as floundered (rules 1.

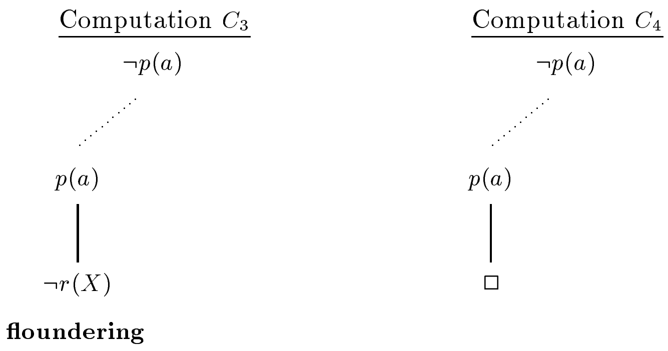

Fig. 2. Some computations for example 2

and 2. being applied) (such a tree is a fully floundered SLDNF-tree), mark $\neg A$ as "floundered" in H and select another literal in H (if that is possible).

We call *extended computation* a computation applying these rules.

Definition 8. A *truly floundered goal* of an extended computation is a goal satisfying:
- it is not in the scope of a negative call (i.e. it belongs to the main derivation).
- each literal in the goal is a non ground negative literal or a ground negative literal marked as floundered.

Extended computation closely corresponds to extended SLDNF-resolution: an extended computation either does not terminate or builds an extended SLDNF-tree. Moreover a terminated extended computation which does not contain truly floundered goals (although it may contain floundered goals) builds an SLDNF-tree (in the sense of [Llo87]). Thus floundered goals which are not truly floundered are irrelevant. In our view floundering should no longer be viewed as an error. Avoiding it may be seen as an optimisation question. The problem is now to build true floundering free programs.

We have shown in Sect. 3 that true floundering is independent of the selection rule. The definition of SLDNF-resolution of [Llo87] (and so our extended one), unlike later re-definitions, has a one-step view of subsidiary derivations. So the selection rule concerns only the main derivation. However Prop. 6 can be applied to terminated extended computations.

Let's call *true floundering answer* (of an extended computation), a pair formed by the goal itself and the computed answer at a truly floundered goal. Let's call *success answer*, the computed answer at a success goal. The following is a corollary of Prop. 6.

Corollary 9. *Let P be a normal program and G a normal goal.*
Two terminated extended computations of $P \bigcup \{G\}$ calculate the same (up to a renaming) successes and true floundering answers.

As long as no floundering occurs, extended computation is as efficient as computation because in this case an extended computation is a computation. Moreover, for some programs and goals, computations may flounder under some selection rules while extended computations terminate and are complete.

5 Related works

Drabent ([Dra94]) gives a notion of floundering called *serious floundering*. He remarks that fairness in SLDNF-resolution should be extended. For a goal $\leftarrow \neg A, \bar{L}$ ($\bar{L}$ being a sequence of literals) selecting $\neg A$ may make it impossible to select any literal of $\bar{L}$ (if A loops). He defines a notion of SLDNF1 pre-failed tree (using a constructive negation formulation) in which, in such a case, derivations for $\leftarrow A$ and $\leftarrow \bar{L}$ both may be built. Serious floundering is defined with SLDNF1 pre-failed trees. Unlike true floundering, serious floundering is defined wrt a selection rule.
In this section we give some comparison results between serious floundering and true floundering. Fairness in the following propositions is related to SLDNF1 pre-failed trees ([Dra94]).

We need the notion of safe selection rule (e.g. [Llo87]) but for technical convenience, we use a slightly modified version:

Definition 10. A selection rule is *safe* if it satisfies:
if a non ground negative literal is selected in a goal then the goal only contains non-ground negative literals.

Proposition 11. *Let P be a normal program and G a normal goal. Let R be a safe and fair selection rule. If $P \bigcup \{G\}$ seriously flounders under R then $P \bigcup \{G\}$ truly flounders.*

The interested reader may refer to the appendix for the proof.

A consequence of this proposition is that a non-truly floundered terminated extended computation (its selection rule R is safe and fair) does not seriously flounder under R. Thus, with the main result of [Dra94] we have (intuitively):

Let P be a normal program and G a normal goal. A non-truly floundered terminated extended computation for $P \bigcup \{G\}$ is correct and complete wrt Kunen semantics.

The reciprocal of Prop. 11 holds (i.e. if $P \bigcup \{G\}$ does not seriously flounder under R then $P \bigcup \{G\}$ does not truly flounder). Thus true floundering and serious floundering are in a sense equivalent. However, true floundering is defined in a simpler way (serious floundering is defined wrt a selection rule).

Martelli and Tricomi [MT92] give a new definition of SLDNF-resolution. They consider extended goals that enclose bracketed subgoals coming from negative calls. They define floundering leaf nodes. This notion of floundering is close to ours but some derivations are considered as floundered while we think they should be failed. True froundering could be obtained with some slight changes. However we think that stating independence of the selection rule would be more difficult in this framework than in ours.

[AB94] (which reproduces Apt and Doets's definition of SLDNF-resolution) gives a declarative definition of floundering. But this one is very weak: it involves a program and a goal floundering if there exists a selection rule for which a computation may flounder. Actually, this notion is directly abstracted from the standard notion of floundering.

Many other papers deal with negation and control but, as far as we know, none attempts to redefine floundering.

From a practical point of view, we need to find conditions under which a given program and a goal may truly flounder or not. In our approach, to state that a program (and a goal) does not truly flounder is simply achieved by building a non truly floundered extended SLDNF-tree. In addition, we have also given an efficient (sketch of) implementation which closely corresponds to extended SLDNF-resolution.

6 Conclusion

Avoiding floundering is a major problem in the context of SLDNF-resolution. However, the notion of floundering does not seem to be a suitable one. Firstly, it is too weak. Secondly, it is a computational notion. There is a lack of a new definition abstracted from computation.

We have proposed a stronger notion of floundering - *true floundering* - with a declarative understanding and an efficient computational behaviour. It seems that true floundering gives a good understanding of what floundering is in SLDNF-resolution.

Some recent works ([AD92], [MT92], [MN90]) explore the problem of giving a complete (in the sense that for a program and a goal there always exists an SLDNF-tree) and suitable re-definition of SLDNF-resolution but do not completely treat floundering. True floundering may be incorporated into these works.

The safeness condition may be weakened ([Llo87], [MN90]). True floundering with a weak safeness condition may be defined within the definition of SLDNF-resolution of [MN90].

An implementation of SLDNF-resolution with true floundering is as efficient as a standard implementation as long as no floundering occurs. In some cases, extended computations (i.e. computations in an implementation of SLDNF-resolution with true floundering) may not truly flounder while some standard computations flounder. Moreover an extended computation which terminates and does not truly flounder is correct and complete wrt Kunen semantics. Thus extended computation is more complete than standard computation.

The class of non true floundering programs is quite large because every program that does not flounder (under some selection rule) does not truly flounder. Moreover it has a declarative characterization. We can expect to build true floundering freeness analysis tools that are simpler and more efficient than floundering freeness analysis tools.

Acknowledgements

The author would like to thank Wlodzimierz Drabent for his comments on a draft version of this paper. Thanks to Pierre Deransart, Gérard Ferrand and Jan Maluszynski for some fruitful discussions. Thanks also to some anonymous referees of a previous version of this paper.

References

[AB94] K. R. Apt, R. Bol. Logic Programming and Negation: A Survey. To appear in Journal of Logic Programming.

[AB91] K. R. Apt, M. Bezem. Acyclic programs. *New generation computing*, 29(3) pages 335-363, 1991.

[AD92] K. R. Apt, K. Doets. A new definition of SLDNF-Resolution. *Report CS-R9242, CWI*, 1992.

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

[BFR93] M. Bergère, G. Ferrand, J-H. Réty. Independence of the Computation R for SLD-Resolution with Delay. *Research Report LIFO 93-9*, Orléans, 1993.

[BM90] R. Barbuti, M. Martelli. Recognizing non-floundering logic programs and goals. International Journal on Foundations of Computer Science, 1(2) pages 151-163, 1990.

[CF+93] M. Codish, M. Falaschi, K. Mariott, W. Winsborough. Efficient Analysis of Concurrent Constraint Logic Programs. *proceedings of ICALP'93*, pages 633-644, 1993.

[Cla79] K. L. Clark. Predicate logic as a computational formalism. *Research Report DOC 79/59*, Imperial college, Dept. of computing, London, 1979.

[DM93] P. Deransart, J. Maluszinski. A Grammatical View of Logic Programming. *MIT Press*, 1993.

[Doe94] K. Doets. From Logic to Logic Programming. *MIT Press*, 1994.

[Dra94] W. Drabent. Completeness of SLDNF-Resolution for Non-Floundering Queries. Draft, 1994.

[Kun87] K. Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4, pp. 289-308, 1987.

[Llo87] J. W. Lloyd. Foundations of Logic Programming. *Springer Verlag, 2nd ed.*, 1987.

[MN90] J. Maluszynski, U. Nilsonn. Logic programming and Prolog. *Wiley*, New-York, 1990.

[MSD90] K. Marriott, H. Sondergaard, P. Dart. A characterization of non-floundering logic programs, in Debray and Hermenegildo ed., Proc. NACLP, pages 661-680, Austin 1990, *MIT Press 1990.*

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

[Nai93] L. Naish. Coroutining and the Construction of terminating logic programs. *Australian Computer Science Communications*, 15 (1), pp. 181-190, 1993.

Appendix

A Proof of Proposition 6

The proof is based on the grammatical view of logic programming[1]. In order to deal with negation we extend skeletons to incorporate negative leaf nodes. We define a subset of the set of skeletons - n-complete skeletons - which is intrinsic to the program (and the goal). We reformulate extended SLDNF-resolution in terms of skeletons. Then the result is proven by the existence of a bijection between the set of n-complete skeletons and the set of non-failure leaf nodes of an extended SLDNF-tree.

A.1 Skeletons with Negation.

From now on a *normal program* P and a *goal* are assumed to be fixed.

It is useful to formalize the *normal program* by a finite family $\{Cr\}_{r\in N}$ where each C_r is a *normal clause* $a_0 \leftarrow a_1, ..., a_n$ (a_i *literals*). The symbol r is the *name* of the clause C_r and n is the *arity* of r.

In the same way the *goal* is formalized by a clause $\leftarrow a_1, \cdots, a_n$ (a_i *literals*) and a symbol g ($g \notin N$) which is the name of the goal, n is the arity of g.

We consider two other symbols, '?' and '$\neg$', with arity 0.

Definition 12. A *(partial) skeleton* (with negation) is a finite oriented tree, labelled by $N \bigcup \{g, \text{'?'}, \text{'}\neg\text{'}\}$, such that:

- the degree of a node (number of children) is the arity of its label,
- the root is the only occurrence of g.

A node of a skeleton is *undefined* if its label is '?'. The other nodes are *defined*. A node is *negative* if its label is '$\neg$'. A node is *proper* if it is defined but not negative.

The set of undefined (resp. defined, proper, negative) nodes of the skeleton s is denoted by $undef(s)$ (resp. $def(s)$, $prop(s)$, $neg(s)$).

The skeleton s is said to be *total* if $undef(s) = \emptyset$.

Remark. A skeleton is completely defined by its defined nodes.

Definition 13. A *renaming* for the skeleton s is a family $(C'_u)_{u\in prop(s)}$ where:

1. if u is not the root and u is labelled by r, C'_u is a *variant* of the clause C_r,
2. if u is the root, C'_u is exactly the *goal* clause $\leftarrow a_1, \cdots, a_n$,
3. if $u \neq v$, C'_u and C'_v have no common variable.

Definition 14. Let s be a skeleton equipped with a renaming $(C'_u)_{u\in prop(s)}$.

For each node u except the root, we define a *literal* a_u and, if moreover $u \in prop(s)$, we also define an *equation* e_u as follows:

Let v be the parent node of u. u is the i^{th} child of v ($1 \leq i \leq arity\ of\ the\ label\ of\ v$).

The literal a_u is the i^{th} member of the body of the clause C'_v.

Moreover if $u \in prop(s)$ (so C'_u is also defined) the equation e_u is $a_u = head\ of\ C'_u$.

[1] P. Deransart, J. Maluszinski. A Grammatical View of Logic Programming. MIT press. 1993.

$(e_u)_{u \in prop(s),\ u\ not\ the\ root}$ is said to be *the system of equations* associated with s and the renaming.

An undefined leaf whose associated literal is a positive(resp. negative) literal is said to be a positive(resp. negative) undefined leaf.

Let s be a skeleton equipped with a renaming and S the associated system of equations. The solvability of S does not depend on the particular renaming (another renaming gives only another variant of the system of equations).

Definition 15. A skeleton is said to be *solvable* if (given any renaming) its system of equations is solvable.

Definition 16. Let s be a solvable skeleton, let σ be the mgu of its system of equations. An *undefined leaf* u of the skeleton s is said to be *suspended* if its associated literal is a negative literal $\neg A$ which satisfies:
σA is not ground, or
σA is ground and there is a truely floundered SLDNF-tree for $P \bigcup \{\leftarrow \sigma A\}$.

We denote by $susp(s)$ the set of the suspended leaves of s.

Definition 17. A *skeleton* s is said to be *n-solvable* if, it is solvable (its system of equations admits an mgu σ) and each literal associated with a negative leaf of s is a negative literal $\neg A_i$ which satisfies:
σA_i is ground, and
there is a finitely failed SLDNF-tree for $P \bigcup \{\leftarrow \sigma A_i\}$.

Note that a n-solvable skeleton is not necessarily total (undefined leaves are possible).

Definition 18. A skeleton is *n-complete* if it is n-solvable and all its undefined leaves are suspended.

Definition 19. Let s, s' be two skeletons. s is an *approximation* of s', denoted by $s \leq s'$, if:

- $def(s) \subseteq def(s')$,
- for each *defined* node u of s, the labels of u in s and in s' are the same.

Note that $\leq$ is a *partial order.*

The following notations will be useful.

- If r is a clause name whose arity is n, we denote by $tree(r)$ the tree in which the root is labelled by r and the other nodes are n leaves labelled by '?'.
- We denote by $tree(\text{'}\neg\text{'})$ the tree in which the root is labelled by '$\neg$' and is the only node.
- If $u \in undef(s)$ and $l \in N \bigcup \{\text{'}\neg\text{'}\}$, we denote by $s[u \leftarrow tree(l)]$ the result of the grafting of $tree(l)$ at the node u in s.
  Note that s is an approximation of $s[u \leftarrow tree(l)]$.

The notion of n-complete skeleton is defined without any reference to a top-down computation. The following technical lemma will be a key result to link, in the next section, this notion to extended SLDNF-Resolution.

Lemma 20. *Let s' be a n-complete skeleton and s a n-solvable approximation of s'.*

1. *If $u \in (def(s') \bigcap undef(s)) - susp(s)$ then $s[u \leftarrow tree(label\ of\ u\ in\ s')]$ is again a n-solvable approximation of s'.*
2. *If $s' \neq s$ then $(def(s') \bigcap undef(s)) - susp(s) \neq \emptyset$*

Sketch of proof: A renaming for s' is fixed. We denote by S' the system of equations associated with s' and this renaming, and by S the restriction of S' to $prop(s)$.

1. $prop(s[u \leftarrow tree(label\ of\ u\ in\ s')]) = prop(s) \bigcup \{u\}$.
   If $label\ of\ u\ in\ s'$ is $\neg$ then let $\neg A$ be associated literal with u (it is negative because s' is n-solvable).
   Let σ be an mgu of S, $\sigma(\neg A)$ is ground because u is not suspended.
   moreover, let μ be an mgu of S', $\sigma A = \mu A$.
   So, s' beeing n-solvable, there exists a finite failure tree for σA.
   Else, Let σ be an mgu of S, σe_u has a solution (else s' could not be n-solvable).
2. Each positive undefined leaf of s belongs to $def(s')$ and is trivially not suspended.
   If there is no positive undefined leaves in s then $s = s'$.
   Let $u \in def(s') \bigcap undef(s)$ (there exists because $s \neq s'$).
   Its associated literal is a negative literal $\neg A$ and , let σ be an mgu of S and μ an mgu of S', $\sigma A = \mu A$. Thus s' being n-solvable, u is not suspended.□

Note that, as a consequence of this lemma, an approximation s of a n-complete skeleton s' with $s \neq s'$ cannot be n-complete.

Such an approximation is not necessarily n-solvable, but the approximation $tree(g)$ of s' is trivially n-solvable because there is no equation and no negative leaves.

A.2 Extended SLDNF-resolution.

We will now give a definition of extended SLDNF-resolution in terms of skeletons (with negation).

In our framework the choice of a literal becomes merely the choice of an undefined leaf of a skeleton.

Definition 21. A *selection function* is a function $\mathcal{F}$ defined for each *skeleton* s which is *n-solvable* but *not n-complete* and such that $\mathcal{F}(s) \in undef(s) - susp(s)$.

A selection function $\mathcal{F}$ being fixed, we define the notion of *derivation* (wrt $\mathcal{F}$) on the n-solvable skeletons :

If s and s' are two n-solvable skeletons, s' *derives* from s if, either, there is a clause name r such that $s' = s[\mathcal{F}(s) \leftarrow tree(r)]$, or $s' = s[\mathcal{F}(s) \leftarrow tree(`\neg')]$.

A *derivation* for the goal g is a (finite or infinite) sequence $s_0, s_1, \cdots$ such that $s_0 = tree(g)$ and s_{i+1} derives from s_i for each i.

A finite derivation $(s_0, \cdots, s_n)$ is said to be *from s_0 to s_n*.

A *terminated* derivation is a finite derivation which cannot be extended. The *terminal* skeleton of a terminated derivation is:

either a n-complete skeleton (total or partial),

either a n-solvable skeleton s which has at least one positive undefined leaf, $\mathcal{F}(s)$ for which, for each clause name r, the skeleton $s[\mathcal{F}(s) \leftarrow tree(r)]$ is not n-solvable.

either a n-solvable skeleton s which has at least one negative undefined leaf $\mathcal{F}(s)$, with associated literal $\neg A$, which is not suspended, and there is an SLDNF refutation for σA (σ being an mgu of the system of equations associated with s).

A *successful* (resp. *truely floundered*) derivation is a terminated derivation to a n-complete skeleton which is total (resp. partial and not total).

A *failed* derivation is a terminated derivation which is neither *successful* nor *truly floundered*.

Extended SLDNF-derivation is expressed by this notion of derivation. The *selection rule* (choice of the selected literal) is obviously specified by the selection function $\mathcal{F}$.

Lemma 22. *The mapping* $(s_0, \cdots, s_n) \mapsto s_n$*, where* $(s_0, \cdots, s_n)$ *is a derivation wrt* $\mathcal{F}$*, (*$\mathcal{F}$ *fixed), is* one-to-one.

Proof. At first, note that if $(s_0, \cdots, s_n)$ is a derivation then each s_i is an approximation of s_n. Note also that if $(s_0, \cdots, s_n, s_{n+1})$ is also a derivation then $s_n \neq s_{n+1}$.

Now, consider two derivations $s_0, s_1, \cdots$ and $s'_0, s'_1 \cdots$ from $s = s_0 = s'_0$ such that $s_i = s'_i$ for $i < k$ but $s_k \neq s'_k$. There is a node of s_k and s'_k (the node $\mathcal{F}(s_{k-1}) = \mathcal{F}(s'_{k-1})$) whose labels are different in s_k and s'_k. So s_k and s'_k cannot be approximation of the same skeleton s_n.□

Our major result is that this mapping is one-to-one and *onto* between *non-failure derivations* and *n-complete skeletons*. It remains to prove:

Theorem 23. *For* each *selection function* $\mathcal{F}$*, for each* n-complete skeleton s*, there is a derivation wrt* $\mathcal{F}$ *from* g *to* s*.*

Proof. Justified by Lemma 20, we can define the following derivation where each s_k is a skeleton which is an approximation of s:

$s_0 = tree(g)$,

if $s_k \neq s$ then $s_{k+1} = s_k[\mathcal{F}(s_k) \leftarrow tree(label\ of\ \mathcal{F}(s_k)\ in\ s)]$

There is a step k for which $s_k = s$. So, we have defined a derivation from $tree(g)$.□

Let us translate this result in terms of extended SLDNF-tree.

From a formal viewpoint the set of the derivations for g is already organized as a *non-oriented* tree as follows:

the *nodes* are the finite derivations,

the *label* of the node $(s_0, \cdots, s_n)$ is the skeleton s_n,

the *parent* of the node $(s_0, \cdots, s_n)$ is the node $(s_0, \cdots, s_{n-1})$ if $n \geq 1$,

the root is $(tree(g)) = (s_0)$.

The *derivations* which are terminated or infinite can be identified with the *branches* of the tree. So there are four kinds of branches: *infinite*, *success* when s_n is n-complete and total, *true floundering* when s_n is n-complete and not total, and *failure*.

This non-oriented tree is the *search tree* associated with the goal g. This search tree expresses the notion of extended SLDNF-tree.

So from a theoretical viewpoint this search tree is an abstract tool for expressing and proving properties where variables, renaming, variants, ... are only bothersome details which are not relevant.

The following corollary is the translation of Theorem 23 in terms of search tree: the independence wrt the selection rule is translated into an independence wrt the search tree.

Corollary 24. *Let $\mathcal{S}$ be the* search tree *defined by a* selection function $\mathcal{F}$. *Let $\phi_{\mathcal{F}}$ be the mapping $l \mapsto s$, where l is a* non-failure leaf *of $\mathcal{S}$ and s is the* n-complete skeleton *which is the label of l in $\mathcal{S}$. $\phi_{\mathcal{F}}$ is* one-to-one and onto.

Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be the two search trees *defined by two* selection functions $\mathcal{F}_1$ *and* $\mathcal{F}_2$. *The mapping $(\phi_{\mathcal{F}_2})^{-1} \circ \phi_{\mathcal{F}_1}$ is* one-to-one and onto *between the non-failure leaves of $\mathcal{S}_1$ and the non-failure leaves of $\mathcal{S}_2$. A non-failure leaf of $\mathcal{S}_1$ and a non-failure leaf of $\mathcal{S}_2$ which are associated by this mapping have the* same label *(*n-complete skeleton*) in $\mathcal{S}_1$ and in $\mathcal{S}_2$.*

In terms of *extended SLDNF-trees*, we can say that two extended SLDNF-trees have, in some sense, (up to a renaming of the variables), the *same success leaves*, (hence the same computed answers, instantiation of the goal), and the *same true floundering leaves.* This independence result comes from the correspondence between these leaves and the n-complete skeletons.

As a consequence of this independence, if some extended SLDNF-tree has no true floundering leaf then no extended SLDNF-tree has a true floundering leaf. In other words, true floundering does not depend on the selection rule.

B Proof (sketch) of Proposition 11

By induction on the depth of serious floundering.
Let B be a normal goal.
- If B depth 0 seriously flounders under R then there exists a non-failure branch of an SLDNF1 pre-failed tree for $P \bigcup \{B\}$ via R such that it is finite and the last goal contains only non-ground negative literals (R is safe).
This branch is a truly floundered extended SLDNF-derivation so $P \bigcup \{B\}$ truly flounders.
If B is ground, there is no SLDNF-refutation (def. of serious floundering). So in this case, there is a fully floundered SLDNF-tree of rank 0.
- If B depth k seriously flounders under R then there exists a non-failure branch of an SLDNF pre-failed tree for $P \bigcup \{B\}$ via R such that it is finite and the last goal contains only non-ground negative literals or ground negative literals $\neg A_i$ such that A_i depth k-1 seriously flounders (R is safe and fair).
By induction hypothesis, for each A_i, there exists a fully floundered SLDNF-tree (of rank k-1) for $\leftarrow A_i$. This branch is a truly floundered extended SLDNF-derivation. So $P \bigcup \{G\}$ truly flounders.
If B is ground, there is no SLDNF-refutation. So, in this case, there is a fully floundered SLDNF-tree of rank k.$\square$