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]), emended 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 \( P \cup \{ G \} \) 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 ([Lio87]), 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.