Pruning infinite failure branches in programs with occur-check

Ulrich Neumerkel
Institut für Computersprachen, Technische Universität Wien
ulrich@mips.complang.tuwien.ac.at

Abstract. We propose an improved execution model for Prolog with occur-check. Prolog's incompleteness is reduced in the case of difference-lists and similar data structures which are crucial for practical programming tasks. We show that adding the occur-check to the Prolog-system is not sufficient for using difference-lists in a reliable manner. In fact, most programs where the occur-check fails end up looping infinitely. By propagating the subterm annotations of variables via abstract interpretation we avoid the creation of these undesirable infinite SLD-trees. In addition, our extension provides a better framework for transformations of programs that use difference-lists that cannot be realized in the common framework of infinite trees.

1 Introduction

It is current practice that Prolog systems omit the occur-check. The standard arguments against the occur-check are efficiency reasons. We have experienced that unification with occur-check [Ro71] does not contribute a lot to the executability of logic descriptions. The reason is that most programs which need occur-checks fail to provide useful solutions because they end up in infinite failure branches. Although they will not yield an unsound solution, this behavior is not very satisfactory from a programmer's point of view. We believe that this may be another reason why the occur-check is not considered important by Prolog implementors. As has been noted by [Llo87] the occur-check is most vital to the correct implementation of difference-list programs. Prolog's powerful notation of difference-lists and similar techniques cannot be used to their full extent due to the lack of a reasonable implementation of the occur-check: In most cases, the programmer has to assume tacitly that the original list is complete. The possibility to deal with unknown data is therefore rather restricted. What a programmer would like to have are well-founded difference-lists: If the length of the difference is known, we want to reason about difference-lists in the same convenient way [Pl91] as we are used to reason about ordinary lists with f length. It should be possible to write a structurally similar program with general difference-lists that behaves in the same way as its restricted counterpart.

In this paper we propose a solution to the problem stated above. Our approach allows the use of difference-lists and similar data structures that are still incomplete but whose length of difference is known. E.g.: [1,2]\L-L is treated in recursive predicates in the same way as [1,2]-\[].

Work on efficient and practical implementations of occur-check is rare. The few contributions are devoted to reduce execution overheads preserving the operational
semantics. A recent approach [De91] circumvents the problem by restricting programs to Not Subject To Occur-check-programs which never need an occur check. Discussion on the earlier approaches can be found there.

2 Pitfalls of the occur-check

A case where the occur-check leads to infinite failure branch is discussed. Consider the following simple recursive predicate. Note that suffixd/2 is the same as suffix/2 in [Ste86] but with arguments exchanged in order to outline that the arguments form a difference-list L0-L.

\[
\text{suffixd}([T|\text{L0}], \text{L}) \leftarrow \text{suffixd}(\text{L0}, \text{L}).
\]

Since in the fact suffixd(L,L) a variable occurs twice the occur-check may fail. The second clause will never fail because of an occur-check since all variables in the head are distinct. Queries to suffixd/2 where the occur-check causes the first clause to fail will therefore construct an infinite failure branch. E.g., the query :- suffixd(L,L) succeeds once and ends in an infinite failure branch on backtracking, trying to resolve at the \((n + 1)\)-st step suffixd(L, [V_1, ..., V_n | L]) and trying to produce the infinite term \([\{V_1, ..., V_n\}]^*\). Note that also infinite trees [Col82] do not help: The system would yield an infinite number of unsound solutions, on backtracking, termination is again a problem. Failing completely cannot be a solution because there may be further correct solutions although the occur-check failed: The query :- suffixd([\text{a}\text{L}], \text{L}) will at first attempt to unify \([\text{a}\text{L}] = \text{L}\) so the occur-check will fail, but will find therein the identity substitution, which is the only correct answer. Again, the predicate will fall into an infinite failure branch since the second clause is always applicable. We now identify the cases in which our example predicate has to fail completely: If the first argument is a variable and the second argument is structured and contains the first one, there is no possibility to remove the occurrence of the first argument within the second via forward recursion: Starting from :- suffixd(L, [\text{a} | \text{L}]), the next resolution step will yield a new subgoal :- suffixd(L1, [\text{a}, \text{-} | L1]). Here L has been substituted: L = [\text{-} | L1]. Note that, in our example this is the only case where our program goes into an infinite loop although sufficient information is provided to fail safely.

3 Subterm annotations

A difference-list is a term (or argument pair) of the form \(Xs - Ys\) denoting difference between the longer list \(Xs\) and the shorter list \(Ys\). Since difference-lists are a pure syntactic convention in Prolog, there are also terms that do not describe a difference-list, eg.: \([\text{-}\{\text{L}\}]\). The implicit restriction on correctly used difference-list \(Xs - Ys\) is that \(Xs\) must not occur in \(Ys\), and if ground, \(Ys\) must occur in \(Xs\). While an incorrect difference-list which consists of a pair of rigid terms [Pl91] poses no problem – a failure will occur anyway – open lists may show up the problem discussed. Since we are interested in performing the occur-checks as early as possible we use the