Detecting Non-Provable Goals*

Stefan Brüning

FG Intellektik, FB Informatik, Technische Hochschule Darmstadt
Alexanderstraße 10, D-64283 Darmstadt (Germany)
Phone: [49](6151)16-6632, FAX: [49](6151)16-5326
E-mail: stebr@intellektik.informatik.th-darmstadt.de

Abstract. In this paper we present a method to detect non-provable
goals. The general idea, adopted from cycle unification, is to determine
in advance how terms may be modified during a derivation. Since a com-
plete predetermination is obviously not possible, we analyze how terms
may be changed by, roughly speaking, adding and deleting function sym-
bols. Such changes of a term are encoded by an efficiently decidable clause
set. The satisfiability of such a set ensures that the goal containing the
term under consideration cannot contribute to a successful derivation.

1 Introduction

To develop proof methods which are as general and adequate\(^2\) as possible is one
of the main goals of research in the field of automated theorem proving. General
methods, like the resolution principle [16] and the connection method [1] are
well known. However, calculi developed from these methods are only adequate
if they are augmented by techniques which control the proof process and reduce
the search space.

In this paper we address this problem by proposing a new technique to aug-
ment top-down backward-chaining calculi — like model elimination [14] or SLD-
resolution [13] — such that goals which cannot contribute to a successful deriva-
tion (i.e. a refutation) are identified. This is clearly of importance since such
goals are a source of infinite looping.

In the field of automated theorem proving, a lot of work deals with the
avoidance of redundancy. Well-known techniques (for example cf. [12]) include
clause subsumption, the tautology principle, and the identical ancestor pruning
rule. In the field of logic programming, various techniques to augment SLD-
resolution based interpreters were examined. Most of them (for example cf. [5,
18]) are based on subsumption tests between (sequences of) goals at runtime.

However, all these mechanisms eliminate logical redundancy, ie. a derivation
is pruned if its success would imply the existence of a hopefully smaller successful
derivation. Non-provable goals which are not redundant in this sense cannot be

\(^*\) This research was supported by the Deutsche Forschungsgemeinschaft (DFG) within
project KONNEKTIONSBEWEISER under grant no. Bi 228/8-2.

\(^2\) We will consider a technique as being adequate if, roughly speaking, it solves simpler
problems faster than more difficult ones [2].
handled with these techniques. For instance, techniques based on subsumption are not applicable — except for the detection of identical goals — if the goals under consideration are ground. To cope with this problem mostly only the purity principle [1] is employed. Pure goals, however, are trivially non-provable because they cannot be used to perform any inference step. Thus, the scope of the purity principle is obviously limited.

The approach we have in mind has its roots in cycle unification [4]. Cycle unification deals with the problem to compute a complete set of answer substitutions for a goal \( \leftarrow P(s_1, \ldots, s_n) \) wrt a logic program consisting of one fact \( P(t_1, \ldots, t_n) \) and a recursive two-literal clause \( P(l_1, \ldots, l_n) \leftarrow P(r_1, \ldots, r_n) \).

For that, the maximal number of required iterations through the cycle is determined by studying variable dependencies. But unfortunately even for this small class of problems the question is undecidable in general if the cycle is unrestricted, i.e. \( l_1, \ldots, l_n, r_1, \ldots, r_n \) are arbitrary terms [10].

However, the general idea underlying cycle unification is appealing: Given a clause set and a goal \( G \), try to predetermine what may happen to \( G \) during a derivation. Since a complete predetermination is not feasible one has to restrict the attention to certain aspects of a derivation.

In this paper we adopt this idea and present a technique which, roughly speaking, analyzes how a term may be changed during a derivation by “adding” and “deleting” function symbols. Using this information, many derivation chains that cannot contribute to a refutation can be pruned.

Example 1. Consider the following clause set where “...” stands for a number of additional arguments. The clauses are numbered for reference:

\[
\begin{align*}
1 & \quad \{ \neg P(a, \ldots) \} \\
2 & \quad \{ P(x, \ldots), \neg Q(g(x), \ldots) \} \\
3 & \quad \{ Q(x, \ldots), \neg P(g(x), \ldots) \} \\
4 & \quad \{ P(x, \ldots), R_1(x, \ldots) \} \\
5 & \quad \{ \neg R_1(g(a), \ldots) \} \\
6 & \quad \{ R_1(x, \ldots), \neg R_2(g(x), \ldots) \} \\
7 & \quad \{ R_2(z, \ldots), \neg R_1(g(x), \ldots) \}
\end{align*}
\]

Suppose a derivation starts by selecting clause (1). Assuming a top-down backward-chaining proof procedure, like model elimination, clauses (2) and (3) can be used to derive a goal of the form \( \neg P(g^{2m}(a), \ldots) \). Applying clause (4) once yields a goal \( G \) of the form \( R_1(g^{2n}(a), \ldots) \). In order to derive the empty clause we may now apply clauses (5), (6), and (7).

Considering the term structure of clauses (6) and (7), one can recognize that each subgoal of \( G \) containing the predicate symbol \( R_1 \) is of the form \( R_1(g^{2m}(a), \ldots) \) with \( m < n \). Therefore clause (5) can never be applied to derive the empty clause.

Thus, just by this simple analysis it is possible to decide that a successful derivation starting with clause (1) does not exist. Theorem provers like SETHEO [12] or PTTP [19] do not apply such a kind of analysis and will loop infinitely.

A first method to analyze possible changes of terms in a corresponding way was presented in [7]. The idea is to represent such changes as a formal language which is encoded by a corresponding acceptor. The proposed technique works on arbitrary clause sets and enables to prune derivations where other techniques