Deciding Innermost Loops

René Thiemann\(^1\), Jürgen Giesl\(^2\), and Peter Schneider-Kamp\(^2\)

\(^1\) Institute of Computer Science, University of Innsbruck, Austria
rene.thiemann@uibk.ac.at
\(^2\) LuFG Informatik 2, RWTH Aachen University, Germany
\{giesl,psk\}@informatik.rwth-aachen.de

Abstract. We present the first method to disprove innermost termination of term rewrite systems automatically. To this end, we first develop a suitable notion of an innermost loop. Second, we show how to detect innermost loops: One can start with any technique amenable to find loops. Then our novel procedure can be applied to decide whether a given loop is an innermost loop. We implemented and successfully evaluated our method in the termination prover AProVE.

1 Introduction

Termination is an important property of term rewrite systems (TRSs). Therefore, much effort has been spent on developing and automating powerful techniques for showing (innermost) termination of TRSs. An important application area for these techniques is termination analysis of functional programs. Since the evaluation mechanism of functional languages is mainly term rewriting, one can transform functional programs into TRSs and prove termination of the resulting TRSs to conclude termination of the functional programs \cite{9}. Although “full” rewriting does not impose any evaluation strategy, this approach is sound even if the underlying programming language has an innermost evaluation strategy.

But in order to detect bugs in programs, it is at least as important to prove non-termination of programs or of the corresponding TRSs. Here, the evaluation strategy cannot be ignored, because a non-terminating TRS may still be innermost terminating. Thus, in order to disprove termination of programming languages with an innermost strategy, it is important to develop techniques to disprove innermost termination of TRSs automatically.

Only a few techniques for showing non-termination of TRSs have been introduced so far \cite{12,17,18,20}. Nevertheless, there already exist several tools that are able to prove non-termination of TRSs automatically by finding loops (e.g., AProVE \cite{8}, Jambox \cite{5}, Matchbox \cite{23}, NTI \cite{20}, TORPA \cite{21}, TTT \cite{14}). But up to now, all of these techniques and tools only disprove full and not innermost termination. So they can only be applied to disprove innermost termination if the TRS belongs to a known class where termination and innermost termination coincide \cite{11}. In this paper, we demonstrate how to extend all of these techniques such that they can be directly used for disproving innermost termination for any

\* Supported by the Deutsche Forschungsgemeinschaft (DFG) under grant GI 274/5-2.
kind of TRS. For instance, this is needed for the following program where the resulting TRS is not confluent and hence, does not fall into a known class where innermost and full termination are the same.

Example 1 (Factorial function). The following ACL2 program computes the factorial function where \(x\) is increased from 0 to \(y-1\) and in every iteration the result is multiplied by \(1+x\).

\[
\begin{align*}
(\text{defun factorial } (y) & (\text{fact } 0 \ y)) \\
(\text{defun fact } (x \ y) & (\text{if } (= \ x \ y) \\
& 1 \\
& (\times (+ 1 \ x) \ (\text{fact } (+ 1 \ x) \ y))))
\end{align*}
\]

Using a translation to TRSs suggested by \[22\], we obtain the following TRS \(\mathcal{R}\) where the rules \([5] - [12]\) are needed to handle the built-in functions of ACL2.

\[
\begin{align*}
\text{factorial}(y) & \rightarrow \text{fact}(0, y) & (1) & 0 \times y \rightarrow 0 & (7) \\
\text{fact}(x, y) & \rightarrow \text{if}(x == y, x, y) & (2) & \text{suc}(x) \times y \rightarrow y + (x \times y) & (8) \\
\text{if}(\text{true}, x, y) & \rightarrow \text{suc}(0) & (3) & x == y \rightarrow \text{chk(eq}(x, y)) & (9) \\
\text{if}(\text{false}, x, y) & \rightarrow \text{suc}(x) \times \text{fact(suc}(x), y) & (4) & \text{eq}(x, x) \rightarrow \text{true} & (10) \\
& 0 + y \rightarrow y & (5) & \text{chk(true)} \rightarrow \text{true} & (11) \\
\text{suc}(x) + y & \rightarrow \text{suc}(x + y) & (6) & \text{chk(eq}(x, y)) \rightarrow \text{false} & (12)
\end{align*}
\]

Here, it is crucial to use innermost instead of full rewriting. Otherwise, it would always be possible to rewrite \(s == t \rightarrow_{\mathcal{R}} \text{chk(eq}(s, t)) \rightarrow_{\mathcal{R}} \text{false}\), i.e., terms like \(0 == 0\) could then be evaluated to both \text{true} and \text{false}. In contrast, for innermost rewriting one has to apply rule \([10]\) first if \(s\) and \(t\) are equal.

Note that in this TRS, \(s == t\) is indeed evaluated to \text{false} whenever \(s\) and \(t\) are any terms that are syntactically different. This is essential to model the semantics of ACL2 correctly, since here there are -- like in term rewriting -- no types. At the same time, all functions in ACL2 must be “completely defined”.

So to perform non-termination proofs for languages like ACL2, we need a way to disprove innermost termination. This problem is harder than disproving termination since one has to take care of the evaluation strategy.

In this paper we investigate looping reductions. These are specific kinds of infinite reductions which can be represented in a finite way. To disprove innermost termination of TRSs, we develop an automatic method which in case of success, presents the innermost loop to the user as a counterexample.

For the TRS of Ex. \[1\] there is indeed an innermost loop. It corresponds to the non-terminating reduction of the ACL2 program when calling \text{fact}(n, m)\) for natural numbers \(n > m\). The reason is that the first argument is increased over and over again, and it will never become equal to \(m\).

The paper is organized as follows. In Sect.\[2\] we extend the notion of a loop to innermost rewriting. Then as the main contribution of the paper, we describe a