nonreach – A Tool for Nonreachability Analysis*

Florian Meßner and Christian Sternagel[0000−0001−9864−1014]

University of Innsbruck, Innsbruck, Austria
{florian.g.messner,christian.sternagel}@uibk.ac.at

Abstract. We introduce nonreach, an automated tool for nonreachability analysis that is intended as a drop-in addition to existing termination and confluence tools for term rewriting. Our preliminary experimental data suggests that nonreach can improve the performance of existing termination tools.

Keywords: Term rewriting · Nonreachability analysis · Narrowing · Termination · Confluence · Infeasibility.

1 Introduction

Nonreachability analysis is an important part of automated tools like TTT2 [2] (for proving termination of rewrite systems) and ConCon [4] (for proving confluence of conditional rewrite systems). Many similar systems compete against each other in the annual termination (TermComp)[1] and confluence (CoCo)[2] competitions, both of which will run as part of TACAS's TOOLympics[3] in 2019.

Our intention for nonreach is to become a valuable component of all of the above mentioned tools by providing a fast and powerful back end for reachability analysis. This kind of analysis is illustrated by the following example.

Example 1. Suppose we have a simple program for multiplication represented by a term rewrite system (TRS, for short) consisting of the following rules:

$$\mathsf{add}(\mathsf{0}, y) \to y \qquad \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y))$$
$$\mathsf{mul}(\mathsf{0}, y) \to \mathsf{0} \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(\mathsf{mul}(x, y), y)$$

For checking termination we have to make sure that there is no infinite sequence of recursive calls. One specific subproblem for doing so is to check whether it is possible to reach $t = \mathsf{MUL}(\mathsf{s}(y), z)$[4] from $s = \mathsf{ADD}(\mathsf{mul}(w, x), x)$ for arbitrary instantiations of the variables w, x, y, and z. In other words, we have to check *nonreachability* of the target t from the source s.

* This work is supported by the Austrian Science Fund (FWF): project P27502.

1 http://termination-portal.org/wiki/Termination_Competition

2 http://project-coco.uibk.ac.at/

3 https://tacas.info/toolympics.php

4 We differentiate between recursive calls and normal argument evaluation by capitalization of function symbols.

In the remainder we will: comment on the role we intend for nonreach (Section 2), describe how nonreach is built and used (Section 3), give an overview of the techniques that went into nonreach (Section 4), and finally provide some experimental data (Section 5).

2 Role

When looking into implementations of current termination and confluence tools it soon becomes apparent that many tools use the same techniques for proving nonreachability. In light of this observation, one of our main goals for nonreach was to provide a dedicated stand-alone tool for nonreachability that can be reused for example in termination and confluence tools and can in principle replace many existing implementations.

In order to make such a reuse desirable for authors of other tools two things are important: (1) we have to provide a simple but efficient interface, and (2) we should support all existing techniques that can be implemented efficiently.[5]

At the time of writing, we already successfully use nonreach as back end in the termination tool $\mathsf{T_TT_2}$ [2]. To this end, we incorporated support for external nonreachability tools into $\mathsf{T_TT_2}$, with interaction purely via standard input and output. More specifically, while at the moment only YES/NO/MAYBE answers are required, the interface is general enough to support more detailed certificates corroborating such answers. The external tool is launched once per termination proof and supplied with those nonreachability problems that could not be handled by the existing techniques of $\mathsf{T_TT_2}$. Our next goal is to achieve the same for the confluence tool ConCon [4].

Furthermore, a new *infeasibility* category will be part of CoCo 2019.[6] Infeasibility is a concept from conditional term rewriting but can be seen as a variant of nonreachability [3]. Thus, we plan to enter the competition with nonreach.

Another potential application of nonreach is dead code detection or showing that some error can never occur.

3 Installation and Usage

Our tool nonreach is available from a public *bitbucket*[7] repository which can be obtained using the following command:

```
git clone git@bitbucket.org:fmessner/nonreach.git
```

To compile and run nonreach, you need an up to date installation of *Haskell's stack*.[8] The source code is compiled by invoking `stack build` in the project

[5] Efficiency is especially important under the consideration that, for example, termination tools may sometimes have to check thousands of nonreachability problems within a single termination proof.

[6] http://project-coco.uibk.ac.at/2019/categories/infeasibility.php

[7] https://bitbucket.org/fmessner/nonreach

[8] https://docs.haskellstack.org/en/stable/README

directory containing the `stack.yaml` file. In order to install the executable in the local `bin` path (`~/.local/bin/` on Linux), run `stack install` instead.

Usage. The execution of nonreach is controlled by several command line flags. The only mandatory part is a rewrite system (with respect to which nonreachability should be checked). This may be passed either as literal string (flag `-d "..."`) or as file (flag `-f filename`). Either way, the input follows the formats for (conditional) rewrite systems that are used for TermComp and CoCo.

In addition to a rewrite system we may provide the nonreachability problems to be worked on (if we do not provide any problems, nonreach will wait indefinitely). For a single nonreachability problem the simple format `s -> t` is used, where s and t are terms and we are interested in nonreachability of the target t from the source s. Again, there are several ways to pass problems to nonreach:

- We can provide white-space-separated lists of problems either literally on the command line (flag `-P "..."`) or through a file (flag `-p filename`).
- Alternatively, a single infeasibility problem can be provided as part of the input rewrite system as specified by the new infeasibility category of CoCo 2019.
- Otherwise, nonreach waits for individual problems on standard input.

For each given problem nonreach produces one line of output: In its default mode the output is `NO` whenever nonreachability can be established and either `MAYBE` or `TIMEOUT`, otherwise. When given an infeasibility problem, the output is `YES` if the problem is infeasible and either `MAYBE` or `TIMEOUT`, otherwise.

Further flags may be used to specify a timeout (in microseconds; flag `-t`) or to give finer control over the individual techniques that are implemented in nonreach (we will mention those in Section 4).

It is high time for an example. Let us check Example 1 using nonreach.

Example 2. Assuming that the TRS of Example 1 is in a file `mul.trs` we can have the following interaction (where we indicate user input by a preceding `>`):

```
nonreach -f mul.trs
> ADD(mul(w,x),x) -> MUL(s(y),z)
NO
```

4 Design and Techniques

In this section we give a short overview of the general design decisions and specific nonreachability techniques that went into nonreach.

Design. Efficiency was at the heart of our concern. On the one hand, from a user-interface perspective, this was the reason to provide the possibility that a single invocation of nonreach for a fixed TRS can work on arbitrarily many reachability problems. On the other hand, this lead us to mostly concentrate on techniques that are known to be fast in practice. The selection of techniques we present below (with the exception of narrowing) satisfies this criterion.

Techniques. Roughly speaking, nonreach uses two different kinds of techniques: (1) *transformations* that result in disjunctions or conjunctions of easier nonreachability problems, and (2) actual *nonreachability checks*. We use the notation $s \twoheadrightarrow t$ for a nonreachability problem with source s and target t.

Reachability checks. The first check we recapitulate is implemented by most termination tools and based on the idea of computing the topmost part of a term that does not change under rewriting (its *cap*) [1]. If the cap of the source s does not unify with the target t, then there are no substitutions σ and τ such that $s\sigma \to^* t\tau$. There are different algorithms to underapproximate such caps. We use etcap, developed by Thiemann together with the second author [5], due to its linear time complexity (by reducing unification to *ground context matching*) but nevertheless simple implementation. With etcap subterms are matched bottom-up with left-hand sides of rules. In case of a match, the corresponding subterm is potentially rewritable and thus replaced by a *hole*, written $\square$, representing a fresh variable. We illustrate etcap on the first two (addition) rules of Example 1.

Example 3. We have $\mathsf{etcap}(\mathsf{s}(\mathsf{0})) = \mathsf{s}(\mathsf{0})$ and $\mathsf{etcap}(\mathsf{s}(\mathsf{add}(\mathsf{s}(z), \mathsf{s}(\mathsf{0})))) = \mathsf{s}(\square)$, since the subterm headed by add matches the second addition rule. Using etcap, we obtain the following nonreachability check

$$\mathsf{reach}_{\mathsf{etcap}}(s \twoheadrightarrow t) = \begin{cases} \mathtt{MAYBE} & \text{if } \mathsf{etcap}(s) \sim t \\ \mathtt{NO} & \text{otherwise} \end{cases}$$

where $\sim$ denotes unifiability of terms.

The second reachability check implemented in nonreach [6,3] is based on the so called *symbol transition graph* (SG for short) of a TRS. Here, the basic idea is to build a graph that encodes the dependencies between root symbols induced by the rules of a TRS. This is illustrated by the following example:

Example 4. Given the TRS $\mathcal{R}$ consisting of the four rules

$$\mathsf{f}(x, x) \to \mathsf{g}(x) \qquad \mathsf{g}(x) \to \mathsf{a} \qquad \mathsf{h}(\mathsf{a}) \to \mathsf{b} \qquad \mathsf{h}(x) \to x$$

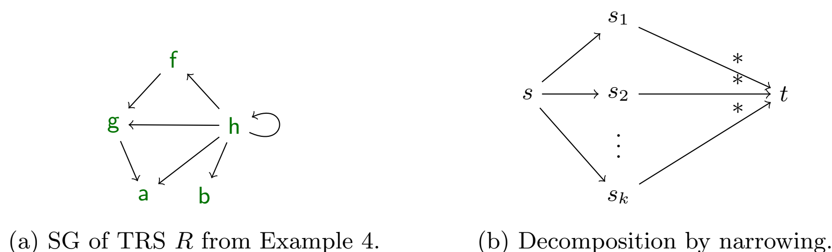
we generate the corresponding SG shown in Figure 1(a) on page 5.

For each rule we generate an edge from the node representing the root symbol of the left-hand side to the node representing the root symbol of the right-hand side. Since in the last rule, the right-hand side is a variable (which can in principle be turned into an arbitrary term by applying a substitution), we have to add edges from h to all nodes (including h itself).

From the graph it is apparent that $\mathsf{f}(x, y)$ is not reachable from $\mathsf{g}(z)$, no matter the substitutions for x, y and z, since there is no path from g to f.

We obtain the following SG based nonreachability check:

$$\mathsf{reach}_{\mathsf{stg}}(s \twoheadrightarrow t) = \begin{cases} \mathtt{MAYBE} & \text{if there is a path from } \mathsf{root}(s) \text{ to } \mathsf{root}(t) \text{ in the graph} \\ \mathtt{NO} & \text{otherwise} \end{cases}$$



(a) SG of TRS R from Example 4.

(b) Decomposition by narrowing.

Fig. 1: A symbol transition graph and the idea of narrowing.

Which reachability checks are applied by nonreach can be controlled by its `-s` flag, which expects a list of checks (in Haskell syntax). Currently the two checks `TCAP` and `STG` are supported.

Transformations. We call the first of our transformations *(problem) decomposition.* This technique relies on root-reachability checks to decompose a problem into a *conjunction* of strictly smaller subproblems. Thus, we are done if any of these subproblems is nonreachable.

Decomposition of a problem $s \twoheadrightarrow t$ only works if source and target are of the form $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$, respectively. Now the question is whether t can be reached from s involving at least one root-rewrite step. If this is not possible, we know that for reachability from s to t, we need all t_i to be reachable from the corresponding s_i.

For the purpose of checking root-nonreachability, we adapt the two reachability checks from above:

$$\mathsf{rootreach}_{\mathsf{etcap}}(s,t) = \begin{cases} \texttt{True} & \text{if there is a rule } \ell \to r \text{ such that } \mathsf{etcap}(s) \sim \ell \\ \texttt{False} & \text{otherwise} \end{cases}$$

$$\mathsf{rootreach}_{\mathsf{stg}}(s,t) = \begin{cases} \texttt{True} & \text{if there is a non-empty path from } \mathsf{root}(s) \\ & \text{to } \mathsf{root}(t) \text{ in the SG} \\ \texttt{False} & \text{otherwise} \end{cases}$$

If at least one of these checks returns `False`, we can decompose the initial problem $s \twoheadrightarrow t$ into a conjunction of problems $s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$.

The second transformation is based on *narrowing*. Without going into too much technical detail let us start from the following consideration. Given a reachability problem $s \twoheadrightarrow t$, assume that t is reachable from s. Then either the corresponding rewrite sequence is empty (in which case s and t are unifiable) or there is at least one initial rewrite step. Narrowing is the tool that allows us to capture all possible first steps (one for each rule $l \to r$ and each subterm of s that unifies with l), of the form $s \to s_i \to^* t$. This idea is captured in Figure 1(b).

Now, decomposition (which is always applicable and thus has to be handled with care), transforms a given reachability problem $s \twoheadrightarrow t$ into a disjunction of

new problems (that is, this time we have to show NO for all of these problems in order to conclude NO for the initial one) $s \stackrel{?}{\sim} t$ or $s_1 \twoheadrightarrow t$ or $\ldots$ or $s_k \twoheadrightarrow t$, where the first one is a unifiability problem and the remaining ones are again reachability problems.

The maximal number of narrowing applications is specified by the `-l` flag.

5 Experiments

In order to obtain test data for nonreach we started from the *Termination Problem Data Base* (TPDB, for short),[9] the data base of problems that are used also in TermComp. Then we used a patched version of the termination tool TTT2 to obtain roughly 20,000 non-trivial reachability problems with respect to 730 TRSs. These are available from a public *bitbucket* repository[10] alongside a small script, that runs nonreach on all the problems and displays the overall results. All these problems could, at the time of creating the dataset, not be solved with the reachability checks used in the competition strategy of TTT2. The current version[11] of nonreach in its default configuration can prove nonreachability of 369 problems of this set. While this does not seem much, we strongly suspect that the majority of problems in our set are actually reachable.

References

1. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and Disproving Termination of Higher-Order Functions. In: Proc. 5th FroCoS. LNAI, vol. 3717, pp. 216–231. Springer (2005). doi:10.1007/11559306_12
2. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: Proc. 20th RTA. LNCS, vol. 5595, pp. 295–304. Springer (2009). doi:10.1007/978-3-642-02348-4_21
3. Sternagel, C., Yamada, A.: Reachability analysis for termination and confluence of rewriting. In: Proc. 25th TACAS (2019), submitted
4. Sternagel, T., Middeldorp, A.: Conditional confluence (system description). In: Proc. Joint 25th RTA and 12th TLCA. LNCS, vol. 8560, pp. 456–465. Springer (2014). doi:10.1145/2676724.2693171
5. Thiemann, R., Sternagel, C.: Certification of Termination Proofs using CeTA. In: Proc. 22nd International Conference on Theorem Proving in Higher Order Logics. LNCS, vol. 5674, pp. 452–468. Springer (2009). doi:10.1007/978-3-642-03359-9_31
6. Yamada, A.: Reachability for termination. In: 4th AJSW (2016), http://cl-informatik.uibk.ac.at/users/ayamada/AJSW2016-slides.pdf

[9] http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB
[10] https://bitbucket.org/fmessner/nonreach-testdata
[11] https://bitbucket.org/fmessner/nonreach/src/77945a5/?at=master