Decision Procedures for Loop Detection*

René Thiemann[1], Jürgen Giesl[2], Peter Schneider-Kamp[2]

[1] Institute of Computer Science, University of Innsbruck
Techniker Str. 21a, A-6020 Innsbruck, Austria
rene.thiemann@uibk.ac.at
[2] LuFG Informatik 2, RWTH Aachen
Ahornstr. 55, 52074 Aachen, Germany
{giesl,psk}@informatik.rwth-aachen.de

Abstract. The dependency pair technique is a powerful modular method for automated termination proofs of term rewrite systems. We first show that dependency pairs are also suitable for disproving termination: loops can be detected more easily.
In a second step we analyze how to disprove innermost termination. Here, we present a novel procedure to decide whether a given loop is an innermost loop.
All results have been implemented in the termination prover AProVE.

Keywords. Non-Termination, Decision Procedures, Term Rewriting, Dependency Pairs

1 Introduction

One of the most powerful techniques to prove termination or innermost termination of TRSs automatically is the *dependency pair approach* [1,3]. In [4,7], we showed that dependency pairs can be used as a general framework to combine arbitrary techniques for termination analysis in a modular way. The general idea of this framework is to solve termination problems by repeatedly decomposing them into sub-problems. We call this new concept the "dependency pair *framework*" ("DP framework") to distinguish it from the old "dependency pair *approach*". In particular, this framework also facilitates the development of new methods for termination and *non-termination* analysis.

Example 1 (Factorial function). The following ACL2 program computes the factorial function.

$$\begin{array}{l}(\mathit{defun}\ \mathsf{factorial}\ (x)\ (\mathsf{fact}\ 0\ x))\\(\mathit{defun}\ \mathsf{fact}\ (x\ y)\ (\mathsf{if}\ (==\ x\ y)\\\quad 1\\\quad (\times\ (\mathsf{fact}\ (+\ 1\ x)\ y)\ (+\ 1\ x))))\end{array}$$

Using the translation of [15] we obtain the following TRS $\mathcal{R}$ where the rules (5) – (12) are needed to handle the built-in functions of ACL2.

* Supported by the DFG grant GI 274/5-1.

$$\mathsf{factorial}(x) \to \mathsf{fact}(\mathsf{0}, x) \tag{1}$$
$$\mathsf{fact}(x, y) \to \mathsf{if}(x == y, x, y) \tag{2}$$
$$\mathsf{if}(\mathsf{true}, x, y) \to \mathsf{s}(\mathsf{0}) \tag{3}$$
$$\mathsf{if}(\mathsf{false}, x, y) \to \mathsf{fact}(\mathsf{s}(x), y) \times \mathsf{s}(x) \tag{4}$$
$$\mathsf{0} + y \to y \tag{5}$$
$$\mathsf{s}(x) + y \to \mathsf{s}(x + y) \tag{6}$$
$$\mathsf{0} \times y \to \mathsf{0} \tag{7}$$
$$\mathsf{s}(x) \times y \to y + (x \times y) \tag{8}$$
$$x == y \to \mathsf{chk}(\mathsf{eq}(x, y)) \tag{9}$$
$$\mathsf{eq}(x, x) \to \mathsf{true} \tag{10}$$
$$\mathsf{chk}(\mathsf{true}) \to \mathsf{true} \tag{11}$$
$$\mathsf{chk}(\mathsf{eq}(x, y)) \to \mathsf{false} \tag{12}$$

Note that the rules for equality can compare terms of arbitrary "type", e.g., it will be detected that false is not equal to 0. This is essential to model the semantics of ACL2, since here there are – like in term rewriting – no types. At the same time, all functions in ACL2 must be "completely defined".

Since in ACL2 every function must be terminating to guarantee consistency, it is essential to know whether a termination proof cannot be obtained due to non-termination. The goal is then to identify and present the infinite sequence to the user such that the bug can be fixed. Here, we will investigate a specific kind of infinite reductions which can be represented in a finite way: a looping reduction proves non-termination.

It is not only a hard problem to find a loop due to the huge associated search space, but with the translated ACL2-programs we encounter a second problem. The rules of equality of $\mathcal{R}$ do not work correctly for standard rewriting since it is always possible to rewrite $s == t \to_{\mathcal{R}} \mathsf{chk}(\mathsf{eq}(s, t)) \to_{\mathcal{R}} \mathsf{false}$, i.e., all terms are not equal, and some terms are equal and not equal at the same time. As this obviously does not correspond to the ACL2 semantics, one cannot conclude non-termination of the ACL2-program from non-termination of $\mathcal{R}$.

This is not the case for innermost rewriting since there the equality rules work correctly. One has to first apply rule (10) if s and t are equal. And indeed, $\mathcal{R}$ is an innermost confluent TRS which models the semantics of equality in a correct way. But then we need a way to disprove innermost termination, a problem which is harder than disproving termination since one has to take care of the evaluation strategy.

However, in this paper we will present a technique which will disprove innermost termination of $\mathcal{R}$ and present an *innermost loop* to the user as counterexample. This loop corresponds to the non-terminating reduction of the ACL2 program which does not terminate if one calls $\mathsf{fact}(n, 0)$ for a natural number $n > 0$. The reason is that the first argument is increased over and over again, and it will never become equal to 0.

The paper is organized as follows. After recapitulating the basics of the DP framework in Sect. 2, we present two significant improvements: in Sect. 3 we show how to use the DP framework to prove *non-termination* with the help of loops and in Sect. 4 we present a way to detect loops for standard rewriting. Then in Sect. 5 we extend the notion of a loop to innermost rewriting, and describe a novel decision procedure in Sect. 6 which detects whether a loop for standard rewriting is still a loop in the innermost case.[3] Sect. 7 contains related work and summarizes our results.

2 The Dependency Pair Framework

We refer to [2] for the basics of rewriting and to [1,3,4,7] for motivations and details on dependency pairs. We only regard finite signatures and TRSs. $\mathcal{T}(\mathcal{F},\mathcal{V})$ is the set of terms over the signature $\mathcal{F}$ and the infinite set of variables $\mathcal{V} = \{x, y, z, \ldots, \alpha, \beta, \ldots\}$. A *TRS* $\mathcal{R}$ is a (finite) set of rules $\ell \to r$. We use the notation $\{x/t_1, \ldots, x_n/t_n\}$ for the substitution which replaces every variable x_i by the corresponding term t_i.

For a TRS $\mathcal{R}$ over $\mathcal{F}$, the *defined symbols* are $\mathcal{D} = \{\mathrm{root}(l) \mid l \to r \in \mathcal{R}\}$. For every $\mathsf{f} \in \mathcal{F}$ let $\mathsf{f}^\sharp$ be a fresh *tuple symbol* with the same arity as f. The set of tuple symbols is denoted by $\mathcal{F}^\sharp$. If $t = \mathsf{g}(t_1, \ldots, t_m)$ with $\mathsf{g} \in \mathcal{D}$, we let $t^\sharp$ denote $\mathsf{g}^\sharp(t_1, \ldots, t_m)$.

Definition 2 (Dependency Pair). *The set of* dependency pairs *for a TRS* $\mathcal{R}$ *is* $DP(\mathcal{R}) = \{l^\sharp \to t^\sharp \mid l \to r \in \mathcal{R}, t \text{ is a subterm of } r, \mathrm{root}(t) \in \mathcal{D}\}$.

Example 3. In the TRS of Ex. 1, the defined symbols are $\mathsf{factorial}$, fact, if, $+$, $\times$, $==$, and chk. Hence, the dependency pairs of the TRS are the following ones.

$$\mathsf{factorial}^\sharp(x) \to \mathsf{fact}^\sharp(\mathsf{0}, x) \tag{13}$$
$$\mathsf{fact}^\sharp(x, y) \to \mathsf{if}^\sharp(x == y, x, y) \tag{14}$$
$$\mathsf{fact}^\sharp(x, y) \to x ==^\sharp y \tag{15}$$
$$\mathsf{if}^\sharp(\mathsf{false}, x, y) \to \mathsf{fact}(\mathsf{s}(x), y) \times^\sharp \mathsf{s}(x) \tag{16}$$
$$\mathsf{if}^\sharp(\mathsf{false}, x, y) \to \mathsf{fact}^\sharp(\mathsf{s}(x), y) \tag{17}$$
$$\mathsf{s}(x) +^\sharp y \to x +^\sharp y \tag{18}$$
$$\mathsf{s}(x) \times^\sharp y \to y +^\sharp (x \times y) \tag{19}$$
$$\mathsf{s}(x) \times^\sharp y \to x \times^\sharp y \tag{20}$$
$$x ==^\sharp y \to \mathsf{chk}^\sharp(\mathsf{eq}(x, y)) \tag{21}$$
$$x ==^\sharp y \to \mathsf{eq}^\sharp(x, y) \tag{22}$$

For termination, we try to prove that there are no infinite *chains* of dependency pairs. Intuitively, a dependency pair corresponds to a function call and a chain represents a possible sequence of calls that can occur during a reduction. In the following definition, $\mathcal{P}$ is usually a set of dependency pairs.

[3] All results described in this draft including the proofs are available in [5] and [14].

Definition 4 (Chain). *Let $\mathcal{P}, \mathcal{R}$ be TRSs. A (possibly infinite)* $(\mathcal{P}, \mathcal{R})$-chain *is a reduction of the form*

$$s_1 \to_{\mathcal{P}} s_2 \to_{\mathcal{R}}^* s_3 \to_{\mathcal{P}} s_4 \to_{\mathcal{R}}^* \dots$$

where $\mathcal{P}$-steps are only allowed at the root. It is an innermost *$(\mathcal{P}, \mathcal{R})$-chain iff*

$$s_1 \xrightarrow{\mathsf{i}}_{\mathcal{P}} s_2 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^* s_3 \xrightarrow{\mathsf{i}}_{\mathcal{P}} s_4 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^* \dots$$

where again $\mathcal{P}$-steps are only allowed at the root. Here, "$\xrightarrow{\mathsf{i}}$" denotes innermost reductions w.r.t. $\mathcal{R}$, i.e., all direct subterms of the redex have to be in $\mathcal{R}$-normal form.

Example 5. One can use the dependency pairs (14) and (17) of Ex. 3 to build a chain:

$$\begin{array}{lll} \mathsf{fact}^\sharp(0, \mathsf{s}(0)) & \xrightarrow{\mathsf{i}}_{\{(14)\}} & \mathsf{if}^\sharp(0 == \mathsf{s}(0), 0, \mathsf{s}(0)) \\ & \xrightarrow{\mathsf{i}}_{\mathcal{R}}^* & \mathsf{if}^\sharp(\mathsf{false}, 0, \mathsf{s}(0)) \\ & \xrightarrow{\mathsf{i}}_{\{(17)\}} & \mathsf{fact}^\sharp(\mathsf{s}(0), \mathsf{s}(0)) \end{array}$$

The following theorem states that an infinite reduction corresponds to an infinite chain and vice versa.

Theorem 6 (Termination Criterion [1]). *A TRS $\mathcal{R}$ is (innermost) non-terminating iff there is an infinite (innermost) $(DP(\mathcal{R}), \mathcal{R})$-chain.*

The idea of the DP framework [4] is to treat a set of dependency pairs $\mathcal{P}$ together with the TRS $\mathcal{R}$ and to prove absence of infinite $(\mathcal{P}, \mathcal{R})$-chains instead of examining $\to_{\mathcal{R}}$. Formally, a *dependency pair problem* ("DP problem") consists of two TRSs $\mathcal{P}$ and $\mathcal{R}$ (where initially, $\mathcal{P} = DP(\mathcal{R})$) and a flag $e \in \{\mathbf{t}, \mathbf{i}\}$ standing for "**t**ermination" or "**i**nnermost termination". Instead of "$(\mathcal{P}, \mathcal{R})$-chains" we also speak of "$(\mathcal{P}, \mathcal{R}, \mathbf{t})$-chains" and instead of "innermost $(\mathcal{P}, \mathcal{R})$-chains" we speak of "$(\mathcal{P}, \mathcal{R}, \mathbf{i})$-chains". Our goal is to show that there is an infinite $(\mathcal{P}, \mathcal{R}, e)$-chain. In this case, we call the problem *infinite*, otherwise it is finite.[4]

Termination techniques should now operate on DP problems instead of TRSs. We refer to such techniques as *dependency pair processors* ("DP processors"). Formally, a DP processor is a function *Proc* which takes a DP problem as input and returns a new set of DP problems which then have to be solved instead. Alternatively, it can also return "no". A DP processor *Proc* is *sound* if for all DP problems d, d is finite whenever $Proc(d)$ is not "no" and all DP problems in $Proc(d)$ are finite. *Proc* is *complete* if for all DP problems d, d is infinite whenever $Proc(d)$ is "no" or when $Proc(d)$ contains an infinite DP problem.

Soundness of a DP processor *Proc* is required to prove termination (in particular, to conclude that d is finite if $Proc(d) = \emptyset$). Completeness is needed to prove non-termination (in particular, to conclude that d is infinite if $Proc(d) = \mathsf{no}$).

[4] To ease readability we use a simpler definition of *DP problems* than [4,7,14], since this simple definition suffices for the new results of this paper. Moreover, we also use a simplified notion of *infinite DP problems*.

So termination proofs in the DP framework start with the initial DP problem $(DP(\mathcal{R}), \mathcal{R}, e)$, where e depends on whether one wants to prove termination or innermost termination. Then this problem is transformed repeatedly by sound DP processors. If the final processors return empty sets of DP problems, then termination is proved. If one of the processors returns "no" and all processors used before were complete, then one has disproved termination of the TRS $\mathcal{R}$.

Example 7. If d_0 is the initial DP problem $(DP(\mathcal{R}), \mathcal{R}, e)$ and there are sound processors $Proc_0$, $Proc_1$, $Proc_2$ with $Proc_0(d_0) = \{d_1, d_2\}$, $Proc_1(d_1) = \emptyset$, and $Proc_2(d_2) = \emptyset$, then one can conclude termination. But if $Proc_1(d_1) = \mathsf{no}$, and both $Proc_0$ and $Proc_1$ are complete, then one can conclude non-termination.

3 Loops

Almost all techniques for automated termination analysis try to *prove termination* and there are hardly any methods to *prove non-termination*. But detecting non-termination automatically would be very helpful when debugging programs.

We show that the DP framework is particularly suitable for combining both termination and *non*-termination analysis. We introduce a DP processor which tries to detect infinite DP problems in order to answer "no". Then, if all previous processors were complete, we can conclude non-termination of the original TRS. An important advantage of the DP framework is that it can couple the search for a proof and a disproof of termination: Processors which try to prove termination are also helpful for the non-termination proof because they transform the initial DP problem into sub-problems, where most of them can easily be proved finite. So they detect those sub-problems which could cause non-termination. Therefore, the non-termination processors should only operate on these sub-problems and thus, they only have to regard a subset of the rules when searching for non-termination. On the other hand, processors that try to disprove termination are also helpful for the termination proof, even if some of the previous processors were incomplete. The reason is that there are many indeterminisms in a termination proof attempt, since usually many DP processors can be applied to a DP problem. Thus, if one can find out that a DP problem is infinite, one knows that one has reached a "dead end" and should backtrack.

Example 8. Using sound and complete processors we can simplify the initial DP problem $(DP(\mathcal{R}), \mathcal{R}, e)$ of Ex. 1 and 3 to the DP problem $(\{(14), (17)\}, \{(9) - (12)\}, e)$. Hence, if the TRS is not (innermost) terminating then it is only due to the recursion in the factorial function.

An obvious approach to find infinite reductions is to search for a term s which evaluates to a term $C[s\mu]$ containing an instance of s. A TRS with such reductions is called *looping*. Clearly, a naive search for looping terms is very costly.

In contrast to "looping TRSs", when adapting the concept of *loopingness* to DP problems, we only have to consider terms s occurring in dependency pairs

and we do not have to regard any contexts C. The reason is that such contexts are already removed by the construction of dependency pairs. Thm. 10 shows that in this way one can indeed detect all looping TRSs.

Definition 9 (Looping DP Problem [5]). *A DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is* looping *iff there is a $(\mathcal{P}, \mathcal{R})$-chain*

$$s_1 \to_{\mathcal{P}} s_2 \to_{\mathcal{P}\cup\mathcal{R}} s_3 \to_{\mathcal{P}\cup\mathcal{R}} \cdots \to_{\mathcal{P}\cup\mathcal{R}} s_m \to_{\mathcal{P}\cup\mathcal{R}} s_1\mu$$

with $m \geq 1$. Of course, whenever a rule of $\mathcal{P}$ is used then the corresponding reduction must be at the root position.

The requirement that the first rule must be from $\mathcal{P}$ corresponds to the requirement that the first reduction in a chain is done with $\mathcal{P}$.

Theorem 10 (Looping TRSs and Looping DP Problems [5]). *A TRS $\mathcal{R}$ is looping iff the DP problem $(DP(\mathcal{R}), \mathcal{R}, \mathbf{t})$ is looping.*

Example 11. Consider the remaining DP problem $(\{(14), (17)\}, \{(9) - (12)\}, e)$. If $e = \mathbf{t}$ then this DP problem is looping since

$$\begin{array}{lll} \mathsf{fact}^\sharp(x, 0) & \to_{\{(14)\}} & \mathsf{if}^\sharp(x == 0, x, 0) \\ & \to_{\mathcal{R}} & \mathsf{if}^\sharp(\mathsf{chk}(\mathsf{eq}(x, 0)), x, 0) \\ & \to_{\mathcal{R}} & \mathsf{if}^\sharp(\mathsf{false}, x, 0) \\ & \to_{\{(17)\}} & \mathsf{fact}^\sharp(\mathsf{s}(x), \mathsf{s}(0)) \\ & = & \mathsf{fact}^\sharp(x, 0)\ \{x/\mathsf{s}(x)\} \end{array}$$

is a chain where the last term is an instance of the starting term.

Our goal is to detect looping DP problems (in the termination case) since they are infinite and hence, if all preceding DP processors were complete, then termination is disproved. We will discuss the innermost case in Sect. 5.

Lemma 12 (Looping and Infinite DP Problems). *If $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is looping, then $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is infinite.*

Now we can define the DP processor for proving non-termination.

Theorem 13 (Non-Termination Processor [5]). *The following DP processor Proc is sound and complete. For a DP problem $(\mathcal{P}, \mathcal{R}, e)$, Proc returns*

- *"*no*", if $(\mathcal{P}, \mathcal{R}, e)$ is looping and $e = \mathbf{t}$*
- *$\{(\mathcal{P}, \mathcal{R}, e)\}$, otherwise*

It only remains to detect looping DP problems. We consider this problem in the upcoming section.

4 Detecting Looping DP Problems

Our criteria to detect looping DP problems automatically use *narrowing*.

Definition 14 (Narrowing). *Let $\mathcal{R}$ be a TRS. A term t* narrows *to s, denoted $t \rightsquigarrow_{\mathcal{R},\delta,p} s$, iff there is a substitution δ, a (variable-renamed) rule $l \to r \in \mathcal{R}$ and a non-variable position p of t where $\delta = mgu(t|_p, l)$ and $s = t[r]_p\delta$. Let $\rightsquigarrow_{\mathcal{R},\delta}$ be the relation which permits narrowing steps on all positions p. Let $\rightsquigarrow_{(\mathcal{P},\mathcal{R}),\delta}$ denote $\rightsquigarrow_{\mathcal{P},\delta,\varepsilon} \cup \rightsquigarrow_{\mathcal{R},\delta}$, where ε is the root position. Moreover, $\rightsquigarrow^*_{(\mathcal{P},\mathcal{R}),\delta}$ is the smallest relation containing $\rightsquigarrow_{(\mathcal{P},\mathcal{R}),\delta_1} \circ \ldots \circ \rightsquigarrow_{(\mathcal{P},\mathcal{R}),\delta_n}$ for all $n \geq 0$ and all substitutions where $\delta = \delta_1 \ldots \delta_n$.*

To find loops, we narrow the right-hand side t of a dependency pair $s \to t$ until one reaches a term s' such that $s\delta$ *semi-unifies* with s' (i.e., $s\delta\sigma\mu = s'\sigma$ for some substitutions σ and μ). Here, δ is the substitution used for narrowing. Then we indeed have a loop as in Def. 9 since $s\delta\sigma \to_{\mathcal{P}} t\delta\sigma \to^*_{\mathcal{P},\mathcal{R}} s'\sigma = (s\delta\sigma)\mu$. Semi-unification encompasses both matching and unification and algorithms for semi-unification can for example be found in [9,12].

Theorem 15 (Loop Detection by Narrowing [5]). *Let $(\mathcal{P}, \mathcal{R}, e)$ be a DP problem. If there is an $s \to t \in \mathcal{P}$ such that $t \rightsquigarrow^*_{(\mathcal{P},\mathcal{R}),\delta} s'$ and $s\delta$ semi-unifies with s', then $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is looping.*

Example 16. We now try to detect a loop of the DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ of Ex. 11 by narrowing. Starting with the right-hand side of (14) we obtain

$$\begin{aligned} \mathsf{if}^\sharp(x == y, x, y) &\rightsquigarrow_{\mathcal{R},\{\ \}} \mathsf{if}^\sharp(\mathsf{chk}(\mathsf{eq}(x, y)), x, y) \\ &\rightsquigarrow_{\mathcal{R},\{\ \}} \mathsf{if}^\sharp(\mathsf{false}, x, y) \\ &\rightsquigarrow_{\mathcal{P},\{\ \}} \mathsf{fact}^\sharp(\mathsf{s}(x), y) \end{aligned}$$

where the resulting substitution δ is empty in this case. Since the instantiated left-hand side $\mathsf{fact}^\sharp(x, y)$ of (14) matches the resulting term $\mathsf{fact}^\sharp(\mathsf{s}(x), y)$ we know that the DP problem is looping and the TRS of Ex. 1 does not terminate.

Further ways to find loops are backward narrowing where one narrows with the reversed TRS $(\mathcal{P} \cup \mathcal{R})^{-1}$, and one can also permit narrowing into variables, i.e., in Def. 14 the position p may be a variable position of t [5]. Sometimes these extensions are required, but of course they increase the search space.

5 Innermost Loops

Unfortunately, if one does not consider full rewriting but innermost rewriting, then loopingness does not imply non-termination, since $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ is not stable. The reason is that from $s \xrightarrow{\mathsf{i}}^+_{\mathcal{R}} C[s\mu]$ one cannot deduce $s\mu \xrightarrow{\mathsf{i}}^+_{\mathcal{R}} C\mu[s\mu^2]$. And even if this is possible, then it might be the problem that later on for some larger n the reduction $s\mu^n \xrightarrow{\mathsf{i}}^+_{\mathcal{R}} C\mu^n[s\mu^{n+1}]$ is not possible.

As an example consider the TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{g}(x)) \to \mathsf{f}(\mathsf{g}(\mathsf{g}(x))), \mathsf{g}(\mathsf{g}(\mathsf{g}(x))) \to \mathsf{a}\}$. By choosing $s = \mathsf{f}(\mathsf{g}(x))$, $C = \Box$, and $\mu = \{x/\mathsf{g}(x)\}$ we obtain the following reduction.

$$s \xrightarrow{\mathsf{i}}_{\mathcal{R}} C[s\mu] = \mathsf{f}(\mathsf{g}(\mathsf{g}(x))) \xrightarrow{\mathsf{i}}_{\mathcal{R}} C[C\mu[s\mu^2]] = \mathsf{f}(\mathsf{g}(\mathsf{g}(\mathsf{g}(x))))$$

Now the only possible next reduction step yields the normal form $\mathsf{f}(\mathsf{a})$, such that one cannot build an infinite reduction as in the termination case. And indeed, the TRS $\mathcal{R}$ is innermost terminating.

To solve this problem, one can define that a TRS $\mathcal{R}$ is *innermost looping* iff there is a term s, a substitution μ, and a context C such that $s\mu^n \xrightarrow{\mathsf{i}}{}^{+}_{\mathcal{R}} C\mu^n[s\mu^{n+1}]$ for every natural number n. A similar definition was already used in [5, Footnote 6]. Then indeed, innermost loopingness implies innermost non-termination. However the following example shows that this definition does not really correspond to a loop in the intuitive way where one has the same reduction over and over again.

Example 17. Consider the TRS $\mathcal{R}$ with the following rules.

$$\begin{aligned}\mathsf{f}(x, y) &\to \mathsf{f}(\mathsf{s}(x), \mathsf{g}(\mathsf{h}(x, 0)))\\ \mathsf{h}(\mathsf{s}(x), y) &\to \mathsf{h}(x, \mathsf{s}(y))\\ \mathsf{g}(\mathsf{h}(x, y)) &\to \mathsf{i}(y)\end{aligned}$$

Then $\mathcal{R}$ is "innermost looping", as for $s = \mathsf{f}(\mathsf{s}(x), \mathsf{g}(\mathsf{h}(x, 0)))$, $C = \Box$, and $\mu = \{x/\mathsf{s}(x)\}$ there are the following reductions.

$$\begin{aligned}s\mu^n &= \mathsf{f}(\mathsf{s}(x), \mathsf{g}(\mathsf{h}(x, 0)))\mu^n\\ &= \mathsf{f}(\mathsf{s}^{n+1}(x), \mathsf{g}(\mathsf{h}(\mathsf{s}^n(x), 0)))\\ &\xrightarrow{\mathsf{i}}{}^{n}_{\mathcal{R}} \mathsf{f}(\mathsf{s}^{n+1}(x), \mathsf{g}(\mathsf{h}(x, \mathsf{s}^n(0))))\\ &\xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{f}(\mathsf{s}^{n+1}(x), \mathsf{i}(\mathsf{s}^n(0)))\\ &\xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{f}(\mathsf{s}^{n+2}(x), \mathsf{g}(\mathsf{h}(\mathsf{s}^{n+1}(x), 0)))\\ &= C\mu^n[s\mu^{n+1}]\end{aligned}$$

The problem is that the reductions from $s\mu^n$ to $C\mu^n[s\mu^{n+1}]$ depend on n. In this example it might still be possible to represent and check the reductions in a finite way as they have a regular structure, but in general the problem is not even semi-decidable.

Suppose we want to solve the undecidable problem whether a function i over the naturals is total. Since term rewriting is Turing-complete we can assume that there are corresponding rules for i which compute that function by innermost rewriting. But then we can add the three rules of $\mathcal{R}$ and totality of i is equivalent to the question whether s, μ, and C as above form an innermost loop, since we obtain a loop iff all terms $\mathsf{i}(\mathsf{s}^n(0))$ for $n \in \mathbb{N}$ have a normal form.

So, the problem with the requirement $s\mu^n \xrightarrow{\mathsf{i}}{}^{+}_{\mathcal{R}} C\mu^n[s\mu^{n+1}]$ is that although there are some intermediate terms in the infinite reduction that have a regular structure, it is possible that for every n the reduction from $s\mu^n$ to $C\mu^n[s\mu^{n+1}]$ is completely different.

However, in the infinite reduction that corresponds to a loop for standard rewriting there is a strong regularity in the reductions from $s\mu^n$ to $C\mu^n[s\mu^{n+1}]$. For every n one can apply exactly the same rules in exactly the same order at exactly the same positions. Hence, one only has to give the reduction of s to $C[s\mu]$ to know how to continue for $s\mu, s\mu^2, \ldots$. This gives rise to the our final definition of innermost looping.

Definition 18 (Innermost Looping TRS [14]). *A TRS $\mathcal{R}$ is* innermost looping *iff there is a term s, a context C, a substitution μ, a number $m \geq 1$, and if there are rules $\ell_1 \to r_1, \ldots, \ell_m \to r_m \in \mathcal{R}$ and positions $p_1, \ldots, p_m$ such that for all $n \in \mathbb{N}$ there is the following reduction.*[5]

$$s_1\mu^n \xrightarrow{\mathrm{i}}_{\ell_1 \to r_1, p_1} s_2\mu^n \xrightarrow{\mathrm{i}}_{\ell_2 \to r_2, p_2} s_3\mu^n \ldots s_m\mu^n \xrightarrow{\mathrm{i}}_{\ell_m \to r_m, p_m} C\mu^n[s\mu^{n+1}]$$

Note that Def. 18 essentially is the requirement that one has a loop for standard rewriting where all steps in the corresponding infinite reduction are innermost steps. Hence, one can represent an innermost loop in the same way as a loop for termination: just give the reduction $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} s_m \to_{\mathcal{R}} C[s\mu]$, i.e., choosing $n = 0$. The only problem is to check whether a given loop really is an innermost loop. Before we tackle this problem in the next section, we will state the desired fact that innermost loopingness implies innermost non-termination, and we will lift the notion of innermost loopingness to DP problems where again we will show that even for innermost rewriting, loopingness of a TRS coincides with loopingness of the initial DP problem.

Lemma 19 (Innermost Loops and Innermost Non-Termination). *Let $\mathcal{R}$ be a TRS. If $\mathcal{R}$ is innermost looping then $\mathcal{R}$ is not innermost terminating.*

Definition 20 (Innermost Looping DP Problem [14]). *A DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{i})$ is* innermost looping *iff there are rules $\ell_1 \to r_1 \in \mathcal{P}, \ell_2 \to r_2, \ldots, \ell_m \to r_m \in \mathcal{P} \cup \mathcal{R}$ and positions $p_1, \ldots, p_m$ and a substitution μ such that*

$$s_1\mu^n \xrightarrow{\mathrm{i}}_{\ell_1 \to r_1, p_1} s_2\mu^n \xrightarrow{\mathrm{i}}_{\ell_2 \to r_2, p_2} s_3\mu^n \ldots s_m\mu^n \xrightarrow{\mathrm{i}}_{\ell_m \to r_m, p_m} s_1\mu^{n+1}$$

for all $n \in \mathbb{N}$. As in Def. 9, whenever a rule of $\mathcal{P}$ is used then the corresponding position must be the root position.

The following theorem states that loopingness implies non-termination.

Theorem 21 (Innermost Non-Termination Processor [14]). *The following DP processor Proc is sound and complete. For a DP problem $(\mathcal{P}, \mathcal{R}, e)$, Proc returns*

- no, *if $(\mathcal{P}, \mathcal{R}, e)$ is innermost looping and $e = \mathbf{i}$*
- $\{(\mathcal{P}, \mathcal{R}, e)\}$, *otherwise*

[5] Here, $\xrightarrow{\mathrm{i}}_{\ell \to r, p}$ denotes an innermost rewrite step with the rule $\ell \to r$ at position p.

Like in the termination case we have the same advantages when using dependency pairs to search for innermost loops: There is no need to search for a context and one can simplify the initial DP problem before trying to disprove innermost termination.

The following theorem shows that one can detect all innermost looping TRSs by looking for innermost loops in the corresponding DP problems.

Theorem 22 (Innermost Looping TRSs and Innermost Looping DP Problems [14]). *A TRS $\mathcal{R}$ is innermost looping iff $(DP(\mathcal{R}), \mathcal{R}, \mathbf{i})$ is innermost looping.*

6 Detecting Innermost Loops

We remain with the problem to detect innermost looping DP problems. Up to now we only have a way to detect loops with the help of narrowing. Such a loop definitely is a good starting point to find an innermost loop because for every innermost loop there is a loop which satisfies the additional requirements of Def. 20.

Example 23. In Ex. 16 we have detected the following looping reduction.

$$\begin{aligned}
\mathsf{fact}^\sharp(x,y) &\to_{\mathcal{P},\epsilon} \mathsf{if}^\sharp(x == y, x, y)\\
&\to_{\mathcal{R},1} \mathsf{if}^\sharp(\mathsf{chk}(\mathsf{eq}(x,y)), x, y)\\
&\to_{\mathcal{R},1} \mathsf{if}^\sharp(\mathsf{false}, x, y)\\
&\to_{\mathcal{P},\epsilon} \mathsf{fact}^\sharp(\mathsf{s}(x), y)\\
&= \quad \mathsf{fact}^\sharp(x,y)\{x/\mathsf{s}(x)\}
\end{aligned}$$

To check whether this is an innermost loop we have to check for $\mu = \{x/\mathsf{s}(x)\}$ and for all $n \in \mathbb{N}$ whether the corresponding steps are innermost steps when instantiating the terms with μ^n. The main problem in this example is the third reduction since the redex contains the subterm $\mathsf{eq}(x,y)\mu^n$ which might not be a normal form w.r.t. the rule $\mathsf{eq}(x,x) \to \mathsf{true}$ for some n.

In the remainder of this section we will show the main result that it is indeed decidable whether a given loop is an innermost loop. For example, it will turn out that the loop in our example is indeed an innermost loop.

Note that a looping reduction

$$s_1\mu^n \to_{\ell_1 \to r_1, p_1} s_2\mu^n \to_{\ell_2 \to r_2, p_2} \ldots s_m\mu^n \to_{\ell_m \to r_m, p_m} s\mu^{n+1}$$

is innermost looping iff every direct subterm $s_i|_{p_i \cdot j}\ \mu^n$ of every redex $s_i\mu^n|_{p_i}$ is in normal form. Since a term t is in normal form iff t does not contain a redex w.r.t. $\mathcal{R}$, we can reformulate the question about innermost loopingness in terms of so-called *redex problems*.

Definition 24 (Redex, Matching, and Identity Problems [14]). *Let s and ℓ be terms, let μ be a substitution with finite domain. Then a* redex problem

is a triple $(s \mid\rhd \ell, \mu)$*, a* matching problem *is a triple* $(s \rhd \ell, \mu)$*, and an* identity problem *is a triple* $(s \cong \ell, \mu)$.

A redex problem $(s \mid\rhd \ell, \mu)$ *is* solvable *iff there is a natural number* n*, a position* p*, and a substitution* σ *such that* $s\mu^n|_p = \ell\sigma$*, a matching problem is* solvable *iff there is a natural number* n *and a substitution* σ *such that* $s\mu^n = \ell\sigma$*, and an identity problem is* solvable *iff there is a natural number* n *such that* $s\mu^n = \ell\mu^n$.

Obviously, for every term s the redex problem $(s \mid\rhd \mu, \ell)$ is not solvable for any $\ell \to r \in \mathcal{R}$ iff $s\mu^n$ is in normal form w.r.t. $\mathcal{R}$ for all $n \in \mathbb{N}$.

Example 25. We consider the loop of Ex. 23. It is innermost looping iff for $\mu = \{x/\mathsf{s}(x)\}$ all redex problems $(s \mid\rhd \ell, \mu)$ are not solvable where s is chosen from $\{x, y, \mathsf{eq}(x, y), \mathsf{false}\}$ and ℓ is a left-hand side of rules (9) – (12).

To answer the question whether a redex problem $(s \mid\rhd \ell, \mu)$ is solvable, we have to look for three unknowns: the position p, the substitution σ, and the number n. We will now eliminate these unknowns one by one and start with the position p. This will result in matching problems. Then in a second step we will further transform matching problems into identity problems where only the number n is unknown. Finally, we will present a novel algorithm to decide identity problems. Therefore, at the end of this section we will have a decision procedure for redex problems, and thus also for the question whether a given loop respects the strategy.

To start with simplifying a redex problem $(s \mid\rhd \ell, \mu)$ into a finite disjunction of matching problems, note that since the position can be chosen freely from all terms $s, s\mu, s\mu^2, \ldots$ it is not possible to just unroll the possible choices for the position. But the following theorem shows that it is indeed possible to reduce redex problems to matching problems.

Theorem 26 (Solving Redex Problems [14]). *Let* $(s \mid\rhd \ell, \mu)$ *be a redex problem. Let* $\mathcal{W} = \bigcup_{i \in \mathbb{N}} \mathcal{V}(s\mu^i)$*. Then* $(s \mid\rhd \ell, \mu)$ *is solvable iff* ℓ *is a variable or if the matching problem* $(u \rhd \ell, \mu)$ *is solvable for some non-variable subterm* u *of a term in* $\{s\} \cup \{x\mu \mid x \in \mathcal{W}\}$.

Note that the set $\mathcal{W}$ is finite since it is a subset of the finite set of variables $\mathcal{V}(s) \cup \bigcup_{x \in Dom(\mu)} \mathcal{V}(x\mu)$. Hence, Thm. 26 can easily be automated.

Example 27. We use Thm. 26 for the redex problems of Ex. 25. Since there are no new variables occurring when applying μ we obtain $\mathcal{W} = \{x, y\}$. Thus, one of the redex problems is solvable iff one of the matching problems $(s \rhd \ell, \mu)$ is solvable where s is now chosen from $\{\mathsf{s}(x), \mathsf{eq}(x, y), \mathsf{false}\}$. (The variables have been dropped, but one has to consider the new term $\mathsf{s}(x)$ from the substitution.)

Now the question whether a given matching problem is solvable remains. This amounts to detecting the unknown number n and the matcher σ. Our next aim is again to reduce this problem to a conjunction of identity problems where there is no matcher σ any more. However, we first have to generalize the notion of a matching problem $(s \rhd \ell, \mu)$ which includes one pair of terms $s \rhd \ell$ into a matching problem which allows a set of pairs of terms.

Definition 28 (Matching Problem [14]). *A* matching problem $(\mathcal{M}, \mu)$ *consists of a set* $\mathcal{M}$ *of pairs* $\{s_1 \trianglerighteq \ell_1, \ldots, s_k \trianglerighteq \ell_k\}$ *together with a substitution* μ*. A matching problem* $(\mathcal{M}, \mu)$ *is* solvable *iff there is a substitution* σ *and a number* $n \in \mathbb{N}$ *such that for all* $1 \leq j \leq k$ *the equality* $s_j\mu^n = \ell_j\sigma$ *is valid.*

If $\mathcal{M}$ *only contains one pair* $s \trianglerighteq \ell$ *then we identify* $(\mathcal{M}, \mu)$ *with* $(s \trianglerighteq \ell, \mu)$*, and if* μ *is clear from the context we write* $\mathcal{M}$ *as an abbreviation for* $(\mathcal{M}, \mu)$*.*

We now give a set of four transformation rules which either detect that a matching problem is not solvable (indicated by $\bot$), or which transform a matching problem into *solved form*. And once we have reached a matching problem in solved form, it is possible to translate it into identity problems.

Definition 29 (Transformation of Matching Problems [14]). *Let* $\mathcal{V}$ *be the set of variables and let* $\mu = \{x_1/t_1, \ldots, x_m/t_m\}$ *be a substitution. We define* $\mathcal{V}_{incr} = \{x \in \mathcal{V} \mid$ *there is some* $n \in \mathbb{N}$ *with* $x\mu^n \notin \mathcal{V}\}$ *as the set of* increasing variables*.*

For each matching problem $(\mathcal{M}, \mu)$ *with* $\mathcal{M} = \mathcal{M}' \uplus \{s \trianglerighteq \ell\}$ *with* $\ell \notin \mathcal{V}$ *there is a corresponding transformation rule.*

1. $\mathcal{M} \Rightarrow \{s'\mu \trianglerighteq \ell' \mid s' \trianglerighteq \ell' \in \mathcal{M}\}$*, if* $s \in \mathcal{V}_{incr}$
2. $\mathcal{M} \Rightarrow \bot$*, if* $s \in \mathcal{V} \setminus \mathcal{V}_{incr}$
3. $\mathcal{M} \Rightarrow \bot$*, if* $s = f(\ldots)$*,* $\ell = g(\ldots)$*, and* $f \neq g$
4. $\mathcal{M} \Rightarrow \mathcal{M}' \cup \{s_1 \trianglerighteq \ell_1, \ldots, s_k \trianglerighteq \ell_k\}$*, if* $s = f(s_1, \ldots, s_k)$*,* $\ell = f(\ell_1, \ldots, \ell_k)$

Otherwise, a matching problem is in solved form*.*

The following theorem shows that every matching problem $(s \trianglerighteq \ell, \mu)$ can be reduced to a set of identity problems.

Theorem 30 (Solving Matching Problems [14]). *Let* $(\mathcal{M}, \mu)$ *be a matching problem.*

1. *The transformation rules of Def. 29 are confluent and terminating.*
2. *If* $\mathcal{M} \Rightarrow \bot$ *then* $\mathcal{M}$ *is not solvable.*
3. *If* $\mathcal{M} \Rightarrow \mathcal{M}'$ *then* $\mathcal{M}$ *is solvable iff* $\mathcal{M}'$ *is solvable.*
4. $\mathcal{M}$ *is solvable iff* $\mathcal{M} \Rightarrow^* \mathcal{M}'$ *for some matching problem* $\mathcal{M}' = \{s_1 \trianglerighteq x_1, \ldots, s_k \trianglerighteq x_k\}$ *in solved form, such that for all* $i \neq j$ *with* $x_i = x_j$ *the identity problem* $(s_i \cong s_j, \mu)$ *is solvable.*

Example 31. We illustrate the transformation rules of Def. 29 by continuing Ex. 27, where at the end we had to analyze the matching problems $(s \trianglerighteq \ell, \mu)$ where $s \in \{\mathsf{s}(x), \mathsf{eq}(x, y), \mathsf{false}\}$ and ℓ is a left-hand side of one of the rules (9) – (12).

- The matching problems $(\mathsf{s}(x) \trianglerighteq \ell, \mu)$ and $(\mathsf{false} \trianglerighteq \ell, \mu)$ can be reduced to $\bot$ by rule (3). The same is valid for all matching problems $(\mathsf{eq}(x, y) \trianglerighteq \ell, \mu)$ except for the one where ℓ is the left-hand side $\mathsf{eq}(x, x)$.

- The remaining matching problem $(\mathsf{eq}(x, y) \gtrdot \mathsf{eq}(x, x), \mu)$ is transformed by rule (4) into its solved form $\{x \gtrdot x, y \gtrdot x\}$. Hence, by Thm. 30 all matching problems are not solvable iff the identity problem $(x \approxeq y, \mu)$ is not solvable.

It only remains to give an algorithm which decides solvability of an identity problem. The full algorithm is presented in Fig. 1, and we will explain the performed steps one by one.

Input: An identity problem $(s \approxeq t, \mu)$.
Output: "Yes", if the identity problem is solvable, and "No", if it is not.

1. While μ contains a cycle of length $n > 1$ do $\mu := \mu^n$.
2. $S := \emptyset$
3. If $s = t$ then stop with result "Yes".
4. If there is a shared position p of s and t such that $s|_p = f(\dots)$ and $t|_p = g(\dots)$ and $f \neq g$ then stop with result "No".
5. If there is a shared position p of s and t such that $s|_p = x$, $t|_p = g(\dots)$, and x is not an increasing variable then stop with result "No".
   Repeat this step with s and t exchanged.
6. If there is a shared position p of s and t such that $s|_p = x$, $t|_p = y$, $x \neq y$, and $x, y \notin Dom(\mu)$ then stop with result "No".
7. Add the triple $(x, p, t|_p)$ to S for all shared positions p of s and t such that $x = s|_p \neq t|_p$ where x is an increasing variable.
   Repeat this step with s and t exchanged.
8. If $(x, p_1, u_1) \in S$ and $(x, p_2, u_2) \in S$ where
   (a) u_1 and u_2 are not unifiable or where
   (b) $u_1 = u_2$ and $p_1 < p_2$,
   then stop with result "No".
9. $s := s\mu, t := t\mu$
10. Continue with step (3).

Fig. 1. An algorithm to decide solvability of identity problems

First we replace the substitution μ by μ^n such that μ^n does not contain *cycles*. Here, a substitution δ contains a cycle of length n iff $\delta = \{x_1/x_2, x_2/x_3, \dots, x_n/x_1, \dots\}$ where the x_i are pairwise different variables. Obviously, if δ contains a cycle of length n then in δ^n all variables $x_1, \dots, x_n$ do not belong to the domain any more. Thus, step (1) will terminate and afterwards μ does not contain cycles of length 2 or more.

Note that like in Thm. 26 where we replaced μ by $\delta = \mu^{j-i}$, the identity problem $(s \approxeq t, \mu)$ is solvable iff $(s \approxeq t, \mu^n)$ is solvable. Hence, after step (1) we still have to decide solvability of $(s \approxeq t, \mu)$ for the modified μ. The advantage is that now μ has a special structure. For all $x \in Dom(\mu)$ either x is an increasing variable or for some n the term $x\mu^n$ is a variable which is not in the domain of μ. And for such substitutions μ the terms $s, s\mu, s\mu^2, \dots$ finally become *stationary* at each position, i.e., for every position p there is some n such that either all

terms $s\mu^n|_p, s\mu^{n+1}|_p, s\mu^{n+2}|_p, \ldots$ are of the form $f(\ldots)$, or all these terms are the same variable $x \notin Dom(\mu)$. Therefore, it is possible to define $s\mu^\infty$ as the limit of all terms $s, s\mu, s\mu^2, \ldots$.

If the identity problem is solvable then there is some n such that $s\mu^n = t\mu^n$ which is detected in step (3). The reason is that with steps (9) and (10) one iterates over all term pairs (s,t), $(s\mu, t\mu)$, $(s\mu^2, t\mu^2)$, ...

On the other hand it might be the case that an identity problem has a *stationary conflict*, i.e., $s\mu^\infty \neq t\mu^\infty$. Then the identity problem is unsolvable since $s\mu^n = t\mu^n$ implies $s\mu^\infty = t\mu^\infty$. However, if the terms $s\mu^\infty$ and $t\mu^\infty$ differ, then there is some position p such that the symbols at position p differ, or $s\mu^\infty|_p$ is a variable and $t\mu^\infty|_p$ is not a variable (or vice versa), or both $s\mu^\infty|_p$ and $t\mu^\infty|_p$ are different variables. Hence, if we choose n high enough then p is stationary for both $s\mu^n|_p$ and $t\mu^n|_p$. But then one of three cases in steps (4)-(6) will hold. The reason is that all variables in $s\mu^\infty$ and $t\mu^\infty$ are not from the domain of μ.

Up to now we can detect all solvable identity problems and all identity problems which are not solvable due to a stationary conflict. However, there remain other identity problems which are neither solvable nor do they possess a stationary conflict. As an example consider $(x \approxeq y, \{x/\mathsf{f}(x), y/\mathsf{f}(y)\})$. Then $s\mu^\infty = \mathsf{f}(\mathsf{f}(\mathsf{f}(\ldots))) = t\mu^\infty$ but this identity problem is not solvable since $x\mu^n = \mathsf{f}^n(x) \neq \mathsf{f}^n(y) = y\mu^n$. We call these identity problems *infinite*.

The remaining steps (2), (7), and (8) are used to detect infinite identity problems. In the set S we store subproblems (x, p, u) such that whenever the identity problem is solvable, then $x\mu^m = u\mu^m$ must hold for some m to make the terms $s\mu^n$ and $t\mu^n$ equal at position p.

We give some intuitive arguments why the two abortion criteria in step (8) are correct. For (8a) notice that if u_1 and u_2 are not unifiable then $x\mu^m$ cannot be both $u_1\mu^m$ and $u_2\mu^m$, which would be a conflict. And if in the problem to make $x\mu^m$ equal to $u_1\mu^m$ we again produce the same problem at a lower position, then this will continue forever. Hence, the problem is not solvable, which will be detected by step (8b).

The following theorem shows that indeed all answers of the algorithm are correct and it also shows that we will always get an answer. However, especially the termination proof is quite involved since we have to show that the criteria in step (8) suffice to detect all infinity identity problems.

Theorem 32 (Solving Identity Problems [14]). *The algorithm in Fig. 1 to decide solvability of identity problems is correct and it terminates.*

Example 33. We illustrate the algorithm with the identity problem $(x \approxeq y, \mu)$ where $\mu = \{x/\mathsf{f}(y, u_0), y/\mathsf{f}(z, u_0), z/\mathsf{f}(x, u_0), u_0/u_1, u_1/u_0\}$.

As μ contains a cycle of length 2 we replace μ by $\mu^2 = \{x/\mathsf{f}(\mathsf{f}(z, u_0), u_1), y/\mathsf{f}(\mathsf{f}(x, u_0), u_1), z/\mathsf{f}(\mathsf{f}(y, u_0), u_1)\}$. Since $x\mu^\infty = \mathsf{f}(\mathsf{f}(\mathsf{f}(\ldots, u_1), u_0), u_1) = y\mu^\infty$ we know that the problem is solvable or infinite. Hence, the steps (4)-(6) will never succeed. We start with $s = x$ and $t = y$. Since the terms are different we add (x, ϵ, y) and (y, ϵ, x) to S. In the next iteration we have $s = \mathsf{f}(\mathsf{f}(z, u_0), u_1)$ and $t = \mathsf{f}(\mathsf{f}(x, u_0), u_1)$. Again, the terms are different and we add $(x, 11, z)$

and $(z, 11, x)$ to S. The next iteration yields the new triples $(y, 1111, z)$ and $(z, 1111, y)$, and after having applied μ three times we get the two last triples $(x, 111111, y)$ and $(y, 111111, x)$. Then due to (8b) the algorithm terminates with "No".

By simply combining the theorems of this section we have finally obtained a decision procedure which can solve the question whether a loop is also an innermost loop.

Corollary 34 (Deciding Innermost Loops [14]). *For every looping reduction of a TRS or of a DP problem it is decidable whether that reduction is an innermost loop.*

Example 35. At the end of Ex. 31 we knew that the given loop is an innermost loop iff $(x \cong y, \mu)$ is not solvable where $\mu = \{x/\mathsf{s}(x)\}$. We apply the algorithm of Fig. 1 to show that this identity problem is indeed not solvable, and hence we show that the loop is also innermost looping and thus, the TRS of Ex. 1 is not innermost terminating.

Since μ only contains cycles of length 1, we skip step (1). So, let $s = x$ and $t = y$. Then none of the steps (3)-(6) is applicable. Hence, we add (x, ϵ, y) to S and continue with $s = \mathsf{s}(x)$ and $t = y$. Then, in step (5) the algorithm is stopped with answer "No" due to a stationary conflict.

7 Conclusion and Related Work

We have shown how one can identify looping DP problems by narrowing. This is related to the concept of *forward closures* of a TRS $\mathcal{R}$ [10]. However, our approach differs from forward closures by starting from the rules of another TRS $\mathcal{P}$ and by also allowing narrowings with $\mathcal{P}$'s rules on root level. (The reason is that we prove non-termination within the DP framework.) Moreover, we also regard backward narrowing, and we allow narrowing into variables. The latter is partially done in [13] as well.

There are only a few papers on automatically proving *non-termination* of TRSs. Recently, [16,17] (Matchbox and Torpa) presented methods for proving non-termination of *string rewrite systems* (i.e., TRSs where all function symbols have arity 1). Similar to our approach, [16] uses (forward) narrowing and [17] uses ancestor graphs which correspond to (backward) narrowing. However, our approach differs substantially from [16,17]: our technique works within the DP framework, whereas [16,17] operate on the whole set of rules. Therefore, we can benefit from all previous DP processors which decompose the initial DP problem into smaller sub-problems and identify those parts which could cause non-termination. Moreover, we regard full term rewriting instead of string rewriting. Therefore, we use semi-unification to detect loops, whereas for string rewriting, matching is sufficient.

In [13] (NTI) forward closures[6] are used in combination with semi-unification to find loops of TRSs, but as in [16,17] their method works directly on the level

[6] In [13] forward closures are called *unfoldings*.

of the TRS without using dependency pairs. To increase the power they perform a preprocessing step which corresponds to one narrowing step into variables. However, it is easy to give examples where one needs two narrowing steps into variables showing that our approach of narrowing into variables at any time is strictly more powerful.

Both forward and backward narrowing are subsumed by overlap closures [8], a technique by which one can go in both directions at the same time. But if one allows narrowing into variables then our technique is of incomparable power to overlap closures. E.g., the loop of the TRS $\{\mathsf{f}(x,y,x,y,z) \to \mathsf{f}(0,1,z,z,z), \mathsf{a} \to \mathsf{0}, \mathsf{a} \to \mathsf{1}\}$ of [18] cannot be detected by overlap closures, but using narrowing into variables will find the loop.

Finally, we also considered innermost rewriting where we developed the notion of an innermost loop, both for TRSs and for DP problems. Moreover, we presented a decision procedure to detect whether a loop is an innermost loop.

All techniques described in this paper have been implemented in our termination prover AProVE [6]. Since AProVE is currently the only tool (participating in the annual termination competition [11]) which can handle innermost nontermination, it is not surprising that AProVE has achieved the highest score in disproving innermost termination at the competition. However, due to the combination with dependency pairs, AProVE also had the highest score for disproving termination.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
3. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
4. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, LNAI 3452, pages 301–331, 2005.
5. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*, LNAI 3717, pages 216–231, 2005.
6. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
7. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
8. J. Guttag, D. Kapur, and D. Musser. On proving uniform termination and restricted termination of rewriting systems. *SIAM J. Computation*, 12:189–214, 1983.
9. D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. *Theoretical Computer Science*, 81(2):169–187, 1991.
10. D. Lankford and D. Musser. A finite termination criterion. Unpublished Draft. USC Information Scienes Institue, 1978.

11. C. Marché and H. Zantema. The termination competition. In *Proc. RTA '07*, LNCS 4533, pages 303–313, 2007.
12. A. Oliart and W. Snyder. A fast algorithm for uniform semi-unification. In *Proc. CADE '98*, LNCS 1421, pages 239–253, 1998.
13. É. Payet. Detecting non-termination of term rewriting systems using an unfolding operator. In *Proc. LOPSTR '06*, LNCS 4407, pages 177–193, 2006.
14. R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, RWTH Aachen University, 2007. Available as technical report AIB-2007-17, http://aib.informatik.rwth-aachen.de/2007/2007-17.pdf.
15. D. Vroon. Personal communication, 2007.
16. J. Waldmann. Matchbox: A tool for match-bounded string rewriting. In *Proc. 15th RTA*, LNCS 3091, pages 85–94, 2004.
17. H. Zantema. Termination of string rewriting proved automatically. *Journal of Automated Reasoning*, 34:105–139, 2005.
18. X. Zhang. Overlap closures do not suffice for termination of general term rewriting systems. *Information Processing Letters*, 37(1):9–11, 1991.