Proving and Disproving Termination of Higher-Order Functions*

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp

LuFG Informatik II, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany
{giesl|thiemann|psk}@informatik.rwth-aachen.de

Abstract. The dependency pair technique is a powerful modular method for automated termination proofs of term rewrite systems (TRSs). We present two important extensions of this technique: First, we show how to prove termination of *higher-order* functions using dependency pairs. To this end, the dependency pair technique is extended to handle (untyped) applicative TRSs. Second, we introduce a method to prove *non-termination* with dependency pairs, while up to now dependency pairs were only used to verify termination. Our results lead to a framework for combining termination and non-termination techniques for first- and higher-order functions in a very flexible way. We implemented and evaluated our results in the automated termination prover AProVE.

1 Introduction

One of the most powerful techniques to prove termination or innermost termination of TRSs automatically is the *dependency pair approach* [4, 12, 13]. In [16], we recently 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 analysis. After recapitulating the basics of the DP framework in Sect. 2, we present two new significant improvements: in Sect. 3 we extend the framework in order to handle *higher-order* functions and in Sect. 4 we show how to use the DP framework to prove *non-termination*. Sect. 5 summarizes our results and describes their empirical evaluation with the system AProVE. All proofs can be found in [17].

2 The Dependency Pair Framework

We refer to [5] for the basics of rewriting and to [4, 13, 16] 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 $\mathcal{V} = \{x, y, z, \ldots, \alpha, \beta, \ldots\}$. $\mathcal{R}$ is a *TRS over* $\mathcal{F}$ if $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ for all rules $l \to r \in \mathcal{R}$.

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

We will present a method for termination analysis of untyped higher-order functions which do not use λ-abstraction. Due to the absence of λ, such functions can be represented in curried form as *applicative* first-order TRSs (cf. e.g., [22]). A signature $\mathcal{F}$ is *applicative* if it only contains nullary function symbols and a binary symbol $'$ for function application. Moreover, any TRS $\mathcal{R}$ over $\mathcal{F}$ is called *applicative*. So instead of a term $\mathsf{map}(\alpha, x)$ we write $'(' (\mathsf{map}, \alpha), x)$. To ease readability, we use $'$ as an infix-symbol and we let $'$ associate to the left. Then this term can be written as $\mathsf{map}\,'\,\alpha\,'\,x$. This is very similar to the usual notation of higher-order functions where application is just denoted by juxtaposition (i.e., here one would write $\mathsf{map}\;\alpha\;x$ instead of $\mathsf{map}\,'\,\alpha\,'\,x$).

Example 1. The function map is used to apply a function to all elements in a list. Instead of the higher-order rules $\mathsf{map}(\alpha, \mathsf{nil}) \to \mathsf{nil}$ and $\mathsf{map}(\alpha, \mathsf{cons}(x, xs)) \to \mathsf{cons}(\alpha(x), \mathsf{map}(\alpha, xs))$, we encode it by the following first-order TRS.

$$\mathsf{map}\,'\,\alpha\,'\,\mathsf{nil} \to \mathsf{nil} \tag{1}$$

$$\mathsf{map}\,'\,\alpha\,'\,(\mathsf{cons}\,'\,x\,'\,xs) \to \mathsf{cons}\,'\,(\alpha\,'\,x)\,'\,(\mathsf{map}\,'\,\alpha\,'\,xs) \tag{2}$$

A TRS is terminating if all reductions are finite, i.e., if all applications of functions encoded in the TRS terminate. So intuitively, the TRS $\{(1), (2)\}$ is terminating iff map terminates whenever its arguments are terminating terms.

For a TRS $\mathcal{R}$ over $\mathcal{F}$, the *defined symbols* are $\mathcal{D} = \{\mathrm{root}(l) \mid l \to r \in \mathcal{R}\}$ and the *constructors* are $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. For every $\mathsf{f} \in \mathcal{F}$ let $\mathsf{f}^\sharp$ be a fresh *tuple symbol* with the same arity as f, where we often write F for $\mathsf{f}^\sharp$. 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 is a subterm of r,* $\mathrm{root}(t) \in \mathcal{D}\}$.

Example 3. In the TRS of Ex. 1, the only defined symbol is $'$ and map, cons, and nil are constructors. Let AP denote the tuple symbol for $'$. Then we have the following dependency pairs where s is the term $\mathsf{AP}(\mathsf{map}\,'\,\alpha,\ \mathsf{cons}\,'\,x\,'\,xs)$.

$$s \to \mathsf{AP}(\mathsf{cons}\,'\,(\alpha\,'\,x),\ \mathsf{map}\,'\,\alpha\,'\,xs) \tag{3}$$

$$s \to \mathsf{AP}(\mathsf{cons},\ \alpha\,'\,x) \tag{4}$$

$$s \to \mathsf{AP}(\alpha,\ x) \tag{5}$$

$$s \to \mathsf{AP}(\mathsf{map}\,'\,\alpha,\ xs) \tag{6}$$

$$s \to \mathsf{AP}(\mathsf{map},\ \alpha) \tag{7}$$

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. We always assume that different occurrences of dependency pairs are variable disjoint and consider substitutions whose domains may be infinite. In the following definition, $\mathcal{P}$ is usually a set of dependency pairs.

Definition 4 (Chain). *Let $\mathcal{P}, \mathcal{R}$ be TRSs. A (possibly infinite) sequence of pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ from $\mathcal{P}$ is a $(\mathcal{P}, \mathcal{R})$-*chain *iff there is a substitution σ with $t_i\sigma \to_{\mathcal{R}}^* s_{i+1}\sigma$ for all i. It is an* innermost *$(\mathcal{P}, \mathcal{R})$-chain iff $t_i\sigma \xrightarrow{\mathrm{i}}{}_{\mathcal{R}}^* s_{i+1}\sigma$ and $s_i\sigma$ is in normal form w.r.t. $\mathcal{R}$ for all i. Here, "$\xrightarrow{\mathrm{i}}{}_{\mathcal{R}}$" denotes innermost reductions.*

Example 5. "(6), (6)" is a chain: an instance of (6)'s right-hand side $\mathsf{AP}(\mathsf{map}\,'\,\alpha_1,\ xs_1)$ can reduce to an instance of its left-hand side $\mathsf{AP}(\mathsf{map}\,'\,\alpha_2, \mathsf{cons}\,'\,x_2\,'\,xs_2)$.

Theorem 6 (Termination Criterion [4]). *A TRS $\mathcal{R}$ is (innermost) terminating iff there is no infinite (innermost) $(DP(\mathcal{R}), \mathcal{R})$-chain.*

The idea of the DP framework [16] 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")[1] 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 "$\underline{\mathbf{t}}$ermination" or "$\underline{\mathbf{i}}$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 no infinite $(\mathcal{P}, \mathcal{R}, e)$-chain. In this case, we call the problem *finite*.

A DP problem $(\mathcal{P}, \mathcal{R}, e)$ that is not finite is called *infinite*. But in addition, $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is already *infinite* whenever $\mathcal{R}$ is not terminating and $(\mathcal{P}, \mathcal{R}, \mathbf{i})$ is already *infinite* whenever $\mathcal{R}$ is not innermost terminating. Thus, there can be DP problems which are both finite and infinite. For example, the DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ with $\mathcal{P} = \{\mathsf{F}(\mathsf{f}(x)) \to \mathsf{F}(x)\}$ and $\mathcal{R} = \{\mathsf{f}(\mathsf{f}(x)) \to \mathsf{f}(x),\ \mathsf{a} \to \mathsf{a}\}$ is finite since there is no infinite $(\mathcal{P}, \mathcal{R}, \mathbf{t})$-chain, but also infinite since $\mathcal{R}$ is not terminating. Such DP problems do not cause any difficulties, cf. [16]. If one detects an infinite problem during a termination proof attempt, one can abort the proof, since termination has been disproved (if all proof steps were "complete", i.e., if they preserved the termination behavior).

A DP problem $(\mathcal{P}, \mathcal{R}, e)$ is *applicative* iff $\mathcal{R}$ is a TRS over an applicative signature $\mathcal{F}$, and for all $s \to t \in \mathcal{P}$, we have $t \notin \mathcal{V}$, $\{\mathrm{root}(s), \mathrm{root}(t)\} \subseteq \mathcal{F}^\sharp$, and all function symbols below the root of s or t are from $\mathcal{F}$. We also say that such a problem is an applicative DP problem *over* $\mathcal{F}$. Thus, in an applicative DP problem $(\mathcal{P}, \mathcal{R}, e)$, the pairs $s \to t$ of $\mathcal{P}$ must have a shape which is similar to the original dependency pairs (i.e., the roots of s and t are tuple symbols which do not occur below the root). This requirement is needed in Sect. 3.3 in order to transform applicative terms back to ordinary functional form.

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 $Proc(d) = \varnothing$). Completeness is needed to prove non-termination (in particular, to conclude that d is infinite if $Proc(d) = \mathsf{no}$).

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

[1] To ease readability we use a simpler definition of *DP problems* than [16], since this simple definition suffices for the new results of this paper.

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 $Proc_0(d_0) = \{d_1, d_2\}$, $Proc_1(d_1) = \varnothing$, and $Proc_2(d_2) = \varnothing$, 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 DP Processors for Higher-Order Functions

Since we represent higher-order functions by first-order applicative TRSs, all existing techniques and DP processors for first-order TRSs can also be used for higher-order functions. However, most termination techniques rely on the outermost function symbol when comparing terms. This is also true for dependency pairs and standard reduction orders. Therefore, they usually fail for applicative TRSs since here, all terms except variables and constants have the same root symbol ${}'$. For example, a direct termination proof of Ex. 1 is impossible with standard reduction orders and difficult[2] with dependency pairs.

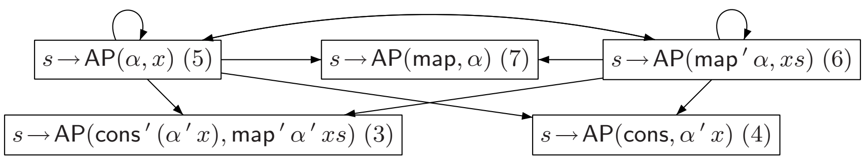
Therefore, in Sect. 3.1 and Sect. 3.2 we improve the most important processors of the DP framework in order to be successful on applicative TRSs. Moreover, we introduce a new processor in Sect. 3.3 which removes the symbol ${}'$ and transforms applicative TRSs and DP problems into ordinary (functional) form again. Sect. 5 shows that these contributions indeed yield a powerful termination technique for higher-order functions. Sect. 3.4 is a comparison with related work.

3.1 A DP Processor Based on the Dependency Graph

The *dependency graph* determines which pairs can follow each other in chains.

Definition 8 (Dependency Graph). *Let* $(\mathcal{P}, \mathcal{R}, e)$ *be a DP problem. The nodes of the* $(\mathcal{P}, \mathcal{R}, e)$-dependency graph *are the pairs of* $\mathcal{P}$ *and there is an arc from* $s \to t$ *to* $u \to v$ *iff* $s \to t, u \to v$ *is an* $(\mathcal{P}, \mathcal{R}, e)$*-chain.*

Example 9. For Ex. 1, we obtain the following $(\mathcal{P}, \mathcal{R}, e)$-dependency graph for both $e = \mathbf{t}$ and $e = \mathbf{i}$. The reason is that the right-hand sides of (3), (4), and (7) have $\mathsf{cons}\,{}'\,(\alpha\,{}'\,x)$, cons, or map as their first arguments. No instance of these terms reduces to an instance of $\mathsf{map}\,{}'\,\alpha$ (which is the first argument of s).



A set $\mathcal{P}' \neq \varnothing$ of dependency pairs is a *cycle* iff for all $s \to t$ and $u \to v$ in $\mathcal{P}'$, there is a path from $s \to t$ to $u \to v$ traversing only pairs of $\mathcal{P}'$. A cycle $\mathcal{P}'$ is a

[2] It needs complex DP processors or base orders (e.g., non-linear polynomial orders).

strongly connected component (SCC) if $\mathcal{P}'$ is not a proper subset of another cycle. As absence of infinite chains can be proved separately for each SCC, termination proofs can be modularized by decomposing a DP problem into sub-problems.

Theorem 10 (Dependency Graph Processor [16]). *For a DP problem $(\mathcal{P}, \mathcal{R}, e)$, let Proc return $\{(\mathcal{P}_1, \mathcal{R}, e), \ldots, (\mathcal{P}_n, \mathcal{R}, e)\}$, where $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are the SCCs of the $(\mathcal{P}, \mathcal{R}, e)$-dependency graph. Then Proc is sound and complete.*

For Ex. 1, we start with the initial DP problem $(\mathcal{P}, \mathcal{R}, e)$, where $\mathcal{P} = \{(3), \ldots, (7)\}$. The only SCC of the dependency graph is $\{(5), (6)\}$. So the above processor transforms $(\mathcal{P}, \mathcal{R}, e)$ into $(\{(5), (6)\}, \mathcal{R}, e)$, i.e., (3), (4), and (7) are deleted.

Unfortunately, the dependency graph is not computable. Therefore, for automation one constructs an *estimated* graph containing at least all arcs of the real graph. The existing estimations that are used for automation [4, 18] assume that all subterms with defined root could possibly be evaluated. Therefore, they use a function $\textsc{cap}$, where $\textsc{cap}(t)$ results from replacing all subterms of t with defined root symbol by different fresh variables. To estimate whether $s \to t$ and $u \to v$ form a chain, one checks whether $\textsc{cap}(t)$ unifies with u (after renaming their variables). Moreover, if one regards termination instead of innermost termination, one first has to linearize $\textsc{cap}(t)$, i.e., multiple occurrences of the same variable in $\textsc{cap}(t)$ are renamed apart. Further refinements of this estimation can be found in [18]; however, they rely on the same function $\textsc{cap}$.

These estimations are not suitable for applicative TRSs. The problem is that there, *all* subterms except variables and constants have the defined root symbol $'$ and are thus replaced by variables when estimating the arcs of the dependency graph. So for Ex. 1, the estimations assume that (3) could be followed by any dependency pair in chains. The reason is that the right-hand side of (3) is $\mathsf{AP}(\mathsf{cons}\,'\,(\alpha\,'\,x), \mathsf{map}\,'\,\alpha\,'\,xs)$ and $\textsc{cap}$ replaces both arguments of AP by fresh variables, since their root symbol $'$ is defined. The resulting term $\mathsf{AP}(y, z)$ unifies with the left-hand side of every dependency pair. Therefore, the estimated dependency graph contains additional arcs from (3) to every dependency pair.

The problem is that these estimations do not check whether subterms with defined root can really be reduced further when being instantiated. For example, the first argument $\mathsf{cons}\,'\,(\alpha\,'\,x)$ of (3)'s right-hand side can never become a redex for any instantiation. The reason is that all left-hand sides of the TRS have the form $\mathsf{map}\,'\,t_1\,'\,t_2$. Thus, one should not replace $\mathsf{cons}\,'\,(\alpha\,'\,x)$ by a fresh variable.

Therefore, we now refine $\textsc{cap}$'s definition. If a subterm can clearly never become a redex, then it is not replaced by a variable anymore. Here, $\textsc{icap}$ is used for **<u>i</u>**nnermost termination proofs and $\textsc{tcap}$ differs from $\textsc{icap}$ by renaming multiple occurrences of variables, which is required when proving full **<u>t</u>**ermination.

Definition 11 ($\textsc{icap}$, $\textsc{tcap}$). *Let $\mathcal{R}$ be a TRS over $\mathcal{F}$, let $f \in \mathcal{F} \cup \mathcal{F}^\sharp$.*

(i) $\textsc{icap}(x) = x$ *for all* $x \in \mathcal{V}$
(ii) $\textsc{icap}(f(t_1, \ldots, t_n)) = f(\textsc{icap}(t_1), \ldots, \textsc{icap}(t_n))$ *iff* $f(\textsc{icap}(t_1), ..., \textsc{icap}(t_n))$ *does not unify with any left-hand side of a rule from* $\mathcal{R}$
(iii) $\textsc{icap}(f(t_1, \ldots, t_n))$ *is a fresh variable, otherwise*

We define $\textsc{tcap}$ *like* $\textsc{icap}$ *but in (i),* $\textsc{tcap}(x)$ *is a different fresh variable for* *every occurrence of x. Moreover in (ii), we use* $\textsc{tcap}(t_i)$ *instead of* $\textsc{icap}(t_i)$.

Now one can detect that (3) should not be connected to any pair in the dependency graph, since $\textsc{icap}(\mathsf{AP}(\mathsf{cons}'\,(\alpha'\,x), \mathsf{map}'\,\alpha'\,xs)) = \mathsf{AP}(\mathsf{cons}'\,y, z)$ does not unify with left-hand sides of dependency pairs. Similar remarks hold for TCAP. This leads to the following improved estimation.[3]

Definition 12 (Improved Estimated Dependency Graph). *In the* estimated $(\mathcal{P}, \mathcal{R}, \mathbf{t})$*-dependency graph there is an arc from* $s \to t$ *to* $u \to v$ *iff* $\textsc{tcap}(t)$ *and* u *are unifiable. In the* estimated $(\mathcal{P}, \mathcal{R}, \mathbf{i})$*-dependency graph there is an arc from* $s \to t$ *to* $u \to v$ *iff* $\textsc{icap}(t)$ *and* u *are unifiable by an mgu* μ *(after renaming their variables) such that* $s\mu$ *and* $u\mu$ *are in normal form w.r.t.* $\mathcal{R}$.

Now the estimated graph is identical to the real dependency graph in Ex. 9.

Theorem 13 (Soundness of the Improved Estimation). *The dependency graph is a subgraph of the estimated dependency graph.*

Of course, the new estimation of dependency graphs from Def. 12 is also useful for non-applicative TRSs and DP problems. The benefits of our improvements (also for ordinary TRSs) is demonstrated by our experiments in Sect. 5.

3.2 DP Processors Based on Orders and on Usable Rules

Classical techniques for automated termination proofs try to find a *reduction order* $\succ$ such that $l \succ r$ holds for all rules $l \to r$. In practice, most orders are *simplification orders* [10]. However, termination of many important TRSs cannot be proved with such orders directly. Therefore, the following processor allows us to use such orders in the DP framework instead. It generates constraints which should be satisfied by a *reduction pair* [23] $(\succsim, \succ)$ where $\succsim$ is reflexive, transitive, monotonic, and stable and $\succ$ is a stable well-founded order compatible with $\succsim$ (i.e., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$). Now one can use existing techniques to search for suitable relations $\succsim$ and $\succ$, and in this way, classical simplification orders can prove termination of TRSs where they would have failed otherwise.

For a problem $(\mathcal{P}, \mathcal{R}, e)$, the constraints require that at least one rule in $\mathcal{P}$ is strictly decreasing (w.r.t. $\succ$) and all remaining rules in $\mathcal{P}$ and $\mathcal{R}$ are weakly decreasing (w.r.t. $\succsim$). Requiring $l \succsim r$ for $l \to r \in \mathcal{R}$ ensures that in chains $s_1 \to t_1, s_2 \to t_2, \ldots$ with $t_i\sigma \to_{\mathcal{R}}^* s_{i+1}\sigma$, we have $t_i\sigma \succsim s_{i+1}\sigma$. Hence, if a reduction pair satisfies these constraints, then the strictly decreasing pairs of $\mathcal{P}$ cannot occur infinitely often in chains. Thus, the following processor deletes these pairs from $\mathcal{P}$. For any TRS $\mathcal{P}$ and any relation $\succ$, let $\mathcal{P}_\succ = \{s \to t \in \mathcal{P} \mid s \succ t\}$.

Theorem 14 (Reduction Pair Processor [16]). *Let* $(\succsim, \succ)$ *be a reduction pair. Then the following DP processor Proc is sound and complete. For a DP problem* $(\mathcal{P}, \mathcal{R}, e)$*, Proc returns*

- $\{(\mathcal{P} \setminus \mathcal{P}_\succ, \mathcal{R}, e)\}$*, if* $\mathcal{P}_\succ \cup \mathcal{P}_\succsim = \mathcal{P}$ *and* $\mathcal{R}_\succsim = \mathcal{R}$
- $\{(\mathcal{P}, \mathcal{R}, e)\}$*, otherwise*

[3] Moreover, TCAP and ICAP can also be combined with further refinements to approximate dependency graphs [4, 18].

DP problems $(\mathcal{P}, \mathcal{R}, \mathbf{i})$ for *innermost* termination can be simplified by replacing the second component $\mathcal{R}$ by those rules from $\mathcal{R}$ that are *usable* for $\mathcal{P}$ (i.e., by the *usable rules* of $\mathcal{P}$). Then by Thm. 14, a weak decrease $l \succsim r$ is not required for all rules but only for the usable rules. As defined in [4], the *usable rules* of a term t contain all f-rules for all function symbols f occurring in t. Moreover, if f's rules are usable and there is a rule $\mathsf{f}(\ldots) \to r$ in $\mathcal{R}$ whose right-hand side r contains a symbol g, then g is usable, too. The *usable rules* of a TRS $\mathcal{P}$ are defined as the usable rules of its right-hand sides.

For instance, after applying the dependency graph processor to Ex. 1, we have the remaining dependency pairs (5) and (6) with the right-hand sides $\mathsf{AP}(\alpha, x)$ and $\mathsf{AP}(\mathsf{map}\,{}'\,\alpha, xs)$. While $\mathsf{AP}(\alpha, x)$ has no usable rules, $\mathsf{AP}(\mathsf{map}\,{}'\,\alpha, xs)$ contains the defined function symbol ${}'$ and therefore, all ${}'$-rules are usable.

This indicates that the definition of usable rules has to be improved to handle applicative TRSs successfully. Otherwise, whenever ${}'$ occurs in the right-hand side of a dependency pair, then *all* rules (except rules of the form $\mathsf{f} \to \ldots$) would be usable. The problem is that the current definition of "usable rules" assumes that all ${}'$-rules can be applied to any subterm with the root symbol ${}'$.

Thus, we refine the definition of usable rules. Now a subterm starting with ${}'$ only influences the computation of the usable rules if some suitable instantiation of this subterm would start new reductions. To detect this, we again use the function ICAP from Def. 11. For example, $\mathsf{map}\,{}'\,\alpha$ can never be reduced if α is instantiated by a normal form, since $\mathsf{map}\,{}'\,\alpha$ does not unify with the left-hand side of any rule. Therefore, the right-hand side $\mathsf{AP}(\mathsf{map}\,{}'\,\alpha, xs)$ of (6) should not have any usable rules.[4]

Definition 15 (Improved Usable Rules). *For a DP problem* $(\mathcal{P}, \mathcal{R}, \mathbf{i})$*, we define the* usable rules $\mathcal{U}(\mathcal{P}) = \bigcup_{s \to t \in \mathcal{P}} \mathcal{U}(t)$. *Here* $\mathcal{U}(t) \subseteq \mathcal{R}$ *is the smallest set with:*

- *If* $t = f(t_1, \ldots, t_n)$, $f \in \mathcal{F} \cup \mathcal{F}^\sharp$, *and* $f(\textsc{icap}(t_1), \ldots, \textsc{icap}(t_n))$ *unifies with a left-hand side* l *of a rule* $l \to r \in \mathcal{R}$*, then* $l \to r \in \mathcal{U}(t)$.
- *If* $l \to r \in \mathcal{U}(t)$*, then* $\mathcal{U}(r) \subseteq \mathcal{U}(t)$.
- *If* t' *is a subterm of* t*, then* $\mathcal{U}(t') \subseteq \mathcal{U}(t)$.

Theorem 16 (Usable Rule Processor). *For a DP problem* $(\mathcal{P}, \mathcal{R}, e)$*, let Proc return* $\{\, (\mathcal{P}, \mathcal{U}(\mathcal{P}), \mathbf{i}) \,\}$ *if* $e = \mathbf{i}$ *and* $\{\, (\mathcal{P}, \mathcal{R}, e) \,\}$ *otherwise. Then Proc is sound.*[5]

Example 17. In Ex. 1, now the dependency pairs in the remaining DP problem $(\{(5), (6)\}, \mathcal{R}, \mathbf{i})$ have no usable rules. Thus, Thm. 16 transforms this DP problem into $(\{(5), (6)\}, \varnothing, \mathbf{i})$. Then with the processor of Thm. 14 we try to find a reduction pair such that (5) and (6) are decreasing. Any simplification order $\succ$ (even the embedding order) makes both pairs strictly decreasing: $s \succ \mathsf{AP}(\alpha, x)$ and $s \succ \mathsf{AP}(\mathsf{map}\,{}'\,\alpha, xs)$ for $s = \mathsf{AP}(\mathsf{map}\,{}'\,\alpha, \mathsf{cons}\,{}'\,x\,{}'\,xs)$. Thus, both dependency pairs are removed and the resulting DP problem $(\varnothing, \varnothing, \mathbf{i})$ is transformed

[4] Our new definition of usable rules can also be combined with other techniques to reduce the set of usable rules [14] and it can also be applied for dependency graph estimations or other DP processors that rely on usable rules [16, 18].

[5] Incompleteness is due to our simplified definition of "DP problems". With the full definition of "DP problems" from [16], the processor is complete [16, Lemma 12].

into the empty set by the dependency graph processor of Thm. 10. So innermost termination of the map-TRS from Ex. 1 can now easily be proved automatically. Note that this TRS is non-overlapping and thus, it belongs to a well-known class where innermost termination implies termination.

Similar to the improved estimation of dependency graphs in the previous section, the new improved definition of usable rules from Def. 15 is also beneficial for ordinary non-applicative TRSs, cf. Sect. 5.

In [32], we showed that under certain conditions, the usable rules of [4] can also be used to prove full instead of just innermost termination (for arbitrary TRSs). Then, even for termination, it is enough to require $l \succsim r$ just for the usable rules in Thm. 14. This result also holds for the new improved usable rules of Def. 15, provided that one uses TCAP instead of ICAP in their definition.

3.3 A DP Processor to Transform Applicative to Functional Form

Some applicative DP problems can be transformed (back) to ordinary functional form. In particular, this holds for problems resulting from first-order functions (encoded by currying). This transformation is advantageous: e.g., the processor in Thm. 14 is significantly more powerful for DP problems in functional form, since standard reduction orders focus on the root symbol when comparing terms.

Example 18. We extend the map-TRS by the following rules for minus and div. Note that a direct termination proof with simplification orders is impossible.

$$\mathsf{minus}\,'x\,'\mathsf{0} \to x \qquad (8) \qquad\qquad \mathsf{div}\,'\mathsf{0}\,'(\mathsf{s}\,'y) \to \mathsf{0} \qquad (10)$$

$$\mathsf{minus}\,'(\mathsf{s}\,'x)\,'(\mathsf{s}\,'y) \to \mathsf{minus}\,'x\,'y \;(9) \quad \mathsf{div}\,'(\mathsf{s}\,'x)\,'(\mathsf{s}\,'y) \to \mathsf{s}\,'(\mathsf{div}\,'(\mathsf{minus}\,'x\,'y)\,'(\mathsf{s}\,'y)) \;(11)$$

While map is really a higher-order function, minus and div correspond to first-order functions. It again suffices to verify innermost termination, since this TRS $\mathcal{R}$ is non-overlapping. The improved estimated dependency graph has three SCCs corresponding to map, minus, and div. Thus, by the dependency graph and the usable rule processors (Thm. 10 and 16), the initial DP problem $(DP(\mathcal{R}), \mathcal{R}, \mathbf{i})$ is transformed into three new problems. The first problem $(\{(5), (6)\}, \varnothing, \mathbf{i})$ for map can be solved as before. The DP problems for minus and div are:

$$(\{\mathsf{AP}(\mathsf{minus}\,'(\mathsf{s}\,'x),\ \mathsf{s}\,'y) \to \mathsf{AP}(\mathsf{minus}\,'x, y)\},\ \varnothing,\ \mathbf{i}) \qquad (12)$$

$$(\{\mathsf{AP}(\mathsf{div}\,'(\mathsf{s}\,'x),\ \mathsf{s}\,'y) \to \mathsf{AP}(\mathsf{div}\,'(\mathsf{minus}\,'x\,'y),\ \mathsf{s}\,'y)\},\ \{(8),(9)\},\ \mathbf{i}) \qquad (13)$$

Since (12) and (13) do not contain map anymore, one would like to change them back to conventional functional form. Then they could be replaced by the following DP problems. Here, every (new) function symbol is labelled by its arity.

$$(\{\mathsf{MINUS}^2(\mathsf{s}^1(x), \mathsf{s}^1(y)) \to \mathsf{MINUS}^2(x,y)\},\ \varnothing,\ \mathbf{i}) \qquad (14)$$

$$\begin{aligned}&(\{\mathsf{DIV}^2(\mathsf{s}^1(x), \mathsf{s}^1(y)) \to \mathsf{DIV}^2(\mathsf{minus}^2(x,y), \mathsf{s}^1(y))\},\\ &\ \{\mathsf{minus}^2(x, \mathsf{0}^0) \to x,\ \mathsf{minus}^2(\mathsf{s}^1(x), \mathsf{s}^1(y)) \to \mathsf{minus}^2(x,y)\},\ \mathbf{i})\end{aligned} \qquad (15)$$

These DP problems are easy to solve: for example, the constraints of the reduction pair processor (Thm. 14) are satisfied by the polynomial order which maps $\mathsf{s}^1(x)$ to $x+1$, $\mathsf{minus}^2(x,y)$ to x, and every other symbol to the sum of its arguments. Thus, termination could immediately be proved automatically.

Now we characterize those applicative TRSs which correspond to first-order functions and can be translated into functional form. In these TRSs, for any function symbol f there is a number n (called its *arity*) such that f only occurs in terms of the form $\mathsf{f}\,'\,t_1\,'\,\ldots\,'\,t_n$. So there are no applications with too few or too many arguments. Moreover, there are no terms $x\,'\,t$ where the first argument of $'$ is a variable. Def. 19 extends this idea from TRSs to DP problems.

Definition 19 (Arity and Proper Terms). *Let $(\mathcal{P},\mathcal{R},e)$ be an applicative DP problem over $\mathcal{F}$. For each $\mathsf{f} \in \mathcal{F} \setminus \{\,'\,\}$ let* $\mathrm{arity}(\mathsf{f}) = max\{\, n \mid \mathsf{f}\,'\,t_1\,'\,\ldots\,'\,t_n$ *or* $(\mathsf{f}\,'\,t_1\,'\,\ldots\,'\,t_n)^\sharp$ *occurs in* $\mathcal{P}\cup\mathcal{R}\,\}$*. A term t is* proper *iff $t \in \mathcal{V}$ or $t = \mathsf{f}\,'\,t_1\,'\,\ldots\,'\,t_n$ or $t = (\mathsf{f}\,'\,t_1\,'\,\ldots\,'\,t_n)^\sharp$ where in the last two cases,* $\mathrm{arity}(\mathsf{f}) = n$ *and all t_i are proper. Moreover, $(\mathcal{P},\mathcal{R},e)$ is* proper *iff all terms in $\mathcal{P}\cup\mathcal{R}$ are proper.*

The DP problems (12) and (13) for minus and div are proper. Here, minus and div have arity 2, s has arity 1, and $\mathsf{0}$ has arity 0. But the problem $(\{(5),(6)\},\varnothing,\mathbf{i})$ for map is not proper as (5) contains the subterm $\mathsf{AP}(\alpha,x)$ with $\alpha \in \mathcal{V}$.

The following transformation translates proper terms from applicative to functional form. To this end, $\mathsf{f}\,'\,t_1\,'\,\ldots\,'\,t_n$ is replaced by $\mathsf{f}^n(\ldots)$, where n is f's arity (as defined in Def. 19) and f^n is a new n-ary function symbol. In this way, (12) and (13) were transformed into (14) and (15) in Ex. 18.

Definition 20 (Transformation $\mathcal{A}$). *$\mathcal{A}$ maps every proper term from $\mathcal{T}(\mathcal{F}\cup\mathcal{F}^\sharp,\mathcal{V})$ to a term from $\mathcal{T}(\{\mathsf{f}^n,\mathsf{F}^n \mid \mathsf{f}\in\mathcal{F}\setminus\{\,'\,\},\ \mathrm{arity}(\mathsf{f})=n\},\mathcal{V})$:*

- $\mathcal{A}(x) = x$ *for all* $x \in \mathcal{V}$
- $\mathcal{A}(\mathsf{f}\,'\,t_1\,'\,\ldots\,'\,t_n) = \mathsf{f}^n(\mathcal{A}(t_1),\ldots,\mathcal{A}(t_n))$ *for all* $\mathsf{f}\in\mathcal{F}\setminus\{\,'\,\}$
- $\mathcal{A}((\mathsf{f}\,'\,t_1\,'\,\ldots\,'\,t_n)^\sharp) = \mathsf{F}^n(\mathcal{A}(t_1),\ldots,\mathcal{A}(t_n))$ *for all* $\mathsf{f}\in\mathcal{F}\setminus\{\,'\,\}$

For any TRS $\mathcal{R}$ with only proper terms, let $\mathcal{A}(\mathcal{R}) = \{\mathcal{A}(l)\to\mathcal{A}(r) \mid l\to r\in\mathcal{R}\}$.

We now define a DP processor which replaces proper DP problems $(\mathcal{P},\mathcal{R},e)$ by $(\mathcal{A}(\mathcal{P}),\mathcal{A}(\mathcal{R}),e)$. Its soundness is due to the fact that every $(\mathcal{P},\mathcal{R},e)$-chain results in an $(\mathcal{A}(\mathcal{P}),\mathcal{A}(\mathcal{R}),e)$-chain, i.e., that $t_i\sigma \to_{\mathcal{R}}^* s_{i+1}\sigma$ implies $\mathcal{A}(t_i)\sigma' \to_{\mathcal{A}(\mathcal{R})}^* \mathcal{A}(s_{i+1})\sigma'$ for some substitution σ'. The reason is that t_i and s_{i+1} are proper and while σ may introduce non-proper terms, every chain can also be constructed with a substitution $\overline{\sigma}$ where all $\overline{\sigma}(x)$ are proper. Thus, while soundness and completeness of the following processor might seem intuitive, the formal proof including this construction is quite involved and can be found in [17].

Theorem 21 (DP Processor for Transformation in Functional Form). *For any DP problem $(\mathcal{P},\mathcal{R},e)$, let Proc return $\{(\mathcal{A}(\mathcal{P}),\mathcal{A}(\mathcal{R}),e)\}$ if $(\mathcal{P},\mathcal{R},e)$ is proper and $\{(\mathcal{P},\mathcal{R},e)\}$ otherwise. Then Proc is sound and complete.*

With the processor of Thm. 21 and our new improved estimation of dependency graphs (Def. 12), it does not matter anymore for the termination proof whether first-order functions are represented in ordinary functional or in applicative form: in the latter case, dependency pairs with non-proper right-hand sides are not in SCCs of the improved estimated dependency graph. Hence, after applying the dependency graph processor of Thm. 10, all remaining DP problems are proper and can be transformed into functional form by Thm. 21.

As an alternative to the processor of Thm. 21, one can also couple the transformation $\mathcal{A}$ with the reduction pair processor from Thm. 14. Then a DP problem $(\mathcal{P}, \mathcal{R}, e)$ is transformed into $\{(\mathcal{P} \setminus \{s \to t \mid \mathcal{A}(s) \succ \mathcal{A}(t)\}, \mathcal{R}, e)\}$ if $(\mathcal{P}, \mathcal{R}, e)$ is proper, if $\mathcal{A}(\mathcal{P})_\succ \cup \mathcal{A}(\mathcal{P})_\succsim = \mathcal{A}(\mathcal{P})$, and if $\mathcal{A}(\mathcal{R})_\succsim = \mathcal{A}(\mathcal{R})$ holds for some reduction pair $(\succsim, \succ)$. An advantage of this alternative processor is that it can be combined with our results from [32] on applying usable rules for termination instead of innermost termination proofs, cf. Sect. 3.2.

3.4 Comparison with Related Work

Most approaches for higher-order functions in term rewriting use *higher-order TRSs*. While there exist powerful termination criteria for higher-order TRSs (e.g., [7, 29]), the main *automated* termination techniques for such TRSs are *simplification orders* (e.g., [20]) which fail on functions like div in Ex. 18.

Exceptions are the *monotonic higher-order semantic path order* [8] and the existing variants of dependency pairs for higher-order TRSs. However, these variants require considerable restrictions (e.g., on the TRSs [31] or on the orders that may be used [3, 24, 30].) So in contrast to our results, they are less powerful than the original dependency pair technique when applied to first-order functions.

Termination techniques for higher-order TRSs often handle a richer language than our results. But these approaches are usually difficult to automate (there are hardly any implementations of these techniques available). In contrast, it is very easy to integrate our results into existing termination provers for ordinary first-order TRSs using dependency pairs (and first-order reduction orders).

Other approaches represent higher-order functions by first-order TRSs [1, 2, 19, 25, 33], similar to us. However, they mostly use *monomorphic* types (this restriction is also imposed in some approaches for higher-order TRSs [8]). In other words, there the types are only built from basic types and type constructors like $\to$ or $\times$, but there are no *type variables*, i.e., no polymorphic types. Then terms like "$\mathsf{map}\,{}'\,\mathsf{minus}\,{}'\,xs$" and "$\mathsf{map}\,{}'\,(\mathsf{minus}\,{}'\,x)\,{}'\,xs$" cannot both be well typed, but one needs different map-symbols for arguments of different types. In contrast, our approach uses untyped term rewriting. Hence, it can be applied for termination analysis of polymorphic or untyped functional languages. Moreover, [25] and [33] only consider extensions of the lexicographic path order, whereas we can also handle non-simply terminating TRSs like Ex. 18.

4 A DP Processor for Proving Non-Termination

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. As shown by our experiments in Sect. 5, our new processor also successfully handles non-terminating higher-order functions if they are represented by first-order TRSs. 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.

To prove non-termination within the DP framework, in Sect. 4.1 we introduce *looping* DP problems and in Sect. 4.2 we show how to detect such DP problems automatically. Finally, Sect. 4.3 is a comparison with related work.

4.1 A DP Processor Based on Looping DP Problems

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. 23 shows that in this way one can indeed detect all looping TRSs.

Definition 22 (Looping DP Problems). *A DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is* looping *iff there is a $(\mathcal{P}, \mathcal{R})$-chain $s_1 \to t_1, s_2 \to t_2, \ldots$ with $t_i\sigma \to_{\mathcal{R}}^* s_{i+1}\sigma$ for all i such that $s_1\sigma$ matches $s_k\sigma$ for some $k > 1$ (i.e., $s_1\sigma\mu = s_k\sigma$ for a substitution μ).*

Theorem 23. *A TRS $\mathcal{R}$ is looping iff the DP problem $(DP(\mathcal{R}), \mathcal{R}, \mathbf{t})$ is looping.*

Example 24. Consider Toyama's example $\mathcal{R} = \{\mathsf{f}(0,1,x) \to \mathsf{f}(x,x,x), \mathsf{g}(y,z) \to y, \mathsf{g}(y,z) \to z\}$ and $\mathcal{P} = DP(\mathcal{R}) = \{\mathsf{F}(0,1,x) \to \mathsf{F}(x,x,x)\}$. We have the $(\mathcal{P}, \mathcal{R})$-chain $\mathsf{F}(0,1,x_1) \to \mathsf{F}(x_1,x_1,x_1)$, $\mathsf{F}(0,1,x_2) \to \mathsf{F}(x_2,x_2,x_2)$, since $\mathsf{F}(x_1,x_1,x_1)\sigma \to_{\mathcal{R}}^* \mathsf{F}(0,1,x_2)\sigma$ for $\sigma(x_1) = \sigma(x_2) = \mathsf{g}(0,1)$. As the term $\mathsf{F}(0,1,x_1)\sigma$ matches $\mathsf{F}(0,1,x_2)\sigma$ (they are even identical), the DP problem $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is looping.

Our goal is to detect looping DP problems. In the termination case, every looping DP problem is infinite and hence, if all preceding DP processors were complete, then termination is disproved. However, the definition of "looping" from Def. 22 cannot be used for innermost termination: in Ex. 24, $(DP(\mathcal{R}), \mathcal{R}, \mathbf{t})$

is looping, but $(DP(\mathcal{R}), \mathcal{R}, \mathbf{i})$ is finite and $\mathcal{R}$ is innermost terminating.[6]

Nevertheless, for *non-overlapping* DP problems, $(\mathcal{P}, \mathcal{R}, \mathbf{i})$ is infinite whenever $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is infinite. So here loopingness of $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ indeed implies that $(\mathcal{P}, \mathcal{R}, \mathbf{i})$ is infinite. We call $(\mathcal{P}, \mathcal{R}, e)$ *non-overlapping* if $\mathcal{R}$ is non-overlapping and no left-hand side of $\mathcal{R}$ unifies with a non-variable subterm of a left-hand side of $\mathcal{P}$.

Lemma 25 (Looping and Infinite DP Problems).

(a) If $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is looping, then $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is infinite.
(b) If $(\mathcal{P}, \mathcal{R}, \mathbf{t})$ is infinite and non-overlapping, then $(\mathcal{P}, \mathcal{R}, \mathbf{i})$ is infinite.

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

Theorem 26 (Non-Termination Processor). *The following DP processor Proc is sound and complete. For a DP problem $(\mathcal{P}, \mathcal{R}, e)$, Proc returns*

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

4.2 Detecting Looping DP Problems

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

Definition 27 (Narrowing). *Let $\mathcal{R}$ be a TRS which may also have rules $l \to r$ with $\mathcal{V}(r) \not\subseteq \mathcal{V}(l)$ or $l \in \mathcal{V}$. 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 $\delta = \delta_1 \ldots \delta_n$.*

Example 28. Let $\mathcal{R} = \{\mathsf{f}(x,y,z) \to \mathsf{g}(x,y,z),\ \mathsf{g}(\mathsf{s}(x),y,z) \to \mathsf{f}(z,\mathsf{s}(y),z)\}$ and $\mathcal{P} = DP(\mathcal{R}) = \{\mathsf{F}(x,y,z) \to \mathsf{G}(x,y,z),\ \mathsf{G}(\mathsf{s}(x),y,z) \to \mathsf{F}(z,\mathsf{s}(y),z)\}$. The term $\mathsf{G}(x,y,z)$ can only be narrowed by the rule $\mathsf{G}(\mathsf{s}(x'),y',z') \to \mathsf{F}(z',\mathsf{s}(y'),z')$ on the root position and hence, we obtain $\mathsf{G}(x,y,z) \rightsquigarrow_{\mathcal{P},[x/\mathsf{s}(x'),\, y'/y,\, z'/z],\varepsilon} \mathsf{F}(z,\mathsf{s}(y),z)$.

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., $s\delta\mu_1\mu_2 = s'\mu_1$ for some substitutions μ_1 and μ_2). Here, δ is the substitution used for narrowing. Then we indeed have a loop as in Def. 22 by defining $\sigma = \delta\mu_1$ and $\mu = \mu_2$. Semi-unification encompasses both matching and unification and algorithms for semi-unification can for example be found in [21, 27].

Theorem 29 (Loop Detection by Forward Narrowing). *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.*

---

[6] One can adapt "loopingness" to the innermost case: $(\mathcal{P}, \mathcal{R}, \mathbf{i})$ is *looping* iff there is an *innermost* $(\mathcal{P}, \mathcal{R})$-chain $s_1 \to t_1, s_2 \to t_2, \ldots$ such that $t_i\sigma\mu^n \xrightarrow{\mathsf{i}}^*_{\mathcal{R}} s_{i+1}\sigma\mu^n$, $s_1\sigma\mu = s_k\sigma$, and $s_i\sigma\mu^n$ is in normal form for all i *and all* $n \geq 0$. Then loopingness implies that the DP problem is infinite, but now one has to examine *infinitely* many instantiations $s_i\sigma\mu^n$ and $t_i\sigma\mu^n$. Nevertheless, one can also formulate sufficient conditions for loopingness in the innermost case which are amenable to automation.

Example 30. We continue with Ex. 28. We had $\mathsf{G}(x,y,z) \leadsto_{(\mathcal{P},\mathcal{R}),\delta} \mathsf{F}(z,\mathsf{s}(y),z)$ where $\delta = [x/\mathsf{s}(x'),\, y'/y,\, z'/z]$. Applying δ to the left-hand side $s = \mathsf{F}(x,y,z)$ of the first dependency pair yields $\mathsf{F}(\mathsf{s}(x'),y,z)$. Now $\mathsf{F}(\mathsf{s}(x'),y,z)$ semi-unifies with $\mathsf{F}(z,\mathsf{s}(y),z)$, since $\mathsf{F}(\mathsf{s}(x'),y,z)\mu_1\mu_2 = \mathsf{F}(z,\mathsf{s}(y),z)\mu_1$ for the substitutions $\mu_1 = [z/\mathsf{s}(x')]$ and $\mu_2 = [y/\mathsf{s}(y)]$. (However, the first term does not match or unify with the second.) Thus, $(\mathcal{P},\mathcal{R},\mathbf{t})$ is looping and $\mathcal{R}$ does not terminate.

However, while the DP problem of Toyama's example (Ex. 24) is looping, this is not detected by Thm. 29. The reason is that the right-hand side $\mathsf{F}(x,x,x)$ of the only dependency pair cannot be narrowed. Therefore, we now introduce a "backward" variant[7] of the above criterion which narrows with the reversed TRSs $\mathcal{P}^{-1}$ and $\mathcal{R}^{-1}$. Of course, in general $\mathcal{P}^{-1}$ and $\mathcal{R}^{-1}$ may also have rules $l \to r$ with $\mathcal{V}(r) \not\subseteq \mathcal{V}(l)$ or $l \in \mathcal{V}$. However, the usual definition of narrowing can immediately be extended to such TRSs, cf. Def. 27.

Theorem 31 (Loop Detection by Backward Narrowing). *Let $(\mathcal{P},\mathcal{R},e)$ be a DP problem. If there is an $s \to t \in \mathcal{P}$ such that $s \leadsto^*_{(\mathcal{P}^{-1},\mathcal{R}^{-1}),\delta} t'$ and t' semi-unifies with $t\delta$, then $(\mathcal{P},\mathcal{R},\mathbf{t})$ is looping.*

Example 32. To detect that Toyama's example (Ex. 24) is looping, we start with the left-hand side $s = \mathsf{F}(0,1,x)$ and narrow 0 to $\mathsf{g}(0,z)$ using $y \to \mathsf{g}(y,z) \in \mathcal{R}^{-1}$. Then we narrow 1 to $\mathsf{g}(y',1)$ by $z' \to \mathsf{g}(y',z')$. Therefore we obtain $\mathsf{F}(0,1,x) \leadsto^*_{(\mathcal{P}^{-1},\mathcal{R}^{-1}),[y/0,\,z'/1]} \mathsf{F}(\mathsf{g}(0,z),\mathsf{g}(y,1),x)$. Now $t' = \mathsf{F}(\mathsf{g}(0,z),\mathsf{g}(y,1),x)$ (semi-)unifies with the corresponding right-hand side $t = \mathsf{F}(x,x,x)$ using $\mu_1 = [x/\mathsf{g}(0,1),\, y/0,\, z/1]$. Thus, $(DP(\mathcal{R}),\mathcal{R},\mathbf{t})$ is looping and the TRS is not terminating.

However, there are also TRSs where backward narrowing fails and forward narrowing succeeds.[8] Note that Ex. 24 where forward narrowing fails is not right-linear and that the example in Footnote 8 where backward narrowing fails is not left-linear. In fact, our experiments show that most looping DP problems $(\mathcal{P},\mathcal{R},\mathbf{t})$ can be detected by forward narrowing if $\mathcal{P}\cup\mathcal{R}$ is right-linear and by backward narrowing if $\mathcal{P}\cup\mathcal{R}$ is left-linear. Therefore, we use the non-termination processor of Thm. 26 with the following heuristic in our system AProVE [15]:

- If $\mathcal{P}\cup\mathcal{R}$ is right- and not left-linear, then use forward narrowing (Thm. 29).
- Otherwise, we use backward narrowing (Thm. 31). If $\mathcal{P}\cup\mathcal{R}$ is not left-linear, then moreover we also permit narrowing steps in variables (i.e., $t|_p \in \mathcal{V}$ is permitted in Def. 27). The reason is that then there are looping DP problems which otherwise cannot be detected by forward or backward narrowing.[9]
- Moreover, to obtain a finite search space, we use an upper bound on the number of times that a rule from $\mathcal{P}\cup\mathcal{R}$ can be used for narrowing.

[7] Thus, non-termination can be investigated both by forward and by backward analysis. In that sense, non-termination is similar to several other properties of programs for which both forward and backward analysis techniques are used. A well-known such property is *strictness* in lazy functional languages. Here, classical forward and backward analysis techniques are [26] and [35], respectively.

[8] An example is $\mathcal{R} = \{\mathsf{f}(x,x) \to \mathsf{f}(0,1),\ 0 \to \mathsf{a},\ 1 \to \mathsf{a}\}$, $\mathcal{P} = DP(\mathcal{R}) = \{\mathsf{F}(x,x) \to \mathsf{F}(0,1)\}$.

[9] An example is the well-known TRS of Drosten [11]. Nevertheless, then there are also looping DP problems which cannot even be found when narrowing into variables.

4.3 Comparison with Related Work

We use narrowing to identify looping DP problems. 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.

There are only few papers on automatically proving *non-termination* of TRSs. An early work is [28] which detects TRSs that are not *simply* terminating (but they may still terminate). Recently, [36, 37] presented methods for proving non-termination of *string rewrite systems* (i.e., TRSs where all function symbols have arity 1). Similar to our approach, [36] uses (forward) narrowing and [37] uses ancestor graphs which correspond to (backward) narrowing. However, our approach differs substantially from [36, 37]: our technique works within the DP framework, whereas [36, 37] 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. Finally, we also presented a condition to disprove *innermost* termination, whereas [36, 37] only try to disprove full termination.

5 Experiments and Conclusion

The DP framework is a general concept for combining termination techniques in a modular way. We presented two important improvements: First, we extended the framework in order to handle higher-order functions, represented as applicative first-order TRSs. To this end, we developed three new contributions: a refined approximation of dependency graphs, an improved definition of usable rules, and a new processor to transform applicative DP problems into functional form. The advantages of our approach, also compared to related work, are the following: it is simple and very easy to integrate into any termination prover based on dependency pairs (e.g., AProVE [15], CiME [9], TTT [19]). Moreover, it encompasses the original DP framework, e.g., it is at least as successful on ordinary first-order functions as the original dependency pair technique. Finally, our approach treats untyped higher-order functions, i.e., it can be used for termination analysis of polymorphic and untyped functional languages.

As a second extension within the DP framework, we introduced a new processor for disproving termination automatically (an important problem which was hardly tackled up to now). A major advantage of our approach is that it combines techniques for proving and for disproving termination in the DP framework, which is beneficial for both termination and non-termination analysis.

We implemented all these contributions in the newest version of our termination prover AProVE [15]. Due to the results of this paper, AProVE 1.2 was the most powerful tool for both termination and non-termination proofs of TRSs at the *Annual International Competition of Termination Tools 2005* [34]. In the following table, we compare AProVE 1.2 with its predecessor AProVE 1.1d-γ, which was the winning tool for TRSs at the competition in 2004. While AProVE 1.1d-γ already contained our results on non-termination analysis, the contributions on handling applicative TRSs from Sect. 3 were missing. For the experiments, we used the same setting as in the competition with a timeout of 60 seconds for each example (where however most proofs take less than two seconds).

	higher-order (61 TRSs)		*non-term* (90 TRSs)		*TPDB* (838 TRSs)	
	t	**n**	**t**	**n**	**t**	**n**
AProVE 1.2	43	8	25	61	639	95
AProVE 1.1d-γ	13	7	24	60	486	92

Here, "*higher-order*" is a collection of untyped versions of typical higher-order functions from [2, 3, 6, 24, 25, 33] and "*non-term*" contains particularly many non-terminating examples. "*TPDB*" is the *Termination Problem Data Base* used in the annual termination competition [34]. It consists of 838 (innermost) termination problems for TRSs from different sources. In the tables, **t** and **n** are the numbers of TRSs where **t**ermination resp. **n**on-termination could be proved.

AProVE 1.2 solves the vast majority of the examples in the "*higher-order*"- and the "*non-term*"-collection. This shows that our results for higher-order functions and non-termination are indeed successful in practice. In contrast, the first column demonstrates that previous techniques for automated termination proofs often fail on applicative TRSs representing higher-order functions. Finally, the last two columns show that our contributions also increase power substantially on ordinary non-applicative TRSs (which constitute most of the TPDB). For further details on our experiments and to download AProVE, the reader is referred to `http://www-i2.informatik.rwth-aachen.de/AProVE/`.

References

1. T. Aoto and T. Yamada. Termination of simply typed term rewriting systems by translation and labelling. In *Proc. RTA '03*, LNCS 2706, pages 380–394, 2003.
2. T. Aoto and T. Yamada. Termination of simply typed applicative term rewriting systems. In *Proc. HOR '04*, Report AIB-2004-03, RWTH, pages 61–65, 2004.
3. T. Aoto and T. Yamada. Dependency pairs for simply typed term rewriting. In *Proc. RTA '05*, LNCS 3467, pages 120–134, 2005.
4. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
5. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge, 1998.
6. R. Bird. *Introduction to Functional Prog. using* Haskell. Prentice Hall, 1998.
7. F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proc. RTA '04*, LNCS 3091, pages 24–39, 2004.
8. C. Borralleras and A. Rubio. A monotonic higher-order semantic path ordering. In *Proc. LPAR '01*, LNAI 2250, pages 531–547, 2001.
9. E. Contejean, C. Marché, B. Monate, and X. Urbain. CiME. `http://cime.lri.fr`.
10. N. Dershowitz. Termination of rewriting. *J. Symb. Comp.*, 3:69–116, 1987.
11. K. Drosten. *Termersetzungssysteme: Grundlagen der Prototyp-Generierung algebraischer Spezifikationen.* Springer, 1989.
12. J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *Appl. Algebra in Engineering, Communication and Computing*, 12(1,2):39–72, 2001.

13. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
14. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving dependency pairs. In *Proc. LPAR '03*, LNAI 2850, pages 165–179, 2003.
15. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *Proc. RTA '04*, LNCS 3091, pages 210–220, 2004.
16. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The DP framework: Combining techniques for autom. termination proofs. In *Proc. LPAR '04*, LNAI 3452, 2005.
17. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. Technical Report AIB-2005-03, RWTH Aachen, 2005. Available from `http://aib.informatik.rwth-aachen.de`.
18. N. Hirokawa and A. Middeldorp. Automating the DP method. In *Proc. CADE '03*, LNAI 2741, pages 32–46, 2003. Full version in *Information and Computation*.
19. N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool. In *Proc. RTA '05*, LNCS 3467, pages 175–184, 2005.
20. J.-P. Jouannaud and A. Rubio. Higher-order recursive path orderings. In *Proc. LICS '99*, pages 402–411, 1999.
21. D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. *Theoretical Computer Science*, 81(2):169–187, 1991.
22. R. Kennaway, J. W. Klop, R. Sleep, and F.-J. de Vries. Comparing curried and uncurried rewriting. *Journal of Symbolic Computation*, 21(1):15–39, 1996.
23. K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *Proc. PPDP '99*, LNCS 1702, pages 48–62, 1999.
24. K. Kusakari. On proving termination of term rewriting systems with higher-order variables. *IPSJ Transactions on Programming*, 42(SIG 7 (PRO 11)):35–45, 2001.
25. M. Lifantsev and L. Bachmair. An LPO-based termination ordering for higher-order terms without λ-abstraction. In *Proc. TPHOLs '98*, LNCS 1479, 1998.
26. A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. 4th Int. Symp. on Programming*, LNCS 83, pages 269–281, 1980.
27. A. Oliart and W. Snyder. A fast algorithm for uniform semi-unification. In *Proc. CADE '98*, LNCS 1421, pages 239–253, 1998.
28. D. A. Plaisted. A simple non-termination test for the Knuth-Bendix method. In *Proc. CADE '86*, LNCS 230, pages 79–88, 1986.
29. J. van de Pol. Termination of higher-order rewrite systems. PhD, Utrecht, 1996.
30. M. Sakai, Y. Watanabe, and T. Sakabe. An extension of dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.
31. M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Trans. on Inf. & Sys.*, 2005. To appear.
32. R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved modular termination proofs using dependency pairs. In *Proc. IJCAR '04*, LNAI 3097, pages 75–90, 2004.
33. Y. Toyama. Termination of S-expression rewriting systems: Lexicographic path ordering for higher-order terms. In *Proc. RTA '04*, LNCS 3091, pages 40–54, 2004.
34. TPDB web page. `http://www.lri.fr/~marche/termination-competition/`.
35. P. Wadler and J. Hughes. Projections for strictness analysis. In *Proc. 3rd Int. Conf. Functional Prog. Lang. & Comp. Arch.*, LNCS 274, pages 385–407, 1987.
36. J. Waldmann. `Matchbox`: A tool for match-bounded string rewriting. In *Proc. 15th RTA*, LNCS 3091, pages 85–94, 2004.
37. H. Zantema. TORPA: Termination of string rewriting proved automatically. *Journal of Automated Reasoning*, 2005. To appear.