A Dependency Pair Framework for Relative Termination of Term Rewriting[⋆]

Jan-Christoph Kassing, Grigory Vartanyan, and Jürgen Giesl

RWTH Aachen University, Aachen, Germany
{kassing,giesl}@cs.rwth-aachen.de, grigory.vartanyan@rwth-aachen.de

Abstract. *Dependency pairs* are one of the most powerful techniques for proving termination of term rewrite systems (TRSs), and they are used in almost all tools for termination analysis of TRSs. Problem #106 of the RTA List of Open Problems asks for an adaption of dependency pairs for *relative termination*. Here, infinite rewrite sequences are allowed, but one wants to prove that a certain subset of the rewrite rules cannot be used infinitely often. Dependency pairs were recently adapted to *annotated dependency pairs (ADPs)* to prove almost-sure termination of probabilistic TRSs. In this paper, we develop a novel adaption of ADPs for relative termination. We implemented our new ADP framework in our tool AProVE and evaluate it in comparison to state-of-the-art tools for relative termination of TRSs.

1 Introduction

Termination is an important topic in program verification. There is a wealth of work on automatic termination analysis of term rewrite systems (TRSs) which can also be used to analyze termination of programs in many other languages. Essentially all current termination tools for TRSs (e.g., AProVE [13], NaTT [36], MU-TERM [15], T$_\mathsf{T}$T$_2$ [27], etc.) use *dependency pairs (DPs)* [1, 11, 12, 16, 17].

A combination of two TRSs (a *main* TRS $\mathcal{R}$ and a *base* TRS $\mathcal{B}$) is "*relatively terminating*" if there is no rewrite sequence that uses infinitely many steps with rules from $\mathcal{R}$ (whereas rules from $\mathcal{B}$ may be used infinitely often). Relative termination of TRSs has been studied since decades [8], and approaches based on relative rewriting are used for many applications, e.g., in complexity analysis [3, 6, 7, 29, 37], for proving confluence [19, 25], for certifying confluence proofs [30], for proving termination of narrowing [20, 31, 34], and for proving liveness [26].

However, while techniques and tools for analyzing ordinary termination of TRSs are very powerful due to the use of DPs, a direct application of standard DPs to analyze relative termination is not possible. Therefore, most existing approaches for automated analysis of relative termination are quite restricted in power. Hence, one of the largest open problems regarding DPs is Problem #106 of the RTA List of Open Problems [5]: *Can we use the dependency pair*

⋆ funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and DFG Research Training Group 2236 UnRAVeL

method to prove relative termination? A first major step towards an answer to this question was presented in [21] by giving criteria for $\mathcal{R}$ and $\mathcal{B}$ that allow the use of ordinary DPs for relative termination.

Recently, we adapted DPs to analyze probabilistic innermost term rewriting, by using so-called *annotated dependency pairs (ADPs)* [23] or *dependency tuples (DTs)* [22] (which were originally proposed for innermost complexity analysis of TRSs [32]).[1] In these adaptions, one considers all *defined* function symbols in the right-hand side of a rule at once, whereas ordinary DPs consider them separately.

In this paper, we show that considering the defined symbols on right-hand sides separately (as for classical DPs) does not suffice for relative termination. On the other hand, we do not need to consider all of them at once either (i.e., we do not have to use the notions of ADPs or DTs from [22, 23, 32]). Instead, we introduce a new definition of ADPs that is suitable for relative termination and develop a corresponding ADP framework for automated relative termination proofs of TRSs. Moreover, while ADPs and DTs were only applicable for *innermost* rewriting in [22, 23, 32], we now adapt ADPs to *full* (relative) rewriting, i.e., we do not impose any specific evaluation strategy. So while [21] presented conditions under which the *ordinary classical* DP framework can be used to prove relative termination, in this paper we develop the first *specific* DP framework for relative termination.

Structure: We start with preliminaries on relative rewriting in Sect. 2. In Sect. 3 we recapitulate the core processors of the DP framework and show that classical DPs are unsound for relative termination in general. Moreover, we state the main results of [21] on criteria when ordinary DPs may nevertheless be used for relative termination. Afterwards, we introduce our novel notion of *annotated dependency pairs* for relative termination in Sect. 4 and present a corresponding new ADP framework in Sect. 5. We implemented our framework in the tool AProVE and in Sect. 6, we evaluate our implementation in comparison to other state-of-the-art tools. All proofs can be found in [24].

2 Relative Term Rewriting

We assume familiarity with term rewriting [2] and regard (finite) TRSs over a (finite) signature Σ and a set of variables $\mathcal{V}$.

Example 1. Consider the following TRS $\mathcal{R}_{\mathsf{divL}}$, where $\mathsf{divL}(x, xs)$ computes the number that results from dividing x by each element of the list xs. As usual, natural numbers are represented by the function symbols $\mathsf{0}$ and s, and lists are represented via nil and cons. Then $\mathsf{divL}(\mathsf{s}^{24}(\mathsf{0}), \mathsf{cons}(\mathsf{s}^4(\mathsf{0}), \mathsf{cons}(\mathsf{s}^3(\mathsf{0}), \mathsf{nil})))$ evaluates to $\mathsf{s}^2(\mathsf{0})$, because $(24/4)/3 = 2$. Here, $\mathsf{s}^2(\mathsf{0})$ stands for $\mathsf{s}(\mathsf{s}(\mathsf{0}))$, etc.

$$
\begin{array}{rlr@{\qquad}rlr}
\mathsf{minus}(x, \mathsf{0}) & \to x & (1) & \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) & \to \mathsf{s}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{s}(y))) & (4)\\
\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to \mathsf{minus}(x, y) & (2) & \mathsf{divL}(x, \mathsf{nil}) & \to x & (5)\\
\mathsf{div}(\mathsf{0}, \mathsf{s}(y)) & \to \mathsf{0} & (3) & \mathsf{divL}(x, \mathsf{cons}(y, xs)) & \to \mathsf{divL}(\mathsf{div}(x, y), xs) & (6)
\end{array}
$$

[1] As shown in [23], using ADPs instead of DTs leads to a more elegant, more powerful, and less complicated framework, and to completeness of the underlying *chain criterion*.

A TRS $\mathcal{R}$ induces a *rewrite relation* $\to_{\mathcal{R}} \subseteq \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$ on terms where $s \to_{\mathcal{R}} t$ holds if there is a $\pi \in \mathrm{Pos}(s)$, a rule $\ell \to r \in \mathcal{R}$, and a substitution σ such that $s|_\pi = \ell\sigma$ and $t = s[r\sigma]_\pi$. For example, $\mathsf{minus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0})) \to_{\mathcal{R}_{\mathsf{divL}}} \mathsf{minus}(\mathsf{0}, \mathsf{0}) \to_{\mathcal{R}_{\mathsf{divL}}} \mathsf{0}$. We call a TRS $\mathcal{R}$ *terminating* (abbreviated SN, for "strongly normalizing") if $\to_{\mathcal{R}}$ is well founded. Using the DP framework, one can easily prove that $\mathcal{R}_{\mathsf{divL}}$ is SN (see Sect. 3.1). In particular, in each application of the recursive divL-rule (6), the length of the list in divL's second argument is decreased by one.

In the relative setting, one considers two TRSs $\mathcal{R}$ and $\mathcal{B}$. We say that $\mathcal{R}$ is *relatively terminating* w.r.t. $\mathcal{B}$ (i.e., $\mathcal{R}/\mathcal{B}$ is SN) if there is no infinite $(\to_{\mathcal{R}} \cup \to_{\mathcal{B}})$-rewrite sequence that uses an infinite number of $\to_{\mathcal{R}}$-steps. We refer to $\mathcal{R}$ as the *main* and $\mathcal{B}$ as the *base* TRS.

Example 2. Let $\mathcal{R}_{\mathsf{divL}}$ be the *main* TRS. Since the order of the list elements does not affect the termination of $\mathcal{R}_{\mathsf{divL}}$, this algorithm also works for multisets. To abstract lists to multisets, we add the *base* TRS $\mathcal{B}_{\mathsf{mset}} = \{(7)\}$.

$$\mathsf{cons}(x, \mathsf{cons}(y, zs)) \to \mathsf{cons}(y, \mathsf{cons}(x, zs)) \tag{7}$$

$\mathcal{B}_{\mathsf{mset}}$ is non-terminating, since it can switch elements in a list arbitrarily often. However, $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset}}$ is SN as each application of Rule (6) still reduces the list length. Indeed, termination of $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset}}$ can also be shown via the approach of [21], because it allows us to apply (standard) DPs in this example, see Ex. 13.

However, if $\mathcal{B}_{\mathsf{mset}}$ is replaced by the base TRS $\mathcal{B}_{\mathsf{mset2}}$ with the rule

$$\mathsf{divL}(z, \mathsf{cons}(x, \mathsf{cons}(y, zs))) \to \mathsf{divL}(z, \mathsf{cons}(y, \mathsf{cons}(x, zs))), \tag{8}$$

then $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset2}}$ remains terminating, but the approach of [21] is no longer applicable, see Ex. 14. In contrast, with our new DP framework in Sect. 4 and 5, termination of such examples can be proved automatically.[2]

We will use the following four examples to illustrate the problems that one has to take into account when analyzing relative termination. So these examples show why a naive adaption of dependency pairs does not work in the relative setting and why we need our new notion of *annotated dependency pairs*. The examples represent different types of infinite rewrite sequences that can lead to non-termination in the relative setting: *redex-duplicating*, *redex-creating* (or "-emitting"), and *ordinary infinite sequences*.

Example 3 (Redex-Duplicating). Consider the TRSs $\mathcal{R}_1 = \{\mathsf{a} \to \mathsf{b}\}$ and $\mathcal{B}_1 = \{\mathsf{f}(x) \to \mathsf{d}(\mathsf{f}(x), x)\}$ from [21, Ex. 4]. $\mathcal{R}_1/\mathcal{B}_1$ is not SN due to the infinite rewrite sequence $\underline{\mathsf{f}(\mathsf{a})} \to_{\mathcal{B}_1} \mathsf{d}(\mathsf{f}(\mathsf{a}), \underline{\mathsf{a}}) \to_{\mathcal{R}_1} \mathsf{d}(\underline{\mathsf{f}(\mathsf{a})}, \mathsf{b}) \to_{\mathcal{B}_1} \mathsf{d}(\mathsf{d}(\mathsf{f}(\mathsf{a}), \underline{\mathsf{a}}), \mathsf{b}) \to_{\mathcal{R}_1} \mathsf{d}(\mathsf{d}(\underline{\mathsf{f}(\mathsf{a})}, \mathsf{b}), \mathsf{b}) \to_{\mathcal{B}_1} \ldots$ The reason is that $\mathcal{B}_1$ can be used to duplicate an arbitrary $\mathcal{R}_1$-redex infinitely often.

Example 4 (Redex-Creating on Parallel Position). Next, consider $\mathcal{R}_2 = \{\mathsf{a} \to \mathsf{b}\}$ and $\mathcal{B}_2 = \{\mathsf{f} \to \mathsf{d}(\mathsf{f}, \mathsf{a})\}$. $\mathcal{R}_2/\mathcal{B}_2$ is not SN as we have the infinite rewrite sequence

[2] To ease the presentation, the rule (8) only switches the first two elements in a list. Our approach also succeeds on a more complicated variant where the elements of lists in divL's second argument can be permuted arbitrarily. We included such an example in the benchmark collection that we used for our evaluation in Sect. 6.

$\underline{\mathsf{f}} \to_{\mathcal{B}_2} \mathsf{d}(\mathsf{f}, \underline{\mathsf{a}}) \to_{\mathcal{R}_2} \mathsf{d}(\underline{\mathsf{f}}, \mathsf{b}) \to_{\mathcal{B}_2} \mathsf{d}(\mathsf{d}(\mathsf{f}, \underline{\mathsf{a}}), \mathsf{b}) \to_{\mathcal{R}_2} \mathsf{d}(\mathsf{d}(\underline{\mathsf{f}}, \mathsf{b}), \mathsf{b}) \to_{\mathcal{B}_2} \ldots$ Here, $\mathcal{B}_2$ can create an $\mathcal{R}_2$-redex infinitely often (where in the right-hand side $\mathsf{d}(\mathsf{f}, \mathsf{a})$ of $\mathcal{B}_2$'s rule, the $\mathcal{B}_2$-redex f and the created $\mathcal{R}_2$-redex a are on parallel positions).

Example 5 (Redex-Creating on Position Above). Let $\mathcal{R}_3 = \{\mathsf{a}(x) \to \mathsf{b}(x)\}$ and $\mathcal{B}_3 = \{\mathsf{f} \to \mathsf{a}(\mathsf{f})\}$. $\mathcal{R}_3/\mathcal{B}_3$ is not SN as we have $\underline{\mathsf{f}} \to_{\mathcal{B}_3} \underline{\mathsf{a}}(\mathsf{f}) \to_{\mathcal{R}_3} \mathsf{b}(\underline{\mathsf{f}}) \to_{\mathcal{B}_3} \mathsf{b}(\underline{\mathsf{a}}(\mathsf{f})) \to_{\mathcal{R}_3} \mathsf{b}(\mathsf{b}(\underline{\mathsf{f}})) \to_{\mathcal{B}_3} \ldots$, i.e., again $\mathcal{B}_3$ can be used to create an $\mathcal{R}_3$-redex infinitely often. In the right-hand side $\mathsf{a}(\mathsf{f})$ of $\mathcal{B}_3$'s rule, the position of the created $\mathcal{R}_3$-redex $\mathsf{a}(\ldots)$ is above the position of the $\mathcal{B}_3$-redex f.

Example 6 (Ordinary Infinite). Finally, consider $\mathcal{R}_4 = \{\mathsf{a} \to \mathsf{b}\}$ and $\mathcal{B}_4 = \{\mathsf{b} \to \mathsf{a}\}$. Here, the base TRS $\mathcal{B}_4$ can neither duplicate nor create an $\mathcal{R}_4$-redex infinitely often, but in combination with the main TRS $\mathcal{R}_4$ we obtain the infinite rewrite sequence $\mathsf{a} \to_{\mathcal{R}_4} \mathsf{b} \to_{\mathcal{B}_4} \mathsf{a} \to_{\mathcal{R}_4} \mathsf{b} \to_{\mathcal{B}_4} \ldots$ Thus, $\mathcal{R}_4/\mathcal{B}_4$ is not SN.

3 DP Framework

We first recapitulate dependency pairs for ordinary (non-relative) rewriting in Sect. 3.1 and summarize existing results on DPs for relative rewriting in Sect. 3.2.

3.1 Dependency Pairs for Ordinary Term Rewriting

We recapitulate DPs and the two most important processors of the DP framework, and refer to, e.g., [1, 11, 12, 16, 17] for more details. As an example, we show how to prove termination of $\mathcal{R}_{\mathsf{divL}}$ without the base $\mathcal{B}_{\mathsf{mset}}$. We decompose the signature $\Sigma = \mathcal{C} \uplus \mathcal{D}$ of a TRS $\mathcal{R}$ such that $f \in \mathcal{D}$ if $f = \mathrm{root}(\ell)$ for some rule $\ell \to r \in \mathcal{R}$. The symbols in $\mathcal{C}$ and $\mathcal{D}$ are called *constructors* and *defined symbols* of $\mathcal{R}$, respectively. For every $f \in \mathcal{D}$, we introduce a fresh *annotated* (or "marked") symbol $f^{\#}$ of the same arity. Let $\mathcal{D}^{\#}$ denote the set of all annotated symbols, and let $\Sigma^{\#} = \Sigma \uplus \mathcal{D}^{\#}$. To ease readability, we often use capital letters like F instead of $\mathsf{f}^{\#}$. For any term $t = f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ with $f \in \mathcal{D}$, let $t^{\#} = f^{\#}(t_1, \ldots, t_n)$. For each rule $\ell \to r$ and each subterm t of r with defined root symbol, one obtains a *dependency pair* $\ell^{\#} \to t^{\#}$. Let $\mathcal{DP}(\mathcal{R})$ denote the set of all dependency pairs of the TRS $\mathcal{R}$.

Example 7. For $\mathcal{R}_{\mathsf{divL}}$ from Ex. 1, we obtain the following five dependency pairs.

$$\mathsf{M}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{M}(x, y) \quad (9) \qquad \mathsf{DL}(x, \mathsf{cons}(y, xs)) \to \mathsf{D}(x, y) \quad (12)$$

$$\mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{M}(x, y) \quad (10) \qquad \mathsf{DL}(x, \mathsf{cons}(y, xs)) \to \mathsf{DL}(\mathsf{div}(x, y), xs) \quad (13)$$

$$\mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{D}(\mathsf{m}(x, y), \mathsf{s}(y)) \quad (11)$$

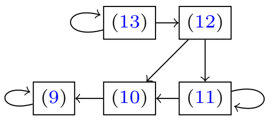
The DP framework operates on *DP problems* $(\mathcal{P}, \mathcal{R})$ where $\mathcal{P}$ is a (finite) set of DPs, and $\mathcal{R}$ is a (finite) TRS. A (possibly infinite) sequence $t_0, t_1, t_2, \ldots$ with $t_i \xrightarrow{\varepsilon}_{\mathcal{P}} \circ \to_{\mathcal{R}}^{*} t_{i+1}$ for all i is a $(\mathcal{P}, \mathcal{R})$-*chain*. Here, $\xrightarrow{\varepsilon}$ are rewrite steps at the root. A chain represents subsequent "function calls" in evaluations. Between two function calls (corresponding to steps with $\mathcal{P}$, called **p**-steps) one can evaluate the arguments using arbitrary many steps with $\mathcal{R}$ (called **r**-steps). So **r**-steps are rewrite steps that are needed in order to enable another **p**-step at a position above later on. Hence, $\mathsf{DL}(\mathsf{s}(0), \mathsf{cons}(\mathsf{s}(0), \mathsf{nil})), \mathsf{DL}(\mathsf{s}(0), \mathsf{nil})$ is a $(\mathcal{DP}(\mathcal{R}_{\mathsf{divL}}), \mathcal{R}_{\mathsf{divL}})$-chain, as $\mathsf{DL}(\mathsf{s}(0), \mathsf{cons}(\mathsf{s}(0), \mathsf{nil})) \xrightarrow{\varepsilon}_{\mathcal{DP}(\mathcal{R}_{\mathsf{divL}})} \mathsf{DL}(\mathsf{div}(\mathsf{s}(0), \mathsf{s}(0)), \mathsf{nil}) \to^*_{\mathcal{R}_{\mathsf{divL}}} \mathsf{DL}(\mathsf{s}(0), \mathsf{nil})$.

A DP problem $(\mathcal{P}, \mathcal{R})$ is called *terminating (SN)* if there is no infinite $(\mathcal{P}, \mathcal{R})$-chain. The main result on DPs is the *chain criterion* which states that a TRS $\mathcal{R}$ is SN iff $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$ is SN. The key idea of the DP framework is a *divide-and-conquer* approach which applies *DP processors* to transform DP problems into simpler sub-problems. A *DP processor* Proc has the form $\mathrm{Proc}(\mathcal{P}, \mathcal{R}) = \{(\mathcal{P}_1, \mathcal{R}_1), \ldots, (\mathcal{P}_n, \mathcal{R}_n)\}$, where $\mathcal{P}, \mathcal{P}_1, \ldots, \mathcal{P}_n$ are sets of DPs and $\mathcal{R}, \mathcal{R}_1, \ldots, \mathcal{R}_n$ are TRSs. Proc is *sound* if $(\mathcal{P}, \mathcal{R})$ is SN whenever $(\mathcal{P}_i, \mathcal{R}_i)$ is SN for all $1 \leq i \leq n$. It is *complete* if $(\mathcal{P}_i, \mathcal{R}_i)$ is SN for all $1 \leq i \leq n$ whenever $(\mathcal{P}, \mathcal{R})$ is SN.

So for a TRS $\mathcal{R}$, one starts with the initial DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R})$ and applies sound (and preferably complete) DP processors until all sub-problems are "solved" (i.e., processors transform them to the empty set). This allows for modular termination proofs, as different techniques can be applied on each sub-problem.

One of the most important processors is the *dependency graph processor*. The $(\mathcal{P}, \mathcal{R})$-*dependency graph* indicates which DPs can be used after each other in chains. Its set of nodes is $\mathcal{P}$ and there is an edge from $s_1 \to t_1$ to $s_2 \to t_2$ if there are substitutions σ_1, σ_2 with $t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2$. Any infinite $(\mathcal{P}, \mathcal{R})$-chain corresponds to an infinite path in the dependency graph, and since the graph is finite, this infinite path must end in a strongly connected component (SCC).[3] Hence, it suffices to consider the SCCs of this graph independently.

Theorem 8 (Dep. Graph Processor). *For the SCCs* $\mathcal{P}_1, \ldots, \mathcal{P}_n$ *of the* $(\mathcal{P}, \mathcal{R})$-*dependency graph,* $\mathrm{Proc}_{\mathtt{DG}}(\mathcal{P}, \mathcal{R}) = \{(\mathcal{P}_1, \mathcal{R}), \ldots, (\mathcal{P}_n, \mathcal{R})\}$ *is sound and complete.*


$$\begin{tikzcd}
 & (13) \arrow[loop left] \arrow[r] & (12) \arrow[dl] \arrow[d] \\
(9) \arrow[loop left] & (10) \arrow[l] & (11) \arrow[l] \arrow[loop right]
\end{tikzcd}$$

While the exact dependency graph is not computable in general, there are several techniques to over-approximate it automatically [1, 12, 16]. The $(\mathcal{DP}(\mathcal{R}_{\mathsf{divL}}), \mathcal{R}_{\mathsf{divL}})$-dependency graph for our example is on the right. Here, $\mathrm{Proc}_{\mathtt{DG}}(\mathcal{DP}(\mathcal{R}_{\mathsf{divL}}),\mathcal{R}_{\mathsf{divL}})$ yields $\big(\{(9)\},\mathcal{R}_{\mathsf{divL}}\big)$, $\big(\{(11)\},\mathcal{R}_{\mathsf{divL}}\big)$, and $\big(\{(13)\},\mathcal{R}_{\mathsf{divL}}\big)$.

The second crucial processor adapts classical reduction orders to DP problems. A *reduction pair* $(\succsim, \succ)$ consists of two relations on terms such that $\succsim$ is reflexive, transitive, and closed under contexts and substitutions, and $\succ$ is a well-founded order that is closed under substitutions but does not have to be closed under contexts. Moreover, $\succsim$ and $\succ$ must be compatible, i.e., $\succsim \circ \succ \circ \succsim\; \subseteq\; \succ$. The *reduction pair processor* requires that all rules and dependency pairs are weakly decreasing, and it removes those DPs that are strictly decreasing.

Theorem 9 (Reduction Pair Processor). *Let* $(\succsim, \succ)$ *be a reduction pair such that* $\mathcal{P} \cup \mathcal{R} \subseteq \succsim$. *Then* $\mathrm{Proc}_{\mathtt{RPP}}(\mathcal{P}, \mathcal{R}) = \{(\mathcal{P} \setminus \succ, \mathcal{R})\}$ *is sound and complete.*

For example, one can use reduction pairs based on polynomial interpretations [28]. A *polynomial interpretation* Pol is a $\Sigma^{\#}$-algebra which maps every function

[3] Here, a set $\mathcal{P}'$ of dependency pairs is an *SCC* if it is a maximal cycle, i.e., it is a maximal set such that for any $s_1 \to t_1$ and $s_2 \to t_2$ in $\mathcal{P}'$ there is a non-empty path from $s_1 \to t_1$ to $s_2 \to t_2$ which only traverses nodes from $\mathcal{P}'$.

symbol $f \in \Sigma^{\#}$ to a polynomial $f_{\mathrm{Pol}} \in \mathbb{N}[\mathcal{V}]$. $\mathrm{Pol}(t)$ denotes the *interpretation* of a term t by the $\Sigma^{\#}$-algebra Pol. Then Pol induces a reduction pair $(\succsim, \succ)$ where $t_1 \succsim t_2$ $(t_1 \succ t_2)$ holds if the inequation $\mathrm{Pol}(t_1) \geq \mathrm{Pol}(t_2)$ $(\mathrm{Pol}(t_1) > \mathrm{Pol}(t_2))$ is true for all instantiations of its variables by natural numbers.

For the three remaining DP problems $\big(\{(9)\}, \mathcal{R}_{\mathsf{divL}}\big)$, $\big(\{(11)\}, \mathcal{R}_{\mathsf{divL}}\big)$, and $\big(\{(13)\}, \mathcal{R}_{\mathsf{divL}}\big)$ in our example, we can apply the reduction pair processor using the polynomial interpretation which maps $\mathsf{0}$ and nil to 0, $\mathsf{s}(x)$ to $x+1$, $\mathsf{cons}(y, xs)$ to $xs+1$, $\mathsf{DL}(x, xs)$ to xs, and all other symbols to their first arguments. Since (9), (11), and (13) are strictly decreasing, $\mathrm{Proc}_{\mathtt{RPP}}$ transforms all three remaining DP problems into DP problems of the form $(\varnothing, \ldots)$. As $\mathrm{Proc}_{\mathtt{DG}}(\varnothing, \ldots) = \varnothing$ and all processors used are sound, this means that there is no infinite chain for the initial DP problem $(\mathcal{DP}(\mathcal{R}_{\mathsf{divL}}), \mathcal{R}_{\mathsf{divL}})$ and thus, $\mathcal{R}_{\mathsf{divL}}$ is SN.

3.2 Dependency Pairs for Relative Termination

Up to now, we only considered DPs for ordinary termination of TRSs. The easiest idea to use DPs in the relative setting is to start with the DP problem $(\mathcal{DP}(\mathcal{R} \cup \mathcal{B}), \mathcal{R} \cup \mathcal{B})$. This would prove termination of $\mathcal{R} \cup \mathcal{B}$, which implies termination of $\mathcal{R}/\mathcal{B}$, but ignores that the rules in $\mathcal{B}$ do not have to terminate. Since termination of DP problems is already defined via a relative condition (finite chains can only have finitely many **p**-steps but there may exist rewrite sequences with infinitely many **r**-steps that are no chains), another idea for proving termination of $\mathcal{R}/\mathcal{B}$ is to start with the DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R} \cup \mathcal{B})$, which only considers the DPs of $\mathcal{R}$. However, this is unsound in general.

Example 10. The only defined symbol of $\mathcal{R}_2$ from Ex. 4 is a. Since the right-hand side of $\mathcal{R}_2$'s rule does not contain defined symbols, we would get the DP problem $(\varnothing, \mathcal{R}_2 \cup \mathcal{B}_2)$, which is SN as it has no DP. Thus, we would falsely conclude that $\mathcal{R}_2/\mathcal{B}_2$ is SN. Similarly, this approach would also falsely "prove" SN for Ex. 3 and 5. Thus, the standard notion of DPs is unsound for relative termination.

In [21], it was shown that under certain conditions on $\mathcal{R}$ and $\mathcal{B}$, starting with the DP problem $(\mathcal{DP}(\mathcal{R} \cup \mathcal{B}_a), \mathcal{R} \cup \mathcal{B})$ for a subset $\mathcal{B}_a \subseteq \mathcal{B}$ is sound for relative termination.[4] The two conditions on the TRSs are *dominance* and being *non-duplicating*. We say that $\mathcal{R}$ *dominates* $\mathcal{B}$ if defined symbols of $\mathcal{R}$ do not occur in the right-hand sides of rules of $\mathcal{B}$. A TRS is *non-duplicating* if no variable occurs more often on the right-hand side of a rule than on its left-hand side.

Theorem 11 (First Main Result of [21], Sound and Complete). *Let $\mathcal{R}$ and $\mathcal{B}$ be TRSs such that $\mathcal{B}$ is non-duplicating and $\mathcal{R}$ dominates $\mathcal{B}$. Then the DP problem $(\mathcal{DP}(\mathcal{R}), \mathcal{R} \cup \mathcal{B})$ is SN iff $\mathcal{R}/\mathcal{B}$ is SN.*

Theorem 12 (Second Main Result of [21], only Sound). *Let $\mathcal{R}$ and $\mathcal{B} = \mathcal{B}_a \uplus \mathcal{B}_b$ be TRSs. If $\mathcal{B}_b$ is non-duplicating, $\mathcal{R} \cup \mathcal{B}_a$ dominates $\mathcal{B}_b$, and the DP problem $(\mathcal{DP}(\mathcal{R} \cup \mathcal{B}_a), \mathcal{R} \cup \mathcal{B})$ is SN, then $\mathcal{R}/\mathcal{B}$ is SN.*

[4] As before, for the construction of $\mathcal{DP}(\mathcal{R} \cup \mathcal{B}_a)$, only the root symbols of left-hand sides of $\mathcal{R} \cup \mathcal{B}_a$ are considered to be "defined".

Example 13. For the main TRS $\mathcal{R}_{\mathsf{divL}}$ from Ex. 1 and base TRS $\mathcal{B}_{\mathsf{mset}}$ from Ex. 2 we can apply Thm. 11 and consider the DP problem $(\mathcal{DP}(\mathcal{R}_{\mathsf{divL}}), \mathcal{R}_{\mathsf{divL}} \cup \mathcal{B}_{\mathsf{mset}})$, since $\mathcal{B}_{\mathsf{mset}}$ is non-duplicating and $\mathcal{R}_{\mathsf{divL}}$ dominates $\mathcal{B}_{\mathsf{mset}}$. As for $(\mathcal{DP}(\mathcal{R}_{\mathsf{divL}}),\mathcal{R}_{\mathsf{divL}})$, the DP framework can prove that $(\mathcal{DP}(\mathcal{R}_{\mathsf{divL}}), \mathcal{R}_{\mathsf{divL}} \cup \mathcal{B}_{\mathsf{mset}})$ is SN. In this way, the tool NaTT which implements the results of [21] proves that $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset}}$ is SN. Note that sophisticated techniques like DPs are needed to prove SN for $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset}}$ because classical (simplification) orders already fail to prove termination of $\mathcal{R}_{\mathsf{divL}}$.

Example 14. As mentioned in Ex. 2, if we consider $\mathcal{B}_{\mathsf{mset2}}$ with the rule

$$\mathsf{divL}(z, \mathsf{cons}(x, \mathsf{cons}(y, zs))) \to \mathsf{divL}(z, \mathsf{cons}(y, \mathsf{cons}(x, zs))) \qquad (8)$$

instead of $\mathcal{B}_{\mathsf{mset}}$ as the base TRS, then $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset2}}$ is still terminating, but we cannot use Thm. 11 since $\mathcal{R}_{\mathsf{divL}}$ does not dominate $\mathcal{B}_{\mathsf{mset2}}$. If we try to split $\mathcal{B}_{\mathsf{mset2}}$ as in Thm. 12, then $\varnothing \neq \mathcal{B}_a \subseteq \mathcal{B}_{\mathsf{mset2}}$ implies $\mathcal{B}_a = \mathcal{B}_{\mathsf{mset2}}$, but $\mathcal{B}_{\mathsf{mset2}}$ is non-terminating. Therefore, all previous tools for relative termination fail in proving that $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset2}}$ is SN. In Sect. 4 we will present our novel DP framework which can prove relative termination of relative TRSs like $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset2}}$.

As remarked in [21], Thm. 11 and 12 are unsound if one only considers *minimal* chains, i.e., if for a DP problem $(\mathcal{P}, \mathcal{R})$ one only considers chains $t_0, t_1, \ldots$, where all t_i are $\mathcal{R}$-terminating. In the DP framework for ordinary rewriting, the restriction to minimal chains allows the use of further processors, e.g., based on *usable rules* [12, 17] or the *subterm criterion* [17]. As shown in [21], usable rules and the subterm criterion can nevertheless be applied if $\mathcal{B}$ is *quasi-terminating* [4], i.e., $\{t \mid s \to_{\mathcal{B}}^* t\}$ is finite for every term s. This restriction would also be needed to integrate processors that rely on minimality into our new framework in Sect. 4.

4 Annotated Dependency Pairs for Relative Termination

As shown in Sect. 3.2, up to now there only exist criteria [21] that state when it is sound to apply *ordinary* DPs for proving relative termination, but there is no *specific* DP-based technique to analyze relative termination directly. For ordinary termination, we create a separate DP for each occurrence of a defined symbol in the right-hand side of a rule (and no DP is created for rules without defined symbols in their right-hand sides). This would work to detect *ordinary infinite* sequences like the one in Ex. 6 in the relative setting, i.e., such an infinite sequence would give rise to an infinite chain. However, as shown in Ex. 10, this would not suffice to detect infinite redex-creating sequences as in Ex. 4 and 5. Thus, ordinary DPs are unsound for analyzing relative termination.

To solve this problem, we now adapt the concept of *annotated dependency pairs* (ADPs) for relative termination. ADPs were introduced in [23] to prove innermost almost-sure termination of probabilistic term rewriting. In the relative setting, we can use similar dependency pairs as in the probabilistic setting, but with a different rewrite relation $\hookrightarrow$ to deal with non-innermost steps. Compared to [21], we (a) remove the requirement of dominance, which will be handled by the dependency graph processor, and (b) allow for ADP processors that are specifically designed for the relative setting before possibly moving to ordinary DPs.

The requirement that $\mathcal{B}$ must be non-duplicating remains, since relative non-termination because of duplicating rules is not necessarily due to the relation between the left-hand side and the subterms with defined root symbols in the right-hand side of a rule. Therefore, this cannot be captured by (A)DPs, i.e., DPs do not help in analyzing redex-duplicating sequences as in Ex. 3, where the crucial redex a is not generated from a "function call" in the right-hand side of a rule, but it just corresponds to a duplicated variable. To handle TRSs $\mathcal{R}/\mathcal{B}$ where $\mathcal{B}_{dup} \subseteq \mathcal{B}$ is duplicating, one can move the duplicating rules to the main TRS $\mathcal{R}$ and try to prove relative termination of $(\mathcal{R} \cup \mathcal{B}_{dup})/(\mathcal{B} \setminus \mathcal{B}_{dup})$ instead, or one can try to find a reduction pair $(\succsim, \succ)$ where $\succ$ is closed under contexts such that $\mathcal{R} \cup \mathcal{B} \subseteq \succsim$ and $\mathcal{B}_{dup} \subseteq \succ$. Then it suffices to prove relative termination of $(\mathcal{R} \setminus \succ)/(\mathcal{B} \setminus \succ)$ instead.

We will now define a notion of DPs that can detect infinite redex-creating sequences as in Ex. 4 with $\mathcal{R}_2 = \{\mathsf{a} \to \mathsf{b}\}$ and $\mathcal{B}_2 = \{\mathsf{f} \to \mathsf{d}(\mathsf{f}, \mathsf{a})\}$: $\underline{\mathsf{f}} \to_{\mathcal{B}_2} \mathsf{d}(\mathsf{f}, \underline{\mathsf{a}}) \to_{\mathcal{R}_2} \mathsf{d}(\underline{\mathsf{f}}, \mathsf{b}) \to_{\mathcal{B}_2} \mathsf{d}(\mathsf{d}(\mathsf{f}, \underline{\mathsf{a}}), \mathsf{b}) \to_{\mathcal{R}_2} \ldots$ To this end, (1) we need a DP for the rule $\mathsf{a} \to \mathsf{b}$ to track the reduction of the created $\mathcal{R}_2$-redex a, although b is a constructor. Moreover, (2) both defined symbols f and a in the right-hand side of the rule $\mathsf{f} \to \mathsf{d}(\mathsf{f}, \mathsf{a})$ have to be considered simultaneously: We need f to create an infinite number of $\mathcal{R}_2$-redexes, and we need a since it is the created $\mathcal{R}_2$-redex. Hence, for rules from the base TRS $\mathcal{B}_2$, we have to consider all possible pairs of defined symbols in their right-hand sides simultaneously.[5] This is not needed for the main TRS $\mathcal{R}_2$, i.e., if the f-rule were in the main TRS, then the f in the right-hand side could be considered separately from the a that it generates. Therefore, we distinguish between *main* and *base ADPs* (that are generated from the main and the base TRS, respectively).

As in [23], we now annotate defined symbols directly in the original rewrite rule instead of extracting annotated subterms from its right-hand side. In this way, we may have terms containing several annotated symbols, which allows us to consider pairs of defined symbols in right-hand sides simultaneously. At the same time, an ADP maintains the information on the positions of the subterms in the original right-hand side. (This information will be needed for the "completeness" of the chain criterion in Thm. 23, i.e., it allows us to obtain an *equivalent* characterization of relative termination via chains of ADPs.[6])

Definition 15 (Annotations). *For $t \in \mathcal{T}\left(\Sigma^{\#}, \mathcal{V}\right)$ and $\mathcal{X} \subseteq \Sigma^{\#} \cup \mathcal{V}$, let $\mathrm{Pos}_{\mathcal{X}}(t)$ be the set of all positions of t with symbols or variables from $\mathcal{X}$. For $\Phi \subseteq \mathrm{Pos}_{\mathcal{D} \cup \mathcal{D}^{\#}}(t)$, $\#_{\Phi}(t)$ is the variant of t where the symbols at positions from Φ are annotated and all other annotations are removed. Thus, $\mathrm{Pos}_{\mathcal{D}^{\#}}(\#_{\Phi}(t)) = \Phi$,*

[5] For relative termination, it suffices to consider *pairs* of defined symbols. The reason is that to "track" a non-terminating reduction, one only has to consider a single redex plus possibly another redex of the base TRS which may later create a redex of the main TRS again.

[6] This is the main advantage of ADPs over related formalisms like *dependency tuples* [22, 32] where this information on the positions is lost. Therefore, as shown in [23] for almost-sure termination analysis of probabilistic term rewriting, using ADPs instead of DTs leads to a more elegant, more powerful, and less complicated framework.

and $\#_{\varnothing}(t)$ removes all annotations from t, where we often write $\flat(t)$ instead of $\#_{\varnothing}(t)$. Moreover, for a singleton $\{\pi\}$, we often write $\#_{\pi}$ instead of $\#_{\{\pi\}}$. We write $t \trianglelefteq_{\#}^{\pi} s$ if $\pi \in \mathrm{Pos}_{\mathcal{D}^{\#}}(s)$ and $t = \flat(s|_{\pi})$ (i.e., t results from a subterm of s with annotated root symbol by removing its annotations). We also write $\trianglelefteq_{\#}$ instead of $\trianglelefteq_{\#}^{\pi}$ if π is irrelevant.

Example 16. If $\mathsf{f} \in \mathcal{D}$, then we have $\#_1(\mathsf{f}(\mathsf{f}(x))) = \#_1(\mathsf{F}(\mathsf{F}(x))) = \mathsf{f}(\mathsf{F}(x))$ and $\flat(\mathsf{F}(\mathsf{F}(x))) = \mathsf{f}(\mathsf{f}(x))$. Moreover, we have $\mathsf{f}(x) \trianglelefteq_{\#}^{1} \mathsf{f}(\mathsf{F}(x))$.

While in [23] all defined symbols on the right-hand sides of rules were annotated, we now define our novel variant of *annotated dependency pairs* for relative rewriting. As explained before Def. 15, we have to track (at most) two redexes for base ADPs and only one redex for main ADPs.

Definition 17 (Annotated Dependency Pair). *A rule $\ell \to r$ with $\ell \in \mathcal{T}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$, $r \in \mathcal{T}(\Sigma^{\#}, \mathcal{V})$, and $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$ is called an* annotated dependency pair (ADP)*. Let $\mathcal{D}$ be the defined symbols of $\mathcal{R} \cup \mathcal{B}$, and for $n \in \mathbb{N}$, let $\mathcal{A}_n(\ell \to r) = \{\ell \to \#_{\Phi}(r) \mid \Phi \subseteq \mathrm{Pos}_{\mathcal{D}}(r), |\Phi| = \min(n, |\mathrm{Pos}_{\mathcal{D}}(r)|)\}$. The* canonical main ADPs *for $\mathcal{R}$ are $\mathcal{A}_1(\mathcal{R}) = \bigcup_{\ell \to r \in \mathcal{R}} \mathcal{A}_1(\ell \to r)$ and the* canonical base ADPs *for $\mathcal{B}$ are $\mathcal{A}_2(\mathcal{B}) = \bigcup_{\ell \to r \in \mathcal{B}} \mathcal{A}_2(\ell \to r)$.*

So the left-hand side of an ADP is just the left-hand side of the original rule. The right-hand side results from the right-hand side of the original rule by replacing certain defined symbols f with $f^{\#}$.

Example 18. The canonical ADPs of Ex. 4 are $\mathcal{A}_1(\mathcal{R}_2) = \{\mathsf{a} \to \mathsf{b}\}$ and $\mathcal{A}_2(\mathcal{B}_2) = \{\mathsf{f} \to \mathsf{d}(\mathsf{F}, \mathsf{A})\}$ and for Ex. 5 we get $\mathcal{A}_1(\mathcal{R}_3) = \{\mathsf{a}(x) \to \mathsf{b}(x)\}$ and $\mathcal{A}_2(\mathcal{B}_3) = \{\mathsf{f} \to \mathsf{A}(\mathsf{F})\}$. For $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset2}}$ from Ex. 1 and 14, the ADPs $\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}})$ are

$$\mathsf{minus}(x, \mathsf{0}) \to x \quad (14) \qquad \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{D}(\mathsf{minus}(x, y), \mathsf{s}(y))) \quad (18)$$

$$\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{M}(x, y) \quad (15) \qquad \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{div}(\mathsf{M}(x, y), \mathsf{s}(y))) \quad (19)$$

$$\mathsf{div}(\mathsf{0}, \mathsf{s}(y)) \to \mathsf{0} \quad (16) \qquad \mathsf{divL}(x, \mathsf{cons}(y, xs)) \to \mathsf{DL}(\mathsf{div}(x, y), xs) \quad (20)$$

$$\mathsf{divL}(x, \mathsf{nil}) \to x \quad (17) \qquad \mathsf{divL}(x, \mathsf{cons}(y, xs)) \to \mathsf{divL}(\mathsf{D}(x, y), xs) \quad (21)$$

and $\mathcal{A}_2(\mathcal{B}_{\mathsf{mset2}})$ contains $\mathsf{divL}(z, \mathsf{cons}(x, \mathsf{cons}(y, zs))) \to \mathsf{DL}(z, \mathsf{cons}(y, \mathsf{cons}(x, zs)))$ (22)

In [23], ADPs were only used for innermost rewriting. We now modify their rewrite relation and define what happens with annotations inside the substitutions during a rewrite step. To simulate redex-creating sequences as in Ex. 5 with ADPs (where the position of the created redex $\mathsf{a}(\ldots)$ is above the position of the creating redex f), ADPs should be able to rewrite above annotated arguments without removing their annotation (we will demonstrate that in Ex. 25). Thus, for an ADP $\ell \to r$ with a variable $\ell|_{\pi} = x$, we use a *variable reposition function (VRF)* to indicate which occurrence of x in r should keep the annotations if one rewrites an instance of ℓ where the subterm at position π is annotated. So a VRF maps positions of variables in the left-hand side of a rule to positions of the same variable in the right-hand side.

Definition 19 (Variable Reposition Function). *Let $\ell \to r$ be an ADP. A function $\varphi : \mathrm{Pos}_{\mathcal{V}}(\ell) \to \mathrm{Pos}_{\mathcal{V}}(r) \uplus \{\bot\}$ is called a* variable reposition function (VRF) *for $\ell \to r$ iff $\ell|_\pi = r|_{\varphi(\pi)}$ whenever $\varphi(\pi) \neq \bot$.*

Example 20. For the ADP $\mathsf{a}(x) \to \mathsf{b}(x)$ for $\mathcal{R}_3$ from Ex. 5, if x on position 1 of the left-hand side is instantiated by F, then the VRF $\varphi(1) = 1$ indicates that this ADP rewrites $\mathsf{A}(\mathsf{F})$ to $\mathsf{b}(\mathsf{F})$, while $\varphi(1) = \bot$ means that it rewrites $\mathsf{A}(\mathsf{F})$ to $\mathsf{b}(\mathsf{f})$.

With VRFs we can define the rewrite relation for ADPs w.r.t. full rewriting.

Definition 21 ($\hookrightarrow_{\mathcal{P}}$). *Let $\mathcal{P}$ be a set of ADPs. A term $s \in \mathcal{T}\left(\Sigma^{\#}, \mathcal{V}\right)$ rewrites to t using $\mathcal{P}$ (denoted $s \hookrightarrow_{\mathcal{P}} t$) if there are an ADP $\ell \to r \in \mathcal{P}$, a substitution σ, a position $\pi \in \mathrm{Pos}_{\mathcal{D} \cup \mathcal{D}^{\#}}(s)$ such that $\flat(s|_\pi) = \ell\sigma$, a VRF φ for $\ell \to r$, and*[7]

$$\begin{array}{lll} t = s[\#_{\Phi}(r\sigma)]_\pi & \text{if } \pi \in \mathrm{Pos}_{\mathcal{D}^{\#}}(s) & (\mathbf{pr}) \\ t = s[\#_{\Psi}(r\sigma)]_\pi & \text{if } \pi \in \mathrm{Pos}_{\mathcal{D}}(s) & (\mathbf{r}) \end{array}$$

with $\Psi = \{\varphi(\rho).\tau \mid \rho \in \mathrm{Pos}_{\mathcal{V}}(\ell),\ \varphi(\rho) \neq \bot,\ \rho.\tau \in \mathrm{Pos}_{\mathcal{D}^{\#}}(s|_\pi)\}$ and $\Phi = \mathrm{Pos}_{\mathcal{D}^{\#}}(r) \cup \Psi$.

So Ψ considers all positions of annotated symbols in $s|_\pi$ that are below positions ρ of variables in ℓ. If the VRF maps ρ to a variable position ρ' in r, then the annotations below $\pi.\rho$ in s are kept in the resulting subterm at position $\pi.\rho'$ after the rewriting.

Rewriting with $\mathcal{P}$ is like ordinary term rewriting, while considering and modifying annotations. Note that we represent a DP resulting from a rule as well as the original rule by just one ADP. So the ADP $\mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{D}(\mathsf{minus}(x, y), \mathsf{s}(y)))$ represents both the DP resulting from div in the right-hand side of the rule (4), and the rule (4) itself (by simply disregarding all annotations of the ADP).

Similar to the classical DP framework, our goal is to track specific reduction sequences. As before, there are **p**-steps where a DP is applied at the position of an annotated symbol. These steps may introduce new annotations. Moreover, between two **p**-steps there can be several **r**-steps.

A step of the form (**pr**) at position π in Def. 21 represents a **p**- or an **r**-step (or both), where an **r**-step is only possible if one later rewrites an annotated symbol at a position above π. All annotations are kept during this step except for annotations of subterms that correspond to variables of the applied rule. Here, the used VRF φ determines which of these annotations are kept and which are removed. As an example, with the canonical ADP $\mathsf{a}(x) \to \mathsf{b}(x)$ from $\mathcal{A}_1(\mathcal{R}_3)$ we can rewrite $\mathsf{A}(\mathsf{F}) \hookrightarrow_{\mathcal{A}_1(\mathcal{R}_3)} \mathsf{b}(\mathsf{F})$ as in Ex. 20. Here, we have $\pi = \varepsilon$, $\flat(s|_\varepsilon) = \mathsf{a}(\mathsf{f}) = \ell\sigma$, $r = \mathsf{b}(x)$, and the VRF φ with $\varphi(1) = 1$ such that the annotation of F in A's argument is kept in the argument of b.

[7] In [23] there were two additional cases in the definition of the corresponding rewrite relation. One of them was needed for processors that restrict the rules applicable for **r**-steps (e.g., based on usable rules), and the other case was needed to ensure that the innermost evaluation strategy is not affected by the application of ADP processors. This is unnecessary here since we consider full rewriting. On the other hand, VRFs are new compared to [23], since they are not needed for innermost rewriting.

A step of the form (**r**) rewrites at the position of a non-annotated defined symbol, and represents just an **r**-step. Hence, we remove all annotations from the right-hand side r of the ADP. However, we may have to keep the annotations inside the substitution, hence we move them according to the VRF. For example, we obtain the rewrite step $\mathsf{s}(\mathsf{D}(\underline{\mathsf{minus}(\mathsf{s}(0), \mathsf{s}(0))}, \mathsf{s}(0))) \hookrightarrow_{\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}})} \mathsf{s}(\mathsf{D}(\mathsf{minus}(0, 0), \mathsf{s}(0)))$ using the ADP $\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{M}(x, y)$ (15) and any VRF.

A *(relative) ADP problem* has the form $(\mathcal{P}, \mathcal{S})$, where $\mathcal{P}$ and $\mathcal{S}$ are finite sets of ADPs. $\mathcal{P}$ is the set of all main ADPs and $\mathcal{S}$ is the set of all base ADPs. Now we can define chains in the relative setting.

Definition 22 (Chains and Terminating ADP Problems). *Let $(\mathcal{P}, \mathcal{S})$ be an ADP problem. A sequence of terms $t_0, t_1, \ldots$ with $t_i \in \mathcal{T}\left(\Sigma^{\#}, \mathcal{V}\right)$ is a* $(\mathcal{P}, \mathcal{S})$-chain *if we have $t_i \hookrightarrow_{\mathcal{P} \cup \mathcal{S}} t_{i+1}$ for all $i \in \mathbb{N}$. The chain is called* infinite *if infinitely many of these rewrite steps use $\hookrightarrow_{\mathcal{P}}$ with Case* (**pr**)*. We say that an ADP problem $(\mathcal{P}, \mathcal{S})$ is* terminating (SN) *if there is no infinite $(\mathcal{P}, \mathcal{S})$-chain.*

Note the two different forms of relativity in Def. 22: In a finite chain, we may not only use infinitely many steps with $\mathcal{S}$ but also infinitely many steps with $\mathcal{P}$ where Case (**r**) applies. Thus, an ADP problem $(\mathcal{P}, \mathcal{S})$ without annotated symbols or without any main ADPs (i.e., where $\mathcal{P} = \varnothing$) is obviously SN. Finally, we obtain our desired chain criterion.

Theorem 23 (Chain Criterion for Relative Rewriting). *Let $\mathcal{R}$ and $\mathcal{B}$ be TRSs such that $\mathcal{B}$ is non-duplicating. Then $\mathcal{R}/\mathcal{B}$ is SN iff the ADP problem $(\mathcal{A}_1(\mathcal{R}), \mathcal{A}_2(\mathcal{B}))$ is SN.*

Example 24. The infinite rewrite sequence of Ex. 4 can be simulated by the following infinite chain using $\mathcal{A}_1(\mathcal{R}_2) = \{\mathsf{a} \to \mathsf{b}\}$ and $\mathcal{A}_2(\mathcal{B}_2) = \{\mathsf{f} \to \mathsf{d}(\mathsf{F}, \mathsf{A})\}$.

$$\underline{\mathsf{F}} \hookrightarrow_{\mathcal{A}_2(\mathcal{B}_2)} \mathsf{d}(\mathsf{F}, \underline{\mathsf{A}}) \hookrightarrow_{\mathcal{A}_1(\mathcal{R}_2)} \mathsf{d}(\underline{\mathsf{F}}, \mathsf{b}) \hookrightarrow_{\mathcal{A}_2(\mathcal{B}_2)} \mathsf{d}(\mathsf{d}(\mathsf{F}, \underline{\mathsf{A}}), \mathsf{b}) \hookrightarrow_{\mathcal{A}_1(\mathcal{R}_2)} \cdots$$

The steps with $\hookrightarrow_{\mathcal{A}_2(\mathcal{B}_2)}$ use Case (**pr**) at the position of the annotated symbol F and the steps with $\hookrightarrow_{\mathcal{A}_1(\mathcal{R}_2)}$ use (**pr**) as well. For this infinite chain, we indeed need two annotated symbols in the right-hand side of the base ADP: If A were not annotated (i.e., if we had the ADP $\mathsf{f} \to \mathsf{d}(\mathsf{F}, \mathsf{a})$), then the step with $\hookrightarrow_{\mathcal{A}_1(\mathcal{R}_2)}$ would just use Case (**r**) and the chain would not be considered "infinite". If F were not annotated (i.e., if we had the ADP $\mathsf{f} \to \mathsf{d}(\mathsf{f}, \mathsf{A})$), then we would have the step $\mathsf{f} \hookrightarrow_{\mathcal{A}_2(\mathcal{B}_2)} \mathsf{d}(\mathsf{f}, \mathsf{a})$ which uses Case (**r**) and removes all annotations from the right-hand side. Hence, again the chain would not be considered "infinite".

Example 25. The infinite rewrite sequence of Ex. 5 is simulated by the following chain with $\mathcal{A}_1(\mathcal{R}_3) = \{\mathsf{a}(x) \to \mathsf{b}(x)\}$ and $\mathcal{A}_2(\mathcal{B}_3) = \{\mathsf{f} \to \mathsf{A}(\mathsf{F})\}$.

$$\underline{\mathsf{F}} \hookrightarrow_{\mathcal{A}_2(\mathcal{B}_3)} \underline{\mathsf{A}}(\mathsf{F}) \hookrightarrow_{\mathcal{A}_1(\mathcal{R}_3)} \mathsf{b}(\underline{\mathsf{F}}) \hookrightarrow_{\mathcal{A}_2(\mathcal{B}_3)} \mathsf{b}(\underline{\mathsf{A}}(\mathsf{F})) \hookrightarrow_{\mathcal{A}_1(\mathcal{R}_3)} \mathsf{b}(\mathsf{b}(\underline{\mathsf{F}})) \hookrightarrow_{\mathcal{A}_2(\mathcal{B}_3)} \cdots$$

Here, it is important to use the VRF $\varphi(1) = 1$ for $\mathsf{a}(x) \to \mathsf{b}(x)$ which keeps the annotation of A's argument F when rewriting with $\mathcal{A}_1(\mathcal{R}_3)$, i.e., these steps must yield $\mathsf{b}(\mathsf{F})$ instead of $\mathsf{b}(\mathsf{f})$ to generate further subterms $\mathsf{A}(\ldots)$ afterwards.

5 The Relative ADP Framework

Now we present processors for our novel relative ADP framework. An *ADP processor* Proc has the form $\mathrm{Proc}(\mathcal{P},\mathcal{S}) = \{(\mathcal{P}_1,\mathcal{S}_1),\ldots,(\mathcal{P}_n,\mathcal{S}_n)\}$, where $\mathcal{P},\mathcal{P}_1,\ldots,\mathcal{P}_n$, $\mathcal{S}_1,\ldots,\mathcal{S}_n$ are sets of ADPs. Proc is *sound* if $(\mathcal{P},\mathcal{S})$ is SN whenever $(\mathcal{P}_i,\mathcal{S}_i)$ is SN for all $1 \leq i \leq n$. It is *complete* if $(\mathcal{P}_i,\mathcal{S}_i)$ is SN for all $1 \leq i \leq n$ whenever $(\mathcal{P},\mathcal{S})$ is SN. To prove relative termination of $\mathcal{R}/\mathcal{B}$, we start with the canonical ADP problem $(\mathcal{A}_1(\mathcal{R}),\mathcal{A}_2(\mathcal{B}))$ and apply sound (and preferably complete) ADP processors until all sub-problems are transformed to the empty set.

In Sect. 5.1, we present two processors to remove (base) ADPs, and in Sect. 5.2 and 5.3, we adapt the main processors of the classical DP framework from Sect. 3.1 to the relative setting. As mentioned, the soundness and completeness proofs for our processors and the chain criterion (Thm. 23) can be found in [24].

5.1 Derelatifying Processors

The following two *derelatifying* processors can be used to switch from ADPs to ordinary DPs, similar to Thm. 11 and 12. We extend $\flat$ to ADPs and sets of ADPs $\mathcal{S}$ by defining $\flat(\ell \to r) = \ell \to \flat(r)$ and $\flat(\mathcal{S}) = \{\ell \to \flat(r) \mid \ell \to r \in \mathcal{S}\}$.

If the ADPs in $\mathcal{S}$ contain no annotations anymore, then it suffices to use ordinary DPs. The corresponding set of DPs for a set of ADPs $\mathcal{P}$ is defined as $\mathrm{dp}(\mathcal{P}) = \{\ell^\# \to t^\# \mid \ell \to r \in \mathcal{P}, t \trianglelefteq_\# r\}$.

Theorem 26 (Derelatifying Processor (1)). *Let $(\mathcal{P},\mathcal{S})$ be an ADP problem such that $\flat(\mathcal{S}) = \mathcal{S}$. Then $\mathrm{Proc}_{\mathtt{DRP1}}(\mathcal{P},\mathcal{S}) = \varnothing$ is sound and complete iff the ordinary DP problem $(\mathrm{dp}(\mathcal{P}),\flat(\mathcal{P}\cup\mathcal{S}))$ is SN.*

Furthermore, similar to Thm. 12, we can always move ADPs from $\mathcal{S}$ to $\mathcal{P}$, but such a processor is only sound and not complete. However, it may help to satisfy the requirements of Thm. 26 by moving ADPs with annotations from $\mathcal{S}$ to $\mathcal{P}$ such that the ordinary DP framework can be used afterwards.

Theorem 27 (Derelatifying Processor (2)). *Let $(\mathcal{P},\mathcal{S})$ be an ADP problem, and let $\mathcal{S} = \mathcal{S}_a \uplus \mathcal{S}_b$. Then $\mathrm{Proc}_{\mathtt{DRP2}}(\mathcal{P},\mathcal{S}) = \{(\mathcal{P}\cup\mathtt{split}(\mathcal{S}_a),\mathcal{S}_b)\}$ is sound. Here,* $\mathtt{split}(\mathcal{S}_a) = \{\ell \to \#_\pi(r) \mid \ell \to r \in \mathcal{S}_a, \pi \in \mathrm{pos}_{\mathcal{D}^\#}(r)\}$.

So if $\mathcal{S}_a$ contains an ADP with two annotations, then we split it into two ADPs, where each only contains a single annotation.

Example 28. There are also redex-creating examples that are terminating, e.g., $\mathcal{R}_2 = \{\mathsf{a} \to \mathsf{b}\}$ and the base TRS $\mathcal{B}_2' = \{\mathsf{f}(\mathsf{s}(y)) \to \mathsf{d}(\mathsf{f}(y),\mathsf{a})\}$. Relative (and full) termination of this example can easily be shown by using the second derelatifying processor from Thm. 27 to replace the base ADP $\mathsf{f}(\mathsf{s}(y)) \to \mathsf{d}(\mathsf{F}(y),\mathsf{A})$ by the main ADPs $\mathsf{f}(\mathsf{s}(y)) \to \mathsf{d}(\mathsf{F}(y),\mathsf{a})$ and $\mathsf{f}(\mathsf{s}(y)) \to \mathsf{d}(\mathsf{f}(y),\mathsf{A})$. Then the processor of Thm. 26 is used to switch to the ordinary DPs $\mathsf{F}(\mathsf{s}(y)) \to \mathsf{F}(y)$ and $\mathsf{F}(\mathsf{s}(y)) \to \mathsf{A}$.

5.2 Relative Dependency Graph Processor

Next, we develop a dependency graph processor in the relative setting. The definition of the dependency graph is analogous to the one in the standard setting

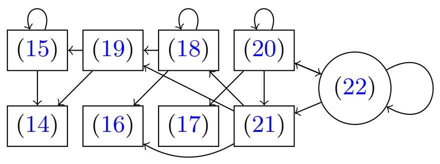


Fig. 1: $(\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}}), \mathcal{A}_2(\mathcal{B}_{\mathsf{mset2}}))$-Dep. Graph

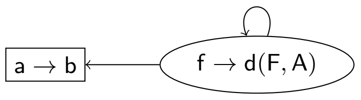


Fig. 2: $(\mathcal{A}_1(\mathcal{R}_2), \mathcal{A}_2(\mathcal{B}_2))$-Dep. Graph

and thus, the same techniques can be used to over-approximate it automatically.

Definition 29 (Relative Dependency Graph). *Let $(\mathcal{P}, \mathcal{S})$ be an ADP problem. The* $(\mathcal{P}, \mathcal{S})$-dependency graph *has the set of nodes $\mathcal{P} \cup \mathcal{S}$ and there is an edge from $\ell_1 \to r_1$ to $\ell_2 \to r_2$ if there exist substitutions σ_1, σ_2 and a term $t \trianglelefteq_\# r_1$ such that $t^\# \sigma_1 \to^*_{\flat(\mathcal{P}\cup\mathcal{S})} \ell_2^\# \sigma_2$.*

So similar to the standard dependency graph, there is an edge from an ADP $\ell_1 \to r_1$ to $\ell_2 \to r_2$ if the rules of $\flat(\mathcal{P} \cup \mathcal{S})$ (without annotations) can reduce an instance of a subterm t of r_1 to an instance of ℓ_2, if one only annotates the roots of t and ℓ_2 (i.e., then the rules can only be applied below the root).

Example 30. The dependency graph for the ADP problem $(\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}}), \mathcal{A}_2(\mathcal{B}_{\mathsf{mset2}}))$ from Ex. 18 is shown in Fig. 1. Here, nodes from $\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}})$ are denoted by rectangles and the node from $\mathcal{A}_2(\mathcal{B}_{\mathsf{mset2}})$ is a circle.

To detect possible ordinary infinite rewrite sequences as in Ex. 6, we again have to regard SCCs of the dependency graph, where we only need to consider SCCs that contain a node from $\mathcal{P}$, because otherwise, all steps in the SCC are relative (base) steps. However, in the relative ADP framework, non-termination can also be due to chains representing redex-creating sequences. Here, it does not suffice to look at SCCs. Thus, the relative dependency graph processor differs substantially from the corresponding processor for ordinary rewriting (and also from the corresponding processor for the probabilistic ADP framework in [23]).

Example 31 (Dependency Graph for Redex-Creating TRSs). For $\mathcal{R}_2$ and $\mathcal{B}_2$ from Ex. 4, the dependency graph for $(\mathcal{A}_1(\mathcal{R}_2), \mathcal{A}_2(\mathcal{B}_2))$ from Ex. 24 is in Fig. 2. Here, we cannot regard the SCC $\{\mathsf{f} \to \mathsf{d}(\mathsf{F}, \mathsf{A})\}$ separately, as we need $\mathcal{A}_1(\mathcal{R}_2)$'s rule $\mathsf{a} \to \mathsf{b}$ to reduce the created redex. To find the ADPs that can reduce the created redexes, we have to regard the outgoing paths from the SCCs of $\mathcal{S}$ to ADPs of $\mathcal{P}$.

The structure that we are looking for in the redex-creating case is a path from an SCC to a node from $\mathcal{P}$ (i.e., a form of a *lasso*), which is *minimal* in the sense that if we reach a node from $\mathcal{P}$, then we stop and do not move further along the edges of the graph. Moreover, the SCC needs to contain an ADP with more than one annotated symbol, as otherwise the generation of the infinitely many $\mathcal{P}$-redexes would not be possible. Here, it suffices to look at SCCs in the graph restricted to only $\mathcal{S}$-nodes (i.e., in the $(\flat(\mathcal{P}), \mathcal{S})$-dependency graph). The reason is that if the SCC contains a node from $\mathcal{P}$, then as mentioned above, we have to prove anyway that the SCC does not give rise to infinite chains.

Definition 32 ($\mathtt{SCC}^{(\mathcal{P},\mathcal{S})}_{\mathcal{P}'}$, $\mathtt{Lasso}$). *Let $(\mathcal{P},\mathcal{S})$ be an ADP problem. For any $\mathcal{P}' \subseteq \mathcal{P} \cup \mathcal{S}$, let $\mathtt{SCC}^{(\mathcal{P},\mathcal{S})}_{\mathcal{P}'}$ denote the set of all SCCs of the $(\mathcal{P},\mathcal{S})$-dependency graph that contain an ADP from $\mathcal{P}'$. Moreover, let $\mathcal{S}_{>1} \subseteq \mathcal{S}$ denote the set of all ADPs from $\mathcal{S}$ with more than one annotation. Then the set of all* minimal lassos *is defined as $\mathtt{Lasso} = \{\mathcal{Q} \cup \{n_1, \ldots, n_k\} \mid \mathcal{Q} \in \mathtt{SCC}^{(\flat(\mathcal{P}),\mathcal{S})}_{\mathcal{S}_{>1}}$, $n_1, \ldots, n_k$ is a path in the $(\flat(\mathcal{P}),\mathcal{S})$-dependency graph such that $n_1 \in \mathcal{Q}$ and $n_k \in \flat(\mathcal{P})\}$.*

We remove the annotations of ADPs which do not have to be considered anymore for **p**-steps due to the dependency graph, but we keep the ADPs for possible **r**-steps and thus, consider them as relative (base) ADPs.

Theorem 33 (Dep. Graph Processor). *Let $(\mathcal{P},\mathcal{S})$ be an ADP problem. Then*

$$\mathrm{Proc}_{\mathtt{DG}}(\mathcal{P},\mathcal{S}) = \{(\,\mathcal{P} \cap \mathcal{Q},\ (\mathcal{S} \cap \mathcal{Q}) \cup \flat(\,(\mathcal{P} \cup \mathcal{S}) \setminus \mathcal{Q}\,)\,) \mid \mathcal{Q} \in \mathtt{SCC}^{(\mathcal{P},\mathcal{S})}_{\mathcal{P}} \cup \mathtt{Lasso}\}$$

is sound and complete.

Example 34. For $(\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}}), \mathcal{A}_2(\mathcal{B}_{\mathsf{mset2}}))$ from Ex. 30 we have three SCCs $\{(15)\}$, $\{(18)\}$, and $\{(20),(22)\}$ containing nodes from $\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}})$. The set $\{(22)\}$ is the only SCC of $(\flat(\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}})), \mathcal{A}_2(\mathcal{B}_{\mathsf{mset2}}))$ and there are paths from that SCC to the ADPs (20) and (21) of $\mathcal{P}$. However, they are not in $\mathtt{Lasso}$, because the SCC $\{(22)\}$ does not contain an ADP with more than one annotation. Hence, we result in the three new ADP problems $(\{(15)\}, \{\flat(22)\} \cup \flat(\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}}) \setminus \{(15)\}))$, $(\{(18)\}, \{\flat(22)\} \cup \flat(\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}}) \setminus \{(18)\}))$, and $(\{(20)\}, \{(22)\} \cup \flat(\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}}) \setminus \{(20)\}))$. For the first two of these new ADP problems, we can use the derelatifying processor of Thm. 26 and prove SN via ordinary DPs, since their base ADPs do not contain any annotated symbols anymore.

The dependency graph processor in combination with the derelatifying processors of Thm. 26 and 27 already subsumes the techniques of Thm. 11 and 12. The reason is that if $\mathcal{R}$ dominates $\mathcal{B}$, then there is no edge from an ADP of $\mathcal{A}_2(\mathcal{B})$ to any ADP of $\mathcal{A}_1(\mathcal{R})$ in the $(\mathcal{A}_1(\mathcal{R}), \mathcal{A}_2(\mathcal{B}))$-dependency graph. Hence, there are no minimal lassos and the dependency graph processor just creates ADP problems from the SCCs of $\mathcal{A}_1(\mathcal{R})$ where the base ADPs do not have any annotations anymore. Then Thm. 26 allows us to switch to ordinary DPs. For example, if we consider $\mathcal{B}_{\mathsf{mset}}$ instead of $\mathcal{B}_{\mathsf{mset2}}$, then the dependency graph processor yields the three sub-problems for the SCCs $\{(15)\}$, $\{(18)\}$, and $\{(20)\}$, where the base ADPs do not contain annotations anymore. Then, we can move to ordinary DPs via Thm. 26.

Compared to Thm. 11 and 12, the dependency graph allows for more precise over-approximations than just "dominance" to detect when the base ADPs do not depend on the main ADPs. Moreover, the derelatifying processors of Thm. 26 and 27 allow us to switch to the ordinary DP framework also for sub-problems which result from the application of other processors of our relative ADP framework. In other words, Thm. 26 and 27 allow us to apply this switch in a modular way, even if their prerequisites do not hold for the initial canonical ADP problem (i.e., even if the prerequisites of Thm. 11 and 12 do not hold for the whole TRSs).

5.3 Relative Reduction Pair Processor

Next, we adapt the reduction pair processor to ADPs for relative rewriting. While the reduction pair processor for ADPs in the probabilistic setting [23] was restricted to polynomial interpretations, we now allow arbitrary reduction pairs using a similar idea as in [18] for complexity analysis via DPs.

To find out which ADPs cannot be used for infinitely many **p**-steps, the idea is not to compare the annotated left-hand side with the whole right-hand side, but just with the set of its annotated subterms. To combine these subterms in the case of ADPs with two or no annotated symbols, we extend the signature by two fresh *compound* symbols c_0 and c_2 of arity 0 and 2, respectively. Similar to [18], we have to use c-*monotonic* and c-*invariant* reduction pairs.

Definition 35 (c-Monotonic, c-Invariant). *For $r \in \mathcal{T}\left(\Sigma^\#, \mathcal{V}\right)$, we define* $\operatorname{ann}(r) = \mathsf{c}_0$ *if r does not contain any annotation,* $\operatorname{ann}(r) = t^\#$ *if $t \trianglelefteq_\# r$ and r only contains one annotated symbol, and* $\operatorname{ann}(r) = \mathsf{c}_2(r_1^\#, r_2^\#)$ *if $r_1 \trianglelefteq_\#^{\pi_1} r$, $r_2 \trianglelefteq_\#^{\pi_2} r$, and $\pi_1 <_{lex} \pi_2$ where $<_{lex}$ is the (total) lexicographic order on positions.*

A reduction pair $(\succsim, \succ)$ is called c-monotonic *if $\mathsf{c}_2(s_1, t) \succ \mathsf{c}_2(s_2, t)$ and $\mathsf{c}_2(t, s_1) \succ \mathsf{c}_2(t, s_2)$ for all $s_1, s_2, t \in \mathcal{T}\left(\Sigma^\#, \mathcal{V}\right)$ with $s_1 \succ s_2$. Moreover, it is* c-invariant *if $\mathsf{c}_2(s_1, s_2) \sim \mathsf{c}_2(s_2, s_1)$ and $\mathsf{c}_2(s_1, \mathsf{c}_2(s_2, s_3)) \sim \mathsf{c}_2(\mathsf{c}_2(s_1, s_2), s_3)$ for $\sim = \succsim \cap \precsim$ and all $s_1, s_2, s_3 \in \mathcal{T}\left(\Sigma^\#, \mathcal{V}\right)$.*

So for example, reduction pairs based on polynomial interpretations are c-monotonic and c-invariant if $\mathsf{c}_2(x, y)$ is interpreted by $x + y$.

For an ADP problem $(\mathcal{P}, \mathcal{S})$, now the reduction pair processor has to orient the non-annotated rules $\flat(\mathcal{P} \cup \mathcal{S})$ weakly and for all ADPs $\ell \to r$, it compares the annotated left-hand side $\ell^\#$ with $\operatorname{ann}(r)$. In strictly decreasing ADPs, one can then remove all annotations and consider them as relative (base) ADPs again.

Theorem 36 (Reduction Pair Processor). *Let $(\mathcal{P}, \mathcal{S})$ be an ADP problem and let $(\succsim, \succ)$ be a c-monotonic and c-invariant reduction pair such that $\flat(\mathcal{P} \cup \mathcal{S}) \subseteq \succsim$ and $\ell^\# \succsim \operatorname{ann}(r)$ for all $\ell \to r \in \mathcal{P} \cup \mathcal{S}$. Moreover, let $\mathcal{P}_\succ \subseteq \mathcal{P} \cup \mathcal{S}$ such that $\ell^\# \succ \operatorname{ann}(r)$ for all $\ell \to r \in \mathcal{P}_\succ$. Then $\operatorname{Proc}_{\mathsf{RPP}}(\mathcal{P}, \mathcal{S}) = \{(\mathcal{P} \setminus \mathcal{P}_\succ, (\mathcal{S} \setminus \mathcal{P}_\succ) \cup \flat(\mathcal{P}_\succ))\}$ is sound and complete.*

Example 37. For the remaining ADP problem $(\{(20)\}, \{(22)\} \cup \flat(\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}}) \setminus \{(20)\}))$ from Ex. 34, we can apply the reduction pair processor using the polynomial interpretation from the end of Sect. 3.1 which maps $\mathsf{0}$ and nil to 0, $\mathsf{s}(x)$ to $x + 1$, $\mathsf{cons}(y, xs)$ to $xs + 1$, $\mathsf{DL}(x, xs)$ to xs, and all other symbols to their first arguments. Then, (20) is oriented strictly (i.e., it is in $\mathcal{P}_\succ$), and (22) and all other base ADPs are oriented weakly. Hence, we remove the annotation from (20) and move it to the base ADPs. Now there is no main ADP anymore, and thus the dependency graph processor returns $\varnothing$. This proves SN for $(\mathcal{A}_1(\mathcal{R}_{\mathsf{divL}}), \mathcal{A}_2(\mathcal{B}_{\mathsf{mset2}}))$, hence $\mathcal{R}_{\mathsf{divL}} / \mathcal{B}_{\mathsf{mset2}}$ is also SN.

Example 38. Regard the ADPs $\mathsf{a} \to \mathsf{b}$ and $\mathsf{f} \to \mathsf{d}(\mathsf{F}, \mathsf{A})$ for the redex-creating Ex. 4 again. When using a polynomial interpretation Pol that maps c_0 to 0 and $\mathsf{c}_2(x, y)$ to $x + y$, then for the reduction pair processor one has to satisfy $\operatorname{Pol}(\mathsf{A}) \geq 0$ and $\operatorname{Pol}(\mathsf{F}) \geq \operatorname{Pol}(\mathsf{F}) + \operatorname{Pol}(\mathsf{A})$, i.e., one cannot make any of the ADPs strictly decreasing.

In contrast, for the variant with the terminating base rule $\mathsf{f}(\mathsf{s}(y)) \to \mathsf{d}(\mathsf{f}(y), \mathsf{a})$ from Ex. 28, we have the ADPs $\mathsf{a} \to \mathsf{b}$ and $\mathsf{f}(\mathsf{s}(y)) \to \mathsf{d}(\mathsf{F}(y), \mathsf{A})$. Here, the second constraint is $\mathrm{Pol}(\mathsf{F}(\mathsf{s}(y))) \geq \mathrm{Pol}(\mathsf{F}(y)) + \mathrm{Pol}(\mathsf{A})$. To make one of the ADPs strictly decreasing, one can set $\mathrm{Pol}(\mathsf{F}(x)) = x$, $\mathrm{Pol}(\mathsf{s}(x)) = x + 1$, and $\mathrm{Pol}(\mathsf{A}) = 1$ or $\mathrm{Pol}(\mathsf{A}) = 0$. Then the reduction pair processor removes the annotations from the strictly decreasing ADP and the dependency graph processor proves SN.

6 Evaluation and Conclusion

In this paper, we introduced the first notion of (annotated) dependency pairs and the first DP framework for relative termination, which also features suitable dependency graph and reduction pair processors for relative ADPs. Of course, further classical DP processors can be adapted to our relative ADP framework as well. For example, in our implementation of the novel ADP framework in our tool AProVE [13], we also included a straightforward adaption of the classical *rule removal processor* [11], see [24].[8] While the soundness proofs for the processors in the new relative ADP framework are more involved than in the standard DP framework, the new processors themselves are quite analogous to their original counterparts and thus, adapting an existing implementation of the ordinary DP framework to the relative ADP framework does not require much effort. In future work, we will investigate how to use our new form of ADPs for full (instead of innermost) rewriting also in the probabilistic setting and for complexity analysis.

To evaluate the new relative ADP framework, we compared its implementation in "*new* AProVE" to all tools that participated in the most recent *termination competition (TermComp 2023)* [14] on relative rewriting, i.e., NaTT [36], $\mathsf{T_TT_2}$ [27], MultumNonMulta [9], and "*old* AProVE" which did not yet contain the contributions of the current paper. In *TermComp 2023*, 98 benchmarks were used for relative termination. However, these benchmarks only consist of examples where the main TRS $\mathcal{R}$ dominates the base TRS $\mathcal{B}$ (i.e., which can be handled by Thm. 11 from [21]) or which can already be solved via simplification orders directly.

Therefore, we extended the collection by 32 new "typical" examples for relative rewriting, including both $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset}}$ from Ex. 1 and 2, and our leading example $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset2}}$ from Ex. 2 and 14 (where only *new* AProVE can prove SN). Except for $\mathcal{R}_{\mathsf{divL}}/\mathcal{B}_{\mathsf{mset}}$, in these examples $\mathcal{R}$ does not dominate $\mathcal{B}$. Most of these examples adapt well-known classical TRSs from the *Termination Problem Data Base* [33] used at *TermComp* to the relative setting. Moreover, 5 of our new examples illustrate the application of relative termination for proving confluence, i.e., in these examples one can prove confluence with the approach of [19] via our new technique for relative termination proofs.

[8] This processor works analogously to the preprocessing at the beginning of Sect. 4 which can be used to remove duplicating rules: For an ADP problem $(\mathcal{P}, \mathcal{S})$, it tries to find a reduction pair $(\succsim, \succ)$ where $\succ$ is closed under contexts such that $\flat(\mathcal{P} \cup \mathcal{S}) \subseteq \succsim$. Then for $\mathcal{P}_\succ \subseteq \mathcal{P} \cup \mathcal{S}$ with $\flat(\mathcal{P}_\succ) \subseteq \succ$, the processor replaces the ADP by $(\mathcal{P} \setminus \mathcal{P}_\succ, \mathcal{S} \setminus \mathcal{P}_\succ)$.

In the following table, the number in the "YES" ("NO") row indicates for how many of the 130 examples the respective tool could prove (disprove) relative termination and "MAYBE" refers to the benchmarks where the tool could not solve the problem within the timeout of 300 s per example. The numbers in brackets are the respective results when only considering our new 32 examples. "AVG(s)" gives the average runtime of the tool on solved examples in seconds.

	new AProVE	NaTT	*old* AProVE	$\mathsf{T_TT_2}$	MultumNonMulta
YES	91 (32)	68 (10)	48 (5)	39 (3)	0 (0)
NO	13 (0)	5 (0)	13 (0)	7 (0)	13 (0)
MAYBE	26 (0)	57 (22)	69 (27)	84 (29)	117 (32)
AVG(s)	5.11	0.41	4.02	1.67	1.60

The table clearly shows that while *old* AProVE was already the second most powerful tool for relative termination, the integration of the ADP framework in *new* AProVE yields a substantial advance in power (i.e., it only fails on 26 of the examples, compared to 57 and 69 failures of NaTT and *old* AProVE, respectively). In particular, previous tools (including *old* AProVE) often have problems with relative TRSs where the main TRS does not dominate the base TRS, whereas the ADP framework can handle such examples.

A special form of relative TRSs are *relative string rewrite systems (SRSs)*, where all function symbols have arity 1. Due to the base ADPs with two annotated symbols on the right-hand side, here the ADP framework is less powerful than dedicated techniques for string rewriting. For the 403 relative SRSs at *TermComp 2023*, the ADP framework only finds 71 proofs, mostly due to the dependency graph and the rule removal processor, while termination analysis via AProVE's standard strategy for relative SRSs succeeds on 209 examples, and the two most powerful tools for relative SRSs at *TermComp 2023* (MultumNonMulta and Matchbox [35]) succeed on 274 and 269 examples, respectively.

Another special form of relative rewriting is *equational rewriting*, where one has a set of equations E which correspond to relative rules that can be applied in both directions. In [10], DPs were adapted to equational rewriting. However, this approach requires E-unification to be decidable and finitary (i.e., for (certain) pairs of terms, it has to compute finite complete sets of E-unifiers). This works well if E are AC- or C-axioms, and for this special case, dedicated techniques like [10] are more powerful than our new ADP framework for relative termination. For example, on the 76 AC- and C-benchmarks for equational rewriting at *TermComp 2023*, the relative ADP framework finds 36 proofs, while dedicated tools for AC-rewriting like AProVE's equational strategy or MU-TERM [15] succeed on 66 and 64 examples, respectively. However, in general, the requirement of a finitary E-unification algorithm is a hard restriction. In contrast to existing tools for equational rewriting, our new ADP framework can be used for arbitrary (non-duplicating) relative rules.

For details on our experiments, our collection of examples, and for instructions on how to run our implementation in AProVE via its *web interface* or locally, see: https://aprove-developers.github.io/RelativeDTFramework/

References

[1] T. Arts and J. Giesl. "Termination of Term Rewriting Using Dependency Pairs". In: *Theoretical Computer Science* 236.1-2 (2000), pp. 133–178. DOI: 10.1016/S0304-3975(99)00207-8.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.

[3] T. Baudon, C. Fuhs, and L. Gonnord. "Analysing Parallel Complexity of Term Rewriting". In: *Proc. LOPSTR '22*. LNCS 13474. 2022, pp. 3–23. DOI: 10.1007/978-3-031-16767-6_1.

[4] N. Dershowitz. "Termination of Rewriting". In: *Journal of Symbolic Computation* 3.1 (1987), pp. 69–115. DOI: https://doi.org/10.1016/S0747-7171(87)80022-6.

[5] N. Dershowitz. *The RTA List of Open Problems*. URL: https://www.cs.tau.ac.il/~nachum/rtaloop/.

[6] F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder. "Lower Bounds for Runtime Complexity of Term Rewriting". In: *Journal of Automated Reasoning* 59.1 (2017), pp. 121–163. DOI: 10.1007/S10817-016-9397-X.

[7] C. Fuhs. "Transforming Derivational Complexity of Term Rewriting to Runtime Complexity". In: *Proc. FroCoS '19*. LNCS 11715. 2019, pp. 348–364. DOI: 10.1007/978-3-030-29007-8_20.

[8] A. Geser. "Relative Termination". PhD thesis. University of Passau, Germany, 1990. URL: https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui/Ulmer_Informatik_Berichte/1991/UIB-1991-03.pdf.

[9] A. Geser, D. Hofbauer, and J. Waldmann. "Sparse Tiling Through Overlap Closures for Termination of String Rewriting". In: *Proc. FSCD '19*. LIPIcs 131. 2019, 21:1–21:21. DOI: 10.4230/LIPICS.FSCD.2019.21.

[10] J. Giesl and D. Kapur. "Dependency Pairs for Equational Rewriting". In: *Proc. RTA '01*. LNCS 2051. 2001, pp. 93–108. DOI: 10.1007/3-540-45127-7_9.

[11] J. Giesl, R. Thiemann, and P. Schneider-Kamp. "The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs". In: *Proc. LPAR '04*. LNCS 3452. 2004, pp. 301–331. DOI: 10.1007/978-3-540-32275-7_21.

[12] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. "Mechanizing and Improving Dependency Pairs". In: *Journal of Automated Reasoning* 37.3 (2006), pp. 155–203. DOI: 10.1007/s10817-006-9057-7.

[13] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. "Analyzing Program Termination and Complexity Automatically with AProVE". In: *Journal of Automated Reasoning* 58.1 (2017), pp. 3–31. DOI: 10.1007/s10817-016-9388-y.

[14] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. "The Termination and Complexity Competition". In: *Proc. TACAS '19*. LNCS 11429. Website of *TermComp*: https://termination-portal.org/wiki/

Termination_Competition. 2019, pp. 156–166. DOI: 10.1007/978-3-030-17502-3_10.

[15] R. Gutiérrez and S. Lucas. "MU-TERM: Verify Termination Properties Automatically (System Description)". In: *Proc. IJCAR '20*. LNCS 12167. 2020, pp. 436–447. DOI: 10.1007/978-3-030-51054-1_28.

[16] N. Hirokawa and A. Middeldorp. "Automating the Dependency Pair Method". In: *Information and Computation* 199.1-2 (2005), pp. 172–199. DOI: 10.1016/j.ic.2004.10.004.

[17] N. Hirokawa and A. Middeldorp. "Tyrolean Termination Tool: Techniques and Features". In: *Information and Computation* 205.4 (2007), pp. 474–511. DOI: 10.1016/J.IC.2006.08.010.

[18] N. Hirokawa and G. Moser. "Automated Complexity Analysis Based on the Dependency Pair Method". In: *Proc. IJCAR '08*. LNCS 5195. 2008, pp. 364–379. DOI: 10.1007/978-3-540-71070-7_32.

[19] N. Hirokawa and A. Middeldorp. "Decreasing Diagrams and Relative Termination". In: *Journal of Automated Reasoning* 47.4 (2011), pp. 481–501. DOI: 10.1007/S10817-011-9238-X.

[20] J. Iborra, N. Nishida, and G. Vidal. "Goal-Directed and Relative Dependency Pairs for Proving the Termination of Narrowing". In: *Proc. LOPSTR '09*. LNCS 6037. 2009, pp. 52–66. DOI: 10.1007/978-3-642-12592-8_5.

[21] J. Iborra, N. Nishida, G. Vidal, and A. Yamada. "Relative Termination via Dependency Pairs". In: *Journal of Automated Reasoning* 58.3 (2017), pp. 391–411. DOI: 10.1007/S10817-016-9373-5.

[22] J.-C. Kassing and J. Giesl. "Proving Almost-Sure Innermost Termination of Probabilistic Term Rewriting Using Dependency Pairs". In: *Proc. CADE '23*. LNCS 14132. 2023, pp. 344–364. DOI: 10.1007/978-3-031-38499-8_20.

[23] J.-C. Kassing, S. Dollase, and J. Giesl. "A Complete Dependency Pair Framework for Almost-Sure Innermost Termination of Probabilistic Term Rewriting". In: *Proc. FLOPS '24*. LNCS 14659. To appear. Long version at *CoRR* abs/2309.00344. 2024. DOI: 10.48550/arXiv.2309.00344.

[24] J.-C. Kassing, G. Vartanyan, and J. Giesl. "A Dependency Pair Framework for Relative Termination of Term Rewriting". In: *CoRR* abs/2404.15248 (2024). DOI: 10.48550/arXiv.2404.15248.

[25] D. Klein and N. Hirokawa. "Confluence of Non-Left-Linear TRSs via Relative Termination". In: *Proc. LPAR '18*. LNCS 7180. 2012, pp. 258–273. DOI: 10.1007/978-3-642-28717-6_21.

[26] A. Koprowski and H. Zantema. "Proving Liveness with Fairness Using Rewriting". In: *Proc. FroCoS '05*. LNCS 3717. 2005, pp. 232–247. DOI: 10.1007/11559306_13.

[27] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. "Tyrolean Termination Tool 2". In: *Proc. RTA '09*. LNCS 5595. 2009, pp. 295–304. DOI: 10.1007/978-3-642-02348-4_21.

[28] D. S. Lankford. *On Proving Term Rewriting Systems are Noetherian*. Memo MTP-3, Math. Dept., Louisiana Technical University, Ruston, LA, 1979.

URL: https://www.ens-lyon.fr/LIP/REWRITING/TERMINATION/Lankford_Poly_Term.pdf.

[29] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl. “Complexity Analysis for Term Rewriting by Integer Transition Systems”. In: *Proc. FroCoS ’17*. LNCS 10483. 2017, pp. 132–150. DOI: 10.1007/978-3-319-66167-4_8.

[30] J. Nagele, B. Felgenhauer, and H. Zankl. “Certifying Confluence Proofs via Relative Termination and Rule Labeling”. In: *Logical Methods in Computer Science* 13.2 (2017). DOI: 10.23638/LMCS-13(2:4)2017.

[31] N. Nishida and G. Vidal. “Termination of Narrowing via Termination of Rewriting”. In: *Applicable Algebra in Engineering, Communication and Computing* 21.3 (2010), pp. 177–225. DOI: 10.1007/S00200-010-0122-4.

[32] L. Noschinski, F. Emmes, and J. Giesl. “Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs”. In: *Journal of Automated Reasoning* 51 (2013), pp. 27–56. DOI: 10.1007/978-3-642-22438-6_32.

[33] TPDB (Termination Problem Data Base). URL: https://github.com/TermCOMP/TPDB.

[34] G. Vidal. “Termination of Narrowing in Left-Linear Constructor Systems”. In: *Proc. FLOPS ’08*. LNCS 4989. 2008, pp. 113–129. DOI: 10.1007/978-3-540-78969-7_10.

[35] J. Waldmann. “Matchbox: A Tool for Match-Bounded String Rewriting”. In: *Proc. RTA ’04*. LNCS 3091. 2004, pp. 85–94. DOI: 10.1007/978-3-540-25979-4_6.

[36] A. Yamada, K. Kusakari, and T. Sakabe. “Nagoya Termination Tool”. In: *Proc. RTA-TLCA ’14*. LNCS 8560. 2014, pp. 466–475. DOI: 10.1007/978-3-319-08918-8_32.

[37] H. Zankl and M. Korp. “Modular Complexity Analysis for Term Rewriting”. In: *Logical Methods in Computer Science* 10.1 (2014). DOI: 10.2168/LMCS-10(1:19)2014.