Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Proving Termination by Dependency Pairs and Inductive Theorem Proving

Carsten Fuhs · Jürgen Giesl · Michael Parting · Peter Schneider-Kamp · Stephan Swiderski

the date of receipt and acceptance should be inserted later

Abstract Current techniques and tools for automated termination analysis of term rewrite systems (TRSs) are already very powerful. However, they fail for algorithms whose termination is essentially due to an *inductive* argument. Therefore, we show how to couple the *dependency pair* method for termination of TRSs with inductive theorem proving. As confirmed by the implementation of our new approach in the tool AProVE, now TRS termination techniques are also successful on this important class of algorithms.

1 Introduction

There are many powerful techniques and tools to prove termination of TRSs automatically. Moreover, tools from term rewriting are also very successful in termination analysis of real programming languages like, e.g., Haskell, Java, and Prolog [18,32,37]. To measure their performance, there is an annual international *Termination Competition*,[1] where the tools compete on a large data base of TRSs. Nevertheless, there exist natural algorithms like the following one where all these tools fail.

Example 1 Consider the following TRS $\mathcal{R}_{sort}$.

$$\begin{aligned} \mathsf{ge}(x, \mathsf{0}) &\to \mathsf{true} \\ \mathsf{ge}(\mathsf{0}, \mathsf{s}(y)) &\to \mathsf{false} \\ \mathsf{ge}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{ge}(x, y) \end{aligned}$$

Supported by the DFG grant GI 274/5-3, the DFG Research Training Group 1298 (*AlgoSyn*), the G.I.F. grant 966-116.6, and the Danish Natural Science Research Council.

Carsten Fuhs · Jürgen Giesl · Michael Parting · Stephan Swiderski
LuFG Informatik 2, RWTH Aachen University, Germany

Peter Schneider-Kamp
Dept. of Mathematics & Computer Science, University of Southern Denmark, Denmark

[1] `http://termination-portal.org/wiki/Termination_Competition`

$$
\begin{aligned}
\mathsf{eql}(0,0) &\to \mathsf{true}\\
\mathsf{eql}(\mathsf{s}(x),0) &\to \mathsf{false}\\
\mathsf{eql}(0,\mathsf{s}(y)) &\to \mathsf{false}\\
\mathsf{eql}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{eql}(x,y)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{max}(\mathsf{empty}) &\to 0\\
\mathsf{max}(\mathsf{add}(x,\mathsf{empty})) &\to x\\
\mathsf{max}(\mathsf{add}(x,\mathsf{add}(y,xs))) &\to \mathsf{if}_1(\mathsf{ge}(x,y),x,y,xs)\\
\mathsf{if}_1(\mathsf{true},x,y,xs) &\to \mathsf{max}(\mathsf{add}(x,xs))\\
\mathsf{if}_1(\mathsf{false},x,y,xs) &\to \mathsf{max}(\mathsf{add}(y,xs))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{del}(x,\mathsf{empty}) &\to \mathsf{empty}\\
\mathsf{del}(x,\mathsf{add}(y,xs)) &\to \mathsf{if}_2(\mathsf{eql}(x,y),x,y,xs)\\
\mathsf{if}_2(\mathsf{true},x,y,xs) &\to xs\\
\mathsf{if}_2(\mathsf{false},x,y,xs) &\to \mathsf{add}(y,\mathsf{del}(x,xs))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{sort}(\mathsf{empty}) &\to \mathsf{empty}\\
\mathsf{sort}(\mathsf{add}(x,xs)) &\to \mathsf{add}(\mathsf{max}(\mathsf{add}(x,xs)),\ \mathsf{sort}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x,xs)),\mathsf{add}(x,xs))))
\end{aligned}
$$

Here, numbers are represented with $\mathsf{0}$ *and* s *(for the successor function) and lists are represented with* empty *(for the empty list) and* add *(for list insertion).*[2] *For any list* xs, $\mathsf{max}(xs)$ *computes its maximum (where* $\mathsf{max}(\mathsf{empty})$ *is* $\mathsf{0}$*), and* $\mathsf{del}(n,xs)$ *deletes the first occurrence of* n *from the list* xs*. If* n *does not occur in* xs*, then* $\mathsf{del}(n,xs)$ *returns* xs*. Algorithms like* max *and* del *are often expressed with conditions. Such conditional rules can be automatically transformed into unconditional ones (cf. e.g. [31]) and we already did this transformation in our example. To sort a non-empty list* ys *(i.e., a list of the form "*$\mathsf{add}(x,xs)$*"),* $\mathsf{sort}(ys)$ *reduces to "*$\mathsf{add}(\mathsf{max}(ys),\mathsf{sort}(\mathsf{del}(\mathsf{max}(ys),ys)))$*". So* $\mathsf{sort}(ys)$ *starts with the maximum of* ys *and then* sort *is called recursively on the list that results from* ys *by deleting the first occurrence of its maximum. Note that*

$$\textit{every non-empty list contains its maximum.} \tag{1}$$

Hence, the list $\mathsf{del}(\mathsf{max}(ys),ys)$ *is shorter than* ys *and thus,* $\mathcal{R}_{sort}$ *is terminating.*

So (1) is the main argument needed for termination of $\mathcal{R}_{sort}$. Thus, when trying to prove termination of TRSs like $\mathcal{R}_{sort}$ automatically, one faces two problems:

(a) One has to detect the main argument needed for termination and one has to find out that the TRS is terminating provided that this argument is valid.
(b) One has to prove that the argument detected in (a) is valid.

In our example, (1) requires a non-trivial induction proof that relies on the max- and del-rules. Such proofs cannot be done by TRS termination techniques, but they could be performed by state-of-the-art inductive theorem provers [5,6,8,9,22,24,41,43,46]. So to solve Problem (b), we would like to couple termination techniques for TRSs (like the *dependency pair* (DP) method which is implemented in virtually every current

[2] We use the name "add" instead of the more common name "cons" to avoid confusion with the pre-defined function symbol "cons" in Lisp and ACL2, cf. Section 5.2.

TRS termination tool) with an inductive theorem prover. Ideally, this prover should perform the validity proof in (b) fully automatically, but of course it is also possible to have user interaction here. However, it still remains to solve Problem (a). Thus, one has to extend the TRS termination techniques such that they can automatically synthesize an argument like (1) and find out that this argument is sufficient in order to complete the termination proof. This is the subject of the current paper.

There is already work on applying inductive reasoning in termination proofs. Some approaches integrate special forms of inductive reasoning into the termination method itself (e.g., to handle algorithms that increase arguments [7,17] or to prove well-foundedness of evaluation using special termination graphs [19,33]). These approaches are successful on certain forms of algorithms, but they cannot handle examples like Example 1 where one needs more general forms of inductive reasoning. Therefore, in this paper our goal is to couple the termination method with an arbitrary (black box) inductive theorem prover which may use any kind of proof techniques.

There exist also approaches where a full inductive theorem prover like Nqthm, ACL2, or Isabelle is used to perform the whole termination proof of a functional program [6,13,26,29,42]. Such approaches could potentially handle algorithms like Example 1 and indeed, Example 1 is similar to an algorithm from [13,42]. In general, to prove termination one has to solve two tasks:

(i) one has to synthesize suitable well-founded orders and
(ii) one has to prove that recursive calls decrease w.r.t. these orders.

If there is just an inductive theorem prover available for the termination proof, then for Task (i) one can only use a fixed small set of orders or otherwise ask the user to provide suitable well-founded orders manually. Moreover, then Task (ii) has to be tackled by the full theorem prover which may often pose problems for automation. In contrast, there are many TRS techniques and tools available that are extremely powerful for Task (i) and that offer several specialized methods to perform Task (ii) fully automatically in a very efficient way. So in most cases, no inductive theorem prover is needed for Task (ii). Nevertheless, there exist important algorithms (like $\mathcal{R}_{sort}$) where Task (ii) indeed requires inductive theorem proving. Thus, we propose to use the "best of both worlds", i.e., to apply TRS techniques whenever possible, but to use an inductive theorem prover for those parts where it is needed.

After recapitulating the DP method in Section 2, in Section 3 we present the main idea for our improvement. To make this improvement powerful in practice, we need the new result that innermost termination of many-sorted term rewriting and of unsorted term rewriting is equivalent. We expect that this observation will be useful also for other applications in term rewriting, since TRSs are usually considered to be unsorted. We use this result in Section 4 where we show how the DP method can be coupled with inductive theorem proving in order to prove termination of TRSs like $\mathcal{R}_{sort}$ automatically.

In Section 5, we discuss how to automate our new technique efficiently. The problem is that the termination argument (which has to be proved by induction) depends on the chosen underlying well-founded order. Hence, the question is how to find a suitable order such that the resulting termination argument is valid and can be proved by an inductive theorem prover. To evaluate our contributions empirically, we implemented our new technique in the termination prover AProVE [15]. In our experiments, we coupled this termination tool with two different inductive theorem provers. One of them was the well-known ACL2 system [24]. ACL2 is a very powerful theorem prover

which also has strong support for automating induction. In general, ACL2's proofs have to be guided by the user who has to formulate appropriate lemmas. However, our experiments indicate that the inductive conjectures that have to be verified for termination proofs are usually not too complex and thus in our experiments, ACL2 could prove all of them fully automatically. In addition to the experiments with ACL2, we also performed experiments where we used our "own" small inductive theorem prover. This prover was inspired by existing tools [6,8,24,41,43,46] and had already been implemented in AProVE before. Although this inductive theorem prover is less powerful than the more elaborated full theorem provers like ACL2, it already sufficed for those inductive arguments that were arising during the termination proofs in our experiments. So the results of this paper indeed allow to couple *any* termination prover implementing DPs with *any* inductive theorem prover.

A preliminary version of this paper was published in [38]. However, the present paper extends [38] substantially (e.g., by a new self-contained proof for the equivalence of many-sorted and unsorted innermost termination (Theorem 7), by the new Section 5.1 on the automation of our technique, by the new coupling of AProVE with ACL2 (Section 5.2) and new experiments in Section 5.3 to evaluate this coupling, and by more examples and more detailed explanations throughout the paper).

2 Dependency Pairs

We assume familiarity with term rewriting [3] and briefly recapitulate the DP method. See e.g. [2,14,16,20,21] for further motivations and extensions.

Definition 2 (Dependency Pairs) For a TRS $\mathcal{R}$, the *defined* symbols $\mathcal{D}_\mathcal{R}$ are the root symbols of left-hand sides of rules. All other function symbols are called *constructors*. For every defined symbol $f \in \mathcal{D}_\mathcal{R}$, we introduce a fresh *tuple symbol* $f^\sharp$ with the same arity. To ease readability, we often write F instead of $f^\sharp$, etc. If $t = f(t_1, \ldots, t_n)$ with $f \in \mathcal{D}_\mathcal{R}$, we write $t^\sharp$ for $f^\sharp(t_1, \ldots, t_n)$. If $\ell \to r \in \mathcal{R}$ and t is a subterm of r with defined root symbol, then the rule $\ell^\sharp \to t^\sharp$ is a *dependency pair* of $\mathcal{R}$. The set of all dependency pairs of $\mathcal{R}$ is denoted $DP(\mathcal{R})$.

For our running example $\mathcal{R}_{sort}$, we obtain the following set $DP(\mathcal{R}_{sort})$, where GE is ge's tuple symbol, etc.

$$\mathsf{GE}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{GE}(x, y) \tag{2}$$
$$\mathsf{EQL}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{EQL}(x, y) \tag{3}$$
$$\mathsf{MAX}(\mathsf{add}(x, \mathsf{add}(y, xs))) \to \mathsf{IF}_1(\mathsf{ge}(x, y), x, y, xs) \tag{4}$$
$$\mathsf{MAX}(\mathsf{add}(x, \mathsf{add}(y, xs))) \to \mathsf{GE}(x, y) \tag{5}$$
$$\mathsf{IF}_1(\mathsf{true}, x, y, xs) \to \mathsf{MAX}(\mathsf{add}(x, xs)) \tag{6}$$
$$\mathsf{IF}_1(\mathsf{false}, x, y, xs) \to \mathsf{MAX}(\mathsf{add}(y, xs)) \tag{7}$$
$$\mathsf{DEL}(x, \mathsf{add}(y, xs)) \to \mathsf{IF}_2(\mathsf{eql}(x, y), x, y, xs) \tag{8}$$
$$\mathsf{DEL}(x, \mathsf{add}(y, xs)) \to \mathsf{EQL}(x, y) \tag{9}$$
$$\mathsf{IF}_2(\mathsf{false}, x, y, xs) \to \mathsf{DEL}(x, xs) \tag{10}$$
$$\mathsf{SORT}(\mathsf{add}(x, xs)) \to \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x, xs)), \mathsf{add}(x, xs))) \tag{11}$$
$$\mathsf{SORT}(\mathsf{add}(x, xs)) \to \mathsf{DEL}(\mathsf{max}(\mathsf{add}(x, xs)), \mathsf{add}(x, xs)) \tag{12}$$
$$\mathsf{SORT}(\mathsf{add}(x, xs)) \to \mathsf{MAX}(\mathsf{add}(x, xs)) \tag{13}$$

In this paper, we only regard the *innermost* rewrite relation $\xrightarrow{\mathsf{i}}$ and prove innermost termination, since techniques for innermost termination are considerably more powerful than those for full termination. For large classes of TRSs (e.g., TRSs resulting from programming languages [18,37] or non-overlapping TRSs like Example 1), innermost termination is sufficient for termination.

For two TRSs $\mathcal{P}$ and $\mathcal{R}$ (where $\mathcal{P}$ usually consists of DPs), an *innermost* $(\mathcal{P},\mathcal{R})$-*chain* is a sequence of (variable-renamed) pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ from $\mathcal{P}$ such that there is a substitution σ (with possibly infinite domain) where $t_i\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}}^{*} s_{i+1}\sigma$ and $s_i\sigma$ is in normal form w.r.t. $\mathcal{R}$, for all i.[3] The main result on DPs states that $\mathcal{R}$ is innermost terminating iff there is no infinite innermost $(DP(\mathcal{R}),\mathcal{R})$-chain.

As an example for a chain, consider "(11), (11)", i.e.,

$$\mathsf{SORT}(\mathsf{add}(x\ ,xs\ )) \to \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x\ ,xs\ )),\mathsf{add}(x\ ,xs\ ))),$$
$$\mathsf{SORT}(\mathsf{add}(x',xs')) \to \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x',xs')),\mathsf{add}(x',xs'))).$$

Indeed, if $\sigma(x) = \sigma(x') = \mathsf{0}$, $\sigma(xs) = \mathsf{add}(\mathsf{s}(\mathsf{0}),\mathsf{empty})$, and $\sigma(xs') = \mathsf{empty}$, then

$$\mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x,xs)),\mathsf{add}(x,xs)))\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}_{sort}}^{!} \mathsf{SORT}(\mathsf{add}(x',xs'))\sigma,$$

where "$\xrightarrow{\mathsf{i}}_{\mathcal{R}_{sort}}^{!}$" denotes zero or more reduction steps to a normal form.

Termination techniques are now called *DP processors* and they operate on pairs of TRSs $(\mathcal{P},\mathcal{R})$ (which are called *DP problems*).[4] Formally, a DP processor *Proc* takes a DP problem as input and returns a set of new DP problems which then have to be solved instead. A processor *Proc* is *sound* if for all DP problems $(\mathcal{P},\mathcal{R})$ with an infinite innermost $(\mathcal{P},\mathcal{R})$-chain there is also a $(\mathcal{P}',\mathcal{R}') \in \mathit{Proc}(\,(\mathcal{P},\mathcal{R})\,)$ with an infinite innermost $(\mathcal{P}',\mathcal{R}')$-chain. Soundness of a DP processor is required to prove innermost termination and in particular, to conclude that there is no infinite innermost $(\mathcal{P},\mathcal{R})$-chain if $\mathit{Proc}(\,(\mathcal{P},\mathcal{R})\,) = \varnothing$.

So innermost termination proofs in the DP framework start with the initial problem $(DP(\mathcal{R}),\mathcal{R})$. Then the problem is simplified repeatedly by sound DP processors. If all DP problems have been simplified to $\varnothing$, then innermost termination is proved. Theorems 3-5 recapitulate three of the most important processors.

Theorem 3 allows us to replace the TRS $\mathcal{R}$ in a DP problem $(\mathcal{P},\mathcal{R})$ by the *usable rules*. These include all rules that can be used to reduce the terms in right-hand sides of $\mathcal{P}$ when their variables are instantiated with normal forms.

Theorem 3 (Usable Rule Processor [2,14]) *Let $\mathcal{R}$ be a TRS. For any function symbol f, let $\mathit{Rls}(f) = \{\ell \to r \in \mathcal{R} \mid \mathrm{root}(\ell) = f\}$. For any term t, the* usable rules *$\mathcal{U}(t)$ are the smallest set such that*

- *$\mathcal{U}(x) = \varnothing$ for every variable x and*
- *$\mathcal{U}(f(t_1,\ldots,t_n)) = \mathit{Rls}(f) \cup \bigcup_{\ell\to r\in \mathit{Rls}(f)} \mathcal{U}(r) \cup \bigcup_{1\le i\le n} \mathcal{U}(t_i)$*

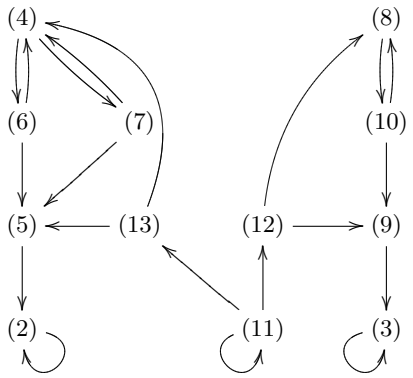
For a TRS $\mathcal{P}$, its usable rules *are $\mathcal{U}(\mathcal{P}) = \bigcup_{s\to t\in\mathcal{P}} \mathcal{U}(t)$. Then the following DP processor Proc is sound: $\mathit{Proc}(\,(\mathcal{P},\mathcal{R})\,) = \{\,(\mathcal{P},\,\mathcal{U}(\mathcal{P}))\,\}$.*

[3] All results of the present paper also hold if one regards *minimal* instead of ordinary innermost chains, i.e., chains where all $t_i\sigma$ are innermost terminating.

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

In Example 1, this processor transforms the initial DP problem $(DP(\mathcal{R}_{sort}), \mathcal{R}_{sort})$ into $(DP(\mathcal{R}_{sort}), \mathcal{R}'_{sort})$. $\mathcal{R}'_{sort}$ is $\mathcal{R}_{sort}$ without the two sort-rules, since sort does not occur in the right-hand side of any DP and thus, its rules are not usable.

The next processor decomposes a DP problem into sub-problems. To this end, one determines which pairs follow each other in innermost chains by constructing an *innermost dependency graph*. For a DP problem $(\mathcal{P}, \mathcal{R})$, the nodes of the innermost dependency graph are the pairs of $\mathcal{P}$, and there is an arc from $s \to t$ to $v \to w$ iff $s \to t, v \to w$ is an innermost $(\mathcal{P}, \mathcal{R})$-chain. In our example, we obtain the following graph.



In general, the innermost dependency graph is not computable, but there exist many techniques to over-approximate this graph automatically, cf. e.g. [2, 20, 25]. In our example, these estimations would even yield the exact innermost dependency graph.

A set $\mathcal{P}' \neq \varnothing$ of DPs is a *cycle* if for every $s \to t, v \to w \in \mathcal{P}'$, there is a non-empty path from $s \to t$ to $v \to w$ traversing only pairs of $\mathcal{P}'$.[5] A cycle $\mathcal{P}'$ is a (non-trivial) *strongly connected component (SCC)* if $\mathcal{P}'$ is not a proper subset of another cycle. The next processor allows us to prove termination separately for each SCC.

Theorem 4 (Dependency Graph Processor [2,14]) *The following DP processor Proc is sound: Proc*$(\,(\mathcal{P}, \mathcal{R})\,) = \{(\mathcal{P}_1, \mathcal{R}), \ldots, (\mathcal{P}_n, \mathcal{R})\}$*, where* $\mathcal{P}_1, \ldots, \mathcal{P}_n$ *are the SCCs of the innermost dependency graph.*

Our graph has the SCCs $\mathcal{P}_1 = \{(2)\}$, $\mathcal{P}_2 = \{(3)\}$, $\mathcal{P}_3 = \{(4), (6), (7)\}$, $\mathcal{P}_4 = \{(8), (10)\}$, and $\mathcal{P}_5 = \{(11)\}$. Thus, $(DP(\mathcal{R}_{sort}), \mathcal{R}'_{sort})$ is transformed into the five new DP problems $(\mathcal{P}_i, \mathcal{R}'_{sort})$ for $1 \leq i \leq 5$ that have to be solved instead. For all problems except $(\{(11)\}, \mathcal{R}'_{sort})$ this is easily possible by the DP processors of this section (and this can also be done automatically by current termination tools). Therefore, we now concentrate on the remaining DP problem $(\{(11)\}, \mathcal{R}'_{sort})$.

The following processor uses so-called reduction pairs to remove dependency pairs from $\mathcal{P}$. A *reduction pair* $(\succsim, \succ)$ consists of a stable monotonic quasi-order $\succsim$ and a stable well-founded order $\succ$, where $\succsim$ and $\succ$ are compatible (i.e., $\succsim \circ \succ \circ \succsim \subseteq \succ$). For a DP problem $(\mathcal{P}, \mathcal{R})$, the processor requires that all DPs in $\mathcal{P}$ are strictly or weakly decreasing and all rules $\mathcal{R}$ are weakly decreasing. Then one can delete all strictly decreasing DPs. Note that both TRSs and relations can be seen as sets of pairs of terms. Thus, $\mathcal{P} \setminus \succ$ denotes $\{s \to t \in \mathcal{P} \mid s \not\succ t\}$.

[5] Note that in standard graph terminology, a path $n_0 \Rightarrow n_1 \Rightarrow \ldots \Rightarrow n_k$ in a directed graph forms a cycle if $n_0 = n_k$ and $k \geq 1$. In our context, we identify cycles with the *set* of elements that occur in it, i.e., we call $\{n_0, n_1, \ldots, n_k\}$ a cycle [16].

Theorem 5 (Reduction Pair Processor [2,14,20]) *Let $(\succsim, \succ)$ be a reduction pair. Then the following DP processor Proc is sound.*

$$Proc(\,(\mathcal{P},\mathcal{R})\,) = \begin{cases} \{\,(\mathcal{P}\setminus\succ,\mathcal{R})\,\}, & \text{if } \mathcal{P}\subseteq\succsim\cup\succ \text{ and } \mathcal{R}\subseteq\succsim \\ \{\,(\mathcal{P},\mathcal{R})\,\}, & \text{otherwise} \end{cases}$$

For the problem $(\{(11)\},\mathcal{R}'_{sort})$, we search for a reduction pair where (11) is strictly decreasing (w.r.t. $\succ$) and the rules in $\mathcal{R}'_{sort}$ are weakly decreasing (w.r.t. $\succsim$). However, this is not satisfied by the orders available in current termination tools.[6] That is not surprising, because termination of this DP problem essentially relies on the argument (1) that every non-empty list contains its maximum.

Recall that our goal is to prove the absence of infinite innermost $(\mathcal{P},\mathcal{R})$-chains. Each such chain would correspond to a reduction of the following form

$$s_1\sigma \;\to_{\mathcal{P}}\; t_1\sigma \;\xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}}\; s_2\sigma \;\to_{\mathcal{P}}\; t_2\sigma \;\xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}}\; s_3\sigma \;\to_{\mathcal{P}}\; t_3\sigma \;\xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}}\; \dots$$

where $s_i \to t_i$ are variable-renamed DPs from $\mathcal{P}$. The reduction pair processor ensures

$$s_1\sigma \;{}_{(}\!\succsim\!{}_{)}\; t_1\sigma \;\succsim\; s_2\sigma \;{}_{(}\!\succsim\!{}_{)}\; t_2\sigma \;\succsim\; s_3\sigma \;{}_{(}\!\succsim\!{}_{)}\; t_3\sigma \;\succsim\; \dots$$

Hence, strictly decreasing DPs (i.e., where $s_i\sigma \succ t_i\sigma$) cannot occur infinitely often in innermost chains and thus, they can be removed from the DP problem.

However, instead of requiring a strict decrease when going from the left-hand side $s_i\sigma$ of a DP to the right-hand side $t_i\sigma$, it would also be sufficient to require a strict decrease when going from the right-hand side $t_i\sigma$ to the *next* left-hand side $s_{i+1}\sigma$. In other words, if every reduction of $t_i\sigma$ to normal form makes the term strictly smaller w.r.t. $\succ$, then we would have $t_i\sigma \succ s_{i+1}\sigma$. Hence, then the DP $s_i \to t_i$ cannot occur infinitely often and could be removed from the DP problem. Our goal is to formulate a new processor based on this idea.

So essentially, we can remove a DP $s \to t$ from the DP problem, if

$$\text{for every normal substitution } \sigma,\ t\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q \text{ implies } t\sigma \succ q. \tag{14}$$

In addition, all DPs and rules still have to be weakly decreasing. A substitution σ is called *normal* iff $\sigma(x)$ is in normal form w.r.t. $\mathcal{R}$ for all variables x. Similarly, the term $t\sigma$ is called a *normal instantiation* of t iff σ is a normal substitution.

So to remove (11) from the remaining DP problem $(\{(11)\},\mathcal{R}'_{sort})$ of Example 1 with the criterion above, we have to use a reduction pair satisfying (14). Here, t is the right-hand side of (11), i.e., $t = \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x,xs)),\mathsf{add}(x,xs)))$.

3 Many-Sorted Rewriting

Now we will weaken the requirement (14) step by step to obtain a condition amenable to automation. The current requirement (14) is still unnecessarily hard. For instance, in our example we also have to regard substitutions like $\sigma(x) = \sigma(xs) = \mathsf{true}$ and require that $t\sigma \succ q$ holds, although intuitively, here x stands for a natural number and

[6] Most of these orders couple a (quasi-)simplification order with an argument filter. Due to the collapsing if_2-rule, one cannot filter away the second argument of del. But then the left-hand side of the DP (11) is embedded in its right-hand side. Thus, all orders of this form fail here.

xs stands for a list (and not a Boolean value). We will show that one does not have to require (14) for *all* normal substitutions, but only for "well-typed" ones. The reason is that if there is an infinite innermost reduction, then there is also an infinite innermost reduction of "well-typed" terms.

First, we make precise what we mean by "well-typed". Recall that up to now we regarded ordinary TRSs over untyped signatures $\mathcal{F}$. The following definition shows how to extend such signatures by (monomorphic) types, cf. e.g. [45].

Definition 6 (Typing) Let $\mathcal{F}$ be an (untyped) signature. A many-sorted signature $\mathcal{F}'$ is a *typed variant* of $\mathcal{F}$ if it contains the same function symbols as $\mathcal{F}$, with the same arities. So f is a symbol of $\mathcal{F}$ with arity n iff f is a symbol of $\mathcal{F}'$ with a type of the form $\tau_1 \times \ldots \times \tau_n \to \tau$. Similarly, a typed variant $\mathcal{V}'$ of the set of variables $\mathcal{V}$ contains the same variables as $\mathcal{V}$, but now every variable has a type τ. We always assume that for every type τ, $\mathcal{V}'$ contains infinitely many variables of type τ. A term over $\mathcal{F}$ and $\mathcal{V}$ is *well typed* w.r.t. $\mathcal{F}'$ and $\mathcal{V}'$ iff

- t is a variable (of some type τ in $\mathcal{V}'$) or
- $t = f(t_1, \ldots, t_n)$ with $n \geq 0$, where all t_i are well typed and have some type τ_i, and where f has type $\tau_1 \times \ldots \times \tau_n \to \tau$ in $\mathcal{F}'$. Then t has type τ.

We only permit typed variants $\mathcal{F}'$ where there exist well-typed ground terms of types $\tau_1, \ldots, \tau_n$ over $\mathcal{F}'$, whenever some $f \in \mathcal{F}'$ has type $\tau_1 \times \ldots \times \tau_n \to \tau$.[7]

A TRS $\mathcal{R}$ over[8] $\mathcal{F}$ and $\mathcal{V}$ is *well typed* w.r.t. $\mathcal{F}'$ and $\mathcal{V}'$ if for all $\ell \to r \in \mathcal{R}$, we have that ℓ and r are well typed and that they have the same type.[9]

For any TRS $\mathcal{R}$ over a signature $\mathcal{F}$, one can use a standard type inference algorithm to compute a typed variant $\mathcal{F}'$ of $\mathcal{F}$ automatically such that $\mathcal{R}$ is well typed. Of course, a trivial solution is to use a many-sorted signature with just one sort (then every term and every TRS are trivially well typed). But to make our approach more powerful, it is advantageous to use the most general typed variant where $\mathcal{R}$ is well typed instead. Here, the set of terms is decomposed into as many types as possible. Then fewer terms are considered to be "well typed" and hence, the condition (14) has to be required for fewer substitutions σ.

For example, let $\mathcal{F} = \{\mathsf{0}, \mathsf{s}, \mathsf{true}, \mathsf{false}, \mathsf{empty}, \mathsf{add}, \mathsf{ge}, \mathsf{eql}, \mathsf{max}, \mathsf{if}_1, \mathsf{del}, \mathsf{if}_2, \mathsf{SORT}\}$. To make $\{(11)\} \cup \mathcal{R}'_{sort}$ well typed, we obtain the typed variant $\mathcal{F}'$ of $\mathcal{F}$ with the types nat, bool, list, and tuple. Here the function symbols have the following types.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{0} & : & \mathsf{nat} & \mathsf{ge}, \mathsf{eql} & : & \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool} \\
\mathsf{s} & : & \mathsf{nat} \to \mathsf{nat} & \mathsf{max} & : & \mathsf{list} \to \mathsf{nat} \\
\mathsf{true}, \mathsf{false} & : & \mathsf{bool} & \mathsf{if}_1 & : & \mathsf{bool} \times \mathsf{nat} \times \mathsf{nat} \times \mathsf{list} \to \mathsf{nat} \\
\mathsf{empty} & : & \mathsf{list} & \mathsf{if}_2 & : & \mathsf{bool} \times \mathsf{nat} \times \mathsf{nat} \times \mathsf{list} \to \mathsf{list} \\
\mathsf{add}, \mathsf{del} & : & \mathsf{nat} \times \mathsf{list} \to \mathsf{list} & \mathsf{SORT} & : & \mathsf{list} \to \mathsf{tuple}
\end{array}$$

Now we show that innermost termination is a *persistent* property, i.e., a TRS is innermost terminating iff it is innermost terminating on well-typed terms. Here, one can use any typed variant where the TRS is well typed. As noted by [30], persistence of innermost termination follows from results of [35], but to our knowledge, it has never

[7] This is not a restriction, as one can simply add new constants to $\mathcal{F}$ and $\mathcal{F}'$.

[8] Note that $\mathcal{F}$ may well contain function symbols that do not occur in $\mathcal{R}$.

[9] W.l.o.g., here one may rename the variables in every rule. Then it is not a problem if the variable x is used with type τ_1 in one rule and with type τ_2 in another rule.

been explicitly stated or applied in the literature before. In the following, we give a new (alternative) self-contained proof. Note that in contrast to innermost termination, full termination is only persistent for very restricted classes of TRSs, cf. [45]. To illustrate this, consider the famous TRS $\mathsf{f}(\mathsf{a},\mathsf{b},x) \to \mathsf{f}(x,x,x),\ \mathsf{g}(x,y) \to x,\ \mathsf{g}(x,y) \to y$ of [40]. If one uses a typed variant where a, b, and the arguments of f have a different type than the arguments and the result of g, then the TRS is terminating on all well-typed terms. In contrast, when ignoring the types, then the f-rule is obviously not terminating when instantiating x with $\mathsf{g}(\mathsf{a},\mathsf{b})$.

Theorem 7 (Persistence) *Let $\mathcal{R}$ be a TRS over $\mathcal{F}$ and $\mathcal{V}$ and let $\mathcal{R}$ be well typed w.r.t. the typed variants $\mathcal{F}'$ and $\mathcal{V}'$. $\mathcal{R}$ is innermost terminating for all well-typed terms w.r.t. $\mathcal{F}'$ and $\mathcal{V}'$ iff $\mathcal{R}$ is innermost terminating (for* all *terms).*

Proof For every (not necessarily well-typed) term t, we define its *result type* $\mathbf{r}(t)$:

$$\mathbf{r}(x) = \rho, \text{ if } x \text{ is of type } \rho \qquad \mathbf{r}(f(t_1,\ldots,t_n)) = \rho, \text{ if } f \text{ is of type } \rho_1 \times \ldots \times \rho_n \to \rho$$

Moreover, a term t is called *semi-well typed* iff there is no type conflict on any path from the root to redexes. Formally, t is semi-well typed iff

- t is a variable or
- $t = f(t_1,\ldots,t_n)$ for a symbol f of type $\rho_1 \times \ldots \times \rho_n \to \rho$, and for every t_i we have:
  - t_i is in normal form or
  - t_i is semi-well typed and $\mathbf{r}(t_i) = \rho_i$.

We first prove the following claim:

$$\begin{array}{l}\text{If } t \text{ is semi-well typed and } t \xrightarrow{\mathsf{i}}_{\mathcal{R}} s,\\ \text{then } s \text{ is also semi-well typed and if } s \text{ is no normal form, then } \mathbf{r}(s) = \mathbf{r}(t).\end{array} \tag{15}$$

We use induction on the position where the reduction $t \xrightarrow{\mathsf{i}}_{\mathcal{R}} s$ takes place. If the reduction is on the root position, then $t = \ell\sigma$ and $s = r\sigma$ for some rule $\ell \to r$ from $\mathcal{R}$. Since r is well typed and σ instantiates all variables of r by normal forms due to the innermost strategy, $r\sigma$ is semi-well typed. Moreover, if $r\sigma$ is no normal form, then r is not a variable and thus, $\mathbf{r}(r\sigma) = \mathbf{r}(\ell\sigma)$, since ℓ and r have the same type.

In the induction step, let the reduction take place on a position $i\,\pi$. Thus, $t = f(t_1,\ldots,t_i,\ldots,t_n)$, $s = f(t_1,\ldots,s_i,\ldots,t_n)$ for some symbol f of type $\rho_1 \times \ldots \times \rho_n \to \rho$ where $t_i \xrightarrow{\mathsf{i}}_{\mathcal{R}} s_i$, t_i is semi-well typed, and $\mathbf{r}(t_i) = \rho_i$. By the induction hypothesis, s_i is also semi-well typed and if s_i is no normal form, then $\mathbf{r}(s_i) = \mathbf{r}(t_i) = \rho_i$. This implies that s is semi-well typed as well. Moreover, obviously $\mathbf{r}(s) = \mathbf{r}(t) = \rho$.

To use the claim (15) for the proof of Theorem 7, for any (not necessarily well-typed) term t and any type τ, we define the term $\mathbf{w}(t,\tau)$ which transforms t into a well-typed term of type τ. To this end, whenever there is a topmost subterm s of t which destroys the well-typedness (i.e., s occurs at a position where a term of type ρ is expected, but s does not have type ρ), then s is replaced by a corresponding new variable $y_{s,\rho}$ of type ρ. More precisely, for any variable x of type ρ and any function symbol f of type $\rho_1 \times \ldots \times \rho_n \to \rho$ we define:

$$\mathbf{w}(x,\tau) = \begin{cases} x & \text{if } \tau = \rho \\ y_{x,\tau} & \text{otherwise} \end{cases} \qquad \mathbf{w}(f(t_1,\ldots,t_n),\tau) = \begin{cases} f(\mathbf{w}(t_1,\rho_1),\ldots,\mathbf{w}(t_n,\rho_n)) & \text{if } \tau = \rho \\ y_{f(t_1,\ldots,t_n),\tau} & \text{otherwise} \end{cases}$$

Now we show the following claim:

$$\text{If } t \text{ is semi-well typed, } t \xrightarrow{\mathsf{i}}_{\mathcal{R}} s, \text{ and } \mathbf{r}(t) = \tau, \text{ then we have } \mathbf{w}(t,\tau) \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathbf{w}(s,\tau). \tag{16}$$

We prove (16) by induction on the position of the reduction $t \xrightarrow{\mathsf{i}}_{\mathcal{R}} s$. If the reduction is on the root position, then we have $t = \ell\sigma$ and $s = r\sigma$ for some rule $\ell \to r$ from $\mathcal{R}$

and some normal substitution σ. For every variable x, let σ' be the substitution with $\sigma'(x) = \mathbf{w}(x\sigma, \rho)$ where ρ is the type of the variable x. Since ℓ and r are well typed and of type τ, we have $\mathbf{w}(\ell\sigma, \tau) = \ell\sigma'$ and $\mathbf{w}(r\sigma, \tau) = r\sigma'$. This implies $\mathbf{w}(t, \tau) = \ell\sigma' \xrightarrow{\mathrm{i}}_{\mathcal{R}} r\sigma' = \mathbf{w}(s, \tau)$.

In the induction step, let the reduction take place on a position $i\,\pi$. Thus, we have $t = f(t_1, \ldots, t_i, \ldots, t_n)$, $s = f(t_1, \ldots, s_i, \ldots, t_n)$ for some function symbol f of type $\rho_1 \times \ldots \times \rho_n \to \rho$ where $t_i \xrightarrow{\mathrm{i}}_{\mathcal{R}} s_i$, t_i is semi-well typed, and $\mathbf{r}(t_i) = \rho_i$. By the induction hypothesis, we obtain $\mathbf{w}(t, \tau) = f(\mathbf{w}(t_1, \rho_1), \ldots, \mathbf{w}(t_i, \rho_i), \ldots, \mathbf{w}(t_n, \rho_n)) \xrightarrow{\mathrm{i}}_{\mathcal{R}} f(\mathbf{w}(t_1, \rho_1), \ldots, \mathbf{w}(s_i, \rho_i), \ldots, \mathbf{w}(t_n, \rho_n)) = \mathbf{w}(s, \tau)$.

Now we show that (15) and (16) imply Theorem 7: If there is a (not necessarily well-typed) term with infinite innermost reduction, then there is also a *minimal* such term (i.e., all its proper subterms are innermost terminating). Hence, its infinite innermost reduction contains a root reduction step. So there exists a (not necessarily well-typed) term t which starts an infinite innermost reduction where the first reduction step is at the root. Hence, t is semi-well typed since t is a (well-typed) left-hand side of a rule where all variables are instantiated by normal forms due to the innermost strategy. By (15), this implies that there is an infinite innermost reduction of semi-well typed terms

$$t \xrightarrow{\mathrm{i}}_{\mathcal{R}} t_1 \xrightarrow{\mathrm{i}}_{\mathcal{R}} t_2 \xrightarrow{\mathrm{i}}_{\mathcal{R}} \ldots$$

and by (16), this results in an infinite reduction of well-typed terms where $\mathbf{r}(t) = \tau$:

$$\mathbf{w}(t, \tau) \xrightarrow{\mathrm{i}}_{\mathcal{R}} \mathbf{w}(t_1, \tau) \xrightarrow{\mathrm{i}}_{\mathcal{R}} \mathbf{w}(t_2, \tau) \xrightarrow{\mathrm{i}}_{\mathcal{R}} \ldots$$

□

We expect that there exist several points where Theorem 7 could simplify innermost termination proofs.[10] In this paper, we use Theorem 7 to weaken the condition (14) required to remove a DP from a DP problem $(\mathcal{P}, \mathcal{R})$. Now one can use any typed variant where $\mathcal{P} \cup \mathcal{R}$ is well typed. To remove $s \to t$ from $\mathcal{P}$, it suffices if

$$\text{for every normal } \sigma \textit{ where } t\sigma \textit{ is well typed, } t\sigma \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q \text{ implies } t\sigma \succ q. \tag{17}$$

4 Coupling Dependency Pairs and Inductive Theorem Proving

Condition (17) is still too hard, because up to now, $t\sigma$ does not have to be ground. We will show (in Theorem 12) that for DP problems $(\mathcal{P}, \mathcal{R})$ satisfying suitable non-overlappingness requirements and where $\mathcal{R}$ is already innermost terminating, (17) can be relaxed to ground substitutions σ. Then $s \to t$ can be removed from $\mathcal{P}$ if

$$\begin{array}{l}\text{for every normal substitution } \sigma \text{ where } t\sigma \text{ is a well-typed } \textit{ground} \text{ term,}\\ t\sigma \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q \text{ implies } t\sigma \succ q.\end{array} \tag{18}$$

Example 8 Innermost termination of $\mathcal{R}$ is really needed to replace (17) by (18). To see this, consider the DP problem $(\mathcal{P}, \mathcal{R})$ with $\mathcal{P} = \{\mathsf{F}(x) \to \mathsf{F}(x)\}$ and the non-innermost terminating TRS $\mathcal{R} = \{\mathsf{a} \to \mathsf{a}\}$.[11] *Let $\mathcal{F} = \{\mathsf{F}, \mathsf{a}\}$. We use a typed variant $\mathcal{F}'$ where*

[10] For example, by Theorem 7 one could switch to termination methods like [28] exploiting types.

[11] One cannot assume that DP problems $(\mathcal{P}, \mathcal{R})$ always have a specific form, e.g., that $\mathcal{P}$ includes $\mathsf{A} \to \mathsf{A}$ whenever $\mathcal{R}$ includes $\mathsf{a} \to \mathsf{a}$. The reason is that a DP problem $(\mathcal{P}, \mathcal{R})$ can result from arbitrary DP processors that were applied before. Hence, one really has to make sure that processors are sound for *arbitrary* DP problems $(\mathcal{P}, \mathcal{R})$.

$\mathsf{F} : \tau_1 \to \tau_2$ *and* $\mathsf{a} : \tau_1$ *for two different types* τ_1 *and* τ_2*. For the right-hand side* $t = \mathsf{F}(x)$ *of the DP, the only well-typed ground instantiation is* $\mathsf{F}(\mathsf{a})$*. Since this term has no normal form* q*, the condition (18) holds. Nevertheless, it is not sound to remove the only DP from* $\mathcal{P}$*, since* $\mathsf{F}(x_1) \to \mathsf{F}(x_1)$, $\mathsf{F}(x_2) \to \mathsf{F}(x_2), \ldots$ *is an infinite innermost* $(\mathcal{P}, \mathcal{R})$*-chain (but there is no infinite innermost* ground *chain).*

To see the reason for the non-overlappingness requirement, consider $(\mathcal{P}, \mathcal{R})$ *with* $\mathcal{P} = \{\mathsf{F}(\mathsf{f}(x)) \to \mathsf{F}(\mathsf{f}(x))\}$ *and* $\mathcal{R} = \{\mathsf{f}(\mathsf{a}) \to \mathsf{a}\}$ *where the left-hand sides of the rules in* $\mathcal{P}$ *and* $\mathcal{R}$ *overlap. Now* $\mathcal{F} = \{\mathsf{F}, \mathsf{f}, \mathsf{a}\}$ *and in the typed variant we have* $\mathsf{F} : \tau_1 \to \tau_2$, $\mathsf{f} : \tau_1 \to \tau_1$, *and* $\mathsf{a} : \tau_1$ *for two different types* τ_1 *and* τ_2*. For the right-hand side* $t = \mathsf{F}(\mathsf{f}(x))$ *of the DP, the only well-typed ground instantiations are* $\mathsf{F}(\mathsf{f}^n(\mathsf{a}))$ *with* $n \geq 1$*. If we take the embedding order* $\succ_{emb}$*, then all well-typed ground instantiations of* t *are* $\succ_{emb}$*-greater than their normal form* $\mathsf{F}(\mathsf{a})$*. So Condition (18) would allow us to remove the only DP from* $\mathcal{P}$*. But again, this is unsound, since there is an infinite innermost* $(\mathcal{P}, \mathcal{R})$*-chain (but no such* ground *chain). In addition, the non-overlappingness requirement will also be needed later on in order to transform Condition (18) into a condition that can be checked by an inductive theorem prover (cf. Footnote 13).*

To prove a condition like (18), we replace (18) by the following condition (19), which is easier to check. Here, we require that for all instantiations $t\sigma$ as above, *every* reduction of $t\sigma$ to its normal form uses a strictly decreasing rule $\ell \to r$ (i.e., a rule with $\ell \succ r$) on a strongly monotonic position π. A position π in a term u is *strongly monotonic* w.r.t. $\succ$ iff $t_1 \succ t_2$ implies $u[t_1]_\pi \succ u[t_2]_\pi$ for all terms t_1 and t_2. So to remove $s \to t$ from $\mathcal{P}$, now it suffices if

> for every normal substitution σ where $t\sigma$ is a well-typed ground term, every reduction "$t\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q$" has the form
>
> $$t\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} u[\ell\delta]_\pi \xrightarrow{\mathsf{i}}_{\mathcal{R}} u[r\delta]_\pi \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q \tag{19}$$
>
> for a rule $\ell \to r \in \mathcal{R}$ where $\ell \succ r$
> and where the position π in u is strongly monotonic w.r.t. $\succ$.[12]

For example, for $\mathcal{R}_{sort}$'s termination proof one may use a reduction pair $(\succsim, \succ)$ based on a *polynomial interpretation* [4,12,27]. A polynomial interpretation $\mathcal{P}ol$ maps every n-ary function symbol f to a polynomial $f_{\mathcal{P}ol}$ over n variables $x_1, \ldots, x_n$ with coefficients from $\mathbb{N}$. This mapping is extended to terms by $[x]_{\mathcal{P}ol} = x$ for all variables x and $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = f_{\mathcal{P}ol}([t_1]_{\mathcal{P}ol}, \ldots, [t_n]_{\mathcal{P}ol})$. Now $s \succ_{\mathcal{P}ol} t$ (resp. $s \succsim_{\mathcal{P}ol} t$) iff $[s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol}$ (resp. $[s]_{\mathcal{P}ol} \geq [t]_{\mathcal{P}ol}$) holds for all instantiations of the variables with natural numbers. For instance, consider the interpretation $\mathcal{P}ol_1$ with

$$\mathsf{0}_{\mathcal{P}ol_1} = \mathsf{empty}_{\mathcal{P}ol_1} = \mathsf{true}_{\mathcal{P}ol_1} = \mathsf{false}_{\mathcal{P}ol_1} = \mathsf{ge}_{\mathcal{P}ol_1} = \mathsf{eql}_{\mathcal{P}ol_1} = 0 \qquad \mathsf{s}_{\mathcal{P}ol_1} = 1 + x_1$$
$$\mathsf{add}_{\mathcal{P}ol_1} = 1 + x_1 + x_2 \qquad \mathsf{max}_{\mathcal{P}ol_1} = x_1$$
$$\mathsf{if}_{1\mathcal{P}ol_1} = 1 + x_2 + x_3 + x_4 \qquad \mathsf{del}_{\mathcal{P}ol_1} = x_2$$
$$\mathsf{if}_{2\mathcal{P}ol_1} = 1 + x_3 + x_4 \qquad \mathsf{SORT}_{\mathcal{P}ol_1} = x_1$$

[12] In special cases, Condition (19) can be automated by k-times *narrowing* the DP $s \to t$ [16]. However, this only works if for any substitution σ, the reduction $t\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} u[\ell\delta]_\pi$ is shorter than a fixed number k. So it fails for TRSs like $\mathcal{R}_{sort}$ where termination relies on an inductive property. Here, the reduction

$$\mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x, xs)), \mathsf{add}(x, xs)))\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}_{sort}} \mathsf{SORT}(\mathsf{if}_2(\mathsf{true}, \ldots, \ldots, \ldots))$$

can be arbitrarily long, depending on σ. Therefore, narrowing the DP (11) a fixed number of times does not help.

When using the reduction pair $(\succsim_{\mathcal{P}ol_1}, \succ_{\mathcal{P}ol_1})$, the DP (11) and all rules of $\mathcal{R}'_{sort}$ are weakly decreasing. Moreover, then Condition (19) is indeed satisfied for the right-hand side t of (11). To see this, note that in *every* reduction $t\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q$ where $t\sigma$ is a well-typed ground term, eventually one has to apply the rule "$\mathsf{if}_2(\mathsf{true}, x, y, xs) \to xs$" which is strictly decreasing w.r.t. $\succ_{\mathcal{P}ol_1}$. This rule is used by the del-algorithm to delete an element, i.e., to reduce the length of the list. Moreover, the rule is used within a context of the form $\mathsf{SORT}(\mathsf{add}(..., \mathsf{add}(..., \quad \ldots \quad \mathsf{add}(..., \Box)\ldots)))$. The polynomials $\mathsf{SORT}_{\mathcal{P}ol_1}$ and $\mathsf{add}_{\mathcal{P}ol_1}$ are strongly monotonic in their first resp. second argument. Thus, the strictly decreasing rule is indeed used on a strongly monotonic position.

To check automatically whether every reduction of $t\sigma$ to normal form uses a strictly decreasing rule on a strongly monotonic position, we add new rules and function symbols to the TRS $\mathcal{R}$ which results in an extended TRS $\mathcal{R}^{\succ}$. Moreover, for every term u we define a corresponding term $u^{\succ}$. For non-overlapping TRSs $\mathcal{R}$, we have the following property, cf. Lemma 10: if $u^{\succ} \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$, then for every reduction $u \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q$, we have $u \succ q$. We now explain how to construct $\mathcal{R}^{\succ}$.

For every $f \in \mathcal{D}_{\mathcal{R}}$, we introduce a new symbol $f^{\succ}$. Now $f^{\succ}(u_1, ..., u_n)$ should reduce to tt in the new TRS $\mathcal{R}^{\succ}$ whenever the reduction of $f(u_1, ..., u_n)$ in the original TRS $\mathcal{R}$ uses a strictly decreasing rule on a strongly monotonic position. Thus, if a rule $f(\ell_1, ..., \ell_n) \to r$ of $\mathcal{R}$ was strictly decreasing (i.e., $f(\ell_1, ..., \ell_n) \succ r$), then we add the rule $f^{\succ}(\ell_1, ..., \ell_n) \to \mathsf{tt}$ to $\mathcal{R}^{\succ}$.[13] Otherwise, a strictly decreasing rule will be used on a strongly monotonic position to reduce an instance of $f(\ell_1, \ldots, \ell_n)$ if this holds for the corresponding instance of the right-hand side r. Hence, then we add the rule $f^{\succ}(\ell_1, ..., \ell_n) \to r^{\succ}$ to $\mathcal{R}^{\succ}$ instead. It remains to define $u^{\succ}$ for any term u over the signature of $\mathcal{R}$. If $u = f(u_1, ..., u_n)$, then we regard the subterms on the strongly monotonic positions of u and check whether their reduction uses a strictly decreasing rule. For any n-ary symbol f, let $mon_{\succ}(f)$ contain those positions from $\{1, \ldots, n\}$ where the term $f(x_1, ..., x_n)$ is strongly monotonic. If $mon_{\succ}(f) = \{i_1, \ldots, i_m\}$, then for $u = f(u_1, ..., u_n)$ we obtain $u^{\succ} = u^{\succ}_{i_1} \vee \ldots \vee u^{\succ}_{i_m}$, if f is a constructor. If f is defined, then a strictly decreasing rule could also be applied on the root position of u. Hence, then $u^{\succ} = u^{\succ}_{i_1} \vee \ldots \vee u^{\succ}_{i_m} \vee f^{\succ}(u_1, ..., u_n)$. Of course, $\mathcal{R}^{\succ}$ also contains appropriate rules for the disjunction "$\vee$".[14] The empty disjunction is represented by ff.

Definition 9 ($\mathcal{R}^{\succ}$) Let $\succ$ be an order on terms and let $\mathcal{R}$ be a TRS over $\mathcal{F}$ and $\mathcal{V}$. We extend $\mathcal{F}$ to a new signature $\mathcal{F}^{\succ} = \mathcal{F} \uplus \{f^{\succ} \mid f \in \mathcal{D}_{\mathcal{R}}\} \uplus \{\mathsf{tt}, \mathsf{ff}, \vee\}$. For any term u over $\mathcal{F}$ and $\mathcal{V}$, we define the term $u^{\succ}$ over $\mathcal{F}^{\succ}$ and $\mathcal{V}$:

$$u^{\succ} = \begin{cases} \bigvee_{i \in mon_{\succ}(f)} u^{\succ}_i, & \text{if } u = f(u_1, \ldots, u_n) \text{ and } f \notin \mathcal{D}_{\mathcal{R}} \\ \bigvee_{i \in mon_{\succ}(f)} u^{\succ}_i \quad \vee \quad f^{\succ}(u_1, \ldots, u_n), & \text{if } u = f(u_1, \ldots, u_n) \text{ and } f \in \mathcal{D}_{\mathcal{R}} \\ \mathsf{ff}, & \text{if } u \in \mathcal{V} \end{cases}$$

Moreover, we define $\mathcal{R}^{\succ} = \quad \{f^{\succ}(\ell_1, ..., \ell_n) \to \mathsf{tt} \mid f(\ell_1, ..., \ell_n) \to r \in \mathcal{R} \cap \succ\}$
$\cup \{f^{\succ}(\ell_1, ..., \ell_n) \to r^{\succ} \mid f(\ell_1, ..., \ell_n) \to r \in \mathcal{R} \setminus \succ\}$
$\cup \mathcal{R} \cup \{\mathsf{tt} \vee b \to \mathsf{tt},\ \mathsf{ff} \vee b \to b\}$.

In our example, the only rules of $\mathcal{R}'_{sort}$ which are strictly decreasing w.r.t. $\succ_{\mathcal{P}ol_1}$ are the last two max-rules and the rule "$\mathsf{if}_2(\mathsf{true}, x, y, xs) \to xs$". So according to

[13] Of course, this is only sound for non-overlapping TRSs. To see this, consider the TRS $\mathcal{R} = \{\mathsf{a} \to \mathsf{b}, \mathsf{a} \to \mathsf{c}\}$ with $\mathsf{a} \succ \mathsf{b}$ and $\mathsf{a} \not\succ \mathsf{c}$. Now $\mathcal{R}^{\succ}$ would contain the rule $\mathsf{a}^{\succ} \to \mathsf{tt}$ although there is a reduction of a to a normal form c that is not $\succ$-smaller than a.

[14] It suffices to include just the rules "$\mathsf{tt} \vee b \to \mathsf{tt}$" and "$\mathsf{ff} \vee b \to b$", since $\mathcal{R}^{\succ}$ is only used for inductive proofs and "$b \vee \mathsf{tt} = \mathsf{tt}$" and "$b \vee \mathsf{ff} = b$" are inductive consequences.

Definition 9, the TRS $\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort}$ contains $\mathcal{R}'_{sort} \cup \{\mathsf{tt} \vee b \to \mathsf{tt},\ \mathsf{ff} \vee b \to b\}$ and the following rules. Here, we already simplified disjunctions of the form "$\mathsf{ff} \vee t$" or "$t \vee \mathsf{ff}$" to t.[15] To ease readability, we wrote "$\mathsf{ge}^{\succ}$" instead of "$\mathsf{ge}^{\succ_{\mathcal{P}ol_1}}$", etc.

$$\begin{aligned}
\mathsf{ge}^{\succ}(x, 0) &\to \mathsf{ff}\\
\mathsf{ge}^{\succ}(0, \mathsf{s}(y)) &\to \mathsf{ff}\\
\mathsf{ge}^{\succ}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{ge}^{\succ}(x, y)
\end{aligned}$$

$$\begin{aligned}
\mathsf{eql}^{\succ}(0, 0) &\to \mathsf{ff}\\
\mathsf{eql}^{\succ}(\mathsf{s}(x), 0) &\to \mathsf{ff}\\
\mathsf{eql}^{\succ}(0, \mathsf{s}(y)) &\to \mathsf{ff}\\
\mathsf{eql}^{\succ}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{eql}^{\succ}(x, y)
\end{aligned}$$

$$\begin{aligned}
\mathsf{max}^{\succ}(\mathsf{empty}) &\to \mathsf{ff}\\
\mathsf{max}^{\succ}(\mathsf{add}(x, \mathsf{empty})) &\to \mathsf{tt}\\
\mathsf{max}^{\succ}(\mathsf{add}(x, \mathsf{add}(y, xs))) &\to \mathsf{tt}\\
\mathsf{if}_1^{\succ}(\mathsf{true}, x, y, xs) &\to \mathsf{max}^{\succ}(\mathsf{add}(x, xs))\\
\mathsf{if}_1^{\succ}(\mathsf{false}, x, y, xs) &\to \mathsf{max}^{\succ}(\mathsf{add}(y, xs))
\end{aligned}$$

$$\begin{aligned}
\mathsf{del}^{\succ}(x, \mathsf{empty}) &\to \mathsf{ff}\\
\mathsf{del}^{\succ}(x, \mathsf{add}(y, xs)) &\to \mathsf{if}_2^{\succ}(\mathsf{eql}(x, y), x, y, xs)\\
\mathsf{if}_2^{\succ}(\mathsf{true}, x, y, xs) &\to \mathsf{tt}\\
\mathsf{if}_2^{\succ}(\mathsf{false}, x, y, xs) &\to \mathsf{del}^{\succ}(x, xs)
\end{aligned}$$

Lemma 10 (Soundness of $\mathcal{R}^{\succ}$) *Let $(\succsim, \succ)$ be a reduction pair and let $\mathcal{R}$ be a non-overlapping TRS over $\mathcal{F}$ and $\mathcal{V}$ with $\mathcal{R} \subseteq \succsim$. For any terms u and q over $\mathcal{F}$ and $\mathcal{V}$ with $u^{\succ} \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$ and $u \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q$, we have $u \succ q$.*

Proof We use induction on the lexicographic combination of the length of the reduction $u \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q$ and of the structure of u.

First let u be a variable. Here, $u^{\succ} = \mathsf{ff}$ and thus, $u^{\succ} \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$ is impossible.

Now let $u = f(u_1, ..., u_n)$. The reduction $u \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q$ starts with $u = f(u_1, ..., u_n) \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}} f(q_1, ..., q_n)$ where the reductions $u_i \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q_i$ are at most as long as $u \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q$. If there is a $j \in mon_{\succ}(f)$ with $u_j^{\succ} \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$, then $u_j \succ q_j$ by induction hypothesis. So $u = f(u_1, ..., u_j, ..., u_n) \succ f(u_1, ..., q_j, ..., u_n) \succsim f(q_1, ..., q_j, ..., q_n) \succsim q$, as $\mathcal{R} \subseteq \succsim$.

Otherwise, $u^{\succ} \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$ means that $f^{\succ}(u_1, \ldots, u_n) \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$. As $\mathcal{R} \subseteq \mathcal{R}^{\succ}$, we have $f^{\succ}(u_1, \ldots, u_n) \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}^{\succ}} f^{\succ}(q_1, \ldots, q_n)$. Since $\mathcal{R}$ is non-overlapping, $\mathcal{R}^{\succ}$ is non-overlapping as well. This implies confluence of $\xrightarrow{\mathrm{i}}_{\mathcal{R}^{\succ}}$, cf. [34]. Hence, we also get $f^{\succ}(q_1, \ldots, q_n) \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$. There is a rule $f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{R}$ and a normal substitution δ with $f(q_1, \ldots, q_n) = f(\ell_1, \ldots, \ell_n)\delta \xrightarrow{\mathrm{i}}_{\mathcal{R}} r\delta \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q$. Note that the q_i only contain symbols of $\mathcal{F}$. Thus, as the q_i are normal forms w.r.t. $\mathcal{R}$, they are also normal forms w.r.t. $\mathcal{R}^{\succ}$. Therefore, as $\mathcal{R}^{\succ}$ is non-overlapping, the only rule of $\mathcal{R}^{\succ}$ applicable

[15] By this simplification, all disjunctions can be eliminated in this example. This would be different for rules with right-hand sides r that contain subterms of the form "$f(\ldots g(\ldots)\ldots)$", where both f and g are defined symbols and where $g(\ldots)$ occurs on a strongly monotonic position of r.

to $f^{\succ}(q_1,\ldots,q_n)$ is the one resulting from $f(\ell_1,\ldots,\ell_n) \to r \in \mathcal{R}$. If $f(\ell_1,\ldots,\ell_n) \succ r$, then that rule would be "$f^{\succ}(\ell_1,\ldots,\ell_n) \to \mathsf{tt}$" and

$$u = f(u_1,\ldots,u_n) \succsim f(q_1,\ldots,q_n) = f(\ell_1,\ldots,\ell_n)\delta \succ r\delta \succsim q.$$

Otherwise, the rule is "$f^{\succ}(\ell_1,...,\ell_n) \to r^{\succ}$", i.e., $f^{\succ}(q_1,...,q_n) = f^{\succ}(\ell_1,...,\ell_n)\delta \xrightarrow{\mathsf{i}}_{\mathcal{R}^{\succ}} r^{\succ}\delta \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$. Since the reduction $r\delta \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q$ is shorter than the original reduction $u \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q$, the induction hypothesis implies $r\delta \succ q$. Thus,

$$u = f(u_1,\ldots,u_n) \succsim f(q_1,\ldots,q_n) = f(\ell_1,\ldots,\ell_n)\delta \succsim r\delta \succ q. \qquad \square$$

With Lemma 10, the condition (19) needed to remove a DP from a DP problem can again be reformulated. To remove $s \to t$ from $\mathcal{P}$, now it suffices if

for every normal substitution σ where $t\sigma$ is a well-typed ground term, we have $t^{\succ}\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$. (20)

So in our example, to remove the DP (11) using the reduction pair $(\succsim_{\mathcal{P}ol_1}, \succ_{\mathcal{P}ol_1})$, we require "$t^{\succ_{\mathcal{P}ol_1}}\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort}} \mathsf{tt}$", where t is the right-hand side of (11), i.e.,

$$t \;=\; \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x,xs)),\mathsf{add}(x,xs))).$$

Since $mon_{\succ_{\mathcal{P}ol_1}}(\mathsf{SORT}) = \{1\}$, $mon_{\succ_{\mathcal{P}ol_1}}(\mathsf{del}) = \{2\}$, $mon_{\succ_{\mathcal{P}ol_1}}(\mathsf{add}) = \{1,2\}$, and $x^{\succ_{\mathcal{P}ol_1}} = xs^{\succ_{\mathcal{P}ol_1}} = \mathsf{ff}$, we have

$$t^{\succ_{\mathcal{P}ol_1}} \;=\; \mathsf{del}^{\succ_{\mathcal{P}ol_1}}(\mathsf{max}(\mathsf{add}(x,xs)),\mathsf{add}(x,xs))$$

when simplifying disjunctions with ff. So to remove (11), we require the following for all normal substitutions σ where $t\sigma$ is well typed and ground.[16]

$$\mathsf{del}^{\succ_{\mathcal{P}ol_1}}(\mathsf{max}(\mathsf{add}(x,xs)),\mathsf{add}(x,xs))\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort}} \mathsf{tt} \tag{21}$$

Note that the rules for $\mathsf{del}^{\succ_{\mathcal{P}ol_1}}$ (given before Lemma 10) compute the *member*-function. In other words, $\mathsf{del}^{\succ_{\mathcal{P}ol_1}}(x,xs)$ holds iff x occurs in the list xs. Thus, (21) is equivalent to the main termination argument (1) of Example 1, i.e., to the observation that every non-empty list contains its maximum. Thus, now we can detect and express termination arguments like (1) within the DP framework.

Our goal is to use inductive theorem provers to verify arguments like (1) or, equivalently, to verify conditions like (20). Indeed, (20) corresponds to the question whether a suitable conjecture is *inductively valid* [5,6,8,9,22,24,41,43,46].

Definition 11 (Inductive Validity) Let $\mathcal{R}$ be a TRS and let s,t be terms over $\mathcal{F}$ and $\mathcal{V}$. We say that $t = s$ is *inductively valid* in $\mathcal{R}$ (denoted $\mathcal{R} \models_{ind} t = s$) iff there exist typed variants $\mathcal{F}'$ and $\mathcal{V}'$ such that $\mathcal{R}$, t, s are well typed where t and s have the same type, and such that $t\sigma \xleftrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} s\sigma$ holds for all substitutions σ over $\mathcal{F}'$ where $t\sigma$ and $s\sigma$ are well-typed ground terms. To make the specific typed variants explicit, we also write "$\mathcal{R} \models^{\mathcal{F}',\mathcal{V}'}_{ind} t = s$".

Of course, in general $\mathcal{R} \models_{ind} t = s$ is undecidable, but it can often be proved automatically by *inductive theorem provers*. By reformulating Condition (20), we now obtain that in a DP problem $(\mathcal{P},\mathcal{R})$, $s \to t$ can be removed from $\mathcal{P}$ if

[16] Note that the restriction to *well-typed ground terms* is crucial. Indeed, (21) does not hold for non-ground or non-well-typed substitutions like $\sigma(x) = \sigma(xs) = \mathsf{true}$.

$$\mathcal{R}^{\succ} \models_{ind} t^{\succ} = \mathsf{tt}. \tag{22}$$

Of course, in addition all DPs in $\mathcal{P}$ and all rules in $\mathcal{R}$ have to be weakly decreasing.

Now we formulate a new DP processor based on Condition (22). Recall that to derive (22) we required a non-overlappingness condition and innermost termination of $\mathcal{R}$. (These requirements ensure that it suffices to regard only *ground* instantiations when proving that reductions of $t\sigma$ to normal form are strictly decreasing, cf. Example 8. Moreover, non-overlappingness is needed for Lemma 10 to make sure that $t^{\succ}\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$ really guarantees that *all* reductions of $t\sigma$ to normal form are strictly decreasing, cf. Footnote 13. Non-overlappingness also ensures that $t^{\succ}\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$ in Condition (20) is equivalent to $t^{\succ}\sigma \xleftrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$ in Condition (22).)

So innermost termination of $\mathcal{R}$ is a condition needed to apply the new induction processor. Of course, innermost termination of $\mathcal{R}$ can again be checked by the DP framework. Therefore, in the following processor, we use a formulation where a DP problem $(\mathcal{P}, \mathcal{R})$ is transformed into *two* new problems. Apart from the new DP problem $(\mathcal{P} \setminus \{s \to t\}, \mathcal{R})$, the processor also generates the problem $(DP(\mathcal{R}), \mathcal{R})$. Absence of infinite innermost $(DP(\mathcal{R}), \mathcal{R})$-chains is equivalent to innermost termination of $\mathcal{R}$. Note that in practice $\mathcal{R}$ only contains the usable rules of $\mathcal{P}$ (since one should have applied the usable rule processor of Theorem 3 before). Then the DP problem $(DP(\mathcal{R}), \mathcal{R})$ means that the TRS consisting just of the usable rules must be innermost terminating. An application of the dependency graph processor of Theorem 4 will therefore transform $(DP(\mathcal{R}), \mathcal{R})$ into DP problems that have already been generated *before*. So (except for algorithms with nested or mutually recursive usable rules), the DP problem $(DP(\mathcal{R}), \mathcal{R})$ obtained by the following processor does not lead to new proof obligations.

In Theorem 12, we restrict ourselves to DP problems $(\mathcal{P}, \mathcal{R})$ with the *tuple property*. This means that for all $s \to t \in \mathcal{P}$, $\mathrm{root}(s)$ and $\mathrm{root}(t)$ are tuple symbols and tuple symbols neither occur anywhere else in s or t nor in $\mathcal{R}$. This is always satisfied for the initial DP problem and it is maintained by almost all DP processors in the literature (including all processors of this paper).

Theorem 12 (Induction Processor) *Let $(\succsim,\succ)$ be a reduction pair with $\mathcal{P}\cup\mathcal{R} \subseteq \succsim$, let $(\mathcal{P}, \mathcal{R})$ have the tuple property where $\mathcal{R}$ is non-overlapping and where there are no critical pairs between $\mathcal{R}$ and $\mathcal{P}$.*[17] *Let $\mathcal{F}'$, $\mathcal{V}'$ be typed variants of $\mathcal{P} \cup \mathcal{R}$'s signature such that $\mathcal{P} \cup \mathcal{R}$ is well typed. Finally, let $s \to t \in \mathcal{P}$ with $\mathcal{R}^{\succ} \models_{ind}^{\mathcal{F}',\mathcal{V}'} t^{\succ} = \mathsf{tt}$. Then the following DP processor Proc is sound.*

$$Proc(\,(\mathcal{P}, \mathcal{R})\,) = \{\ (\mathcal{P} \setminus \{s \to t\},\, \mathcal{R}),\ \ (DP(\mathcal{R}),\, \mathcal{R})\ \}$$

Proof Suppose that there is an infinite innermost $(\mathcal{P}, \mathcal{R})$-chain, i.e., that $\mathcal{P} \cup \mathcal{R}$ is not innermost terminating. By persistence of innermost termination (Theorem 7), there is a well-typed term that is not innermost terminating w.r.t. $\mathcal{P}\cup\mathcal{R}$. Let q be a minimal such term (i.e., q's proper subterms are innermost terminating). Due to the tuple property, w.l.o.g. q either contains no tuple symbol or q contains a tuple symbol only at the root. In the first case, only $\mathcal{R}$-rules can reduce q. Thus, $\mathcal{R}$ is not innermost terminating and there is an infinite innermost $(DP(\mathcal{R}),\, \mathcal{R})$-chain.

Now let $\mathcal{R}$ be innermost terminating. So $\mathrm{root}(q)$ is a tuple symbol and q contains no further occurrences of tuple symbols. Hence, in q's infinite innermost $\mathcal{P} \cup \mathcal{R}$-reduction, $\mathcal{R}$-rules are only applied below the root and $\mathcal{P}$-rules are only applied on the root

[17] More precisely, for all $v \to w$ in $\mathcal{P}$, non-variable subterms of v may not unify with left-hand sides of rules from $\mathcal{R}$ (after variable renaming).

position. Moreover, there are infinitely many $\mathcal{P}$-steps. Hence, this infinite reduction corresponds to an infinite innermost $(\mathcal{P},\mathcal{R})$-chain $s_1 \to t_1, s_2 \to t_2, \ldots$ where $t_i\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} s_{i+1}\sigma$ for all i and all occurring terms are well typed.

Next we show that due to innermost termination of $\mathcal{R}$, there is even an infinite innermost $(\mathcal{P},\mathcal{R})$-chain on well-typed *ground* terms. As shown by Example 8, this step would not hold without innermost termination of $\mathcal{R}$. In other words, if $\mathcal{R}$ is not innermost terminating, then the existence of an infinite innermost $(\mathcal{P},\mathcal{R})$-chain does not imply the existence of an infinite innermost $(\mathcal{P},\mathcal{R})$-chain on *ground* terms. Let δ instantiate all variables in $s_1\sigma$ by ground terms of the corresponding type. (Recall that in any typed variant there are such ground terms.) We define the normal substitution σ' such that $\sigma'(x)$ is the $\mathcal{R}$-normal form of $x\sigma\delta$ for all variables x. This normal form must exist since $\mathcal{R}$ is innermost terminating and it is unique since $\mathcal{R}$ is non-overlapping. Clearly, $t_i\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} s_{i+1}\sigma$ implies $t_i\sigma\delta \to^{*}_{\mathcal{R}} s_{i+1}\sigma\delta$, i.e., $t_i\sigma' \to^{*}_{\mathcal{R}} s_{i+1}\sigma'$. As left-hand sides s_i of DPs do not overlap with rules of $\mathcal{R}$, all $s_i\sigma'$ are in normal form. Due to non-overlappingness of $\mathcal{R}$, $s_{i+1}\sigma'$ is the *only* normal form of $t_i\sigma'$ and thus, it can also be reached by innermost steps, i.e., $t_i\sigma' \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} s_{i+1}\sigma'$. Hence, there is an infinite innermost $(\mathcal{P},\mathcal{R})$-chain on well-typed *ground* terms.

If this chain does not contain infinitely many variable-renamed copies of the DP $s \to t$, then its tail is an infinite innermost $(\mathcal{P} \setminus \{s \to t\}, \mathcal{R})$-chain. Otherwise, $s_{i_1} \to t_{i_1}, s_{i_2} \to t_{i_2}, \ldots$ are variable-renamed copies of $s \to t$ and thus, $t_{i_1}\sigma', t_{i_2}\sigma', \ldots$ are well-typed ground instantiations of t. As $\mathcal{R}^{\succ} \models_{ind} t^{\succ} = \mathsf{tt}$, we have $(t_{i_j}\sigma')^{\succ} = t_{i_j}^{\succ}\sigma' \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$ for all j. Since $t_{i_j}\sigma' \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} s_{i_j+1}\sigma'$, Lemma 10 implies $t_{i_j}\sigma' \succ s_{i_j+1}\sigma'$ for all (infinitely many) j. Moreover, $s_i\sigma' \succsim t_i\sigma'$ and $t_i\sigma' \succsim s_{i+1}\sigma'$ for all i, since $\mathcal{P} \cup \mathcal{R} \subseteq \succsim$. This contradicts the well-foundedness of $\succ$. □

In our example, we ended up with the DP problem $(\{(11)\}, \mathcal{R}'_{sort})$. To remove the DP (11) from the DP problem, we use an inductive theorem prover to prove

$$\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort} \models_{ind} \mathsf{del}^{\succ_{\mathcal{P}ol_1}}(\mathsf{max}(\mathsf{add}(x, xs)), \mathsf{add}(x, xs)) = \mathsf{tt}, \tag{23}$$

i.e., that every non-empty list contains its maximum. The tuple property and the non-overlappingness requirements in Theorem 12 are clearly fulfilled. Moreover, all rules decrease w.r.t. $\succsim_{\mathcal{P}ol_1}$. Hence, the induction processor results in the trivial problem $(\varnothing, \mathcal{R}'_{sort})$ and the problem $(DP(\mathcal{R}'_{sort}), \mathcal{R}'_{sort}) = (\{(2), ..., (10)\}, \mathcal{R}'_{sort})$. The dependency graph processor transforms the latter problem into the problems $(\mathcal{P}_i, \mathcal{R}'_{sort})$ with $1 \le i \le 4$ that had already been solved before, cf. Section 2. For example, both ACL2 and the inductive theorem prover included in AProVE prove (23) automatically. Thus, now AProVE can easily verify termination of the TRS $\mathcal{R}_{sort}$.

5 Automation

In Section 5.1, we present a method to automate the processor of Theorem 12. Afterwards, Section 5.2 discusses how to couple a termination prover like AProVE with inductive theorem provers like ACL2. Finally, experimental results are presented in Section 5.3.

5.1 Generating Orders

In order to automate the new processor of Theorem 12 one has to synthesize a suitable

reduction pair $(\succsim,\succ)$ such that $\mathcal{R}^{\succ} \models_{ind}^{\mathcal{F}',\mathcal{V}'} t^{\succ} = \mathtt{tt}$ holds for some dependency pair $s \to t$ from $\mathcal{P}$. Moreover, both $\mathcal{P}$ and $\mathcal{R}$ have to be weakly decreasing w.r.t. $\succsim$.

In the last years, several techniques have been developed to search for reduction pairs satisfying a given set of inequalities between terms. Typically, one chooses a specific family of orders (e.g., recursive path orders with argument filters [36], Knuth-Bendix orders with argument filters [44], polynomial or matrix orders of a certain shape and a certain range for the coefficients [11,12], etc.). Then the search problem is encoded into a SAT problem and an existing SAT solver is used to generate a reduction pair satisfying the term inequalities.

However, for the induction processor one should choose a reduction pair such that for some DP $s \to t$ the resulting conjecture $t^{\succ} = \mathtt{tt}$ will be inductively valid. Of course, this can hardly be expressed by simple inequalities between terms.

Therefore, we use the following approach for the automation. Recall that inductive validity of $t^{\succ} = \mathtt{tt}$ implies that in every innermost reduction of a normal instantiation $t\sigma$ to normal form, a strictly decreasing rule $\ell \to r$ is applied on a strongly monotonic position. By the definition of usable rules, $\mathcal{U}(t)$ contains all rules that can be applied in innermost reductions of normal instantiations of t. Hence, $\ell \to r$ must be from $\mathcal{U}(t)$. In other words, if we want to find a reduction pair where $t^{\succ} = \mathtt{tt}$ is inductively valid, then we can restrict ourselves to reduction pairs where $\ell \succ r$ holds for some $\ell \to r \in \mathcal{U}(t)$.

To restrict the search space for suitable reduction pairs further, we also take into account that the position where the strictly decreasing rule $\ell \to r$ is applied must be *strongly monotonic*. For any function symbol f, let $\mathit{almon}(t, f)$ be a set of constraints which ensures that whenever an f-rule is applied in the reduction of a normal instantiation of t, then this application *al*ways takes place on a strongly *mon*otonic position. Then we will only search for reduction pairs where there exist a DP $s \to t$ and a rule $\ell \to r \in \mathcal{U}(t)$ such that $\ell \succ r$ and such that the constraints $\mathit{almon}(t, \mathrm{root}(\ell))$ hold. Moreover, the reduction pair should of course orient all DPs in $\mathcal{P}$ and all rules in $\mathcal{R}$ weakly. So to automate the induction pair processor for a DP problem $(\mathcal{P}, \mathcal{R})$ as in Theorem 12, we search for a reduction pair satisfying the following constraint.

$$\bigwedge_{\ell \to r \in \mathcal{R}} \ell \succsim r \wedge \bigwedge_{s \to t \in \mathcal{P}} s \succsim t \wedge \bigvee_{s \to t \in \mathcal{P}} \bigvee_{\ell \to r \in \mathcal{U}(t)} \left(\ell \succ r \wedge \bigwedge_{\varphi \in \mathit{almon}(t, \mathrm{root}(\ell))} \varphi \right) \quad (24)$$

This constraint can be encoded into a propositional formula using existing techniques [11,12,36,44]. If a SAT solver finds a satisfying assignment for the propositional variables, then this solution corresponds to a reduction pair satisfying (24). Thus, for this reduction pair there exist a DP $s \to t \in \mathcal{P}$ and a rule $\ell \to r \in \mathcal{U}(t)$ such that $\ell \succ r$ and all constraints in $\mathit{almon}(t, \mathrm{root}(\ell))$ are true. Now we use an inductive theorem prover to prove $\mathcal{R}^{\succ} \models_{ind}^{\mathcal{F}',\mathcal{V}'} t^{\succ} = \mathtt{tt}$ for the most general typed variant $\mathcal{F}', \mathcal{V}'$ where $\mathcal{P} \cup \mathcal{R}$ is well typed. If this proof succeeds, we can remove $s \to t$ from the set of DPs $\mathcal{P}$ and obtain the new DP problems $(\mathcal{P} \setminus \{s \to t\}, \mathcal{R})$ and $(DP(\mathcal{R}), \mathcal{R})$ as in Theorem 12.

In a full termination prover (like AProVE), the induction processor should of course only be used if simple and fast other DP processors cannot be applied anymore. In these cases, we check whether (24) is satisfied and only then, the inductive theorem prover is called. This turned out to be a successful strategy in our experiments, cf. Section 5.3. In other words, the induction processor was applied whenever it was needed and we did not waste much time by trying to apply it in examples where it was not needed.

Note that the approach to use only reduction pairs satisfying (24) is just a heuristic. In other words, $t^{\succ} = \mathtt{tt}$ could be inductively valid although there is no strictly decreas-

ing rule $\ell \to r$ in $\mathcal{U}(t)$ where all constraints from $almon(t, \mathrm{root}(\ell))$ hold. For example, it could be that not *all* applications of $\ell \to r$ are on strongly monotonic positions, but nevertheless in every reduction of a normal instantiation of t, there is *some* application of $\ell \to r$ on a strongly monotonic position. For instance, suppose that in our example, we replaced the rule

$$\mathsf{sort}(\mathsf{add}(x, xs)) \to \mathsf{add}(\mathsf{max}(\mathsf{add}(x, xs)),\ \mathsf{sort}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x, xs)),\ \mathsf{add}(x, xs))))$$

by the following rule where we added another application of del (which we underlined for readability):

$$\mathsf{sort}(\mathsf{add}(x, xs)) \to \mathsf{add}(\mathsf{max}(\mathsf{add}(x, xs)),\ \mathsf{sort}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x,\ \underline{\mathsf{del}(x,xs)}\ )),\ \mathsf{add}(x, xs))))$$

In the right-hand side t of the resulting SORT-DP, now there would also be an occurrence of del in the first argument of the original outer del. However, del's first argument position is not strongly monotonic if one uses the polynomial order $\succ_{\mathcal{P}ol_1}$. Consequently, the strictly decreasing if_2-rule would now be applied both on a strongly monotonic position and on a position that is not strongly monotonic. Hence, the constraints in $almon(t, \mathsf{if}_2)$ would not hold although $t^{\succ} = \mathsf{tt}$ would be inductively valid as before.

It remains to explain how the constraints $almon(t, f)$ in (24) are constructed for arbitrary terms t and function symbols f. In the following definition, $\mathcal{I}nvokes(f)$ are all terms t whose usable rules include f-rules. Thus, $\mathcal{I}nvokes(f)$ is an approximation (i.e., a superset) of all those terms t where an f-rule could be used when rewriting some normal instantiation $t\sigma$ to normal form. Now $almon(t, f)$ is a set of monotonicity constraints of the form "$i \in mon_{\succ}(g)$". As before, such a constraint means that the i-th argument position of g is strongly monotonic w.r.t. the order $\succ$. Suppose that t has the form $g(t_1, \ldots, t_n)$ and we want to construct the constraints for $almon(t, f)$. Clearly, if a subterm t_i can lead to the application of an f-rule (i.e., if $t_i \in \mathcal{I}nvokes(f)$), then one has to require $i \in mon_{\succ}(g)$ and $almon(t_i, f)$. Moreover, one also requires $almon(r, f)$ for all right-hand sides r of g-rules that could lead to an application of an f-rule (i.e., right-hand sides r where $r \in \mathcal{I}nvokes(f)$).

Definition 13 (*almon*) For any function symbol f, let $\mathcal{I}nvokes(f)$ be the set of all terms t where $\mathcal{U}(t) \cap Rls(f) \neq \varnothing$. For any term $t = g(t_1, \ldots, t_n)$, we define $almon(t, f)$ as the smallest set such that

$$\begin{aligned} almon(g(t_1, \ldots, t_n), f) = \quad & \textstyle\bigcup_{\ell \to r \in Rls(g),\, r \in \mathcal{I}nvokes(f)} almon(r, f) \\ & \cup \textstyle\bigcup_{1 \le i \le n,\, t_i \in \mathcal{I}nvokes(f)} \big(\{i \in mon_{\succ}(g)\} \ \cup\ almon(t_i, f) \big) \end{aligned}$$

As an example, consider the term $t = \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{add}(x, xs)), \mathsf{add}(x, xs)))$ from our leading example. To make sure that all reductions with the strictly decreasing if_2-rule take place on strictly monotonic positions, we would require $almon(t, \mathsf{if}_2)$. This is the following set of constraints:

$$\begin{aligned} & almon(t, \mathsf{if}_2) \\ = \ & \{1 \in mon_{\succ}(\mathsf{SORT})\} \cup \\ & almon(\mathsf{del}(\mathsf{max}(\mathsf{add}(x, xs)), \mathsf{add}(x, xs)), \mathsf{if}_2) && \text{since } \mathsf{del}(\ldots) \in \mathcal{I}nvokes(\mathsf{if}_2) \\ = \ & \{1 \in mon_{\succ}(\mathsf{SORT})\} \cup && \text{since the right-hand side of } \mathsf{del}\text{'s} \\ & almon(\mathsf{if}_2(\mathsf{eql}(x, y), x, y, xs), \mathsf{if}_2) && \text{second rule is in } \mathcal{I}nvokes(\mathsf{if}_2) \end{aligned}$$

$= \{1 \in mon_{\succ}(\mathsf{SORT})\} \cup$ since the right-hand side of if_2's
$almon(\mathsf{add}(y, \mathsf{del}(x, xs)), \mathsf{if}_2)$ second rule is in $\mathcal{I}nvokes(\mathsf{if}_2)$

$= \{1 \in mon_{\succ}(\mathsf{SORT}), 2 \in mon_{\succ}(\mathsf{add})\} \cup$
$almon(\mathsf{del}(x, xs), \mathsf{if}_2)$ since $\mathsf{del}(\ldots) \in \mathcal{I}nvokes(\mathsf{if}_2)$

$= \{1 \in mon_{\succ}(\mathsf{SORT}), 2 \in mon_{\succ}(\mathsf{add})\}$

These two monotonicity constraints are indeed satisfied for the polynomial order $\succ_{\mathcal{P}ol_1}$. Encoding such monotonicity constraints into propositional formulas (in order to use SAT solving for the overall constraint (24)) is easily possible.

The following lemma shows that our definition of *almon* is "sound". In other words, the truth of $almon(t, f)$ indeed implies that for every normal instantiation of t, every reduction with an f-rule takes place on a strongly monotonic position.

Lemma 14 (Soundness of *almon*) *Let t be a term and σ be a normal substitution such that $t\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} u[\ell\delta]_\pi \xrightarrow{\mathsf{i}}_{\mathcal{R}} u[r\delta]_\pi$ for a rule $\ell \to r \in \mathcal{R}$. Let $\mathrm{root}(\ell) = f$ and let all constraints in $almon(t, f)$ hold. Then π is a strongly monotonic position in the term u.*

Proof We use induction on the lexicographic combination of the length of the reduction from $t\sigma$ to $u[\ell\delta]_\pi$ and of the structure of t.

First regard the case where $t\sigma = u[\ell\delta]_\pi$. Since σ is a normal substitution, this means $\mathrm{root}(t|_\pi) = f$ and $t|_\pi\sigma = \ell\delta$. Here, we show the claim that if t contains f at some position π and all constraints in $almon(t, f)$ hold, then π is a strongly monotonic position in t. This claim is easily proved by induction on π. If π is the top position, then π is obviously strongly monotonic. Otherwise, if $\pi = i\,\pi'$, then $t = g(t_1, \ldots, t_n)$ where $t_i \in \mathcal{I}nvokes(f)$. Thus, the truth of $almon(t, f)$ implies $i \in mon_{\succ}(g)$ and the truth of $almon(t_i, f)$. Hence, π' is strongly monotonic in t_i by the induction hypothesis. Together, this means that π is strongly monotonic in t.

Now regard the case where $t\sigma \neq u[\ell\delta]_\pi$ and where none of the reduction steps in $t\sigma \xrightarrow{\mathsf{i}}{}^+_{\mathcal{R}} u[\ell\delta]_\pi$ takes place at the root. Hence, we have $t = g(t_1, \ldots, t_n)$, $u = g(u_1, \ldots, u_n)$, and $\pi = i\,\pi'$ for some $1 \leq i \leq n$. Thus, $t_i \in \mathcal{I}nvokes(f)$ and the truth of $almon(t, f)$ implies $i \in mon_{\succ}(g)$ and the truth of $almon(t_i, f)$. Since the reduction from $t_i\sigma$ to $u_i[\ell\delta]_{\pi'}$ is at most as long as the reduction from $t\sigma$ to $u[\ell\delta]_\pi$ and as t_i is a proper subterm of t, by the induction hypothesis π' is a strongly monotonic position in u_i and thus, π is a strongly monotonic position in u.

Finally, we regard the case where the reduction from $t\sigma$ to $u[\ell\delta]_\pi$ has the form $t\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} \ell'\delta' \xrightarrow{\mathsf{i}}_{\mathcal{R}} r'\delta' \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} u[\ell\delta]_\pi$ and where the step from $\ell'\delta'$ to $r'\delta'$ is the first reduction at the root position. Since δ' is a normal substitution, we have $r' \in \mathcal{I}nvokes(f)$. Hence, as $\mathrm{root}(t) = \mathrm{root}(\ell')$, the truth of $almon(t, f)$ implies that the constraints in $almon(r', f)$ are also true. As the reduction from $r'\delta'$ to $u[\ell\delta]_\pi$ is shorter than the reduction from $t\sigma$ to $u[\ell\delta]_\pi$, the induction hypothesis implies that π is a strongly monotonic position in u. □

5.2 Connecting Termination Tools and Inductive Theorem Provers

We implemented the new processor of Theorem 12 in our termination tool AProVE [15] and coupled it with the small inductive theorem prover for many-sorted conditional term rewriting that was already available in AProVE. To verify "$\mathcal{R}^{\succ} \models_{ind}^{\mathcal{F}',\mathcal{V}'} t^{\succ} = \mathsf{tt}$" as required in Condition (22), our theorem prover requires the TRS $\mathcal{R}^{\succ}$ to be innermost terminating, non-overlapping, sufficiently complete w.r.t. the used typed variants

$\mathcal{F}'$ and $\mathcal{V}'$ (i.e., well-typed ground terms in normal form may not contain defined symbols), and free of mutual recursion. For any left-linear TRS $\mathcal{R}$, innermost termination of $\mathcal{R}^{\succ}$ is implied by innermost termination of $\mathcal{R}$, which is checked by the processor in Theorem 12 anyway. Similarly, non-overlappingness of $\mathcal{R}^{\succ}$ is equivalent to non-overlappingness of $\mathcal{R}$ which is already required in Theorem 12. For many classes of TRSs, sufficient completeness is also easy to check automatically, cf. e.g. [23]. If the TRS is not sufficiently complete, one can automatically add rules for the missing cases where the right-hand sides are arbitrary ground terms of the corresponding type. Finally, for TRSs that encode conditions by mutually recursive unconditional rules, it is easily possible to automatically transform these rules into conditional rules without mutual recursion again. As an example, consider the four rules from $\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort}$ defining the mutually recursive functions $\mathsf{del}^{\succ}$ and $\mathsf{if}_2^{\succ}$:

$$\begin{aligned}
\mathsf{del}^{\succ}(x, \mathsf{empty}) &\to \mathsf{ff}\\
\mathsf{del}^{\succ}(x, \mathsf{add}(y, xs)) &\to \mathsf{if}_2^{\succ}(\mathsf{eql}(x, y), x, y, xs)\\
\mathsf{if}_2^{\succ}(\mathsf{true}, x, y, xs) &\to \mathsf{tt}\\
\mathsf{if}_2^{\succ}(\mathsf{false}, x, y, xs) &\to \mathsf{del}^{\succ}(x, xs)
\end{aligned}$$

Our transformation replaces these rules by the following three conditional rewrite rules:

$$\begin{aligned}
\mathsf{del}^{\succ}(x, \mathsf{empty}) &\to \mathsf{ff}\\
\mathsf{del}^{\succ}(x, \mathsf{add}(y, xs)) &\to \mathsf{tt} \quad &\Leftarrow \quad \mathsf{eql}(x, y) \to^* \mathsf{true}\\
\mathsf{del}^{\succ}(x, \mathsf{add}(y, xs)) &\to \mathsf{del}^{\succ}(x, xs) \quad &\Leftarrow \quad \mathsf{eql}(x, y) \to^* \mathsf{false}
\end{aligned}$$

Afterwards, we can apply our built-in theorem prover to prove inductive validity of conjectures, as in Condition (22).

As stated before, in principle our approach can use any inductive theorem prover as a black box. To demonstrate this, we also coupled AProVE with a well-known powerful inductive theorem prover instead of using our own inductive prover inside AProVE. We experimented with the tool ACL2, as it offers a high degree of automation and can easily be controlled from the command line. Furthermore, many characteristics of Applicative Common Lisp (ACL) correspond to the conditional TRSs which we already used as input for our built-in theorem prover. User-defined functions in ACL are strict in all their arguments, i.e., termination of an ACL program corresponds to innermost termination. Furthermore, ACL functions are defined for all arguments and the evaluation of ACL expressions is deterministic. This corresponds to sufficient completeness and non-overlappingness of our conditional TRSs. There is also a non-strict `if` function in ACL that can be used to encode the conditions of our rewrite rules. To represent a constructor term t in ACL, we use a list that is headed by the constructor symbol $\mathrm{root}(t)$ and whose tail is the list of the arguments. Consider again Condition (23) which has to be shown in order to prove innermost termination of the TRS from Example 1.

$$\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort} \;\models_{ind}\; \mathsf{del}^{\succ_{\mathcal{P}ol_1}}(\mathsf{max}(\mathsf{add}(x, xs)), \mathsf{add}(x, xs)) = \mathsf{tt}.$$

We now explain how TRSs like $\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort}$ are automatically transformed into ACL programs, and how the conjecture "$\mathsf{del}^{\succ_{\mathcal{P}ol_1}}(\ldots) = \mathsf{tt}$" is automatically transformed into a conjecture about this ACL program, such that ACL2 can prove its inductive validity and such that this implies the original condition (23).

We encode $\mathsf{del}^{\succ_{\mathcal{P}ol_1}}$, max, eql, and ge by the ACL function symbols `del`, `max`, `eql`, and `ge`, respectively, as well as `add` and `tt` by the symbols `add` and `tt`, respectively. For equality, ACL offers the built-in function `eq`, and for defining a theorem it has the keyword `defthm`. Then we obtain the following ACL conjecture for our formula:

```
(defthm
  (eq
    (del (max '(add x xs)) '(add x xs))
    tt
  )
)
```

Unfortunately, with this encoding one has the drawback that ACL is untyped. But one needs types in order to prove Condition (23), cf. Footnote 16. In other words, the conjecture only holds for all ground terms of the corresponding types. The solution is to introduce a function `is-type` for each type which tests whether an ACL expression corresponds to a term of that type. To deconstruct an ACL list, we can use the predefined selectors `car` and `cdr` for selecting the head and the tail of a list, respectively.[18] In our example, we introduce the functions `is-bool`, `is-nat` and `is-list`. The function definition for `is-list` can be automatically generated as follows:

```
(defun isa-empty (xs)
  (and
    (consp xs)
    (eq 'empty (car xs))
    (eq (cdr xs) 'nil)
  )
)
(defun isa-add (xs)
  (and
    (consp xs)
    (consp (cdr xs))
    (consp (cddr xs))
    (eq 'add (car xs))
    (is-nat (cadr xs))
    (is-list (caddr xs))
    (eq (cdddr xs) 'nil)
  )
)
(defun is-list (xs)
  (or
    (isa-empty xs)
    (isa-add xs)
  )
)
```

In the definition of `is-list`, the first part of the disjunction tests if `xs` is a list corresponding to empty, i.e., if `xs` is `'(empty)` (which is short for `(quote (cons empty nil))`). The second part checks for a list `'(add y ys)` where `y` is of type nat while `ys` is of type list. Now we can add the types of the variables in our conjecture as a premise:

[18] For brevity, ACL provides abbreviations like `(cadr x)` instead of `(car (cdr x))`, etc.

```
(defthm
  (implies
    (and (is-nat x) (is-list xs))
    (eq
      (del (max '(add x xs)) '(add x xs))
      tt
    )
  )
)
```

The remaining problem is that we need to encode the rules of $\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort}$ as ACL functions. The main task is to transform the pattern matching in the rewrite rules to a case analysis using the selectors `car` and `cdr`. By using these selectors, we can determine which rule would have been applied in the TRS. This rule is always unique as the TRS was non-overlapping. Together with sufficient completeness, this means that we do not need to check any condition for the last case of the case analysis. In our example, we obtain the following ACL function definition for $\mathsf{del}^{\succ_{\mathcal{P}ol_1}}$ which results from the three unconditional rules above. Note that we add a test for the type of the argument as a condition and return an arbitrary ground term of the correct type (e.g., `empty`) if the function is called with an argument of the wrong type.

```
(defun del (x xs)
  (if (and (is-nat x) (is-list xs))
    (if (isa-empty xs)
      'ff
      (if (and (isa-add xs) (eq (equal x (car xs)) 'true))
        'tt
        (del x (cdr xs))
      )
    )
    'empty
  )
)
```

By applying the above transformation also to max, eql, and ge, we obtain a complete definition of the functions used in our conjecture.

For ACL2's automation to be successful on these examples, we add a few hints to the ACL program. The hint `(set-ruler-extenders :all)` is needed to automatically obtain measure functions for the termination proofs of the generated auxiliary functions. For algorithms working on lists, our experiments revealed that generalization often turns a true subgoal into a false one. Thus, on conjectures involving types with a constructor of arity greater than one, we disable generalizations using "`:hints (("Goal" :do-not '(generalize)))`". Finally, as ACL2 is very efficient, we use `(with-prover-time-limit 3 ...)` around `defthm` to limit the proof time to three seconds.

In this way, we have obtained a fully automated system where conjectures generated by AProVE according to the new method of the present paper are automatically transformed into conjectures given to ACL2. Depending on the success of ACL2's proof attempt, AProVE continues the termination proof, possibly calling ACL2 again with new proof goals.

5.3 Experimental Results

To demonstrate the power of our method, [1] features a collection of 19 typical TRSs where an inductive argument is needed for the termination proof. This collection contains several TRSs computing classical arithmetical algorithms as well as many TRSs with standard algorithms for list manipulation like sorting, reversing, etc. The previous version of AProVE was already the most powerful tool for termination of term rewriting at the *Termination Competitions*. Nevertheless, this previous AProVE version as well as all other tools in the competition failed on all 19 examples. In contrast, with a time limit of 60 seconds per example, our new version of AProVE automatically proves termination of 16 of them. At the same time, the new version of AProVE is as successful as the previous one on the remaining examples of the *Termination Problem Data Base*, which is the collection of examples used in the termination competition.

The version of AProVE using our built-in theorem prover and the version using ACL2 both succeed on the same set of examples. The theorems that have to be shown by the inductive prover concern inductive properties of natural numbers (in the notation with "0" and "s") and of lists. These theorems are rather easy to show for the prover and, consequently, solving times are low. There is also no significant difference in the speed of the termination proof when using one or the other inductive prover. Thus, our experiments indicate that one can indeed use *any* inductive theorem prover as a black box to verify the conjectures resulting from our technique. The three examples of the 19 TRSs where AProVE fails would require more advanced termination techniques (like narrowing of rules and the use of more sophisticated orders). In other words, the reason for failure is not due to the inductive theorem prover.

On the web site [1] we present the details of our experiments. The web site also lists the respective conjectures that were automatically generated and given to ACL2. All experiments we conducted on a 2.67 GHz Intel Core i7 CPU running Linux x86_64. We used ACL2 3.6 with SBCL 1.0.29.11 as the Lisp backend.

6 Conclusion

We introduced a new processor in the DP framework which can handle TRSs that terminate because of inductive properties of their algorithms. This processor automatically tries to extract these properties and transforms them into conjectures which are passed to an inductive theorem prover for verification. To obtain a powerful method, we used the new result that innermost termination of many-sorted rewriting and of unsorted rewriting is equivalent (i.e., that innermost termination is persistent). Therefore, it suffices to prove these conjectures only for well-typed terms, even though the original TRSs under examination are untyped. So in contrast to previous approaches that use inductive theorem provers for termination analysis (cf. Section 1), our automation can search for arbitrary reduction pairs instead of being restricted to a fixed small set of orders.

We implemented our approach in the termination tool AProVE and coupled it both with the powerful prover ACL2 and with a small inductive theorem prover within the AProVE tool. Our experiments show that by our new results, the power of AProVE increases strictly, i.e., it now succeeds on several examples where termination could not be proved automatically by any tool before.

Thus, the present paper is a substantial advance in automated termination proving, since it allows the first combination of powerful TRS termination tools with inductive theorem provers. For details on our experiments and to access our implementation via a web-interface, we refer to [1].

In future work, it would be desirable to extend recent approaches for the *certification* of automatically generated termination proofs [10,39] also to the new techniques presented in this paper. Apart from an extension of certification to *innermost* termination, this would of course also involve the certification of the automatically generated inductive proofs.

Acknowledgements We are very grateful to Aart Middeldorp and Hans Zantema for indicating that Theorem 7 follows from results of [35]. We also would like to thank Matt Kaufmann and Sandip Ray for support in understanding and tuning the automation of ACL2. Finally, we thank the referees for many helpful suggestions.

References

1. AProVE web site `http://aprove.informatik.rwth-aachen.de/eval/Induction/`.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
4. A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, 1987.
5. A. Bouhoula and M. Rusinowitch. SPIKE: A system for automatic inductive proofs. In *Proc. AMAST'95*, LNCS 936, pages 576–577, 1995.
6. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
7. J. Brauburger and J. Giesl. Termination analysis by inductive evaluation. In *Proc. CADE'98*, LNAI 1421, pages 254–269, 1998.
8. A. Bundy. The automation of proof by mathematical induction. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 845–911. Elsevier and MIT Press, 2001.
9. H. Comon. Inductionless induction. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 913–962. Elsevier and MIT Press, 2001.
10. E. Contejean, A. Paskevich, X. Urbain, P. Courtieu, O. Pons, and J. Forest. A3PAT, an approach for certified automated termination proofs. In *Proc. PEPM'10*, pages 63–72. ACM Press, 2010.
11. J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2,3):195–220, 2008.
12. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT'07*, LNCS 4501, pages 340–354, 2007.
13. J. Giesl. Termination analysis for functional programs using term orderings. In *Proc. SAS'95*, LNCS 983, pages 154–171, 1995.
14. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR'04*, LNAI 3452, pages 301–331, 2005.
15. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR'06*, LNAI 4130, pages 281–286, 2006.
16. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
17. J. Giesl, R. Thiemann, S. Swiderski, and P. Schneider-Kamp. Proving termination by bounded increase. In *Proc. CADE'07*, LNAI 4603, pages 443–459, 2007.
18. J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming*

Languages and Systems, 2010. To appear. Preliminary version appeared in *Proc. RTA'06*, LNCS 4098, pages 297-312, 2006.
19. I. Gnaedig and H. Kirchner. Termination of rewriting under strategies. *ACM Transactions on Computational Logic*, 10(3), 2008.
20. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.
21. N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.
22. D. Kapur and D. R. Musser. Proof by consistency. *Artificial Intelligence*, 31(2):125–157, 1987.
23. D. Kapur, P. Narendran, and H. Zhang. On sufficient completeness and related properties of term rewriting systems. *Acta Informatica*, 24:395–415, 1987.
24. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer, 2000.
25. M. Korp and A. Middeldorp. Beyond dependency graphs. In *Proc. CADE'09*, LNAI 5663, pages 339–354, 2009.
26. A. Krauss. Certified size-change termination. In *Proc. CADE'07*, LNAI 4603, pages 460–475, 2007.
27. D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.
28. S. Lucas and J. Meseguer. Order-sorted dependency pairs. In *Proc. PPDP'08*, pages 108–119. ACM Press, 2008.
29. P. Manolios and D. Vroon. Termination analysis with calling context graphs. In *Proc. CAV'06*, LNCS 4144, pages 401–414, 2006.
30. A. Middeldorp and H. Zantema. Personal communication, 2008.
31. E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):73–116, 2001.
32. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA'10*, LIPIcs 6, pages 259–276, 2010.
33. S. E. Panitz and M. Schmidt-Schauß. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In *Proc. SAS'97*, LNCS 1302, pages 345–360, 1997.
34. D. A. Plaisted. Equational reasoning and term rewriting systems. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, pages 273–364. Oxford University Press, 1993.
35. J. van de Pol. Modularity in many-sorted term rewriting. Master's thesis, Utrecht University, 1992. Available from `http://homepages.cwi.nl/~vdpol/papers/`.
36. P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving termination using recursive path orders and SAT solving. In *Proc. FroCoS'07*, LNAI 4720, pages 267–282, 2007.
37. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1), 2009.
38. S. Swiderski, M. Parting, J. Giesl, C. Fuhs, and P. Schneider-Kamp. Termination analysis by dependency pairs and inductive theorem proving. In *Proc. CADE'09*, LNAI 5663, pages 322–338, 2009.
39. R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, LNCS 5674, pages 452–468, 2009.
40. Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, 1987.
41. C. Walther. Mathematical induction. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–228. Oxford University Press, 1994.
42. C. Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.
43. C. Walther and S. Schweitzer. About VeriFun. In *Proc. CADE'03*, LNAI 2741, pages 322–327, 2003.
44. H. Zankl, N. Hirokawa, and A. Middeldorp. KBO orientability. *Journal of Automated Reasoning*, 43(2):173–201, 2009.
45. H. Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation*, 17(1):23–50, 1994.
46. H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In *Proc. CADE'88*, LNAI 310, pages 162–181, 1988.