Certification of Termination Proofs using CeTA*

René Thiemann and Christian Sternagel

Institute of Computer Science, University of Innsbruck, Austria
{rene.thiemann|christian.sternagel}@uibk.ac.at

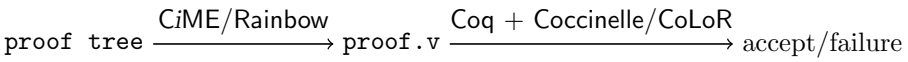
Abstract. There are many automatic tools to prove termination of term rewrite systems, nowadays. Most of these tools use a combination of many complex termination criteria. Hence generated proofs may be of tremendous size, which makes it very tedious (if not impossible) for humans to check those proofs for correctness.
In this paper we use the theorem prover Isabelle/HOL to automatically certify termination proofs. To this end, we first formalized the required theory of term rewriting including three major termination criteria: dependency pairs, dependency graphs, and reduction pairs. Second, for each of these techniques we developed an executable check which guarantees the correct application of that technique as it occurs in the generated proofs. Moreover, if a proof is not accepted, a readable error message is displayed. Finally, we used Isabelle's code generation facilities to generate a highly efficient and certified Haskell program, CeTA, which can be used to certify termination proofs without even having Isabelle installed.

1 Introduction

Termination provers for term rewrite systems (TRSs) became more and more powerful in the last years. One reason is that a proof of termination no longer is just some reduction order which contains the rewrite relation of the TRS. Currently, most provers construct a proof in the dependency pair framework which allows to combine basic termination techniques in a flexible way. Then a termination proof is a tree where at each node a specific technique has been applied. So instead of stating the precedence of some lexicographic path order (LPO) or giving some polynomial interpretation, current termination provers return proof trees which reach sizes of several megabytes. Hence, it would be too much work to check by hand whether these trees really form a valid proof.

That we cannot blindly trust the output of termination provers is regularly demonstrated: Every now and then some tool delivers a faulty proof for some TRS. But most often this is only detected if there is some other prover giving the opposite answer on the same TRS, i.e., that it is nonterminating. To solve this problem, in the last years two systems have been developed which automatically certify or reject a generated termination proof: C*i*ME/Coccinelle [4,6] and Rainbow/CoLoR [3] where Coccinelle and CoLoR are libraries on rewriting for Coq (http://coq.inria.fr), and C*i*ME and Rainbow are used to convert proof trees into Coq-proofs which heavily rely on the theorems within those libraries.

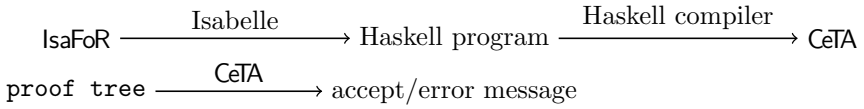
* This research is supported by FWF (Austrian Science Fund) project P18763.



In this paper we present a new combination, CeTA/IsaFoR, to automatically certify termination proofs. Note that the system design has two major differences in comparison to the two existing ones. First, our library IsaFoR (*Isabelle Formalization of Rewriting*, containing 173 definitions, 863 theorems, and 269 functions) is written for the theorem prover Isabelle/HOL[1] [16] and not for Coq.

Second, and more important, instead of generating for each proof tree a new Coq-proof using the auxiliary tools CiME/Rainbow, our library IsaFoR contains several executable "check"-functions (within Isabelle) for each termination technique we formalized. We have formally proven that whenever such a check is accepted, then the termination technique is applied correctly. Hence, we do not need to create an individual Isabelle-proof for each proof tree, but just call the "check"-function for checking the whole tree (which does nothing else but calling the separate checks for each termination technique occurring in the tree). This second difference has several advantages:

- In the other two systems, whenever a proof is not accepted, the user just gets a Coq-error message that some step in the generated Coq-proof failed. In contrast, our functions deliver error messages using notions of term rewriting.
- Since the analysis of the proof trees in IsaFoR is performed by executable functions, we can just apply Isabelle's code-generator [11] to create a certified Haskell program [17], CeTA, leading to the following workflow.



  Hence, to use our certifier CeTA (*Certified Termination Analysis*) you do not have to install any theorem prover, but just execute some binary. Moreover, the runtime of certification is reduced significantly. Whereas the other two approaches take more than one hour to certify all ($\leq$ 580) proofs during the last certified termination competition, CeTA needs less than two minutes for all (786) proofs that it can handle. Note that CeTA can also be used for modular certification. Each single application of a termination technique can be certified—just call the corresponding Haskell-function.

Concerning the techniques that have been formalized, the other two systems offer techniques that are not present in IsaFoR, e.g., LPO or matrix interpretations. Nevertheless, we also feature one new technique that has not been certified so far. Whereas currently only the initial dependency graph estimation of [1] has been certified, we integrated the most powerful estimation which does not require tree automata techniques and is based on a combination of [9,12] where the function tcap is required. Initial problems in the formalization of tcap led to the development of etcap, an equivalent but more efficient version of tcap which

[1] In the remainder of this paper we just write Isabelle instead of Isabelle/HOL.

is also beneficial for termination provers. Replacing tcap by etcap within the termination prover TTT2 [14] reduced the time to estimate the dependency graph by a factor of 2. We will also explain, how to reduce the number of edges that have to be inspected when checking graph decompositions.

Another benefit of our system is its robustness. Every proof which uses weaker techniques than those formalized in IsaFoR is accepted. For example, termination provers can use the graph estimation of [1], as it is subsumed by our estimation.

The paper is structured as follows. In Sect. 2 we recapitulate the required notions and notations of term rewriting and the dependency pair framework (DP framework). Here, we also introduce our formalization of term rewriting within IsaFoR. In Sect. 3–6 we explain our certification of the four termination techniques we currently support: dependency pairs (Sect. 3), dependency graph (Sect. 4), reduction pairs (Sect. 5), and combination of proofs in the dependency pair framework (Sect. 6). However, to increase readability we abstract from our concrete Isabelle code and present the checks for the techniques on a higher level. How we achieved readable error-messages while at the same time having maintainable Isabelle proofs is the topic of Sect. 7. We conclude in Sect. 8 where we show how CeTA is created from IsaFoR and where we give experimental data.

IsaFoR, CeTA, and all details about our experiments are available at CeTA's website http://cl-informatik.uibk.ac.at/software/ceta.

2 Formalizing Term Rewriting

We assume some basic knowledge of term rewriting [2]. Variables are denoted by x, y, z, etc., function symbols by f, g, h, etc., terms by s, t, u, etc., and substitutions by σ, μ, etc. Instead of $f(t_1, \ldots, t_n)$ we write $f(\boldsymbol{t}_n)$. The set of all variables occurring in term t is denoted by $\mathcal{V}\mathsf{ar}(t)$. By $\mathcal{T}(\mathcal{F}, \mathcal{V})$ we denote the set of terms over function symbols from $\mathcal{F}$ and variables from $\mathcal{V}$.

In the following we give an overview of our formalization of term rewriting in IsaFoR. Our main concern is *termination* of rewriting. This property—also known as *strong normalization*—can be stated without considering the structure of terms. Therefore it is part of our Isabelle theory `AbstractRewriting`. An abstract rewrite system (ARS) is represented by the type `('a×'a)set` in Isabelle. Strong normalization (`SN`) of a given ARS $\mathcal{A}$ is equivalent to the absence of an infinite sequence of $\mathcal{A}$-steps. On the lowest level we have to link our notion of strong normalization to the notion of well-foundedness as defined in Isabelle. This is an easy lemma since the only difference is the orientation of the relation, i.e., $\mathtt{SN}(\mathcal{A}) = \mathtt{wf}(\mathcal{A}^{-1})$. At this point we can be sure that our notion of strong normalization is valid.

Now we come to the level of first-order terms (in theory `Term`):

```
datatype ('f,'v)"term" = Var 'v | Fun 'f "('f,'v)term list"
```

Many concepts related to terms are formalized in `Term`, e.g., an induction scheme for terms (as used in textbooks), substitutions, contexts, the (proper) subterm relation etc.

By restricting the elements of some ARS to terms, we reach the level of TRSs

(in theory `Trs`), which in our formalization are just binary relations over terms.

Example 1. As an example, consider the following TRS, encoding rules for subtraction and division on natural numbers.

$$\begin{aligned} \mathsf{minus}(x, \mathsf{0}) &\to x & \mathsf{div}(\mathsf{0}, \mathsf{s}(y)) &\to \mathsf{0} \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{minus}(x, y) & \mathsf{div}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{s}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{aligned}$$

Given a TRS $\mathcal{R}$, $(\ell, r) \in \mathcal{R}$ means that ℓ is the lhs and r the rhs of a rule in $\mathcal{R}$ (usually written as $\ell \to r \in \mathcal{R}$). The *rewrite relation* induced by a TRS $\mathcal{R}$ is denoted by $\to_{\mathcal{R}}$ and has the following definition in IsaFoR:

Definition 2. *Term s rewrites to t by $\mathcal{R}$, iff there are a context C, a substitution σ, and a rule $\ell \to r \in \mathcal{R}$ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$.*

Note that this section contains the only parts where you have to trust our formalization, i.e., you have to believe that $\mathtt{SN}(\to_{\mathcal{R}})$ as defined in IsaFoR really describes "$\mathcal{R}$ is terminating."

3 Certifying Dependency Pairs

Before we introduce dependency pairs [1] formally and give some details about our Isabelle formalization, we recapitulate the ideas that led to the final definition (including a refinement proposed by Dershowitz [7]).

For a TRS $\mathcal{R}$, strong normalization means that there is no infinite derivation $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} t_3 \to_{\mathcal{R}} \cdots$. Additionally we can concentrate on derivations, where t_1 is minimal in the sense that all its proper subterms are terminating. Such terms are called *minimal nonterminating*. The set of all minimally nonterminating terms with respect to a TRS $\mathcal{R}$ is denoted by $\mathcal{T}^{\infty}_{\mathcal{R}}$. Observe that for every term $t \in \mathcal{T}^{\infty}_{\mathcal{R}}$ there is an initial part of an infinite derivation having a specific shape: A (possibly empty) derivation taking place below the root, followed by an application of some rule $\ell \to r \in \mathcal{R}$ at the root, i.e., $t \overset{>\varepsilon *}{\to}_{\mathcal{R}} \ell\sigma \overset{\varepsilon}{\to}_{\mathcal{R}} r\sigma$, for some substitution σ. Furthermore, since $r\sigma$ is nonterminating, there is some subterm u of r, such that $u\sigma \in \mathcal{T}^{\infty}_{\mathcal{R}}$, i.e., $r\sigma = C[u\sigma]$. Then the same reasoning can be used to get a root reduction of $u\sigma$, ..., cf. [1].

To get rid of the additional contexts C a new TRS, $\mathtt{DP}(\mathcal{R})$, is built.

Definition 3. *The set $\mathtt{DP}(\mathcal{R})$ of* dependency pairs *of $\mathcal{R}$ is defined as follows: For every rule $\ell \to r \in \mathcal{R}$, and every subterm u of r such that u is not a proper subterm of ℓ and such that the root of u is defined,*[2] *$\ell^\sharp \to u^\sharp$ is contained in $\mathtt{DP}(\mathcal{R})$. Here $t^\sharp$ is the same as t except that the root of t is marked with the special symbol $\sharp$.*

Example 4. The dependency pairs for the TRS from Ex. 1 consist of the rules

$$\mathsf{M}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{M}(x, y) \quad (\mathsf{MM}) \qquad \mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{M}(x, y) \quad (\mathsf{DM})$$
$$\mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{D}(\mathsf{minus}(x, y), \mathsf{s}(y)) \quad (\mathsf{DD})$$

where we write M instead of $\mathsf{minus}^\sharp$ and D instead of $\mathsf{div}^\sharp$ for brevity.

[2] A function symbol f is *defined* (w.r.t. $\mathcal{R}$) if there is some rule $f(\dots) \to r \in \mathcal{R}$.

Note that after switching to '$\sharp$'-terms, the derivation from above can be written as $t^\sharp \to_{\mathcal{R}}^* \ell^\sharp\sigma \to_{\mathtt{DP}(\mathcal{R})} u^\sharp\sigma$. Hence every nonterminating derivation starting at a term $t \in \mathcal{T}_{\mathcal{R}}^\infty$ can be transformed into an infinite derivation of the following shape where all $\to_{\mathtt{DP}(\mathcal{R})}$-steps are applied at the root.

$$t^\sharp \to_{\mathcal{R}}^* s_1^\sharp \to_{\mathtt{DP}(\mathcal{R})} t_1^\sharp \to_{\mathcal{R}}^* s_2^\sharp \to_{\mathtt{DP}(\mathcal{R})} t_2^\sharp \to_{\mathcal{R}}^* \cdots \tag{1}$$

Therefore, to prove termination of $\mathcal{R}$ it suffices to prove that there is no such derivation. To formalize DPs in Isabelle we modify the signature such that every function symbol now appears in a plain version and in a $\sharp$-version.

```
datatype 'f shp = Sharp 'f ("_♯") | Plain 'f ("_@")
```

Sharping a term is done via

```
fun plain :: "('f,'v)term => ('f shp,'v)term"
where "plain(Var x)    = Var x"
    | "plain(Fun f ss) = Fun f@ (map plain ss)"

fun sharp :: "('f,'v)term => ('f shp,'v)term"
where "sharp(Var x)    = Var x"
    | "sharp(Fun f ss) = Fun f♯ (map plain ss)"
```

Thus $t^\sharp$ in Def. 3 is the same as $\mathtt{sharp}(t)$. Since the function symbols in $\mathtt{DP}(\mathcal{R})$ are of type `'f shp` and the function symbols of $\mathcal{R}$ are of type `'f`, it is not possible to use the same TRS $\mathcal{R}$ in combination with $\mathtt{DP}(\mathcal{R})$. Thus, in our formalization we use the lifting ID—that just applies `plain` to all lhss and rhss in $\mathcal{R}$.

Considering this technicalities and omitting the initial derivation $t^\sharp \to_{\mathcal{R}}^* s_1^\sharp$ from the derivation (1), we obtain

$$s_1^\sharp \to_{\mathtt{DP}(\mathcal{R})} t_1^\sharp \to_{\mathtt{ID}(\mathcal{R})}^* s_2^\sharp \to_{\mathtt{DP}(\mathcal{R})} t_2^\sharp \to_{\mathtt{ID}(\mathcal{R})}^* \cdots$$

and hence a so called infinite $(\mathtt{DP}(\mathcal{R}), \mathtt{ID}(\mathcal{R}))$-*chain*. Then the corresponding *DP problem* $(\mathtt{DP}(\mathcal{R}), \mathtt{ID}(\mathcal{R}))$ is called to be not *finite*, cf. [8]. Notice that in IsaFoR a DP problem is just a pair of two TRSs over arbitrary signatures—similar to [8].

In IsaFoR an infinite chain[3] and finite DP problems are defined as follows.

```
fun ichain where "ichain(𝒫,ℛ) 𝒔 𝒕 𝝈 = (∀i.
  (𝒔 i,𝒕 i) ∈ 𝒫 ∧ (𝒕 i)·(𝝈 i) →ℛ* (𝒔(i+1))·(𝝈(i+1))"

fun finite_dpp where
  "finite_dpp(𝒫,ℛ) = (¬(∃𝒔 𝒕 𝝈. ichain (𝒫,ℛ) 𝒔 𝒕 𝝈))"
```

where '$t \cdot \sigma$' denotes the application of substitution σ to term t.

We formally established the connection between strong normalization and finiteness of the initial DP problem $(\mathtt{DP}(\mathcal{R}), \mathtt{ID}(\mathcal{R}))$. Although this is a well-known theorem, formalizing it in Isabelle was a major effort.

Theorem 5. $\mathtt{wf\_trs}(\mathcal{R}) \wedge \mathtt{finite\_dpp}(\mathtt{DP}(\mathcal{R}), \mathtt{ID}(\mathcal{R})) \longrightarrow \mathtt{SN}(\to_{\mathcal{R}})$.

[3] We also formalized *minimal* chains, but here only present chains for simplicity.

The additional premise $\texttt{wf\_trs}(\mathcal{R})$ ensures two well-formedness properties for $\mathcal{R}$, namely that for every $\ell \to r \in \mathcal{R}$, ℓ is not a variable and that $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(\ell)$.

At this point we can obviously switch from the problem of proving $\texttt{SN}(\to_{\mathcal{R}})$ for some TRS $\mathcal{R}$, to the problem of proving $\texttt{finite\_dpp}(\texttt{DP}(\mathcal{R}), \texttt{ID}(\mathcal{R}))$, and thus enter the realm of the DP framework [8]. Here, the current technique is to apply so-called *processors* to a DP problem, in order to get a set of simpler DP problems. This is done recursively, until the leafs of the so built tree consist of DP problems with empty $\mathcal{P}$-components (and therefore are trivially finite). For this to be correct, the applied processors need to be sound, i.e., every processor $Proc$ has to satisfy the implication

$$(\forall p \in Proc(\mathcal{P}, \mathcal{R}).\ \texttt{finite\_dpp}(p)) \longrightarrow \texttt{finite\_dpp}(\mathcal{P}, \mathcal{R})$$

for every input. The termination techniques that will be introduced in the following sections are all such (sound) processors.

So much to the underlying formalization. Now we will present how the check in IsaFoR certifies a set of DPs $\mathcal{P}$ that was generated by some termination tool for some TRS $\mathcal{R}$. To this end, the function `checkDPs` is used.

$$\texttt{checkDPs}(\mathcal{P},\mathcal{R}) = \texttt{checkWfTRS}(\mathcal{R}) \wedge \texttt{computeDPs}(\mathcal{R}) \subseteq \mathcal{P}$$

Here `checkWfTRS` checks the two well-formedness properties mentioned above (the difference between `wf_trs` and `checkWfTRS` is that only the latter is executable) and `computeDPs` uses Def. 3, which is currently the strongest definition of DPs. To have a robust system, the check does not require that exactly the set of DPs w.r.t. to Def. 3 is provided, but any superset is accepted. Hence we are also able to accept proofs from termination tools that use a weaker definition of $\texttt{DP}(\mathcal{R})$. The soundness result of `checkDPs` is formulated as follows in IsaFoR.

Theorem 6. *If* $\texttt{checkDPs}(\mathcal{P}, \mathcal{R})$ *is accepted then finiteness of* $(\mathcal{P}, \texttt{ID}(\mathcal{R}))$ *implies* $\texttt{SN}(\to_{\mathcal{R}})$.

4 Certifying the Dependency Graph Processor

One important processor to prove finiteness of a DP problem is based on the *dependency graph* [1,8]. The dependency graph of a DP problem $(\mathcal{P}, \mathcal{R})$ is a directed graph $\mathcal{G} = (\mathcal{P}, E)$ where $(s \to t, u \to v) \in E$ iff $s \to t, u \to v$ is a $(\mathcal{P}, \mathcal{R})$-chain. Hence, every infinite $(\mathcal{P}, \mathcal{R})$-chain corresponds to an infinite path in $\mathcal{G}$ and thus, must end in some strongly connected component (SCC) S of $\mathcal{G}$, provided that $\mathcal{P}$ contains only finitely many DPs. Dropping the initial DPs of the chain results in an infinite $(S, \mathcal{R})$-chain.[4] Hence, if for all SCCs S of $\mathcal{G}$ the DP problem $(S, \mathcal{R})$ is finite, then $(\mathcal{P}, \mathcal{R})$ is finite. In practice, this processor allows to prove termination of each block of mutual recursive functions separately.

To certify an application of the dependency graph processor there are two main challenges. First of all, we have to certify that a valid SCC decomposition of $\mathcal{G}$ is used, a purely graph-theoretical problem. Second, we have to generate the edges of $\mathcal{G}$. Since the dependency graph $\mathcal{G}$ is in general not computable, usually

[4] We identify an SCC S with the set of nodes S within that SCC.

estimated graphs $\mathcal{G}'$ are used which contain all edges of the real dependency graph $\mathcal{G}$. Hence, for the second problem we have to implement and certify one estimation of the dependency graph.

Notice that there are various estimations around and that the result of an SCC decomposition depends on the estimation that is used. Hence, it is not a good idea to implement the strongest estimation and then match the result of our decomposition against some given decomposition: problems arise if the termination prover used a weaker estimation and thus obtained larger SCCs.

Therefore, in the upcoming Sect. 4.1 about graph algorithms we just speak of decompositions where the components do not have to be SCCs. Moreover, we will also elaborate on how to minimize the number of tests $(s \to t, u \to v) \in E$. The reason is that in Sect. 4.2 we implemented one of the strongest dependency graph estimations where the test for an edge can become expensive. In Sect. 4.3 we finally show how to combine the results of Sections 4.1 and 4.2.

4.1 Certifying Graph Decompositions

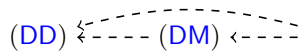
Instead of doing an SCC decomposition of a graph within IsaFoR we base our check on the decomposition that is provided by the termination prover. Essentially, we demand that the set of components is given as a list $\langle C_1, \ldots, C_k \rangle$ in topological order where the component with no incoming edges is listed first. Then we aim at certifying that every infinite path must end in some C_i. Note that the general idea of taking the topological sorted list as input was already publicly mentioned at the "Workshop on the certification of termination proofs" in 2007. In the following we present how we filled the details of this general idea.

The main idea is to ensure that all edges $(p, q) \in E$ correspond to a step forward in the list $\langle C_1, \ldots, C_k \rangle$, i.e., $(p, q) \in C_i \times C_j$ where $i \leq j$. However, iterating over all edges of $\mathcal{G}$ will be costly, because it requires to perform the test $(p, q) \in E$ for all possible edges $(p, q) \in \mathcal{P} \times \mathcal{P}$. To overcome this problem we do not iterate over the edges but over $\mathcal{P}$. To be more precise, we check that

$$\forall (p, q) \in \mathcal{P} \times \mathcal{P}.\ (\exists i \leq j.\ (p, q) \in \mathcal{P}_i \times \mathcal{P}_j) \vee (p, q) \notin E \tag{2}$$

where the latter part of the disjunction is computed only on demand. Thus, only those edges have to be computed, which would contradict a valid decomposition.

Example 7. Consider the set of nodes $\mathcal{P} = \{(\mathsf{DD}), (\mathsf{DM}), (\mathsf{MM})\}$. Suppose that we have to check a decomposition of $\mathcal{P}$ into $L = \langle \{(\mathsf{DD})\}, \{(\mathsf{DM})\}, \{(\mathsf{MM})\} \rangle$ for some graph $\mathcal{G} = (\mathcal{P}, E)$. Then our check has to ensure that the dashed edges in the following illustration do not belong to E.



It is easy to see that (2) is satisfied for every list of SCCs that is given in topological order. What is even more important, whenever there is a valid SCC decomposition of $\mathcal{G}$, then (2) is also satisfied for every subgraph. Hence, regardless of the dependency graph estimation a termination prover might have used, we accept it, as long as our estimation delivers less edges.

However, the criterion is still too relaxed, since we might cheat in the input by listing nodes twice. Consider $\mathcal{P} = \{p_1, \ldots, p_m\}$ where the corresponding graph is arbitrary and $L = \langle \{p_1\}, \ldots, \{p_m\}, \{p_1\}, \ldots, \{p_m\} \rangle$. Then trivially (2) is satisfied, because we can always take the source of edge (p_i, p_j) from the first part of L and the target from the second part of L. To prevent this kind of problem, our criterion demands that the sets C_i in L are pairwise disjoint.

Before we formally state our theorem, there is one last step to consider, namely the handling of singleton nodes which do not form an SCC on their own. Since we cannot easily infer at what position these nodes have to be inserted in the topological sorted list—this would amount to do an SCC decomposition on our own—we demand that they are contained in the list of components.[5]

To distinguish a singleton node without an edge to itself from a "real SCC", we require that the latter ones are marked. Then condition (2) is extended in a way that unmarked components may have no edge to themselves. The advantage of not marking a component is that our IsaFoR-theorem about graph decomposition states that every infinite path will end in some marked component, i.e., here the unmarked components can be ignored.

Theorem 8. *Let $L = \langle C_1, \ldots, C_k \rangle$ be a list of sets of nodes, some of them marked, let $\mathcal{G} = (\mathcal{P}, E)$ be a graph, let α be an infinite path of $\mathcal{G}$. If*

- $\forall (p,q) \in \mathcal{P} \times \mathcal{P}.\ (\exists i < j.\ (p,q) \in C_i \times C_j) \vee (\exists i.\ (p,q) \in C_i \times C_i \wedge C_i \text{ is marked}) \vee (p,q) \notin E$ *and*
- $\forall i \neq j.\ C_i \cap C_j = \emptyset$

then there is some suffix β of α and some marked C_i such that all nodes of β belong to C_i.

Example 9. If we continue with example Ex. 7 where only components {(MM)} and {(DD)} are marked, then our check also analyzes that $\mathcal{G}$ contains no edge from (DM) to itself. If it succeeds, every infinite path will in the end only contain nodes from {(DD)} or only nodes from {(MM)}. In this way, only 4 edges of $\mathcal{G}$ have to be calculated instead of analyzing all 9 possible edges in $\mathcal{P} \times \mathcal{P}$.

4.2 Certifying Dependency Graph Estimations

What is currently missing to certify an application of the dependency graph processor, is to check, whether a singleton edge is in the dependency graph or not. Hence, we have to estimate whether the sequence $s \to t$, $u \to v$ is a chain, i.e., whether there are substitutions σ and μ such that $t\sigma \to_{\mathcal{R}}^* u\mu$. An obvious solution is to just look at the root symbols of t and u—if they are different there is no way that the above condition is met (since all the steps in $t\sigma \to_{\mathcal{R}}^* u\mu$ take place below the root, by construction of the dependency pairs). Although efficient and often good enough, there are more advanced estimations around.

The estimation EDG [1] first replaces via an operation cap all variables and all subterms of t which have a defined root-symbol by distinct fresh variables. Then if $\mathsf{cap}(t)$ and u do not unify, it is guaranteed that there is no edge.

[5] Note that Tarjan's SCC decomposition algorithm produces exactly this list.

The estimation EDG*[12] does the same check and additionally uses the reversed TRS $\mathcal{R}^{-1} = \{r \to \ell \mid \ell \to r \in \mathcal{R}\}$, i.e., it uses the fact that $t\sigma \to^*_{\mathcal{R}} u\mu$ implies $u\mu \to^*_{\mathcal{R}^{-1}} t\sigma$ and checks whether $\mathsf{cap}(u)$ does not unify with t. Of course in the application of $\mathsf{cap}(u)$ we have to take the reversed rules into account (possibly changing the set of defined symbols) and it is not applicable if $\mathcal{R}$ contains a collapsing rule $\ell \to x$ where $x \in \mathcal{V}$.

The last estimation we consider is based on a better version of cap, called tcap [9]. It only replaces subterms with defined symbols by a fresh variable, if there is a rule that unifies with the corresponding subterm.

Definition 10. *Let $\mathcal{R}$ be a TRS.*

- $\mathsf{tcap}(f(\boldsymbol{t}_n)) = f(\mathsf{tcap}(t_1), \ldots, \mathsf{tcap}(t_n))$ *iff* $f(\mathsf{tcap}(t_1), \ldots, \mathsf{tcap}(t_n))$ *does not unify with any variable renamed left-hand side of a rule from* $\mathcal{R}$
- $\mathsf{tcap}(t)$ *is a fresh variable, otherwise*

To illustrate the difference between cap and tcap consider the TRS of Ex. 1 and $t = \mathsf{div}(\mathsf{0}, \mathsf{0})$. Then $\mathsf{cap}(t) = x_{\mathsf{fresh}}$ since div is a defined symbol. However, $\mathsf{tcap}(t) = t$ since there is no division rule where the second argument is $\mathsf{0}$.

Apart from tree automata techniques, currently the most powerful estimation is the one based on tcap looking both forward as in EDG and backward as in EDG*. Hence, we aimed to implement and certify this estimation in IsaFoR.

Unfortunately, when doing so, we had a problem with the domain of variables. The problem was that although we first implemented and certified the standard unification algorithm of [15], we could not directly apply it to compute tcap. The reason is that to generate fresh variables as well as to rename variables in rules apart, we need a type of variables with an infinite domain. One solution would have been to constrain the type of variables where there is a function which delivers a fresh variable w.r.t. any given finite set of variables.

However, there is another and more efficient approach to deal with this problem than the standard approach to rename and then do unification. Our solution is to switch to another kind of terms where instead of variables there is just one special constructor "$\Box$" representing an arbitrary fresh variable. In essence, this data structure represents contexts which do not contain variables, but where multiple holes are allowed. Therefore in the following we speak of ground-contexts and use $C, D, \ldots$ to denote them.

Definition 11. *Let $[\![C]\!]$ be the equivalence class of a ground-context C where the holes are filled with arbitrary terms: $[\![C]\!] = \{C[t_1, \ldots, t_n] \mid t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})\}$.*

Obviously, every ground-context C can be turned into a term t which only contains distinct fresh variables and vice-versa. Moreover, every unification problem between t and ℓ can be formulated as a *ground-context matching problem* between C and ℓ, which is satisfiable iff there is some μ such that $\ell\mu \in [\![C]\!]$.

Since the result of tcap is always a term which only contains distinct fresh variables, we can do the computation of tcap using the data structure of ground-contexts; it only requires an algorithm for ground-context matching. To this end we first generalize ground-context matching problems to multiple pairs (C_i, ℓ_i).

Definition 12. *A* ground-context matching problem *is a set of pairs* $\mathcal{M} = \{(C_1, \ell_1), \ldots, (C_n, \ell_n)\}$. *It is* solvable *iff there is some* μ *such that* $\ell_i\mu \in [\![C_i]\!]$ *for all* $1 \leq i \leq n$. *We sometimes abbreviate* $\{(C, \ell)\}$ *by* (C, ℓ).

To decide ground-context matching we devised a specialized algorithm which is similar to standard unification algorithms, but which has some advantages: it does neither require occur-checks as the unification algorithm, nor is it necessary to preprocess the left-hand sides of rules by renaming (as would be necessary for standard tcap). And instead of applying substitutions on variables, we just need a basic operation on ground-contexts called merge such that $\mathsf{merge}(C, D) = \bot$ implies $[\![C]\!] \cap [\![D]\!] = \emptyset$, and $\mathsf{merge}(C, D) = E$ implies $[\![C]\!] \cap [\![D]\!] = [\![E]\!]$.

Definition 13. *The following rules simplify a ground-context matching problem into solved form (where all terms are distinct variables) or into* $\bot$.

$$
\begin{array}{lrll}
(a) & \mathcal{M} \cup \{(\Box, \ell)\} & \Rightarrow_{\mathsf{match}} \mathcal{M} & \\
(b) & \mathcal{M} \cup \{(f(\boldsymbol{D}_n), f(\boldsymbol{u}_n))\} & \Rightarrow_{\mathsf{match}} \mathcal{M} \cup \{(D_1, u_1), \ldots, (D_n, u_n)\} & \\
(c) & \mathcal{M} \cup \{(f(\boldsymbol{D}_n), g(\boldsymbol{u}_k))\} & \Rightarrow_{\mathsf{match}} \bot & \text{if } f \neq g \text{ or } n \neq k \\
(d) & \mathcal{M} \cup \{(C, x), (D, x)\} & \Rightarrow_{\mathsf{match}} \mathcal{M} \cup \{(E, x)\} & \text{if } \mathsf{merge}(C, D) = E \\
(e) & \mathcal{M} \cup \{(C, x), (D, x)\} & \Rightarrow_{\mathsf{match}} \bot & \text{if } \mathsf{merge}(C, D) = \bot
\end{array}
$$

Rules (a–c) obviously preserve solvability of $\mathcal{M}$ (where $\bot$ represents an unsolvable matching problem). For Rules (d,e) we argue as follows:

$\{(C, x), (D, x)\} \cup \ldots$ is solvable iff
- there is some μ such that $x\mu \in [\![C]\!]$ and $x\mu \in [\![D]\!]$ and $\ldots$ iff
- there is some μ such that $x\mu \in [\![C]\!] \cap [\![D]\!]$ and $\ldots$ iff
- there is some μ such that $x\mu \in [\![\mathsf{merge}(C, D)]\!]$ and $\ldots$ iff
- $\{(\mathsf{merge}(C, D), x)\} \cup \ldots$ is solvable

Since every ground-context matching problem in solved form is solvable, we have devised a decision procedure. It can be implemented in two stages where the first stage just normalizes by the Rules (a–c), and the second stage just applies the Rules (d,e). It remains to implement merge.

$$
\begin{array}{rll}
\mathsf{merge}(\Box, C) & \Rightarrow_{\mathsf{merge}} C & \\
\mathsf{merge}(C, \Box) & \Rightarrow_{\mathsf{merge}} C & \\
\mathsf{merge}(f(\boldsymbol{C}_n), g(\boldsymbol{D}_k)) & \Rightarrow_{\mathsf{merge}} \bot & \text{if } f \neq g \text{ or } n \neq k \\
\mathsf{merge}(f(\boldsymbol{C}_n), f(\boldsymbol{D}_n)) & \Rightarrow_{\mathsf{merge}} f(\mathsf{merge}(C_1, D_1), \ldots, \mathsf{merge}(C_n, D_n)) & \\
f(\ldots, \bot, \ldots) & \Rightarrow_{\mathsf{merge}} \bot &
\end{array}
$$

Note that our implementations of the matching algorithm and the merge function in IsaFoR are slightly different due to different data structures. For example matching problems are represented as lists of pairs, so it may occur that we have duplicates in $\mathcal{M}$. The details of our implementation can be seen in IsaFoR (theory Edg) or in the source of CeTA.

Soundness and completeness of our algorithms are proven in IsaFoR.

Theorem 14.
- *If* $\mathsf{merge}(C, D) \Rightarrow^*_{\mathsf{merge}} \bot$ *then* $[\![C]\!] \cap [\![D]\!] = \emptyset$.
- *If* $\mathsf{merge}(C, D) \Rightarrow^*_{\mathsf{merge}} E$ *then* $[\![C]\!] \cap [\![D]\!] = [\![E]\!]$.
- *If* $(C, \ell) \Rightarrow^*_{\mathsf{match}} \bot$ *then there is no* μ *such that* $\ell\mu \in [\![C]\!]$.
- *If* $(C, \ell) \Rightarrow^*_{\mathsf{match}} \mathcal{M}$ *where* $\mathcal{M}$ *is in solved form, then there exists some* μ *such that* $\ell\mu \in [\![C]\!]$.

Using $\Rightarrow_{\mathsf{match}}$, we can now easily reformulate tcap in terms of ground-context matching which results in the efficient implementation etcap.

Definition 15.
- $\mathsf{etcap}(f(\boldsymbol{t}_n)) = f(\mathsf{etcap}(t_1), \ldots, \mathsf{etcap}(t_n))$ *iff* $(f(\mathsf{etcap}(t_1), \ldots, \mathsf{etcap}(t_n)), \ell) \Rightarrow^*_{\mathsf{match}} \bot$ *for all rules* $\ell \to r \in \mathcal{R}$.
- $\Box$, *otherwise*

One can also reformulate the desired check to estimate the dependency graph whether $\mathsf{tcap}(t)$ does not unify with u in terms of etcap. It is the same requirement as demanding $(\mathsf{etcap}(t), u) \Rightarrow^*_{\mathsf{match}} \bot$. Again, the soundness of this estimation has been proven in IsaFoR where the second part of the theorem is a direct consequence of the first part by using the soundness of the matching algorithm.

Theorem 16. *(a) Whenever* $t\sigma \to^*_{\mathcal{R}} s$ *then* $s \in [\![\mathsf{etcap}(t)]\!]$.
(b) Whenever $t\sigma \to^*_{\mathcal{R}} u\tau$ *then* $(\mathsf{etcap}(t), u) \not\Rightarrow^*_{\mathsf{match}} \bot$ *and* $(\mathsf{etcap}(u), t) \not\Rightarrow^*_{\mathsf{match}} \bot$ *where* $\mathsf{etcap}(u)$ *is computed w.r.t. the reversed TRS* $\mathcal{R}^{-1}$.

4.3 Certifying Dependency Graph Decomposition

Eventually we can connect the results of the previous two subsections to obtain one function to check a valid application of the dependency graph processor.

$$\texttt{checkDepGraphProc}(\mathcal{P}, L, \mathcal{R}) = \texttt{checkDecomposition}(\texttt{checkEdg}(\mathcal{R}), \mathcal{P}, L)$$

where checkEdg just applies the criterion of Thm. 16 (b).

In IsaFoR the soundness result of our check is proven.

Theorem 17. *If* $\texttt{checkDepGraphProc}(\mathcal{P}, L, \mathcal{R})$ *is accepted and if for all* $\mathcal{P}' \in L$ *where* $\mathcal{P}'$ *is marked, the DP problem* $(\mathcal{P}', \mathcal{R})$ *is finite, then* $(\mathcal{P}, \mathcal{R})$ *is finite.*

To summarize, we have implemented and certified the currently best dependency graph estimation which does not use tree automata techniques. Our check-function accepts any decomposition which is based on a weaker estimation, but requires that the components are given in topological order. Since our algorithm computes edges only on demand, the number of tests for an edge is reduced considerably. For example, the five largest graphs in our experiments contain 73,100 potential edges, but our algorithm only has to consider 31,266. This reduced the number of matchings from 13 millions down to 4 millions.

Furthermore, our problem of not being able to generate fresh variables or to rename variables in rules apart led to a more efficient algorithm for tcap based on matching instead of unification: simply replacing tcap by etcap in TTT2 reduced the time for estimating the dependency graph by a factor of two.

5 Certifying the Reduction Pair Processor

One important technique to prove finiteness of a DP problem $(\mathcal{P}, \mathcal{R})$ is the so-called *reduction pair processor*. The general idea is to use a well-founded order where all rules of $\mathcal{P} \cup \mathcal{R}$ are weakly decreasing. Then we can delete all strictly decreasing rules from $\mathcal{P}$ and continue with the remaining dependency pairs.

We first state a simplified version of the reduction pair processor as it is introduced in [8], where we ignore the usable rules refinement.

Theorem 18. *If all the following properties are satisfied, then finiteness of* $(\mathcal{P} \setminus \succ, \mathcal{R})$ *implies finiteness of* $(\mathcal{P}, \mathcal{R})$.

(a) $\succ$ *is a well-founded and stable order*
(b) $\succsim$ *is a stable and monotone quasi-order*
(c) $\succsim \circ \succ \subseteq \succ$ *and* $\succ \circ \succsim \subseteq \succ$
(d) $\mathcal{P} \subseteq \succ \cup \succsim$ *and* $\mathcal{R} \subseteq \succsim$

Of course, to instantiate the reduction pair processor with a new kind of reduction pair, e.g., LPO, polynomial orders,..., we first have to prove the first three properties for that kind of reduction pairs. Since we plan to integrate many reduction pairs, but only want to write the reduction pair processor once, we tried to minimize these basic requirements such that the reduction pair processor still remains sound in total. In the end, we replaced the first three properties by:

(a) $\succ$ is a well-founded and stable *relation*
(b) $\succsim$ is a stable and monotone *relation*
(c) $\succsim \circ \succ \subseteq \succ$

In this way, for every new class of reduction pairs, we do not have to prove transitivity of $\succ$ or $\succsim$ anymore, as it would be required for Thm. 18. Currently, we just support reduction pairs based on polynomial interpretations with negative constants [13], but we plan to integrate other reduction pairs in the future.

For checking an application of a reduction pair processor we implemented a generic function `checkRedPairProc` in Isabelle, which works as follows. It takes as input two functions `checkS` and `checkNS` which have to approximate a reduction pair, i.e., whenever `checkS`(s, t) is accepted, then $s \succ t$ must hold in the corresponding reduction pair and similarly, `checkNS` has to guarantee $s \succsim t$.

Then `checkRedPairProc`(`checkS`, `checkNS`, $\mathcal{P}, \mathcal{P}', \mathcal{R}$) works as follows:

- iterate once over $\mathcal{P}$ to divide $\mathcal{P}$ into $\mathcal{P}_\succ$ and $\mathcal{P}_{\not\succ}$ where the former set contains all pairs of $\mathcal{P}$ where `checkS` is accepted
- ensure for all $s \to t \in \mathcal{R} \cup \mathcal{P}_{\not\succ}$ that `checkNS`(s, t) is accepted, otherwise reject
- accept if $\mathcal{P}_{\not\succ} \subseteq \mathcal{P}'$, otherwise reject

The corresponding theorem in IsaFoR states that a successful application of `checkRedPairProc`$(\ldots, \mathcal{P}, \mathcal{P}', \mathcal{R})$ proves that $(\mathcal{P}, \mathcal{R})$ is finite whenever $(\mathcal{P}', \mathcal{R})$ is finite. Obviously, the first two conditions of `checkRedPairProc` ensure condition (d) of Thm. 18. Note, that it is not required that all strictly decreasing pairs are removed, i.e., our checks may be stronger than the ones that have been used in the termination provers.

6 Certifying the Whole Proof Tree

From Sect. 3–5 we have basic checks for the three techniques of applying dependency pairs (`checkDPs`), the dependency graph processor (`checkDepGraphProc`), and the reduction pair processor (`checkRedPairProc`). For representing proof trees within the DP framework we used the following data structures in IsaFoR.

```
datatype 'f RedPair = NegPolo "('f × (cint × nat list))list"

datatype ('f,'v)DPProof = ...[6]
   | PisEmpty
   | RedPairProc "'f RedPair" "('f,'v)trsL" "('f,'v)DPProof"
   | DepGraphProc "(('f,'v)DPProof option × ('f,'v)trsL)list"

datatype ('f,'v)TRSProof = ...[6]
   | DPTrans "('f shp,'v)trsL" "('f shp,'v)DPProof"
```

The first line fixes the format for reduction pairs, i.e., currently of (linear) polynomial interpretations where for every symbol there is one corresponding entry. E.g., the list $[(f, (-2, [0, 3]))]$ represents the interpretation where $\mathcal{P}ol(f)(x, y) = \mathsf{max}(-2 + 3y, 0)$ and $\mathcal{P}ol(g)(x_1, \ldots, x_n) = 1 + \Sigma_{1 \leq i \leq n} x_i$ for all $f \neq g$.

The datatype `DPProof` represents proof trees for DP problems. Then the check for valid `DPProof`s gets as input a DP problem $(\mathcal{P}, \mathcal{R})$ and a proof tree and tries to certify that $(\mathcal{P}, \mathcal{R})$ is finite. The most basic technique is the one called `PisEmpty`, which demands that the set $\mathcal{P}$ is empty. Then $(\mathcal{P}, \mathcal{R})$ is trivially finite.

For an application of the reduction pair processor, three inputs are required. First, the reduction pair `redp`, i.e., some polynomial interpretation. Second, the dependency pairs $\mathcal{P}'$ that remain after the application of the reduction pair processor. Here, the datatype `trsL` is an abbreviation for lists of rules. And third, a proof that the remaining DP problem $(\mathcal{P}', \mathcal{R})$ is finite. Then the checker just has to call `createRedPairProc`$(\texttt{redp}, \mathcal{P}, \mathcal{P}', \mathcal{R})$ and additionally calls itself recursively on $(\mathcal{P}', \mathcal{R})$. Here, `createRedPairProc` invokes `checkRedPairProc` where `checkS` and `checkNS` are generated from `redp`.

The most complex structure is the one for decomposition of the (estimated) dependency graph. Here, the topological list for the decomposition has to be provided. Moreover, for each subproblem $\mathcal{P}'$, there is an optional proof tree. Subproblems where a proof is given are interpreted as "real SCCs" whereas the ones without proof remain unmarked for the function `checkDepGraphProc`.

The overall function for checking proof trees for DP problems looks as follows.

```
checkDPProof(P,R,PisEmpty) = (P = [])
checkDPProof(P,R,(RedPairProc redp P' prf)) =
  createRedPairProc(redp,P,P',R) ∧ checkDP(P',R,prf)
checkDPProof(P,R,DepGraphProc P's) =
  checkDepGraphProc(P,map (λ(prf0,P').(isSome prf0,P')) P's, R)
  ∧ ⋀_(Some prf,P')∈P's checkDPProof(P',R,prf)
```

[6] CeTA supports even more techniques, cf. CeTA's website for a complete list.

Theorem 19. *If* `checkDPProof(`$\mathcal{P}$`,`$\mathcal{R}$`,prf)` *is accepted then* $(\mathcal{P}, \mathcal{R})$ *is finite.*

Using `checkDPProof` it is now easy to write the final method `checkTRSProof` for proving termination of a TRS, where `computeID` is an implementation of `ID`.

```
checkTRSProof(R,DPTrans P prf) =
  checkDPs(R,P) ∧ checkDPProof(P,computeID(R),prf)
```

For the external usage of CeTA we developed a well documented XML-format, cf. CeTA's website. Moreover, we implemented two XML parsers in Isabelle, one that transforms a given TRS into the internal format, and another that does the same for a given proof. The function `certifyProof`, finally, puts everything together. As input it takes two strings (a TRS and its proof). Then it applies the mentioned parsers and afterwards calls `checkTRSProof` on the result.

Theorem 20. *If* `certifyProof(`$\mathcal{R}$`,prf)` *is accepted then* `SN(`$\to_{\mathcal{R}}$`)`.

To ensure that the parser produces the right TRS, after the parsing process it is checked that when converting the internal data-structures of the TRS back to XML, we get the same string as the input string for the TRS (modulo white-space). This is a major benefit in comparison to the two other approaches where it can and already has happened that the uncertified components Rainbow/CiME produced a wrong proof goal from the input TRS, i.e., they created a termination proof within Coq for a different TRS than the input TRS.

7 Error Messages

To generate readable error messages, our checks do not have a Boolean return type, but a monadic one (isomorphic to `'e option`). Here, `None` represents an accepted check whereas `Some e` represents a rejected check with error message `e`. The theory `ErrorMonad` contains several basic operations like `>>` for conjunction of checks, `<-` for changing the error message, and `isOK` for testing acceptance.

Using the error monad enables an easy integration of readable error messages. For example, the real implementation of `checkTRSProof` looks as follows:

```
fun checkTRSProof where "checkTRSProof R (DPTrans P prf) = (
     checkDPs R P
       <- (λs. ''error ...'' @ showTRS R @ ''...'' @ showTRS P @ s)
  >> checkDPProof P (computeID R) prf
       <- (λs. ''error below switch to dependency pairs'' @ s))"
```

However, since we do not want to adapt the proofs every time the error messages are changed, we setup the Isabelle simplifier such that it hides the details of the error monad, but directly removes all the error handling and turns monadic checks via `isOK(...)` into Boolean ones using the following lemmas.

```
lemma "isOK(m >> n) = isOK(m) ∧ isOK(n)"
lemma "isOK(m <- s) = isOK(m)"
```

Then for example `isOK(checkTRSProof R (DPTrans P prf))` directly simplifies to `isOK(checkDPs R P)` ∧ `isOK(checkDPProof P (computeID R) prf)`.

8 Experiments and Conclusion

Isabelle's code-generator is invoked to create CeTA from IsaFoR. To compile CeTA one auxiliary hand-written Haskell file `CeTA.hs` is needed, which just reads two files (one for the TRS, one for the proof) and then invokes `certifyProof`.

We tested CeTA (version 1.03) using T$_{\text{T}}$T$_2$ as termination prover (TC and TC$^+$). Here, T$_{\text{T}}$T$_2$ uses only the techniques of this paper in the combination TC, whereas in TC$^+$ all supported techniques are tried, including usable rules and nontermination. We compare to C*i*ME/Coccinelle using AProVE [10] or C*i*ME [5] as provers (ACC,CCC), and to Rainbow/CoLoR using AProVE or Matchbox [18] (ARC,MRC) where we take the results of the latest certified termination competition in Nov 2008[7] involving 1391 TRSs from the termination problem database.

We performed our experiments using a PC with a 2.0 GHz processor running Linux where both T$_{\text{T}}$T$_2$ and CeTA where aborted after 60 seconds. The following table summarizes our experiments and the termination competition results.

	TC	TC$^+$	ACC	CCC	ARC	MRC
proved / disproved	401 / 0	572 / 214	532 / 0	531 / 0	580 / 0	458 / 0
certified	391 / 0	572 / 214	437 / 0	485 / 0	558 / 0	456 / 0
rejected	10	0	3	0	0	2
cert. timeouts	0	0	92	46	22	0
total cert. time	33s	113s	6212s	6139s	7004s	3602s

The 10 proofs that CeTA rejected are all for nonterminating TRSs which do not satisfy the variable condition. Since TC supports only polynomial orders as reduction pairs, it can handle less TRSs than the other combinations. But, there are 44 TRSs which are only solved by TC (and TC$^+$), the reason being the time-limit of 60 seconds (19 TRSs), the dependency graph estimation (8 TRSs), and the polynomial order allowing negative constants (17 TRSs).

The second line clearly shows that TC$^+$ (with nontermination and usable rules support) currently is the most powerful combination with 786 certified proofs. Moreover, TC$^+$ can handle 214 nonterminating and 102 terminating TRSs where none of ACC, CCC, ARC, and MRC were successful. The efficiency of CeTA is also clearly visible: the average certification time in TC and TC$^+$ for a single proof is by a factor of 50 faster than in the other combinations.[8]

For more details on the experiments we refer to CeTA's website.

To conclude, we presented a modular and competitive termination certifier, CeTA, which is directly created from our Isabelle library on term rewriting, IsaFoR.

[7] `http://termcomp.uibk.ac.at/`

[8] Note that in the experiments above, for each TRS, each combination might have certified a different proof. In an experiment where the certifiers where run on the same proofs for each TRS (using only techniques that are supported by all certifiers, i.e., EDG and linear polynomials without negative constants), CeTA was even 190 times faster than the other approaches and could certify all 358 proofs, whereas each of the other two approaches failed on more than 30 proofs due to timeouts.

Its main features are that CeTA is available as a stand-alone binary, the efficiency, the dependency graph estimation, nontermination and usable rules support, the error handling, and the robustness.

As each sub-check for a termination technique can be called separately, and as our check to certify a whole termination proof just invokes these sub-checks, it seems possible to integrate other techniques (even if they are proved in a different theorem prover) as long as they are available as executable code. However, we will need a common proof format and a compatible definition.

As future work we plan to certify several other termination techniques where we already made progress in the formalization of semantic labeling and the subterm-criterion. We would further like to contribute to a common proof format.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. F. Blanqui, W. Delobel, S. Coupet-Grimal, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In *Proc. WST '06*, 69–73, 2006.
4. E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In *Proc. FroCoS '07*, LNAI 4720, 148–162, 2007.
5. E. Contejean, C. Marché, B. Monate, and X. Urbain. CiME. http://cime.lri.fr.
6. P. Courtieu, J. Forest, and X. Urbain. Certifying a termination criterion based on graphs, without graphs. In *Proc. TPHOLs '08*, LNCS 5170, 183–198, 2008.
7. N. Dershowitz. Termination dependencies. In *Proc. WST '03*, 27–30, 2003.
8. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, LNAI 3452, 301–331, 2005.
9. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*, LNAI 3717, 216–231, 2005.
10. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. IJCAR '06*, LNAI 4130, 281–286, 2006.
11. Florian Haftmann. *Code generation from Isabelle/HOL theories*, April 2009. http://isabelle.in.tum.de/doc/codegen.pdf.
12. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.
13. N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.
14. M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. RTA'09*, volume 5595 of *LNCS*, 295–304, 2009.
15. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
16. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.
17. S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.
18. J. Waldmann. Matchbox: A tool for match-bounded string rewriting. In *Proc. RTA '04*, LNCS 3091, 85–94, 2004.