University of Innsbruck Institute of Computer Science

**Cumulative Habilitation Thesis**

# A Formalization of Termination Techniques in Isabelle/HOL

René Thiemann

January 21, 2013

für Karin, Hannah und Jonas

# Contents

# Acknowledgements

Clearly, without an appropriate environment it would not have been possible to develop this thesis. Therefore, I want to express my gratitude to those people that made this thesis possible.

First of all, I want to thank Aart Middeldorp for giving me the freedom to freely let me pursue my research interests in the last years, for his support in providing me with the necessary equipment, and for many useful feedbacks.

Insbesondere möchte ich auch Christian Sternagel danken. Es war eine wunderbare Zeit, jemanden vor Ort zu haben, mit dem man über Isabelle diskutieren konnte und gemeinsam Ideen in Formalisierungen und Paper umsetzen konnte.

Mein Dank gilt auch Georg Moser. Besonders während langen Isabelle Entwicklungen war es eine Freude, durch ihn die Gedanken wieder frei zu bekommen: sei es durch kurze Exkurse in die Welt der Komplexität, durch politische Diskussionen in Steering Committees, oder besonders durch den Austausch kurzer Anekdoten über unsere Kleinen.

Of course, I'm also grateful to the whole Computational logic group for a pleasant and friendly research environment.

Natürlich bin ich auch den Entwicklern der Terminierungs Tools AProVE und $\mathsf{T_TT_2}$ dankbar, insbesondere Bertram Felgenhauer, Carsten Fuhs, Christian Kuknat, Christian Sternagel, Fabian Emmes, und Harald Zankl: ohne deren Entwicklung der Beweisausgabe wäre keine Zertifizierung möglich gewesen.

I would also like to express my gratitude to the Isabelle community. Since I started to work with Isabelle, I always got useful feedback on my questions. Moreover, several recent developments on Isabelle have been beneficial for this thesis, like the code generator, the Isabelle collection framework, partial functions, regexp, parallelization, and jedit. Here, special thanks go to Alexander Krauss, Andreas Lochbihler, Florian Haftmann, Lukas Bulwahn, Makarius Wenzel, Peter Lammich, and Tobias Nipkow.

Many thanks also goes to the Austrian Science Fund (FWF). They supported all the selected papers (within the projects P18763 and P22767-N13).

Schließlich möchte ich mich natürlich auch bei meiner Familie bedanken, insbesondere bei Karin: in ihr fand ich immer den so wichtigen Ausgleich, der mich von der sehr technischen Arbeit zurück in die reale Welt führte, und ich bekam immer Rückhalt, selbst in Zeiten, wo ich nur wenig Zeit für die Familie hatte.

# Part I.

# Preface

# 1. Introduction

Termination is the important property of a program that all computation paths produce a result in finite time. Although undecidable in general, much work has been spent on automated termination analysis. As a result, today there are a variety of powerful tools for automatic termination analysis for various languages: e.g., there are AProVE [39], COSTA [1], Julia [93], Matchbox [106], Polytool [82], Terminator [24], and $T_TT_2$ [67]. Several of these tools are competing in the annual international termination competition[1] where the aim is to automatically investigate termination and complexity of programs in different languages, i.e., currently term rewriting, Prolog, Haskell, or Java.

Due to the complexity of the tools itself, it is obvious that there might be bugs in the implementations leading to wrong answers. And indeed, nearly every year in the competition some bugs were spotted since two tools gave contradicting answers, e.g., by providing a termination and a nontermination proof for the same program. Therefore, it has been identified as a key challenge to independently certify the correctness of the generated proofs. Due to the size of these proofs a manual inspection is infeasible and also error-prone. However, it can be done using interactive theorem provers like Coq [11], Isabelle [84], and PVS [88]. These theorem provers are used to model various aspects of mathematics, programming languages, etc. in a logical system and afterwards formally verify desired properties.

As a result, since 2007 also certifiers have entered the termination competition.

- In the A3PAT project[2] [21, 25], the tool C*i*ME was extended by a feature that transforms termination certificates into Coq scripts. These scripts can be checked with help of the underlying Coq library Coccinelle on rewriting.

- Similarly, in the CoLoR project[3] [13], the tool Rainbow produces Coq-scripts for certified termination proofs. It is based on CoLoR, the *Co*q *L*ibrary *o*n *R*ewriting and termination.

- CeTA (Certified Termination Analysis) with its underlying formalization IsaFoR (Isabelle Formalization of Rewriting) is our own certifier which entered the competition in 2009.

With the help of these three certifiers (which we abbreviate by CC for C*i*ME + Coccinelle, RC for Rainbow + CoLoR, and CI for CeTA + IsaFoR), several implementation bugs have been revealed which previously remained undetected. Furthermore, the formalization of the termination techniques also discovered some flaws in pen-and-paper proofs which in at least one case invalidated the main theorem of a termination technique [97].

In this thesis, we report on our work in this area, i.e., the development of CeTA and IsaFoR. It is structured as follows. In Chapter 2 we illustrate four main problems when

[1] `http://www.termination-portal.org/wiki/Termination_Competition`
[2] `http://a3pat.ensiie.fr`
[3] `http://color.inria.fr`

trying to develop a certifier. Related work is addressed in Chapter 3. Our own contributions in this area are discussed in Chapter 4. Here, we also explain which part of each selected paper has been developed by the co-authors. Future work is addressed in Chapter 5 which ends Part I of this thesis. Afterwards, in Part II the selected papers are provided in chronological order.

# 2. Major Problems

In order to develop a certifier for termination proofs three tasks have to be performed.

(i) The definition of termination has to be formally specified.

(ii) Termination methods have to be formalized in some library of a theorem prover.

(iii) Starting from a given certificate, one has to reconstruct a formal termination proof using the termination methods from the previous step.

Once, these tasks have been established, a possible workflow for getting certified termination proofs is as follows, cf. Figure 2.1.

First, an untrusted termination tool is invoked which has to deliver the answer (terminating, nonterminating, don't know) in combination with a certificate which has to contain enough information to reconstruct a proof to validate the answer. As an example, the certificate might be an XML-document which states which termination techniques have been applied and how they have been parametrized.

Afterwards, one can start the reconstruction to obtain a formal proof script. In this step, one has to link the termination techniques in the certificate to those which have been formalized. Moreover, in the proof script several assertions have to be stated which ensure that the termination techniques are applied correctly.

Finally, the formal proof script must be checked within a proof assistant.

In order to maximize the reliability of this certification workflow, one should use proof assistants which are based on the LCF-approach [43, 45, 89], where all generated proofs have to be accepted by some small trusted kernel. In that way, one can be quite certain that during proof checking only correct proofs are accepted. Moreover, for the major properties and definitions in the specification—which are termination and the rewrite relation in our case—one can try to prove that their formalization corresponds to various alternative definitions that are present in the literature [5,100]. For example, we define the rewrite relation as $\to_{\mathcal{R}} := \{(C[\ell\sigma], C[r\sigma]) \mid \ell \to r \in \mathcal{R}\}$ where $C$ ranges over all contexts and $\sigma$ over all substitutions, and afterwards we prove that our definition is equivalent to another standard definition of the rewrite relation: $\to_{\mathcal{R}} = \{(t, t[r\sigma]_p) \mid t|_p = \ell\sigma, \ell \to r \in \mathcal{R}, p \in \mathcal{P}os(t)\}$ where $\mathcal{P}os(t)$ is the set of positions in $t$.

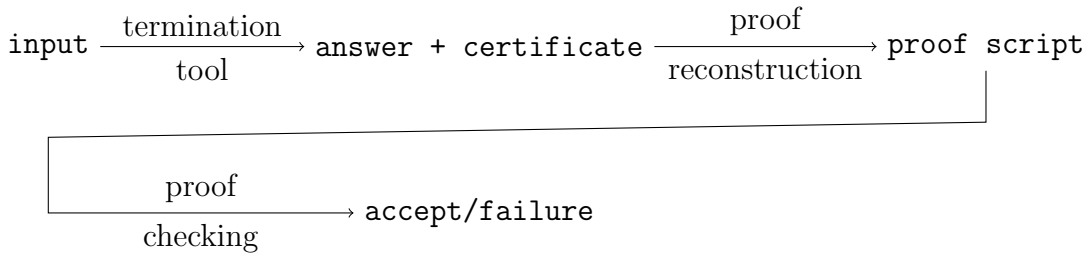



Figure 2.1.: workflow for certified termination proofs

Although the reliability is quite high, we want to mention that there still remain certain risks where we just have to trust the system: the logic of the proof assistant may be inconsistent, there may be bugs in the kernel of the proof assistant as well as in the compiler / interpreter that is used to run the proof assistant, and there may be errors in the operating-system and in the hardware, and the hardware may be damaged. To minimize these risks, several projects have been performed. For example, with Milawa and Jitawa there is a verified theorem prover running in a verified Lisp runtime, i.e., soundness of the theorem prover was proven down to x86 machine code [80]. And in another impressive project, an operating-system kernel was proven to be sound [60].

Despite these risks, we believe that formalized proofs contain far less errors than those which are checked by human—we leave it up to the reader to figure out a list of risks that arise when proofs are solely checked by humans. But even if we trust formalized proofs, we have to establish all three tasks that have been mentioned above.

Whereas the first task is easily solved in our case—since the notions of termination and the rewrite relation of a term rewrite system (TRS) are easily defined—the second task is by far more challenging because of the following two major problems.

**Problem 1: Formalize many termination techniques** Termination tools for term rewriting use several incomparable termination techniques. And for each individual termination technique, we require a fully formalized proof. Here, standard formalization problems may arise: proofs in papers can contain gaps and are often not sufficiently detailed ("w.l.o.g.", "easy to see", "by induction", "we consider only the most interesting case", ...), they may be based on other nontrivial theorems (from Ramsey theory, from graph-theory, ...), they may use nontextual representations like diagrams, and they make use of intuitive arguments which are hard or tedious to describe formally ("we can easily reorder this sequence as follows such that ..."). Moreover, we will encounter the problem, that many termination techniques are using completely different concepts in their soundness proofs: the proofs are based on syntactic criteria, graph theory, automata theory, algebra, combinatorics, etc.

Hence, to get formalized proofs, all gaps have to be filled, all background theories have to be formalized, and one has to find a good formal model for all the concepts that are used in the proofs.

**Problem 2: Combine termination techniques** Even if individual termination techniques have been proven correct, for their combination we need a common semantics, such that all techniques are sound w.r.t. that semantics. For example, for the rewrite relation of a TRS luckily there is only one semantics. However, often termination proofs are performed using the notions of dependency pairs and chains, and here there are at least two different versions of evaluation (chains and minimal chains) [3]. The problem is that some termination techniques are only sound for chains whereas other require minimal chains. Hence, checking proofs which utilize both kinds of termination techniques cannot be done without adaptation: one first has to find some notion of chain which is compatible with all termination techniques.

When considering the third task—proof reconstruction—two additional problems arise.

**Problem 3: Check application of termination techniques** In principle, the certificate contains all major proof steps, i.e., every termination technique that has been applied is

listed in the certificate in combination with its parameters. However, it must be checked that the technique has been applied correctly, which is a problem that may range from easy via complex to undecidable.

For example, some termination methods are based on simple syntactic criteria which can easily be checked, like the application of dependency pairs or the switch from full- to innermost-termination for orthogonal TRSs. However, for the latter technique one can weaken orthogonality to locally confluent overlay TRSs [46]. In that case, checking the preconditions becomes undecidable as local confluence is undecidable.

As another example, consider termination methods based on well-founded orders, like polynomial orders [73], the lexicographic path order (LPO) [58], or the recursive path order (RPO) [27]. For these methods one has to provide an algorithm to check whether two terms are in relation. This problem is in P (for LPO) [71], NP-complete (for RPO), and undecidable for polynomial orders. In the latter case, sufficient decidable criteria like absolute positiveness [56] are used which are again easy to check. Of course, one might enrich the certificates by providing more detailed evidence why two terms are in relation. Then checking can be done more efficiently, but the certificates easily become bulky and are harder to produce.

As a last example, we consider size-change termination which is a PSPACE-complete problem [74]. Here, there is no chance to enlarge the certificate such that checking the proof can be done in polynomial time, unless PSPACE = NP. We shortly illustrate why also in practice an enlarged certificate is not helpful for this technique. One standard algorithm to decide size-change termination works as follows. It computes a (possibly exponentially large) closure of size-change graphs where one has to test that no bad graphs appear in this closure. In principle, one can add the closure to the certificate, but this does not improve the situation, since checking that the provided closure is correct has the same asymptotic complexity as computing the closure.

To summarize, we require methods to guarantee the correct application of termination techniques which is not always easy.

**Problem 4: Obtaining certificates** It is obvious, that for checking a termination proof we require certificates from the tools. These certificates should be sufficiently detailed and in some machine readable format. However, when we started our work, each of the certifiers had its own proof format with varying level of detail. Hence, for a termination tool to support many certifier, the tool had to implement many proof printing procedures: one for each certifier. Putting this into another perspective, we had to convince the tool authors to also support and integrate our proof format.

As a solution to this problem we developed a common proof format for certificates for termination problems: the certification problem format (CPF).[1] The CPF-format was designed as a combination of all previously existing formats by several groups: members from all three certifiers CC, RC, and CI contributed, as well as members from the termination tools AProVE and T$_{\text{T}}$T$_2$. As a result, today termination tools only have to support one output format (CPF) and then all three certifiers can be used to independently certify the answer. Moreover, we also wrote pretty printer which transform CPF proofs into human readable proofs, so that the termination tool authors can also drop their human readable export functionality for those parts of the proof that are covered by CPF.

[1] `http://cl-informatik.uibk.ac.at/software/cpf/`

# 3. Related Work

The alternative certifiers CC and RC clearly have the closest connection to our work. Moreover, there also is a PVS formalization on term rewriting, trs [34, 35]. It was not mentioned before, since as far as we know, it is currently not used for certification of termination proofs.

Although all of the formalizations are on term rewriting there are several differences which we will list in the remainder of this chapter.

**Theorem Prover** CI is based on Isabelle/HOL, CC and RC use Coq, and trs utilizes PVS. As a consequence, the proof of some result in IsaFoR can look quite different from a similar proof of the same result in one of the other formalizations. Moreover, there is no easy way to import or share theorems which have already been integrated in the other formalizations. As a matter of fact, we are only aware that some results are shared among the Coq based formalizations Coccinelle and CoLoR.

**Representation** In CI everything is developed using a deep embedding, whereas in CC and RC also shallow embeddings are used. To shortly illustrate the difference between deep and shallow embedding, consider the following TRS $\mathcal{R}$ for subtraction:

$$\mathsf{minus}(x, \mathsf{0}) \to x$$
$$\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{minus}(x, y)$$

For certification, in both CC and RC first an inductive set or type for the signature is created on the fly within Coq (shallow), which contains exactly the three symbols minus, s, and 0. In CI however, there is no dedicated signature—minus, s, and 0 are just strings. Similarly, for solving arithmetic constraints which may arise from proving termination of $\mathcal{R}$ by a polynomial interpretation, in CC the Coq tactic *omega* is invoked (shallow), whereas for CI an algorithm for solving these constraints has been formalized (deep).

Both designs have their own advantages: using a shallow embedding sometimes allows for more elegant formalizations since the builtin methods from the theorem prover can be used to adequately model or solve a problem; and with a deep embedding, the certification algorithms can also run outside the theorem prover and are less brittle to changes in the theorem prover.

**Support of termination techniques** As we have focussed our work to develop a certifier with a high coverage of termination technique for first order term rewriting, CI contains several techniques in this area which are not available in the other certifier.[1] For example, during the latest full run of termination tools in 2011, where every participating tool was tested on every problem of the termination problem database, 3675 termination and

[1] See `http://cl-informatik.uibk.ac.at/software/ceta/#introduction` for the current list of techniques which are supported by CeTA.

nontermination proofs have been generated for TRSs.[2] In this experiment, there have been dedicated termination tools to support both CI and CC, i.e., some termination tools delivered proofs which are known to be supported by CI and other tools delivered proofs especially for CC. Whereas CC had to refuse 1694 proofs as they contained unsupported techniques, CeTA only had to refuse 525 proofs. And in the meantime, all 3675 proofs can be certified using CeTA version 2.8. These numbers clearly demonstrate a high coverage of CeTA in this area.

However, in other areas the picture is inverted: for example, IsaFoR does not contain any result on higher-order rewriting, but CoLoR does [63].

[2] `http://termcomp.uibk.ac.at/termcomp/competition/certificationResults.seam?cat=10235&comp=260918`

# 4. Contributions

In this chapter, we first explain how the selected papers in the second part of this thesis contribute to solving the four mentioned problems. Here, the papers are listed in order of their publication date. Afterwards, we report on further work in the area of certification, which has been performed in IsaFoR but is not covered in the selected papers.

## 4.1. Certification of Termination Proofs using CeTA (Chapter 6)

### Publication details

René Thiemann and Christian Sternagel. Certification of Termination Proofs using CeTA. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of LNCS, pages 452–468. Springer, 2009.

In this initial paper on IsaFoR and CeTA, we give details on our formalization of three important termination techniques—dependency pairs, dependency graph decomposition, and reduction pairs—a clear contribution to Problem 1.

Note that all of these techniques are also supported by the other certifiers. Therefore, the real novelty in this paper is the approach of dealing with Problem 3. Both RC and CC use a mixture of three possibilities to generate proof scripts for a certificate: invoke executable functions which have been proven correct, perform on-the-fly generation of proofs by untrusted tools, and call built-in solvers of the proof assistant. In contrast, for every termination technique in IsaFoR we always provide an executable function that checks the correct application—even for those techniques that compose other termination techniques.

As a consequence, there is also one check-function `check-trs-termination-proof` which checks full proof trees. It takes a proof and a TRS as input and we have formally proven its correctness: if `check-trs-termination-proof` $\mathcal{R}$ results in "OK" then termination of $\mathcal{R}$ is guaranteed. Hence, our proof scripts are always identical, they just evaluate `check-trs-termination-proof` and demand that the result is "OK".

This allows us to not generate a proof script at all. We can just invoke the code generator of Isabelle [49] to obtain `check-trs-termination-proof` as external program (Haskell, OCaml, SML, or Scala) which can then be compiled and executed. And this is exactly what CeTA is. It consists of a small hand written main function which reads a file, invokes `check-trs-termination-proof` on the input, and accepts the proof if the result is "OK", and otherwise throws an error-message. Hence, for checking proofs one does not have to install and run Isabelle, but one can just compile CeTA and execute it like every other program.

This design has several consequences:

- Since proof checking via CeTA is execution of native code, and does not run within the proof assistant, CeTA is usually faster than the other two certifiers. Moreover, it allows us to also integrate more time intensive termination techniques. For example, the dependency graph estimation EDG$^{***}$ (a combination of the estimations in [41] and [50]), and the termination techniques of semantic labeling [110] (Chapter 9), size-change termination [74], and matchbounds [36] all have complex application criteria where the corresponding check-functions benefit from the speedup that is obtained from running native code.

- If a proof script is rejected, then one can immediately see where the proof is rejected. Depending on the structure of the proof script, it is then more or less easy to figure out which part of the termination proof was rejected. In contrast, if the result of our `check-trs-termination-proof` function were just a Boolean, then it would not be visible which part of the proof was rejected. Therefore, the return type of our check-functions is a disjoint sum: either we return "OK", or a string which is the error message. Since all our check-functions provide detailed human readable error messages, it is easy to figure out which part of the proof is not accepted and why it is not accepted, where it is not required to read any proof script at all. In the paper we further describe how readable error messages can be integrated into the check-functions without becoming an overhead when proving soundness of these functions.

- Since all our formal proofs are done statically (soundness of termination techniques and soundness of check-functions), we are quite robust against any changes in the proof assistant. With every new release of the proof assistant, we just have to replay all our proofs and immediately detect problems which are due to changes in the proof assistant—which can then be easily fixed. However, CeTA will be the same program as before (unless there are severe changes in the code generator), so the changes in the proof assistant will have no impact on checking the certificates. In contrast, if we would have generated proof scripts on the fly, then some problems may occur only during certification which are not visible when replaying the proofs for the theorems in the library.

- As all our check-functions have to be fully executable, they cannot make use of internal tactics, solvers, etc., which might be available in the proof assistant. Hence, it is more tedious to develop these checks, as most parts have to be developed from scratch. Moreover, our design requires that the full formalization is deeply embedded, so we cannot model parts of term rewriting by the builtin constructs of the proof assistant as it is done in both RC and CC.

To summarize, in this paper we formalized well known termination criteria, and presented a new approach to certify termination proofs via code generation. Here, my contributions are the formalization of EDG$^{***}$, polynomial orders, and the idea of using code generation and error messages. My co-author formalized dependency pairs and the underlying theory on abstract reduction systems and term rewrite systems.

## 4.2. Certified Subterm Criterion and Certified Usable Rules (Chapter 7)

### Publication details

Christian Sternagel and René Thiemann. Certified Subterm Criterion and Certified Usable Rules. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of LIPIcs, pages 325–340. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.

This paper is mainly devoted to Problem 1, the formalization of important termination techniques. In the following, we will concentrate on the formalization of usable rules [3, 41, 42] since my co-author developed the part about the subterm criterion [51].

Note that as for the estimation of the dependency graph, there are various definitions of usable rules. To be able to certify many termination proofs we integrated one of the most powerful versions of usable rules, namely the one in [41] which additionally is combined with argument filters as in [42]. Here, we tried to generalize as much as possible.

One of the main results is easily stated where the exact definitions can be seen in Chapter 7: if $\mathcal{R}$ is finite, $(\mathcal{P}, \mathcal{R})$ is a dependency pair problem, $(\succsim, \succ)$ is a reduction pair compatible with an argument filter $\pi$, $\mathcal{U}$ are the usable rules of $(\mathcal{P}, \mathcal{R})$ w.r.t. $\pi$, and $\mathcal{P} \cup \mathcal{U} \cup \mathcal{C}_\varepsilon \subseteq \succsim$, then for a termination proof it is allowed to delete all pairs $s \to t$ from $\mathcal{P}$ which are strictly decreasing, i.e., which satisfy $s \succ t$.

Although this seems exactly like the statement that is given in literature, we want to illustrate that our formalization contains something more general:

- We do not require that $\mathcal{R}$ is a well-formed TRS, i.e., that for every rule $\ell \to r \in \mathcal{R}$ the left-hand side $\ell$ must not be a variable and that all variables of $r$ also occur in $\ell$. As a consequence during certification we do not have to check for well-formedness when applying this techniques. However, it complicates the soundness proof of this termination technique, as it relies upon the fact that $\to_{\mathcal{R}}$ is finitely branching, but non-well-formed TRSs may be infinitely branching.

- We do not make any assumption on the set of variables or function symbols, so that this technique can be applied for every type of variables and function symbols, even for degenerated cases. As an example consider the TRS $\mathcal{C}_\varepsilon$ which in the literature is defined as $\mathcal{C}_\varepsilon = \{\mathsf{c}(x, y) \to x, \mathsf{c}(x, y) \to y\}$ which can be used to make nondeterministic choices. Now let the set of variables be a singleton set, i.e. $x = y$. Then there is no choice anymore and the proof would fail. Therefore, in the formalization we define $\mathcal{C}_\varepsilon := \bigcup_{s,t}\{\mathsf{c}(s, t) \to s, \mathsf{c}(s, t) \to t\}$ where $s$ and $t$ range over all terms. With this definition we can prove the result, and moreover, whenever $\{\mathsf{c}(x, y) \to x, \mathsf{c}(x, y) \to y\} \subseteq \succsim$, then also $\mathcal{C}_\varepsilon \subseteq \succsim$ since $\succsim$ is required to be closed under substitutions. Hence, we do not impose any additional constraint on the reduction pairs that can be used.

Besides these generalizations, the paper illustrates a nice trick to avoid the development of a working list algorithm to compute the usable rules: The usable rules are usually defined via some inductive definition, i.e., they are the least set $\mathcal{U} \subseteq \mathcal{R}$ such that some property $P$ on $\mathcal{U}$ is satisfied. Although a standard working list algorithm for computing the usable rules is easily written, it would require a tedious proof to show termination

of the algorithm, and of course, one would require proofs for soundness and correctness. In contrast, if the usable rules are provided in the certificate, then one just has to check whether they satisfy property $P$: it is not at all essential, that the given set of rules is indeed the least set satisfying $P$.

Note that requiring the usable rules in the certificate is not a severe burden for the termination tools,[1] since they will have to compute the usable rules in any case. Moreover, providing the usable rules in the certificate has the advantage, that one can also show this information when pretty printing the certificate into a human readable format.

To summarize, in this paper we formalized two important termination techniques and generalized them, such that less preconditions have to checked during certification. Moreover, we illustrated how to completely avoid the formalization of a working list algorithm to compute an inductively defined set.

## 4.3. Signature Extensions Preserve Termination – An Alternative Proof via Dependency Pairs (Chapter 8)

### Publication details

Christian Sternagel and René Thiemann. Signature Extensions Preserve Termination – An Alternative Proof via Dependency Pairs. In *Proceedings of the 19th Annual Conference of the EACSL on Computer Science Logic*, volume 6247 of LNCS, pages 514–528. Springer, 2010.

Assume that some property $P$ can be proven for all terms over the signature $\mathcal{F}$ where $\mathcal{F}$ are all those symbols that appear in the TRS $\mathcal{R}$. Now consider an extended signature $\mathcal{F}' \supseteq \mathcal{F}$. The question of signature extensions is the question for which properties $P$ we can conclude that $P$ also holds for all terms over $\mathcal{F}'$.

It is a well-known result that signature extensions are sound for termination and this fact is also used in termination tools which is illustrated in the following example.

**Example 4.1.** Let $\mathcal{R}$ consist of the following rules, and let $\mathcal{F} = \{\mathsf{a}, \mathsf{b}, \mathsf{f}\}$.

$$\mathsf{a}(\mathsf{b}(\mathsf{b}(x))) \to \mathsf{a}(\mathsf{b}(\mathsf{a}(\mathsf{a}(\mathsf{a}(x))))) \quad (1)$$

$$\mathsf{f}(x, y) \to x \quad (2)$$

$$\mathsf{f}(x, y) \to y \quad (3)$$

Using an polynomial order it is possible to remove rules (2) and (3), and it remains to prove termination of $\mathcal{R}' = \{(1)\}$ for all terms over the signature $\mathcal{F}$. Using the result of signature extensions it suffices to prove termination of $\mathcal{R}'$ w.r.t. the signature $\mathcal{F}' = \{\mathsf{a}, \mathsf{b}\}$. Since in $\mathcal{F}'$ every symbol is unary, one can interpret the terms as strings and apply string reversal which results in

$$\mathsf{b}(\mathsf{b}(\mathsf{a}(x))) \to \mathsf{a}(\mathsf{a}(\mathsf{a}(\mathsf{b}(\mathsf{a}(x))))).$$

Afterwards there are no dependency pairs and termination is trivially proven.

Without the result on signature extensions, one would not be able to apply string reversal, and a more complex proof would be required to prove termination of $\mathcal{R}'$ over $\mathcal{F}$.

[1] In the meantime, IsaFoR also contains an algorithm to compute the usable rules. It was developed for other termination techniques where the computation of usable rules occurs as a side-problem, and where adding the usable rules to the certificates would make the certificates too bulky.

In the paper there are essentially three contributions:

(i) There is a new formalized proof that signature extensions are sound for termination, which is simpler than existing proofs. It just requires that signature extensions are sound when considering chains of dependency pairs, which can be proven in a straight-forward way. And using this result in combination with soundness and completeness of dependency pairs already suffices to establish the signature extension result.

(ii) A counter-example is provided which proves that signature extensions are unsound when considering minimal chains of dependency pairs.

(iii) It is shown that under the additional assumption of left-linearity signature extensions are sound for minimal chains.

Especially Contributions (ii) and (iii) are important: In at least two papers [53, 95] signature extensions are used in combination with minimal chains—taking this unsound result for granted. Luckily, in case of uncurrying [53], the major result is still valid (cf. Chapter 11). In contrast, for root-labeling [95] we were able to also construct a counter-example showing that without further restrictions, root-labeling is unsound when considering minimal chains. Hence, Contribution (iii) was important since it allows to use root-labeling at least for left-linear TRSs.

As a consequence, termination tools like AProVE and $\mathsf{T_TT_2}$ now require left-linearity if they want to apply root-labeling when using dependency pairs.

Clearly this paper contributes to Problem 1, as now termination techniques such as string reversal can be certified, however, it shows the overall relevance of formalization, as we have refuted the soundness of an existing termination technique in theory and practice.

Whereas most of the formalization for this paper was mainly done by my co-author, I developed the counter-examples for both signature extensions and root-labeling. Both of us were involved in the process of finding the alternative proof for signature extensions via dependency pairs and in the process of developing a fix for minimal chains—in the form of the additional requirement of left-linearity. Note that in the meantime, I extended the work on signature extensions by integrating the following facts in IsaFoR.

- Signature extensions are sound for innermost rewriting, no matter whether one considers termination of TRSs, chains, or minimal chains.

- Signature extensions are sound for relative termination of $\mathcal{R}$ modulo $\mathcal{S}$[2] if $\mathcal{S}$ is well-formed.

- In general, signature extensions are unsound for relative termination. For the TRSs $\mathcal{R} = \{\mathsf{a} \to \mathsf{b}\}$ and $\mathcal{S} = \{\mathsf{a} \to x\}$, we have relative termination of $\mathcal{R}$ modulo $\mathcal{S}$ if we consider terms over the signature $\mathcal{F} = \{\mathsf{a}, \mathsf{b}\}$, but for the extended signature $\mathcal{F} \cup \{\mathsf{f}\}$ where $\mathsf{f}$ is a binary symbol, relative termination does no longer hold: $\mathsf{a} \to_{\mathcal{S}} \mathsf{f}(\mathsf{a},\mathsf{a}) \to_{\mathcal{R}} \mathsf{f}(\mathsf{a},\mathsf{b}) = C[\mathsf{a}] \to_{\mathcal{S}} C[\mathsf{f}(\mathsf{a},\mathsf{a})] \to_{\mathcal{R}} \ldots$.

[2]Relative termination of $\mathcal{R}$ modulo $\mathcal{S}$ is defined as strong normalization of $\to_{\mathcal{R}} \circ \to_{\mathcal{S}}^*$.

## 4.4. Modular and Certified Semantic Labeling and Unlabeling (Chapter 9)

### Publication details

Christian Sternagel and René Thiemann. Modular and Certified Semantic Labeling and Unlabeling. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications*, volume 10 of LIPIcs, pages 329–344. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011.

This paper addresses Problem 1 by formalizing the important termination technique of semantic labeling [110]. Even more important is the development of a solution to Problem 2, and it also addresses Problem 3.

When applying semantic labeling, one has to find an interpretation such that the rules are a model (or quasi-model) of this interpretation, cf. Chapter 9 for more details. Afterwards, each rule is replaced by several new labeled variants where the function symbols are labeled by their semantics, and where one rule is created for every possible assignment.

The advantage is that in this way arbitrary semantic information can be annotated as labels in the symbols which can then be exploited afterwards, e.g., the precedence for some RPO can be based on the labels. The disadvantage is that the labeled TRS is much larger than the original TRS. To this end the following approach is often used: after labeling, one tries to remove all labeled variants of at least one rule using some termination techniques, and afterwards removes all labels again, in order to avoid the size-increase from the labeling. Whereas this approach is sound for rewriting and for chains, i.e., when working directly on TRSs or on chains of dependency pairs, unfortunately, it is unsound when considering minimal chains.

Hence, with the standard semantics of dependency pairs, one can either use semantic labeling and unlabeling (if one considers chains), or one can use the subterm criterion and usable rules (if one considers minimal chains), but not both at the same time. Since both techniques contribute significantly to the overall power of a termination tool, it would be nice to find a new notion of chain—including a new semantics—where all these techniques are sound. And this is exactly what we developed in this paper.

We proved that semantic labeling and unlabeling are sound w.r.t. to the new semantics, and we also showed that a whole class of termination techniques can be lifted from the old to the new semantics, which includes the subterm criterion and usable rules. As a matter of fact, all but one technique in IsaFoR could be lifted to the new semantics without requiring a new proof, only for string reversal the soundness proof had to be manually adapted.

Concerning certification of semantic labeling, we shortly want to mention that certification is not just checking syntactic criteria or term-order constraints, since another kind of problem arises. When using semantic labeling with quasi-models, one has to add the decreasing rules $\mathcal{D}ec$ in addition to the labeled rules. In fact, some termination tools do not add $\mathcal{D}ec$ but just use a subset $\mathcal{D}ec' \subseteq \mathcal{D}ec$ such that $\to_{\mathcal{D}ec} \subseteq \to^+_{\mathcal{D}ec'}$. To accept these termination proofs, first we have formally proven that it suffices to add such a set $\mathcal{D}ec'$ instead of $\mathcal{D}ec$ itself. As a consequence, for certification one now requires an algorithm to check that $\to_{\mathcal{D}ec} \subseteq \to^+_{\mathcal{D}ec'}$ is satisfied, which we developed in a second step.

For this paper, my co-author developed the formalizations of root-labeling (an instance of semantic labeling which often requires preprocessing using flat-context closures),

whereas the new semantics, the theory on semantic labeling, and the checks for semantic labeling with arbitrary finite models was done by myself.

## 4.5. Termination of Isabelle Functions via Termination of Rewriting (Chapter 10)

### Publication details

Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle Functions via Termination of Rewriting. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving*, volume 6898 of LNCS, pages 152–167. Springer, 2011.

This paper is not related to one of the four mentioned problems. Instead, it describes an application of our certifier. Note that in Isabelle/HOL there are three major alternatives to define recursive functions.

- `primrec`: functions in primitive recursive form
- `fun` or `function`: functions with no syntactic restriction on the recursive calls
- `partial-function`: functions with no syntactic restriction on the recursive calls

Of course, whenever a function can be conveniently written in primitive recursive form, one can use `primrec`. If this is not the case, then one has to use one of the alternatives. For `partial-function` there also is a syntactic restriction, namely that the function must be tail-recursive, or the result of the function must be an option type. Moreover, `partial-function` currently does not yield an induction scheme for tail-recursive functions, and it only provides an inconvenient induction scheme if the result is an option type. Therefore, `fun` and `function` are the most frequently used commands to define functions, where a nice induction scheme is provided, but where termination of the function has to be proven. If one uses `function`, this proof has to be done manually, whereas `fun` first invokes `function`, and afterwards tries to find a termination proof using some heuristics.

To give some statistics, in the archive of formal proofs, `primrec` is used 705 times, `partial-function` is only invoked 17 times, `fun` is applied 1906 times, and `function` is utilized 98 times. In over words, for at least 98 functions, the heuristic of `fun` is not good enough to prove termination of the function. Notice that even in those cases where `fun` was successful, sometimes manual interaction was required—in the form of auxiliary lemmas that can be used by `fun`.

The aim of this paper was simple, we wanted to use existing termination tools to reduce the number of manual termination proofs. To this end, we encode a function $f$ as TRS $\mathcal{R}_f$, use an external termination tool for getting a termination proof for $\mathcal{R}_f$, execute CeTA (within Isabelle/HOL) to formally prove termination of $\mathcal{R}_f$, and prove that from termination of $\mathcal{R}_f$ we can conclude termination of $f$. To this end, we developed a tactic which essentially shows that the computation of $f$ can be simulated by $\mathcal{R}_f$, where every Isabelle/HOL term is encoded into a term over the signature of $\mathcal{R}_f$.

Unfortunately, so far the experimental results are a bit disappointing, since for most of the 98 functions, we have not been able to successfully apply our approach and there are two major reasons for this.

First, our encoding as described in the paper is restricted to first-order functions, whereas many of the 98 examples are higher-order examples. To this end, we later started to extend our work by also providing an encoding for higher-order functions. Using these extensions, we are able to treat functions like "map" and "fold".

Second, for proving termination of $f$ one has to regard side-conditions. Sometimes these are easy to encode, for example if they arise from an if-then-else as in

```
f n = if n > 0 then n * f (n - 1) else 1
```

where one has to find a well-founded order `>` such that `n > n - 1` whenever `n > 0`. Here, the test `n > 0` over naturals is converted into the test whether the term `n > 0` rewrites to true via some TRS which encodes comparison of natural numbers.

However, when using higher-order functions these conditions may become more complex. As an example consider the following function which computes the size of varyadic trees.

```
size (Node ts) = 1 + listsum (map size ts)
```

Here, to prove termination one has to find a well-founded order `>` such that `Node ts > t` whenever `t` $\in$ `set ts` (where `set` converts a list into a set). Using the same solution as before, one would demand that the term `t` $\in$ `set ts` rewrites to true, but the problem is that `t` is a free variable, which would immediately result in nontermination of the TRS. Instead, one can use a function which is dual to membership, namely a selection function, to encode a conditional rewrite rule for `size`: if `select ts` $\to^*$ `t`, then `size` (`Node ts`) $\to$ `size t`.

It remains as interesting future work to also develop suitable encodings for these conditions, and we believe that once this has been established, indeed our approach can be useful to reduce the number of manual termination proofs.

Whereas most of the tactics that are described in the paper have been implemented solely by the first author, I contributed to both the development and the presentation of the tactics on a higher level. Moreover, the extensions to higher-order are completely done by myself, and they have already been presented in an invited tool demonstration at the International Workshop on Higher-Order Rewriting in 2012.

## 4.6. Generalized and Formalized Uncurrying (Chapter 11)

### Publication details

Christian Sternagel and René Thiemann. Generalized and Formalized Uncurrying. In *Proceedings of the 8th International Symposium on the Frontiers of Combining Systems*, volume 6989 of LNCS, pages 243–258. Springer, 2011.

This paper is clearly devoted to partially solve Problem 1, since one important termination technique has been formalized: Uncurrying. Uncurrying as described in [53] works over applicative signatures, where there is one binary application symbol $\circ$, and where all other symbols are constants. It is used to turn applicative TRSs—TRSs where the signature is applicative—into functional form. If rules contain head variables, i.e., subterms of the form $x \circ t_1 \circ \ldots \circ t_n$, uncurrying is performed as far as possible. For example, the applicative rewrite rule

$$\mathsf{map} \circ f \circ (\mathsf{cons} \circ x \circ y) \to \mathsf{cons} \circ (f \circ x) \circ (\mathsf{map} \circ f \circ y)$$

is uncurried into

$$\mathsf{map}(f, \mathsf{cons}(x, y)) \to \mathsf{cons}(f \circ x, \mathsf{map}(f, y))$$

where the subterm $f \circ x$ could not be uncurried. For the approach to be sound, also the uncurrying rules have to be added, i.e., rules like $\mathsf{cons} \circ x \to \mathsf{cons}(x)$ and $\mathsf{cons}(x) \circ y \to \mathsf{cons}(x, y)$.

Note that uncurrying is extremely important to successfully treat applicative TRSs, since several other termination techniques perform badly on applicative signatures.

During the formalization, we immediately spotted one problematic step in the original soundness proof due to our knowledge on signature extensions (cf. Section 4.3 and Chapter 8): in [53], w.l.o.g. it is assumed that signature extensions are sound when regarding minimal chains.

However, in contrast to root-labeling, where we have been able to refute the whole theorem due to this wrong assumption, for uncurrying we could provide an alternative proof which does not rely upon signature extensions: we just dropped the condition that uncurrying is only applied on TRSs over applicative signatures, and generalized the whole proof for arbitrary signatures.

This generalization led to a strictly stronger theorem. For example, using our generalization, one may uncurry the map-rule above, even if the remaining TRS contains some functional rule like $\mathsf{plus}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{plus}(x, y))$. This is not possible using the original technique.

My own part in this work was the full formalization and adaptation of the proofs, whereas my co-author integrated the generalized uncurrying technique in the T$_{\text{T}}$T$_2$ termination tool, which was essential to obtain empirical results for the adapted termination technique.

## 4.7. On the Formalization of Termination Techniques Based on Multiset Orderings (Chapter 12)

### Publication details

René Thiemann, Guillaume Allais, and Julian Nagele. On the Formalization of Termination Techniques Based on Multiset Orderings. In *Proceedings of the 23rd International Conference on Rewriting Techniques and Applications*, volume 15 of LIPIcs, pages 339–354. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.

In this paper we consider two termination techniques which are based on multiset orders: RPO and an approximation of size-change termination (SCNP reduction pairs), hence the paper contributes to solve Problem 1.

To support these techniques, of course we required a formalization of the multiset extension of an order. Although this has already been done in the Isabelle-distribution, we could not use these results, as we require the multiset extension of two orders $\succ$ and $\succsim$ where $\succ$ is well-founded and $\succsim$ is a compatible nonstrict order.

As a motivation why we need this extension, consider polynomials over the naturals: we know that $2x \succsim x$ and $y + 1 \succ y$. And for SCNP reduction pairs, we have to be able to show that the multiset $\{\{2x, y + 1\}\}$ is strictly larger than the multiset $\{\{y, x\}\}$, which is easily possible using our definition, but which is not possible if one only defines the

multiset extension of the strict order $\succ$ and then takes equality as nonstrict order. The reason is that neither $2x \succ x$ nor $2x = x$ is valid.

Also for RPO we require the multiset extension of two orders, since termination provers like AProVE use a variant of RPO, where $\mathsf{f}(\mathsf{g}(x, y), \mathsf{s}(z)) \succ \mathsf{f}(\mathsf{g}(\mathsf{0}, y), z)$. This orientation is possible, if $\mathsf{0}$ has least precedence among all symbols: then $x \succsim \mathsf{0}$ and thus, $\mathsf{g}(x, y) \succsim \mathsf{g}(\mathsf{0}, y)$, which shows that the multiset $\{\{\mathsf{g}(x, y), \mathsf{s}(z)\}\}$ is strictly larger than $\{\{\mathsf{g}(\mathsf{0}, y), z\}\}$.

In addition to the formalization effort of the multiset extension of two orders, we also investigated the difference to the standard multiset extension of one order. Whereas most properties are similar for both variants, we proved one important difference: the decision procedure for checking whether two multisets are in relation becomes NP-complete when using two orders, whereas it is known to be in P for one order. This is clearly an interesting result w.r.t. Problem 3.

After having integrated the required multiset extension, we formalized RPO and SCNP reduction pairs. For RPO we used a definition that contains many extensions which is required for checking proofs that are generated by AProVE. For example, both quasi-precedences and inference rules like $x \succsim \mathsf{0}$ are integrated. For strong normalization we did not use Kruskal's tree theorem [72], but adapted a direct strong normalization proof of the lexicographic path order from [17] which uses computability predicates à la Tait and Girard (a similar and extended proof has also been performed in [57] for the higher-order recursive path order.) During the formalization we detected that the RPO as defined in AProVE is not a reduction order, since it is not closed under substitutions. However, we were able to repair the definition by adding two more inference rules.

Regarding SCNP reduction pairs, there was an even more severe problem. In [20] a nonstandard definition of reduction pair is used, which turned out to be inconsistent. Therefore, we first had to find an alternative definition of reduction pair which could be used to prove the results of the paper. Here, the notion of reduction triples of [51] was helpful, and eventually we could adapt reduction triples to achieve the soundness result for SCNP reduction pairs. Moreover, for SCNP reduction pairs we also had to formalize other multiset extensions of two orders, like the dual-multiset-extension [7].

To summarize, we formalized the multiset extension of two orders, proved that the decision problem for this order is NP-complete, and formalized and corrected both SCNP reduction pairs and a powerful variant of RPO.

My work was the full development of LPO in IsaFoR (which was later extended to RPO by my co-authors), the full formalization of SCNP reduction pairs, and the NP-completeness proof for the multiset extension of two orders. My co-authors integrated the multiset extension for two orders in IsaFoR, and implemented RPO in a memoized version within IsaFoR, since the naive recursive algorithm would immediately result in exponential runtime, even without performing multiset comparisons.

## 4.8. Certification of Nontermination Proofs (Chapter 13)

### Publication details

Christian Sternagel and René Thiemann. Certification of Nontermination Proofs. In *Proceedings of the 3rd International Conference on Interactive Theorem Proving*, volume 7406 of LNCS, pages 266–282. Springer, 2012.

In the last selected paper, we mainly consider Problem 1. However, this time we want to certify nontermination proofs. So where in the previous papers, one was mainly concerned about soundness of termination techniques, here we are looking at completeness of termination techniques, where we not only consider termination but also innermost termination.

Luckily, for many termination techniques, completeness is easy to prove. For example all techniques that just remove rules are immediately sound: whenever the reduced TRS is nonterminating, then so is the full one. Hence, rule removal with any order, dependency graph decomposition, and various reduction pair processors are all complete techniques as they just remove rules and dependency pairs.

However, there is at least one technique which is not trivial, namely the technique which directly proves nontermination of a TRS. In the paper we consider an easy sufficient criterion of nontermination, namely loops, i.e., derivations of the form $t \to_{\mathcal{R}}^{+} C[t\sigma]$ for some context $C$ and substitution $\sigma$. Of course, given such a derivation in the certificate, one can easily check whether all steps in the derivation are indeed correct rewrite steps, and afterwards can guarantee nontermination, since $\to_{\mathcal{R}}$ is closed under substitutions and contexts.

But for innermost termination—where rewriting is performed using the innermost evaluation strategy, i.e., in an innermost rewrite step $C[\ell\sigma] \to_{\mathcal{R}} C[r\sigma]$, all arguments of $\ell\sigma$ must be normal forms—this is not necessarily the case: in general, the innermost rewrite relation is not closed under substitutions. For example, the TRS $\{\mathsf{f}(\mathsf{s}(x)) \to \mathsf{f}(\mathsf{s}(\mathsf{s}(x))), \mathsf{s}(\mathsf{s}(\mathsf{s}(x)))) \to \mathsf{overflow}\}$ is innermost termination, but it has a loop: $t = \mathsf{f}(\mathsf{s}(x)) \to \mathsf{f}(\mathsf{s}(\mathsf{s}(x))) = C[t\sigma]$ for the empty context $C$ and the substitution which replaces $x$ by $\mathsf{s}(x)$.

In [103] it was shown, that the question whether a loop is an innermost loop, i.e., whether we can conclude innermost nontermination, is decidable. To this end, a complex algorithm was provided, which works in three phases. Especially the last phase utilizes some complex algorithm, where one has to decide for given $s$, $t$, and $\sigma$ whether there is some $n$ such $s\sigma^n = t\sigma^n$. Termination of this algorithm is based on Kruskal's tree theorem and in the original proof it is argued via infinite terms which are obtained as limit of the terms $s, s\sigma, s\sigma^2, \ldots$. To ensure that these limit terms are well defined, some preprocessing on $\sigma$ is done before starting the real algorithm.

In contrast, in the formalization we were able to provide a simpler algorithm with simpler proofs: no preprocessing is required, and termination is easy to prove. Moreover, from our algorithm one can easily extract a precise bound $b$ on $n$ which results in a trivial algorithm: there exists an $n$ such that $s\sigma^n = t\sigma^n$ if and only if $s\sigma^b = t\sigma^b$ where $b = (|s| \cdot |t| \cdot |\sigma|)^2$. This result is then used to prove that the whole decision procedure for innermost loops is in P.

The paper also contains some contributions w.r.t. Problem 3, namely it shows how `partial-function` can be used to develop efficient algorithms in Isabelle which cannot be defined via `function`.

For this paper, my co-author implemented a framework for nontermination proofs and integrated completeness results for several techniques. My own work was the complete development and implementation of the new decision procedure for innermost loops in combination with its soundness proof.

In addition to what is written in the paper, I also formalized a result on nonlooping nontermination [30] and integrated sufficient syntactic criteria where nontermination implies innermost nontermination which are based on confluence [40, 41, 46].

## 4.9. Further Contributions

In the area of term rewriting, we integrated several further important results in IsaFoR that are not addressed in the selected papers.

- on confluence: the critical pair lemma and the result that weak orthogonality implies confluence
- on termination: Knuth-Bendix orders, matchbounds, bounded increase, switching between termination and innermost termination, outermost loops, unravelings for conditional rewriting
- on the word problem: Birkhoff's theorem and Knuth-Bendix completion
- on complexity: strongly linear interpretations, triangular matrix interpretations, modular complexity proofs

All of these techniques are complemented with executable functions for certification. As a result, with CeTA we can also check innermost (non)termination proofs, (non)confluence proofs, completion proofs, and equational reasoning (dis)proofs. Clearly, these developments are contributions w.r.t. Problems 1 and 3—if one replaces termination by other properties like completion, confluence, or complexity. Concerning Problem 4, we extended CPF for these kinds of proofs as well, and currently CeTA can certify completion proofs from $\mathsf{mkb}_{\mathsf{TT}}$ [107] and KBCV [99], (non)confluence proofs from CSI [108], and complexity proofs from $\mathsf{T}_{\mathsf{C}}\mathsf{T}$ [4] and CaT [81]. For confluence and complexity, CeTA already participated in the corresponding competitions.[3]

Besides term rewriting, we also developed several auxiliary libraries which became available to all Isabelle users in the archive of formal proofs (AFP).[4] In the following list, we only mention those AFP-entries where the majority of the formalization was performed by ourselves: executable operations on nonlinear multivariate polynomials, executable operations on matrices, executable algorithms to compute (reflexive-)transitive closures, the automatic generation of linear orders for algebraic datatypes, and the formalization of the Babylonian method to compute square roots.

## 4.10. Summary of Contributions

We formalized a large amount of termination techniques for term rewrite systems (Problem 1). All of them can be freely combined using a unified semantics for termination of dependency pair problems (Problem 2). Moreover, for all techniques we developed executable functions which check the application criteria and can thus be used to certify termination proofs (Problem 3). These have to provided in the common CPF format which is supported by the certifiers CC, CI, and RC, as well as the termination tools AProVE, Matchbox, and $\mathsf{T}_{\mathsf{T}}\mathsf{T}_2$ (Problem 4).

The size of our formalization IsaFoR is over 100,000 lines, and the generated certifier CeTA is a Haskell program of over 24,000 lines. Both IsaFoR and CeTA are freely available at `http://cl-informatik.uibk.ac.at/software/ceta/`.

[3] `http://coco.nue.riec.tohoku.ac.jp/2012/` and `http://termcomp.uibk.ac.at/termcomp/competition/competitionSummary.seam?comp=362062`

[4] `http://afp.sourceforge.net`

To evaluate the power of CeTA, we performed the following experiment: we took the most powerful tool from the recent termination competition (AProVE TC 2012) and let it run on all 2778 TRSs of the termination problem database.[5] The results show, that AProVE can prove termination or nontermination of 2101 TRSs and it fails on the remaining 677 TRSs.[6] If we restrict AProVE to only use techniques that are supported by CPF and CeTA (version 2.8), then AProVE can still prove termination or nontermination of 1850 TRSs, i.e., the step from untrusted to certifiable proofs reduces the power by only 12 %. And indeed, all these 1850 generated proofs are correct, as they have been certified by CeTA.

Concerning soundness of termination techniques, the most important observation was the fact that signature extensions are unsound when used in combination with dependency pair problems. As a consequence, we could prove that the termination technique of root-labeling is unsound without additional restrictions. Hence, all tools that implement root-labeling had to be adapted.

In addition to problems in theory, there were also problems in the implementations of the termination tools which became visible during certification. In the following, we list rejected techniques, i.e., techniques whose application was rejected by CeTA at least once, and the underlying reason why these proofs have been rejected.

| termination technique | reason for rejection |
|---|---|
| dependency pairs | no well-formedness check |
| dependency pairs | bug in computation of dependency pairs |
| loop detection | evaluation strategy was ignored |
| bounded increase | bug in implementation of calculus for conditional constraints |
| reduction pair proc. | bug in output of LPO |
| reduction pair proc. | bug in computation of usable rules |

In the first three problems, it was easy to exploit the bug to let the tool give a wrong answer, i.e., we could develop nonterminating TRSs where the tools provided termination "proofs", or vice versa. In the forth problem, it is currently open, whether the bug can be exploited to obtain a wrong answer, however in experiments it was shown, that after fixing the bug, at least for one TRS a termination proof can no longer be detected by that tool, and it is unknown whether the TRS is terminating or not. Finally, in the fifth and sixth problem, the bug just had impact on the output, i.e., the principle answer of termination or nontermination was correct, but the corresponding generated proof was incorrect. For example, in the sixth problem, one termination tool utilizes two methods to compute the usable rules. One, which is used in the search engine and has impact on the internal state (without the bug); and a bogus one for pretty printing the proof.

Several of these bugs have remained undetected for quite some while: they were already present in versions of the tools, that have participated in competitions and have been used for getting empirical data for refereed papers, where no one spotted the mistakes in the generated proofs.

All of these problems have been fixed in the meantime, so that these termination tools became more reliable due to certification.

[5] Version 8.0.6, available at `http://termcomp.uibk.ac.at/status/downloads/`.

[6] The details of the experiments are available at `http://termcomp.uibk.ac.at/termcomp/termexec/categoryList.seam?competitionId=411173`. For technical reasons, the experiments are split into string rewrite systems (SRS standard) and term rewrite systems (TRS standard).

# 5. Future Research

As already mentioned, CeTA can currently handle around 88 % of the termination and nontermination proofs for term rewrite systems since dozens of termination techniques have already been formalized.

This is in stark contrast to other properties of term rewrite systems. For example, in the confluence competition 2012, the tools ACP [2], CSI, and Saigawa [52] were successfully on at least 88 TRSs. In contrast, with the confluence techniques in IsaFoR, only 27 TRSs can be handled, i.e., the power is reduced to only 30 %. Similarly, if one restricts complexity tools like AProVE, CaT, Matchbox, or T$_{\mathsf{C}}$T to only use techniques supported by CeTA, then the power is again significantly reduced, where the most severe loss is for runtime complexity: in the competition 2012, the restricted version of T$_{\mathsf{C}}$T achieved a score of 27 whereas AProVE and T$_{\mathsf{C}}$T reached a score of at least 92.

For both confluence and complexity, the problem is that there are many techniques in these areas which require several nontrivial formalizations. It will be an interesting task to extend IsaFoR towards complexity and confluence to obtain a similar coverage as for termination techniques.

Even more severe is the situation when regarding complexity or termination techniques for other programming languages than term rewriting which are integrated in tools like AProVE, COSTA [1], Julia [93], Polytool, RAML [55], SPEED [47], and Terminator: we are not aware of any formalizations of these important techniques. Here, it looks promising to continue our work by formalizing those methods which prove termination via a transformation to TRSs as in [38] (for Haskell) or in [15, 16, 87] (for Java): we can reuse our existing formalization on rewriting, and there is already an Isabelle/HOL formalization of a Java like language [61, 75].

# Part II.

# Selected Papers

# 6. Certification of Termination Proofs using CeTA


### Publication details

René Thiemann and Christian Sternagel. Certification of Termination Proofs using CeTA. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of LNCS, pages 452–468. Springer, 2009.



### 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.


## 6.1. 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 [21, 25] and Rainbow/CoLoR [12] 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.

$$\texttt{proof tree} \xrightarrow{\text{C}i\text{ME/Rainbow}} \texttt{proof.v} \xrightarrow{\text{Coq + Coccinelle/CoLoR}} \text{accept/failure}$$

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] [84] and not for Coq.

Second, and more important, instead of generating for each proof tree a new Coq-proof using the auxiliary tools C*i*ME/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 [48] to create a certified Haskell program [90], CeTA, leading to the following workflow.

$$\begin{array}{rcl} \text{IsaFoR} & \xrightarrow{\text{Isabelle}} & \text{Haskell program} \xrightarrow{\text{Haskell compiler}} \text{CeTA} \\ \texttt{proof tree} & \xrightarrow{\text{CeTA}} & \text{accept/error message} \end{array}$$

  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 [3] has been certified, we integrated the most powerful estimation which does not require tree automata techniques and is based

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

on a combination of [41, 50] 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 is also beneficial for termination provers. Replacing tcap by etcap within the termination prover T$_{\text{T}}$T$_2$ [67] 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 [3], as it is subsumed by our estimation.

The paper is structured as follows. In Sect. 6.2 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. 6.3–6.6 we explain our certification of the four termination techniques we currently support: dependency pairs (Sect. 6.3), dependency graph (Sect. 6.4), reduction pairs (Sect. 6.5), and combination of proofs in the dependency pair framework (Sect. 6.6). 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. 6.7. We conclude in Sect. 6.8 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`.

## 6.2. Formalizing Term Rewriting

We assume some basic knowledge of term rewriting [5]. Variables are denoted by $x$, $y$, $z$, etc., function symbols by $f$, $g$, $h$, etc., terms by $s$, $t$, $u$, etc., and substitutions by $\sigma$, $\mu$, etc. Instead of $f(t_1, \ldots, t_n)$ we write $f(\vec{t}_n)$. 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., $\texttt{SN}(\mathcal{A}) = \texttt{wf}(\mathcal{A}^{-1})$. 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 6.1.** 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 $\ell$ 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 6.2.** *Term $s$ rewrites to $t$ by $\mathcal{R}$, iff there are a context $C$, a substitution $\sigma$, 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."

## 6.3. Certifying Dependency Pairs

Before we introduce dependency pairs [3] 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 [28]).

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 $\sigma$. 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. [3].

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

**Definition 6.3.** *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 $\ell$ 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 6.4.** The dependency pairs for the TRS from Ex. 6.1 consist of the rules

$$\begin{aligned} \mathsf{M}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{M}(x, y) \quad (\mathsf{MM}) & \mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{M}(x, y) \quad (\mathsf{DM}) \\ & & \mathsf{D}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{D}(\mathsf{minus}(x, y), \mathsf{s}(y)) \quad (\mathsf{DD}) \end{aligned}$$

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

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 $t \in \mathcal{T}^{\infty}_{\mathcal{R}}$

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

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 ♯-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. 6.3 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. [40]. Notice that in IsaFoR a DP problem is just a pair of two TRSs over arbitrary signatures—similar to [40].

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

```
fun ichain where "ichain(𝒫,ℛ) s⃗ t⃗ σ⃗ = (∀i.
  (s⃗ i,t⃗ i) ∈ 𝒫 ∧ (t⃗ i)·(σ⃗ i) →ℛ* (s⃗(i+1))·(σ⃗(i+1))"

fun finite_dpp where
  "finite_dpp(𝒫,ℛ) = (¬(∃s⃗ t⃗ σ⃗. ichain (𝒫,ℛ) s⃗ t⃗ σ⃗))"
```

where '$t \cdot \sigma$' denotes the application of substitution $\sigma$ 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 6.5.** $\mathtt{wf\_trs}(\mathcal{R}) \wedge \mathtt{finite\_dpp}(\mathtt{DP}(\mathcal{R}), \mathtt{ID}(\mathcal{R})) \longrightarrow \mathtt{SN}(\to_{\mathcal{R}})$.

The additional premise $\mathtt{wf\_trs}(\mathcal{R})$ ensures two well-formedness properties for $\mathcal{R}$, namely that for every $\ell \to r \in \mathcal{R}$, $\ell$ 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 $\mathtt{SN}(\to_{\mathcal{R}})$ for some TRS $\mathcal{R}$, to the problem of proving $\mathtt{finite\_dpp}(\mathtt{DP}(\mathcal{R}), \mathtt{ID}(\mathcal{R}))$, and thus enter the realm

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

of the DP framework [40]. 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

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

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. 6.3, 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. 6.3 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 6.6.** *If* $\texttt{checkDPs}(\mathcal{P},\mathcal{R})$ *is accepted, then finiteness of* $(\mathcal{P},\texttt{ID}(\mathcal{R}))$ *implies* $\texttt{SN}(\to_{\mathcal{R}})$.

## 6.4. Certifying the Dependency Graph Processor

One important processor to prove finiteness of a DP problem is based on the *dependency graph* [3, 40]. 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 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.

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

Therefore, in the upcoming Sect. 6.4.1 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. 6.4.2 we implemented one of the strongest dependency graph estimations where the test for an edge can become expensive. In Sect. 6.4.3 we finally show how to combine the results of Sections 6.4.1 and 6.4.2.

### 6.4.1. Certifying Graph Decompositions

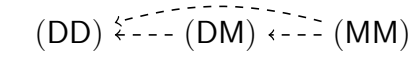
Instead of doing an SCC decomposition of a graph within IsaFoR we base our check on the decomposition that is provided by the termination prover. Essentially, we demand that the set of components is given as a list $\langle C_1, \ldots, C_k \rangle$ in topological order where the component with no incoming 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 6.7.** 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 6.8.** *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 $\alpha$ 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 $\beta$ of $\alpha$ and some marked $C_i$ such that all nodes of $\beta$ belong to $C_i$.*

**Example 6.9.** If we continue with example Ex. 6.7 where only components $\{(\mathsf{MM})\}$ and $\{(\mathsf{DD})\}$ are marked, then our check also analyzes that $\mathcal{G}$ contains no edge from $(\mathsf{DM})$ to itself. If it succeeds, every infinite path will in the end only contain nodes from $\{(\mathsf{DD})\}$ or only nodes from $\{(\mathsf{MM})\}$. In this way, only 4 edges of $\mathcal{G}$ have to be calculated instead of analyzing all 9 possible edges in $\mathcal{P} \times \mathcal{P}$.

### 6.4.2. 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 $\sigma$ and $\mu$ 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 [3] first replaces via an operation $\mathsf{cap}$ 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.

The estimation EDG*[50] 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 $\mathsf{cap}$, called $\mathsf{tcap}$ [41]. It only replaces subterms with defined symbols by a fresh variable, if there is a rule that unifies with the corresponding subterm.

**Definition 6.10.** *Let $\mathcal{R}$ be a TRS.*

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

- $\mathsf{tcap}(f(\vec{t}_n)) = f(\mathsf{tcap}(t_1), \ldots, \mathsf{tcap}(t_n))$ *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. 6.1 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 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 [78], 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 6.11.** *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 $\ell$ can be formulated as a *ground-context matching problem* between $C$ and $\ell$, which is satisfiable iff there is some $\mu$ 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 $(C_i, \ell_i)$.

**Definition 6.12.** *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* $\mu$ *such that* $\ell_i\mu \in [\![C_i]\!]$ *for all* $1 \leq i \leq n$. *We sometimes abbreviate* $\{(C, \ell)\}$ *by* $(C, \ell)$.

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

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

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 $\mu$ such that $x\mu \in [\![C]\!]$ and $x\mu \in [\![D]\!]$ and $\ldots$ iff
- there is some $\mu$ such that $x\mu \in [\![C]\!] \cap [\![D]\!]$ and $\ldots$ iff
- there is some $\mu$ 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.

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

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 6.14.**

- *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* $\mu$ *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* $\mu$ *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 6.15.**

- $\mathsf{etcap}(f(\vec{t}_n)) = f(\mathsf{etcap}(t_1), \ldots, \mathsf{etcap}(t_n))$ *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 $\mathsf{etcap}$. 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 6.16.** *(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* $\mathcal{R}^{-1}$.

### 6.4.3. 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. 6.16 (b).

In IsaFoR the soundness result of our check is proven.

**Theorem 6.17.** *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 T$_{\text{T}}$T$_2$ reduced the time for estimating the dependency graph by a factor of two.

## 6.5. 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 [40], where we ignore the usable rules refinement.

**Theorem 6.18.** *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. 6.18. Currently, we just support reduction pairs based on polynomial interpretations with negative constants [51], 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`$(\texttt{checkS}, \texttt{checkNS}, \mathcal{P}, \mathcal{P}', \mathcal{R})$ 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. 6.18. 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.

## 6.6. Certifying the Whole Proof Tree

From Sect. 6.3–6.5 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)
```

**Theorem 6.19.** *If* `checkDPProof(P,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.

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

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 6.20.** *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 whitespace). 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.

## 6.7. 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)`.

## 6.8. 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 [39] or C*i*ME [22] as provers (ACC,CCC), and to Rainbow/CoLoR using AProVE or Matchbox [106] (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. 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.

[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.

# 7. Certified Subterm Criterion and Certified Usable Rules

**Publication details**

Christian Sternagel and René Thiemann. Certified Subterm Criterion and Certified Usable Rules. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of LIPIcs, pages 325–340. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.

**Abstract**

In this paper we present our formalization of two important termination techniques for term rewrite systems: the subterm criterion and the reduction pair processor in combination with usable rules. For both techniques we developed executable check functions using the theorem prover Isabelle/HOL. These functions are able to certify the correct application of the formalized techniques in a given termination proof. As there are several variants of usable rules, we designed our check function in such a way that it accepts all known variants, even those which are not explicitly spelled out in previous papers.

We integrated our formalization in the publicly available IsaFoR-library. This led to a significant increase in the power of CeTA, a certified termination proof checker that is extracted from IsaFoR.

## 7.1. 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 (DP framework). This allows to combine basic termination techniques in a flexible way. Hence, a termination proof is a tree where at each node a specific technique is applied. So instead of just stating the precedence of some lexicographic path order or giving some polynomial interpretation, current termination provers return proof trees consisting of many different techniques and reaching sizes of several megabytes. Thus, it would be too much work to check by hand whether these trees really form a valid proof. (Additionally, checking by hand does not provide a very high degree of confidence.)

It is regularly demonstrated that we cannot blindly trust in the output of termination provers. Every now and then, some termination prover delivers a faulty proof. Most often, this is only detected if there is another prover giving a contradicting answer on the same problem. To solve this problem, three systems have been developed over the last few years: CiME/Coccinelle [21, 23], Rainbow/CoLoR [12], and CeTA/IsaFoR [104]. These systems either certify or reject a given termination proof. Here, Coccinelle and CoLoR

are libraries on rewriting for Coq (`http://coq.inria.fr`) and IsaFoR is our library on rewriting for Isabelle [84]. (Throughout this paper we just write Isabelle whenever we refer to Isabelle/HOL.)

All of these certifiers can automatically certify termination proofs that are performed within the DP framework. In this framework one tries to simplify so called DP problems $(\mathcal{P}, \mathcal{R})$ by processors until all pairs in $\mathcal{P}$ are removed.

The reduction pair processor [42, 50] is the major technique to remove pairs. Consequently, it has been formalized in all three libraries. One of the conditions of the processor demands that all rules in $\mathcal{R}$ must be weakly decreasing. If this and all other conditions are satisfied then one can remove all strictly decreasing pairs. In this paper, we present the details about the formalization of two important extensions of the reduction pair processor.

The first extension is the subterm criterion [51]. By restricting the used "reduction pair" to the subterm relation in combination with simple projections, it is possible to ignore the $\mathcal{R}$-component of a DP problem. Note that the subterm criterion has independently (and only recently) been formalized for the Coccinelle-library [23]. Here, we present the first Isabelle formalization of this important technique.

The other extension is the integration of usable rules [40–42, 50, 105]. With this extension not all rules in $\mathcal{R}$ have to be weakly decreasing but only the usable rules which are most often a strict subset of $\mathcal{R}$. However, there are several definitions of usable rules where the most powerful ones ([41] and [42]) are incomparable.

$$\text{all rules} \supseteq \text{usable rules ([105])} \supseteq \text{usable rules ([40, 50])} \begin{cases} \text{usable rules ([41])} \\ \text{usable rules ([42])} \end{cases}$$

Although it was often stated that a combination of the definitions of usable rules of [41] and [42] would be possible there never was a refereed paper which showed such a proof. (However, there have been unpublished soundness proofs of such a combined definition.) In this paper we not only present such a combined definition and the first corresponding *formalized* soundness proof, but we also simplified and extended the existing proofs. For example, we never construct filtered terms although we consider usable rules w.r.t. some argument filter. (An independent formalization of usable rules is present in Coccinelle. However, this formalization is unpublished and it only uses the variant of [50]: it does not feature the improvements from [41] and [42].) With these two extensions of the reduction pair processor we could increase the number of TRSs (from the Termination Problem Database) where a proof can be certified by our certifier CeTA by over 50%.

Note that all the proofs that are presented (or omitted) in the following, have been formalized in our Isabelle library IsaFoR. Hence, in the paper we merely give sketches of our "real" proofs. Our goal is to show the general proof outlines and help to understand the full proofs. Our library IsaFoR with all formalized proofs, the executable certifier CeTA, and all details about our experiments are available at CeTA's website:

`http://cl-informatik.uibk.ac.at/software/ceta`

The paper is structured as follows: In Sec. 7.2, we recapitulate the required notions and notations of term rewriting and the DP framework. In Sec. 7.3, we describe our formalization of the subterm criterion. The reduction pair processor with usable rules and its formalization is presented in Sec. 7.4. Then, in Sec. 7.5, we shortly describe how CeTA is obtained from IsaFoR and give a summary about our experiments. We finally conclude in Sec. 7.6.

## 7.2. Preliminaries

*Term Rewriting.* We assume familiarity with term rewriting [5]. Still, we recall the most important notions that are used later on. A *(first-order) term* $t$ over a set of *variables* $\mathcal{V}$ and a set of *function symbols* $\mathcal{F}$ is either a variable $x \in \mathcal{V}$ or an $n$-ary function symbol $f \in \mathcal{F}$ applied to $n$ argument terms $f(\vec{t}_n)$. A *context* $C$ is a term containing exactly one occurrence of the special constant $\Box$ (that is assumed to be distinct from symbols in $\mathcal{F}$). Replacing $\Box$ in a context $C$ by a term $t$ is denoted by $C[t]$. A term $t$ is a *(proper) subterm* of a term $s$—written $(s \rhd t)$ $s \trianglerighteq t$—whenever there exists a context $C$ $(\neq \Box)$, such that $s = C[t]$. We write $s \approx t$ iff $s$ and $t$ are unifiable. An *argument filter* $\pi$ is a mapping from symbols to integers or lists of integers. It induces a mapping from terms to terms where $\pi(x) = x$, $\pi(f(\vec{t}_n)) = \pi(t_i)$ if $\pi(f) = i$, and $\pi(f(\vec{t}_n)) = f(\pi(t_{i_1}), \ldots, \pi(t_{i_k}))$ if $\pi(f) = [i_1, \ldots, i_k]$. Argument filters are also used to indicate which positions in a term are regarded. Then $\pi$ maps symbols to sets of positions. It will be clear from the context which kind of argument filters is used.

A *rewrite rule* is a pair of terms $\ell \to r$ and a TRS $\mathcal{R}$ is a set of rewrite rules. The *rewrite relation (induced by $\mathcal{R}$)* $\to_{\mathcal{R}}$ is the closure under substitutions and under contexts of $\mathcal{R}$, i.e., $s \to_{\mathcal{R}} t$ iff there is a context $C$, a rewrite rule $\ell \to r \in \mathcal{R}$, and a substitution $\sigma$ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. Reductions at the root are denoted by $\to_{\mathcal{R},\epsilon}$.

We say that an element $t$ is *terminating / strongly normalizing (w.r.t. some binary relation $S$)*, and write $\mathsf{SN}_S(t)$, if it cannot start an infinite sequence

$$t = t_1 \; S \; t_2 \; S \; t_3 \; S \cdots .$$

The whole relation is terminating, written $\mathsf{SN}(S)$, if all elements are terminating w.r.t. it. For a TRS $\mathcal{R}$ and a term $t$, we write $\mathsf{SN}(\mathcal{R})$ and $\mathsf{SN}_{\mathcal{R}}(t)$ instead of $\mathsf{SN}(\to_{\mathcal{R}})$ and $\mathsf{SN}_{\to_{\mathcal{R}}}(t)$. We write $S^+$ for the transitive closure of $S$, and $S^*$ is the reflexive transitive closure.

**Lemma 7.1** (Properties of Subterms)**.**

*(a) stability:* $s \rhd t \Longrightarrow s\sigma \rhd t\sigma$

*(b) subterms preserve termination:* $\mathsf{SN}_{\mathcal{R}}(s) \wedge s \rhd t \Longrightarrow \mathsf{SN}_{\mathcal{R}}(t)$.

Let $\to_{\mathsf{SN}(\mathcal{R})}$ denote the restriction of $\to_{\mathcal{R}}$ to terminating terms, i.e., $\{(s,t) \mid s \to_{\mathcal{R}} t \wedge \mathsf{SN}_{\mathcal{R}}(s)\}$. Let $\overset{\rhd}{\to}_{\mathsf{SN}(\mathcal{R})}$ denote the same relation extended by the restriction of $\rhd$ to terminating terms, i.e., $\to_{\mathsf{SN}(\mathcal{R})} \cup \{(s,t) \mid s \rhd t \wedge \mathsf{SN}_{\mathcal{R}}(s)\}$.

**Lemma 7.2** (Termination Properties)**.** *Let $S$ be some binary relation, let $\mathcal{R}$ be a TRS.*

*(a)* $\mathsf{SN}(S) \Longleftrightarrow \mathsf{SN}(S^+)$,

*(b)* $\mathsf{SN}(\overset{\rhd}{\to}_{\mathsf{SN}(\mathcal{R})})$,

*(c)* $\mathsf{SN}_S(s) \wedge (s,t) \in S \Longrightarrow \mathsf{SN}_S(t)$.

*Dependency Pair Framework.* The DP framework [42] is a way to modularize termination proofs. Therefore, we switch from TRSs to so called DP problems, consisting of two TRSs. The *initial DP problem* for a TRS $\mathcal{R}$ is $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$ where $\mathsf{DP}(\mathcal{R})$ are the *dependency pairs* of $\mathcal{R}$. A $(\mathcal{P}, \mathcal{R})$-*chain* is a possibly infinite derivation of the following form:

$$s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to^*_{\mathcal{R}} s_3\sigma_3 \to_{\mathcal{P}} \cdots \qquad (\star)$$

where $s_i \to t_i \in \mathcal{P}$ for all $i > 0$ (this implies that $\mathcal{P}$-steps only occur at the root). If additionally every $t_i\sigma_i$ is terminating w.r.t. $\mathcal{R}$, then the chain is *minimal*. A DP problem $(\mathcal{P}, \mathcal{R})$ is called *finite* [42], if there is no minimal $(\mathcal{P}, \mathcal{R})$-chain. Proving finiteness of a DP problem is done by simplifying $(\mathcal{P}, \mathcal{R})$ by so called *processors* recursively, until the $\mathcal{P}$-components of all remaining DP problems are empty and therefore trivially finite. For this to be correct, the applied processors need to be *sound*. A processor *Proc* is sound whenever for all DP problems $(\mathcal{P}, \mathcal{R})$ we have that finiteness of $(\mathcal{P}', \mathcal{R}')$ for all $(\mathcal{P}', \mathcal{R}') \in \mathit{Proc}(\mathcal{P}, \mathcal{R})$ implies finiteness of $(\mathcal{P}, \mathcal{R})$. The termination techniques that will be introduced in the following sections are all such sound processors.[1]

**Example 7.3.** In the following TRS $\mathcal{R}$ the term $\mathsf{set}(xs)$ evaluates to the list $[x \in xs \mid 0 < x]$ where duplicates are removed:

$$\begin{aligned}
x < \mathsf{0} &\to \bot, && (1)\\
\mathsf{0} < \mathsf{s}(y) &\to \top, && (2)\\
\mathsf{s}(x) < \mathsf{s}(y) &\to x < y, && (3)\\
\mathsf{set}(\mathsf{nil}) &\to \mathsf{nil}, &&\\
\mathsf{set}(x : z) &\to \mathsf{if2}(\mathsf{0} < x, x, z), &&\\
\mathsf{del}(x, \mathsf{nil}) &\to \mathsf{nil}, && (4)\\
\mathsf{del}(x, y : z) &\to \mathsf{if}(x < y, y < x, x, y, z), && (5)\\
\mathsf{if}(\bot, \bot, x, y, z) &\to \mathsf{del}(x, z), && (6)\\
\mathsf{if}(\top, b, x, y, z) &\to y : \mathsf{del}(x, z), && (7)\\
\mathsf{if}(b, \top, x, y, z) &\to y : \mathsf{del}(x, z), && (8)\\
\mathsf{if2}(\top, x, z) &\to x : \mathsf{set}(\mathsf{del}(x, z)),&&\\
\mathsf{if2}(\bot, x, z) &\to \mathsf{set}(z).&&
\end{aligned}$$

After computing the initial DP problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$ we can split it into the three problems $(\{(9)\}, \mathcal{R})$, $(\{(13)\text{–}(16)\}, \mathcal{R})$, and $(\{(10)\text{–}(12)\}, \mathcal{R})$. (This is done by applying the dependency graph processor [3,41,42,50], a well-known technique to perform separate termination proofs for each recursive function.)

$$\begin{aligned}
\mathsf{s}(x) <^\sharp \mathsf{s}(y) &\to x <^\sharp y, && (9)\\
\mathsf{set}^\sharp(x : z) &\to \mathsf{if2}^\sharp(\mathsf{0} < x, x, z), && (10)\\
\mathsf{if2}^\sharp(\top, x, z) &\to \mathsf{set}^\sharp(\mathsf{del}(x, z)), && (11)\\
\mathsf{if2}^\sharp(\bot, x, z) &\to \mathsf{set}^\sharp(z), && (12)\\
\mathsf{del}^\sharp(x, y : z) &\to \mathsf{if}^\sharp(x < y, y < x, x, y, z), && (13)\\
\mathsf{if}^\sharp(\bot, \bot, x, y, z) &\to \mathsf{del}^\sharp(x, z), && (14)\\
\mathsf{if}^\sharp(\top, b, x, y, z) &\to \mathsf{del}^\sharp(x, z), && (15)\\
\mathsf{if}^\sharp(b, \top, x, y, z) &\to \mathsf{del}^\sharp(x, z). && (16)
\end{aligned}$$

## 7.3. The Subterm Criterion

The subterm criterion [51] is a termination technique that can be employed as a processor of the DP framework. It may be seen as a variant of the reduction pair processor with an attached argument filtering [3]. The used orders ($\rhd$ and $\trianglerighteq$) allow to ignore the $\mathcal{R}$ component of a DP problem $(\mathcal{P}, \mathcal{R})$. And the argument filtering is restricted to be a so called *simple projection*. A simple projection $\pi$ maps a term to one of its arguments, i.e., $\pi(f(\vec{t}_n)) = t_i$ for some $0 < i \leqslant n$. For convenience we use $R_\pi$ to denote the 'composition' of the binary relation on terms $R$ and $\pi$, i.e., $(s, t) \in R_\pi$ iff $(\pi(s), \pi(t)) \in R$.

**Theorem 7.4.** *Finiteness of $(\mathcal{P} \setminus \rhd_\pi, \mathcal{R})$ implies finiteness of $(\mathcal{P}, \mathcal{R})$, provided:*

*(a) all rules of $\mathcal{P}$ are oriented by $\trianglerighteq_\pi$ (i.e., $\mathcal{P} \subseteq \trianglerighteq_\pi$)*

[1] To be more precise, in IsaFoR it is shown that all these processors are chain identifying (`chain_identifying_proc`) which is a slightly stronger requirement than soundness [101, Chapter 7]. The reason is that chain identifying processors can easily be combined with semantic labeling [110]. However, we omit the details here and just refer to theory `DpFramework` for the interested reader.

*(b) all lhss and rhss of $\mathcal{P}$ are non-variable and non-constant terms where the roots of rhss are not defined in $\mathcal{R}$ (i.e., $s = f(\vec{s}_n)$ with $n > 0$ and $t = g(\vec{t}_m)$ with $m > 0$ and $g \notin \mathcal{D}_\mathcal{R}$ for all $s \to t \in \mathcal{P}$)*

**Example 7.5.** The DP problem $(\{(9)\}, \mathcal{R})$ from Ex. 7.3 can be solved using the simple projection $\pi(<^\sharp) = 1$, since $\pi(\mathsf{s}(x) <^\sharp \mathsf{s}(y)) = \mathsf{s}(x) \rhd x = \pi(x <^\sharp y)$. Taking $\pi(\mathsf{del}^\sharp) = 2$ and $\pi(\mathsf{if}^\sharp) = 5$ we can remove Pair (13) from $(\{(13)\text{–}(16)\}, \mathcal{R})$. The result $(\{(14)\text{–}(16)\}, \mathcal{R})$ is then solved by the dependency graph processor. Removing a pair from $(\{(10)\text{–}(12)\}, \mathcal{R})$ is impossible as there is no $\pi$ such that Pair (11) is oriented. Note that $<^\sharp, \mathsf{del}^\sharp, \mathsf{if}^\sharp, \ldots \notin \mathcal{D}_\mathcal{R}$ whereas $<, \mathsf{del}, \mathsf{if}, \cdots \in \mathcal{D}_\mathcal{R}$.

Before we can prove Theorem 7.4, we need several lemmas. First, we prove that termination of some element w.r.t. some binary relation $S$ is equivalent to termination of the same element w.r.t. $S^+$. Note that this is a more general result than Lem. 7.2(a) and thus allows termination analysis of a single term, no matter if the whole TRS is terminating.

**Lemma 7.6.** $\mathsf{SN}_S(t) \iff \mathsf{SN}_{S^+}(t)$.

*Proof.* The direction from right to left is trivial. For the other direction assume that $t$ is not terminating w.r.t. $S^+$. Hence $t = t_1\ S^+\ t_2\ S^+\ t_3\ S^+ \cdots$. Let $S'$ denote the restriction of $R$ to terminating terms, i.e., $S' = \{(s,t) \mid s\ S\ t \wedge \mathsf{SN}_S(s)\}$. By definition we have $\mathsf{SN}(S')$ and with Lem. 7.2(a) also $\mathsf{SN}(S'^+)$. Using $\mathsf{SN}_S(t)$ and Lem. 7.2(c) together with the infinite sequence from above, we get $\mathsf{SN}_S(t_i)$ for all $i > 0$, and further $t_1\ S'^+\ t_2\ S'^+\ t_3\ S'^+ \cdots$. This contradicts $\mathsf{SN}(S'^+)$. □

Next consider a general result on infinite sequences conducted in the union of two binary relations $N$ and $S$ where often $N$ is a non-strict relation and $S$ a strongly normalizing relation. Intuitively it states the following: Assume that there is an infinite sequence of steps, where each step is an $N$-step or an $S$-step. Further assume that whenever there is an $N$-step directly followed by an $S$-step, those two steps can be turned into a single $S$-step. Additionally, there is no infinite $S$-sequence starting at the same point as the sequence we are reasoning about. Then, from some point in our sequence on, there are no more $S$-steps, i.e., it ends in $N$-steps. This is a versatile fact that is used at several places inside IsaFoR.

**Lemma 7.7.** *Let $N$ and $S$ be two binary relations over some carrier and $\vec{q}$ an infinite sequence of carrier elements. If*

*(a) $(q_i, q_{i+1}) \in N \cup S$ for all $i > 0$,*

*(b) $N \circ S \subseteq S$, and*

*(c) $\mathsf{SN}_S(q_1)$,*

*then there is some $j$ such that for all $i \geq j$ we have $(q_i, q_{i+1}) \in N \setminus S$.*

*Proof.* For the sake of a contradiction assume that the lemma does not hold. Then, together with (a), we obtain $\forall i > 0.\, \exists j \geq i.\, (q_j, q_{j+1}) \in S$. Using the *Axiom of Choice* we get hold of a choice function $f$ such that

$$\forall i > 0.\, f(i) \geq i \wedge (q_{f(i)}, q_{f(i)+1}) \in S, \qquad (\dagger)$$

i.e., $f(i)$ produces some index of an $S$-step after position $i$ in $\vec{q}$. Using $f$ we define a new sequence $[\cdot]$ of indices inductively

$$[i] = \begin{cases} i & \text{if } i = 1, \\ f([i-1]) + 1 & \text{otherwise.} \end{cases}$$

With (†) we have $f(i) \geq i$ and $(q_{f(i)}, q_{f(i)+1}) \in S$ for all $i > 0$. Since $f(i) \geq i$ there is an $N \cup S$ sequence from every $q_i$ to the corresponding $q_{f(i)}$. Thus we obtain $(q_i, q_{f(i)+1}) \in S^+$ for all $i > 0$ using (b). This immediately implies $(q_{[i]}, q_{[i+1]}) \in S^+$ for all $i > 0$ and thereby $\neg\mathsf{SN}_{S^+}(q_{[1]})$ which is equivalent to $\neg\mathsf{SN}_S(q_{[1]})$ by Lem. 7.6. But $q_{[1]} = q_1$ and thus $\neg\mathsf{SN}_S(q_1)$. Together with (c), this provides the desired contradiction. □

**Lemma 7.8.** $\mathsf{SN}_{\mathcal{R}}(t) \Longrightarrow \mathsf{SN}_{(\rhd \cup \to_{\mathcal{R}})}(t)$.

*Proof.* Assume that $t$ is not terminating w.r.t. $(\rhd \cup \to_{\mathcal{R}})$. Hence, we obtain the infinite sequence $t = t_1\ (\rhd \cup \to_{\mathcal{R}})\ t_2\ (\rhd \cup \to_{\mathcal{R}})\ t_3\ (\rhd \cup \to_{\mathcal{R}}) \cdots$. From the assumption we have $\mathsf{SN}_{\mathcal{R}}(t_1)$ and by Lem. 7.1 and Lem. 7.2(c) we obtain $\mathsf{SN}_{\mathcal{R}}(t_i)$ for all $i > 0$. Thus, $t_i \overset{\rhd}{\to}_{\mathsf{SN}(\mathcal{R})} t_{i+1}$ for all $i$ and since $\mathsf{SN}(\overset{\rhd}{\to}_{\mathsf{SN}(\mathcal{R})})$ by Lem. 7.2(b) we arrive at a contradiction. □

*Proof of Theorem 7.4.* In order to show that finiteness of $(\mathcal{P} \setminus \rhd_\pi, \mathcal{R})$ implies finiteness of $(\mathcal{P}, \mathcal{R})$ we prove its contraposition. Hence, we may assume (in addition to the premises of Theorem 7.4) that there is a minimal infinite $(\mathcal{P}, \mathcal{R})$-chain and have to transform it into a minimal infinite $(\mathcal{P} \setminus \rhd_\pi, \mathcal{R})$-chain. Thus we may assume that for all $i > 0$:

(a) $s_i \to t_i \in \mathcal{P}$,

(b) $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$, and

(c) $\mathsf{SN}_{\mathcal{R}}(t_i\sigma_i)$.

We start by a case distinction on $\exists j > 0.\, \forall i \geq j.\, (s_i, t_i) \in (\mathcal{P} \setminus \rhd_\pi)$. If there is such a $j$, we can combine this with (b) and obtain the desired minimal $(\mathcal{P} \setminus \rhd_\pi, \mathcal{R})$-chain by shifting the original chain $j$ positions to the left. Hence, consider the second case and assume $\forall i > 0.\, \exists j \geq i.\, (s_j, t_j) \notin (\mathcal{P} \setminus \rhd_\pi)$. With (a) and the preconditions of the subterm criterion processor this results in

$$\forall i > 0.\, \exists j \geq i.\, \pi(s_j)\sigma_j \rhd \pi(t_j)\sigma_j. \tag{17}$$

From this point on, the proof mainly runs by instantiating the relations $N$ and $S$ of Lem. 7.7 appropriately and showing the assumptions Lem. 7.7(a)–Lem. 7.7(c) in turn. For $N$ we use the reflexive and transitive closure of the rewrite relation, i.e., $\to_{\mathcal{R}}^*$. For $S$ we use $(\rhd \cup \to_{\mathcal{R}})^+$. Finally, we use the infinite sequence $q$ defined by $q_i = \pi(s_{i+1})\sigma_{i+1}$ (the index shift is needed to establish termination of $q_1$ later on). From (a) and Thm. 7.4(a), together with Lem. 7.1(a) we get

$$\pi(s_i)\sigma_i \trianglerighteq \pi(t_i)\sigma_i. \tag{18}$$

Furthermore, we obtain

$$\pi(t_i)\sigma_i \to_{\mathcal{R}}^* \pi(s_{i+1})\sigma_{i+1}, \tag{19}$$

since the roots of $t_i$ and $s_{i+1}$ are guaranteed to be non-constant symbols and the root of $t_i$ is not a defined symbol by Thm. 7.4(b). In combination we get $\pi(s_i)\sigma_i \trianglerighteq \circ \to_{\mathcal{R}}^* \pi(s_{i+1})\sigma_{i+1}$

and in turn $(q_i, q_{i+1}) \in N \cup S$, thereby discharging assumption Lem. 7.7(a). For our specific relations assumption Lem. 7.7(b) trivially holds. This leaves us with showing termination of $q_1$ with respect to the relation $(\rhd \cup \to_{\mathcal{R}})^+$. From the minimality of the initial chain (c) we know $\mathsf{SN}_{\mathcal{R}}(t_1\sigma_1)$ and by Lem. 7.1 we get $\mathsf{SN}_{\mathcal{R}}(\pi(s_2)\sigma_2)$ and thus $\mathsf{SN}_{\mathcal{R}}(q_1)$. By Lemmas 7.8 and 7.6 we then achieve $\mathsf{SN}_{(\rhd\cup\to_{\mathcal{R}})^+}(q_1)$. At this point (by Lem. 7.7) we get grip of some $j > 0$ such that

$$\forall i \geq j.\, (q_i, q_{i+1}) \in N \setminus S. \tag{20}$$

Now we prove $\forall i \geq j.\, \pi(s_{i+1})\sigma_{i+1} = \pi(t_{i+1})\sigma_{i+1}$ as follows. Assume $i \geq j$ and $\pi(s_{i+1})\sigma_{i+1} \neq \pi(t_{i+1})\sigma_{i+1}$. Then with (18) we get $\pi(s_{i+1})\sigma_{i+1} \rhd \pi(t_{i+1})\sigma_{i+1}$. By (19), this results in $\pi(s_{i+1})\sigma_{i+1} \rhd \circ \to_{\mathcal{R}}^* \pi(s_{i+2})\sigma_{i+2}$ and consequently in $(q_i, q_{i+1}) \in S$ (contradicting 20). Thus $\forall i \geq j.\, q_i = \pi(t_{i+1})\sigma_{i+1}$. However, this contradicts (17). □

## 7.4. Usable Rules

One important technique to prove termination within the DP framework is the reduction pair processor. A *reduction pair* $(\succ, \succsim)$ consists of a well-founded and stable relation $\succ$ in combination with a monotone and stable relation $\succsim$. Further, $\succsim$ has to be compatible with $\succ$, i.e., $\succsim \circ \succ \subseteq \succ$. Note that it is not required that $\succ$ and $\succsim$ are partial orders [104]. Examples for reduction pairs are polynomial orders [51,73,76], matrix orders [32,64], and the lexicographic path order (LPO) [58]. (There are several other classes of reduction pairs. We listed those which have been formalized in IsaFoR.)

The basic version of the reduction pair processor [42, 50] requires that all rules of $\mathcal{R}$ are weakly decreasing w.r.t. $\succsim$ (then $\to_{\mathcal{R}} \subseteq \succsim$) and all pairs of $\mathcal{P}$ are weakly or strictly decreasing. From ($\star$) on page 49 it is easy to see that this implies that every reduction in a $(\mathcal{P}, \mathcal{R})$-chain corresponds to a weak or strict decrease. Thus, the strictly decreasing pairs cannot occur infinitely often and can be removed from $\mathcal{P}$. This technique is already present in IsaFoR and its formalization is described in [104].

**Theorem 7.9.** *Finiteness of $(\mathcal{P} \setminus \succ, \mathcal{R})$ implies finiteness of $(\mathcal{P}, \mathcal{R})$, provided:*

*(a)* $(\succ, \succsim)$ *is a reduction pair,*

*(b)* $\mathcal{P} \subseteq \succ \cup \succsim$,

*(c)* $\mathcal{R} \subseteq \succsim$.

Starting with [105], there have been several papers [41, 42, 50] on how to improve the last requirement. Therefore, $\mathcal{R}$ in (c) is replaced by the *usable rules*.

The main idea of the usable rules is easy to explain: since in chains rewriting is only performed with instances of rhss of $\mathcal{P}$, it should suffice to rewrite with rules of defined symbols that occur in rhss of $\mathcal{P}$. If these usable rules introduce new defined symbols then the rules defining them also have to be considered as usable. Hence, in the remaining DP problem $(\{(10)\text{–}(12)\}, \mathcal{R})$ of Ex. 7.3 only rules (1)–(3) and (4)–(8) are usable. This idea is formally expressed in the following definition.

**Definition 7.10** (Usable Rules)**.** *The function* $\mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(t)$ *defines whether a term $t$ is closed under usable rules $\mathcal{U}$ w.r.t. some TRS $\mathcal{R}$.*

$$\mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(x) = \mathsf{true},$$
$$\mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(f(\vec{t}_n)) = \bigwedge_{1 \leqslant i \leqslant n} \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(t_i) \wedge \bigwedge_{\ell \to r \in \mathcal{R}} (\mathsf{root}(\ell) = f \Longrightarrow \ell \to r \in \mathcal{U}).$$

*A TRS $\mathcal{Q}$ is closed under usable rules whenever all rhss are closed under usable rules, i.e.,*

$$\mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(\mathcal{Q}) = \bigwedge_{\ell\to r\in\mathcal{Q}} \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(r).$$

*A DP problem $(\mathcal{P},\mathcal{R})$ is closed under usable rules iff $\mathcal{P}$ and $\mathcal{U}$ are closed under usable rules.*

$$\mathsf{urClosed}_{\mathcal{U}}(\mathcal{P},\mathcal{R}) = \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(\mathcal{P}) \wedge \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(\mathcal{U}).$$

*The usable rules of DP problem $(\mathcal{P},\mathcal{R})$ are the least set $\mathcal{U}$ satisfying* $\mathsf{urClosed}_{\mathcal{U}}(\mathcal{P},\mathcal{R})$.

Note that there are several other equivalent definitions of usable rules, e.g., one can find definitions of usable rules via so called usable symbols (i.e., root symbols of lhss of usable rules). However, there are also two improvements that yield smaller sets.[2]

The first improvement is to take the reduction pair into account. If certain positions of terms are disregarded then their usable rules do not have to be considered. For example, usually due to the rhs $\mathsf{if2}^\sharp(\mathsf{0} < x, x, z)$ of DP (10) all $<$-rules are usable. However, if we would use a reduction pair based on polynomial orders, where $\mathcal{P}ol(\mathsf{if2}^\sharp(b,x,z)) = z$ then the call to $<$ is ignored by the order. This can be exploited by excluding the $<$-rules from the set of usable rules. This improvement is called *usable rules w.r.t. an argument filter* [42]. Here, argument filters are used to describe which positions in a term are relevant: an argument filter $\pi$ with $\pi(f) = \{i_1,\dots,i_k\}$ indicates that in a term $f(\vec{t}_n)$ only arguments $t_{i_1},\dots,t_{i_k}$ are regarded. This is formalized by the notion of $\pi$-compatibility.

**Definition 7.11** ($\pi$-Compatibility)**.** *A relation $\succsim$ is* $\pi$-compatible *iff for all $n$-ary symbols $f$, all $i$ with $1 \leqslant i \leqslant n$, all $t_1,\dots,t_n$, and all $s$ and $s'$:*

$$i \notin \pi(f) \Longrightarrow f(t_1,\dots,t_{i-1},s,t_{i+1},\dots,t_n) \succsim f(t_1,\dots,t_{i-1},s',t_{i+1},\dots,t_n).$$

To formally define the usable rules w.r.t. an argument filter, a minor modification of Def. 7.10 suffices. Just

$$\text{replace} \qquad \bigwedge_{1\leqslant i\leqslant n} \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(t_i) \qquad \text{by} \qquad \bigwedge_{i\in\pi(f)} \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(t_i).$$

The second improvement to reduce the set of usable rules is performed by taking the structure of terms into account. Observe that in the rhs $\mathsf{if2}^\sharp(\mathsf{0} < x, x, z)$ of DP (10), the first argument of $<$ is $\mathsf{0}$. Hence, only the rules (1) and (2) are possibly applicable, but not the remaining Rule (3). This is not captured by Def. 7.10. As there, all $f$-rules have to be usable whenever the symbol $f$ occurs. On the contrary, in [41], an improved version of usable rules is described which can figure out that Rule (3) is not applicable. To this end, the condition $\mathsf{root}(\ell) = f$ in Def. 7.10 is replaced by a condition based on unification. Demanding $\ell \approx f(\vec{t}_n)$ would be unsound. First $f(\vec{t}_n)$ has to be preprocessed by the function $\mathsf{tcap}$ of [41]. This function keeps only those parts of the input term which cannot be reduced, even if the term is instantiated. All other parts are replaced by fresh variables.

[2] There also is another extension of usable rules, the *generalized usable rules*. It allows a variant of the reduction pair processor where reduction pairs with non-monotone $\succsim$ may be used, cf. [33, Thm. 10]. However, that reduction pair processor is incomparable to both Thm. 7.9 and the upcoming Thm. 7.14. It may be interesting to formalize the soundness of that processor, too, but that would be a different proof.

**Definition 7.12.** *Let $\mathcal{R}$ be a TRS.*[3]

$$\mathsf{tcap}(t) = \begin{cases} f(\mathsf{tcap}(\vec{t}_n)) & \text{if } t = f(\vec{t}_n) \text{ and } \ell \not\approx f(\mathsf{tcap}(\vec{t}_n)) \text{ for all lhss } \ell \text{ of } \mathcal{R}, \\ \text{a fresh variable} & \text{otherwise.} \end{cases}$$

*Here, $\mathsf{tcap}(\vec{t}_n)$ is the list of terms where $\mathsf{tcap}$ is applied to all arguments of $\vec{t}_n$.*

Now the second improvement can be defined formally. Again, it is a minor but crucial modification of Def. 7.10. It suffices to

$$\text{replace } \bigwedge_{\ell\to r\in\mathcal{R}} (\mathsf{root}(\ell) = f \implies \ell \to r \in \mathcal{U}) \text{ by } \bigwedge_{\ell\to r\in\mathcal{R}} (\ell \approx f(\mathsf{tcap}(\vec{t}_n)) \implies \ell \to r \in \mathcal{U}).$$

Hence, incorporating both improvements results in the following definition which now contains a $\pi$ in the superscript to distinguish it from Def. 7.10.

**Definition 7.13** (Improved Closure Under Usable Rules)**.**

$$\begin{aligned} \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(x) &= \mathsf{true}, \\ \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(f(\vec{t}_n)) &= \bigwedge_{i\in\pi(f)} \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(t_i) \wedge \bigwedge_{\ell\to r\in\mathcal{R}} (\ell \approx f(\mathsf{tcap}(\vec{t}_n)) \Longrightarrow \ell \to r \in \mathcal{U}), \\ \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(\mathcal{Q}) &= \bigwedge_{\ell\to r\in\mathcal{Q}} \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(r), \\ \mathsf{urClosed}^{\pi}_{\mathcal{U}}(\mathcal{P},\mathcal{R}) &= \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(\mathcal{P}) \wedge \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(\mathcal{U}). \end{aligned}$$

Note that we do not define *the* improved usable rules of a DP problem $(\mathcal{P},\mathcal{R})$ (as we would, by demanding that $\mathcal{U}$ is the least set satisfying $\mathsf{urClosed}^{\pi}_{\mathcal{U}}(\mathcal{P},\mathcal{R})$). Hence, every set $\mathcal{U}$ that satisfies the closure properties can be used later on. It is easy to see, that the usable rules w.r.t. Def. 7.10 satisfy the closure properties as well as a version of usable rules which only incorporates one of the two improvements. Thus, by this definition we gain the advantage that we can handle several variants of usable rules.

Having defined all necessary notions, we are ready to present the improved reduction pair processor with usable rules where the second part also incorporates [40, Theorem 28] which allows to delete rules by a syntactic criterion.

**Theorem 7.14** (Reduction Pair Processors with Usable Rules)**.** *Let $\mathsf{c}$ be some binary symbol, let $\mathcal{C}_\varepsilon = \{\mathsf{c}(x,y) \to x, \mathsf{c}(x,y) \to y\}$,*[4] *and let $\mathcal{U}$ be some TRS (called the usable rules). For every signature $\mathcal{F}$ and TRS $\mathcal{R}$, let $\mathcal{R}_{\neg\mathcal{F}} = \{\ell \to r \in \mathcal{R} \mid \ell \notin \mathcal{T}(\mathcal{F},\mathcal{V})\}$. Finiteness of $(\mathcal{P} \setminus \succ, \mathcal{R})$ implies finiteness of $(\mathcal{P},\mathcal{R})$, provided:*

*(a) $(\succ, \succsim)$ is a reduction pair,*

[3] Note that in IsaFoR we do not use tcap, but the more efficient and equivalent version etcap which is based on ground-contexts. Moreover, unification is replaced by ground-context matching [104, Section 4.2]. But as the notions of tcap and unification are more common, in the following we stick to these two notions.

[4] The real definition of $\mathcal{C}_\varepsilon$ in IsaFoR is slightly different due to technical reasons. Since there is no restriction on the type of variables, there might be only one variable. To this end $x$ and $y$ are replaced by all possible terms. And as the type of function symbols is also unrestricted there might be no fresh symbol $\mathsf{c}$. However, in IsaFoR the signature is implicit where every arity is allowed. For example, the term $\mathsf{c}(\mathsf{c},\mathsf{c})$ contains one symbol $(\mathsf{c},1)$ and two symbols $(\mathsf{c},0)$ where $(\mathsf{c},1) \neq (\mathsf{c},0)$. In this way, we can always get a fresh symbol $(\mathsf{c},n)$ where $n$ is larger than all arities that occur in $\mathcal{R}$ and we use a constant $(\mathsf{d},0)$ to obtain this high arity. Hence, we define $\mathcal{C}_\varepsilon = \bigcup_{s,t}\{\mathsf{c}(s,t,\mathsf{d},\ldots,\mathsf{d}) \to s, \mathsf{c}(s,t,\mathsf{d},\ldots,\mathsf{d}) \to t\}$.

*(b)* $\mathcal{P} \subseteq \succ \cup \succsim$,

*(c)* $\mathcal{U} \cup \mathcal{C}_\varepsilon \subseteq \succsim$,

*(d)* $\succsim$ *is* $\pi$*-compatible,*

*(e)* $\mathsf{urClosed}^\pi_{\mathcal{U}}(\mathcal{P}, \mathcal{R})$.

*If additionally* $\mathcal{C}_\varepsilon \subseteq \succ$, $\succ$ *is monotone,* $\mathcal{U} \subseteq \mathcal{R}$*, and* $\mathcal{F}$ *is the set of all symbols occurring in rhss of* $\mathcal{P} \cup \mathcal{U}$*, then* $\mathcal{R}$ *can be replaced by* $\mathcal{U}$ *without any strictly decreasing rules and one can remove all rules which contain symbols of* $\mathcal{F}$ *in their left-hand side. Formally, finiteness of* $(\mathcal{P}_{\neg\mathcal{F}} \setminus \succ, \mathcal{U}_{\neg\mathcal{F}} \setminus \succ)$ *implies finiteness of* $(\mathcal{P}, \mathcal{R})$.

Regarding requirement (c), observe that most reduction pairs that are currently used in automated termination tools do satisfy $\mathcal{C}_\varepsilon \subseteq \succsim$. Therefore, requirement (c) of Thm. 7.9 can usually be replaced by $\mathcal{U} \subseteq \succsim$. Requirement (d) is easy to satisfy by choosing an appropriate argument filter $\pi$ which depends on the reduction pair. And if $\mathcal{U}$ is chosen as the usable rules w.r.t. any known definition of usable rules, then condition (e) is satisfied. Thus, for most reduction pairs one has only replaced requirement $\mathcal{R} \subseteq \succsim$ of Thm. 7.9 by the weaker condition $\mathcal{U} \subseteq \succsim$ in Thm. 7.14.

**Example 7.15.** To solve the remaining DP problem $(\{(10)\text{–}(12)\}, \mathcal{R})$ of Ex. 7.5 we use a polynomial order [73] where $\mathcal{P}ol(\mathsf{set}^\sharp(x)) = \mathcal{P}ol(\mathsf{del}(y,x)) = x$, $\mathcal{P}ol(x : y) = \mathcal{P}ol(\mathsf{if}(\ldots, y)) = y + 1$, $\mathcal{P}ol(\mathsf{if2}^\sharp(x,y,z)) = x + z$, $\mathcal{P}ol(x < y) = \mathcal{P}ol(\bot) = \mathcal{P}ol(\top) = 0$, and $\mathcal{P}ol(\mathsf{c}(x,y)) = x + y$. A corresponding compatible argument filter $\pi$ is defined by $\pi(\mathsf{set}^\sharp) = \{1\}$, $\pi(\mathsf{del}) = \pi(:) = \{2\}$, $\pi(\mathsf{if}) = \{5\}$, $\pi(\mathsf{if2}^\sharp) = \{1,3\}$, $\pi(<) = \pi(\bot) = \pi(\top) = \emptyset$, and $\pi(\mathsf{c}) = \{1,2\}$. For this argument filter, the minimal set of usable rules is $\mathcal{U} = \{(1), (2), (4)\text{–}(8)\}$. Note that it would also be accepted if, e.g., Rule (3) would be added.

Then all conditions of Thm. 7.14 are satisfied and one can remove Pair (10) as it is strictly decreasing. The remaining DP problem $(\{(11),(12)\}, \mathcal{R})$ is then easily solved by another application of the reduction pair processor where one chooses $\mathcal{P}ol(\mathsf{set}^\sharp(x)) = 0$, $\mathcal{P}ol(\mathsf{if2}^\sharp(x,y,z)) = 1$, $\mathcal{P}ol(\mathsf{c}(x,y)) = x + y$, and where $\mathcal{U} = \emptyset$.

In theory `UsableRules` we have proven Thm. 7.14. Although a similar proof has already been performed by the authors of [41] and [42]—on paper, not formalized—this proof has never been published in some reviewed article, it is only available in [101]. Moreover, our proof is not just a formalization of the proof in [101], but there are some essential differences which are pointed out in the following.

The standard proof of Thm. 7.14 is by transforming a minimal chain $t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \ldots$ into a chain over *filtered* terms $\pi(t_1)\delta_1 \to^*_{\pi(\mathcal{U})\cup\mathcal{C}_\varepsilon} \pi(s_2)\delta_2 \to_{\pi(\mathcal{P})} \pi(t_2)\delta_2 \ldots$ and then uses the preconditions of the theorem to show that certain pairs of $\pi(\mathcal{P})$ (and therefore also pairs of $\mathcal{P}$) cannot occur infinitely often. Here, one uses a transformation $\mathcal{I}_\pi$ which transforms $\sigma_i$ into $\delta_i$. For the second part of the theorem where $\succ$ is monotone, one requires another transformation $\mathcal{I}$ which does not apply any argument filter. Hence, there are two transformations $\mathcal{I}$ and $\mathcal{I}_\pi$ where for both transformations similar results are shown.

In our formalization we were able to simplify the proofs considerably by not constructing filtered terms. Moreover, we use the same transformation $\mathcal{I}$ for both parts of the theorem.

A problem in the standard proof of the reduction pair processor with usable rules is the implicit assumption that the TRS $\mathcal{R}$ meets the variable condition, i.e., $\mathcal{V}(\ell) \supseteq \mathcal{V}(r)$

and $\ell \notin \mathcal{V}$ for all rules $\ell \to r \in \mathcal{R}$. Although in practice this condition is nearly always satisfied, we have to deal with this assumption, where there are three alternatives. First, one can check that the TRS $\mathcal{R}$ meets the variable condition whenever the theorem is applied on some concrete DP problem $(\mathcal{P}, \mathcal{R})$. This would clearly increase the runtime for certifying a given proof. Second, one can define finiteness of DP problems or soundness of processors in a way that it incorporates the variable condition. However, this will make the development of other processors more complicated which do not care about the variable condition. And third, one can try to prove Thm. 7.14 without assuming the variable condition. This is the alternative we have finally formalized and which does not appear in the literature so far.

For the upcoming formal definition of $\mathcal{I}$ we essentially use a combination of the definitions of [40] and [41] where $x \# xs$ denotes the Isabelle list with head $x$ and tail $xs$.

**Definition 7.16.** *Let $\mathcal{R}$ and $\mathcal{U}$ be two TRSs, let $\mathcal{F}$ be some signature, and let $\mathsf{c}$ be the binary symbol of $\mathcal{C}_\varepsilon$. We define $\mathcal{I}$ as a function from terms to terms as follows:*

$$\begin{aligned}
\mathsf{comb}([t]) &= t,\\
\mathsf{comb}(t \# s \# ts) &= \mathsf{c}(t, \mathsf{comb}(s \# ts)),\\
\mathsf{rewrite}(\mathcal{R}, t) &= \{C[r\sigma]_p \mid \ell \to r \in \mathcal{R}, t = C[u], \mathsf{match}(u, \ell) = \sigma\},\\
\mathcal{I}(x) &= x,\\
\mathcal{I}(f(\vec{t}_n)) &= \begin{cases} f(\vec{t}_n) & \text{if } \neg\mathsf{SN}_{\mathcal{R}}(f(\vec{t}_n)),\\ f(\mathcal{I}(\vec{t}_n)) & \text{if } \mathsf{SN}_{\mathcal{R}}(f(\vec{t}_n)),\ f \in \mathcal{F}, \text{ and } \forall \ell \to r \in \mathcal{R} \setminus \mathcal{U}.\ell \not\approx f(\mathsf{tcap}(\vec{t}_n)),\\ \mathsf{comb}(f(\mathcal{I}(\vec{t}_n)) \# \mathcal{I}(\mathsf{rewrite}(\mathcal{R}, f(\vec{t}_n)))) & \text{otherwise.}\end{cases}
\end{aligned}$$

*$\mathcal{I}$ and $\mathsf{tcap}$ are homeomorphically extended to operate on lists, i.e., $\mathcal{I}(\vec{t}_n) = (\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$.*

The function $\mathsf{comb}$ just combines a non-empty list of terms into one term. It is easy to prove that one can access all terms in the list by rewriting with $\mathcal{C}_\varepsilon$: $\mathsf{comb}([\ldots, t_i, \ldots]) \to^*_{\mathcal{C}_\varepsilon} t_i$.

The function $\mathsf{rewrite}$ computes the list of one-step reducts of a given term by using a sound and complete matching algorithm $\mathsf{match}$. The major difference between $\{s \mid t \to_{\mathcal{R}} s\}$ and $\mathsf{rewrite}(\mathcal{R}, t)$ is that the latter instantiates a rule by the matcher of the lhs and the corresponding redex (as usual), but it never instantiates variables which only occur in the rhs of the rule. For example, if $\mathcal{R} = \{\mathsf{a} \to x\}$ then $\{s \mid \mathsf{a} \to_{\mathcal{R}} s\}$ is the set of all terms, whereas $\mathsf{rewrite}(\mathcal{R}, \mathsf{a}) = [x]$. Hence, $\mathsf{rewrite}$ is sound ($\mathsf{rewrite}(\mathcal{R}, t) \subseteq \{s \mid t \to_{\mathcal{R}} s\}$) but in general not complete. However, under one condition completeness is achieved: whenever $t \to_{\mathcal{R}} s$ by a reduction with a rule that satisfies the variable condition, then $s \in \mathsf{rewrite}(\mathcal{R}, t)$.

The main reason for introducing $\mathsf{rewrite}$ is that without the assumption of the variable condition on $\mathcal{R}$ the set $\{s \mid t \to_{\mathcal{R}} s\}$ may be infinite. Then the definition of $\mathcal{I}$ as in [41]—where $\{\mathcal{I}(s) \mid f(\vec{t}_n) \to_{\mathcal{R}} s\}$ is used instead of $\mathcal{I}(\mathsf{rewrite}(\mathcal{R}, f(\vec{t}_n)))$—does not work in combination with $\mathsf{comb}$, as $\mathsf{comb}$ expects a list (or a *finite* set) as input. Also note that this input must be finite as one finally wants to obtain a single term containing all input terms.

The first case of $\mathcal{I}(f(\vec{t}_n))$ is mainly a technicality. Usually, $\mathcal{I}$ is only defined on terminating terms. To make $\mathcal{I}$ a total function on all terms we inserted the case distinction on $\mathsf{SN}_{\mathcal{R}}(f(\vec{t}_n))$. Termination of $\mathcal{I}$ is then proven using well-founded induction on $\overset{\triangleright}{\to}_{\mathsf{SN}(\mathcal{R})}$ where in this proof the soundness result for $\mathsf{rewrite}$ is crucial.

The transformation $\mathcal{I}$ is constructed in such a way that for every reduction $t \to_{\mathcal{R}} s$ one obtains a weak decrease, provided that the usable rules and $\mathcal{C}_\varepsilon$ are weakly decreasing. Therefore, in the definition of $\mathcal{I}$ there are essentially two cases for a term $f(\vec{t}_n)$. If only usable rules can be used to reduce $f(\vec{t}_n)$ at the root position, then $\mathcal{I}(f(\vec{t}_n))$ is obtained by applying $\mathcal{I}$ on the arguments, resulting in $f(\mathcal{I}(\vec{t}_n))$. The corresponding reduction will then result in a weak decrease as one can also perform the reduction with the usable rules on the transformed term. The condition that only usable rules are applicable is ensured by demanding that no lhs of a non-usable rule in $\mathcal{R} \setminus \mathcal{U}$ can be unified with $f(\mathsf{tcap}(\vec{t}_n))$.

Otherwise, all rules may have been used to rewrite $f(\vec{t}_n)$. Then, in addition to $f(\mathcal{I}(\vec{t}_n))$ we have to store all one-step reducts of $t$. This is done by encoding them in a single term using $\mathsf{comb}$. Now every possible reduct can be accessed using $\mathcal{C}_\varepsilon$. And since $\mathcal{C}_\varepsilon$ is weakly decreasing we obtain a weak decrease. This is proven formally in the upcoming lemma.

**Lemma 7.17** (Properties of $\mathcal{I}$)**.** *Let $(\succsim, \succ)$ be a reduction pair, let $\succsim$ be $\pi$-compatible, let $\mathcal{U} \cup \mathcal{C}_\varepsilon \subseteq \succsim$, let $\mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(\mathcal{U})$, and let the rhss of $\mathcal{U}$ be terms within $\mathcal{T}(\mathcal{F},\mathcal{V})$. Let $\mathsf{SN}_{\mathcal{R}}(t)$, $\mathsf{SN}_{\mathcal{R}}(t\sigma)$, and $\mathsf{SN}_{\mathcal{R}}(f(\vec{t}_n))$.*

*(i) If $\mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(t)$ and $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ then $t\mathcal{I}(\sigma) \succsim^* \mathcal{I}(t\sigma)$.*

*(ii) $\mathcal{I}(t\sigma) \to^*_{\mathcal{C}_\varepsilon} t\mathcal{I}(\sigma)$. And if $t \notin \mathcal{T}(\mathcal{F},\mathcal{V})$ then $\mathcal{I}(t\sigma) \to^+_{\mathcal{C}_\varepsilon} t\mathcal{I}(\sigma)$.*

*(iii) If $f(\vec{t}_n) \to_{\mathcal{R},\epsilon} s$ using a rule $\ell \to r$ and $\mathcal{I}(f(\vec{t}_n)) = f(\mathcal{I}(\vec{t}_n))$ then $\ell \to r \in \mathcal{U}$, $\mathcal{I}(f(\vec{t}_n)) \succsim^* \circ \to_{\mathcal{U}} \circ \succsim^* \mathcal{I}(s)$. And $\mathcal{I}(f(\vec{t}_n)) \to^+_{\mathcal{C}_\varepsilon} \circ \to_{\mathcal{U}} \circ \succsim^* \mathcal{I}(s)$ if $\ell \notin \mathcal{T}(\mathcal{F},\mathcal{V})$.*

*(iv) If $f(\vec{t}_n) \to_{\mathcal{R}} s$ and $\mathcal{I}(f(\vec{t}_n)) = \mathsf{comb}(\ldots)$ then $\mathcal{I}(f(\vec{t}_n)) \to^+_{\mathcal{C}_\varepsilon} \mathcal{I}(s)$.*

*(v) If $t \to_{\mathcal{R}} s$ then $\mathcal{I}(t) \succsim^* \mathcal{I}(s)$. Moreover, $t \to_{\mathcal{U}_{\neg\mathcal{F}}} s$ or $\mathcal{I}(t) \to^+_{\mathcal{C}_\varepsilon} \circ \succsim^* \mathcal{I}(s)$.*

*(vi) If $t \to^*_{\mathcal{R}} s$ then $\mathcal{I}(t) \succsim^* \mathcal{I}(s)$. Moreover, $\mathcal{I}(t) \succsim^* \circ \to_{\mathcal{C}_\varepsilon} \circ \succsim^* \mathcal{I}(s)$ or $t \to^*_{\mathcal{U}_{\neg\mathcal{F}}} s$.*

*Here, $\mathcal{I}$ is homeomorphically extended to substitutions, i.e., $\mathcal{I}(\sigma)(x) = \mathcal{I}(\sigma(x))$.*

In the following proof sketch of Lem. 7.17, all essential points are included, especially those where we deviate from the standard proofs.

*Proof.* The proof of (vi) is a straight-forward induction on the reduction length using (v).

We prove (v), by induction over $t$. First note that $t$ is not a variable. Otherwise, there would be some $x \to r \in \mathcal{R}$, contradicting $\mathsf{SN}_{\mathcal{R}}(t)$. Hence the base case is trivial. In the step-case, we make a case distinction on how $t = f(\vec{t}_n)$ is transformed. The case $\mathcal{I}(t) = \mathsf{comb}(\ldots)$ is solved by (iv). Otherwise, $\mathcal{I}(t) = f(\mathcal{I}(\vec{t}_n))$. For a root reduction we use (iii). Otherwise, $s = f(t_1,\ldots,s_i,\ldots,t_n)$ and $t_i \to_{\mathcal{R}} s_i$. Applying the induction hypothesis is easy, but some additional effort is required to prove $\mathcal{I}(s) = f(\mathcal{I}(t_1),\ldots,\mathcal{I}(s_i),\ldots,\mathcal{I}(t_n))$.

Proving (iv) essentially requires completeness of $\mathsf{rewrite}$. To this end, we first prove that if $f(\vec{t}_n) \to_{\mathcal{R}} s$ then the employed rule $\ell \to r$ must satisfy $\mathcal{V}(\ell) \supseteq \mathcal{V}(r)$, as otherwise $\mathsf{SN}_{\mathcal{R}}(f(\vec{t}_n))$ would not hold. Under this condition, completeness of $\mathsf{rewrite}$ states that $s \in \mathsf{rewrite}(\mathcal{R}, f(\vec{t}_n))$. The remaining proof of (iv) can be done by simple inductions using the definitions of $\mathsf{comb}$ and $\mathcal{C}_\varepsilon$.

To prove (iii), we first show that $\mathcal{I}(f(\vec{t}_n)) = f(\mathcal{I}(\vec{t}_n))$ ensures that the employed rule $\ell \to r$ is usable. The reason is that $f(\vec{t}_n) = \ell\sigma$ implies $f(\mathsf{tcap}(\vec{t}_n)) \approx \ell$ and hence, $\ell \to r \notin \mathcal{R} \setminus \mathcal{U}$ by the definition of $\mathcal{I}$. By the requirement on the rhss of $\mathcal{U}$ we know that $r \in \mathcal{T}(\mathcal{F},\mathcal{V})$. Hence, we can build the following steps using (ii) and (i) in combination

with $\mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(\mathcal{U})$: $\mathcal{I}(f(\vec{t}_n)) = \mathcal{I}(\ell\sigma) \succsim^* \ell\mathcal{I}(\sigma) \to_{\mathcal{U}} r\mathcal{I}(\sigma) \succsim^* \mathcal{I}(r\sigma) = \mathcal{I}(s)$. And if $\ell \notin \mathcal{T}(\mathcal{F},\mathcal{V})$ then we additionally get $\mathcal{I}(\ell\sigma) \to^+_{\mathcal{C}_\varepsilon} \ell\mathcal{I}(\sigma)$ by (ii).

Proving (ii) is a straight-forward induction on $t$.

And finally, for (i), we also use induction on $t$. In the step-case we first prove that $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ in combination with $\mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(f(\vec{t}_n))$ implies $f(\mathcal{I}(\vec{t}_n\sigma)) = \mathcal{I}(f(\vec{t}_n\sigma))$. Then, for all argument positions $i \in \pi(f)$, we apply the induction hypothesis to obtain $t_i\mathcal{I}(\sigma) \succsim^* \mathcal{I}(t_i\sigma)$. Then, by monotonicity of $\succsim$, we obtain $f(\dots, t_i\mathcal{I}(\sigma), \dots) \succsim^* f(\dots, \mathcal{I}(t_i\sigma), \dots)$. For all other positions, $\pi$-compatibility of $\succsim$ provides the same inequality. □

With the help of Lem. 7.17 it is now possible to prove the main result of this section.

*Proof of Thm. 7.14.* Assume that there is a minimal infinite chain where $s_i\sigma_i \to_{\mathcal{P}} t_i\sigma_i \to^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$ and $\mathsf{SN}_{\mathcal{R}}(t_i\sigma_i)$ for all $i$. Let $\mathcal{F}$ be the set of all symbols that occur in rhss of $\mathcal{P} \cup \mathcal{U}$. Then by the conditions of the theorem and by using Lem. 7.17 (i), (vi), and (ii), for all $i$ we conclude

$$s_i\mathcal{I}(\sigma_i) \to_{\mathcal{P}} t_i\mathcal{I}(\sigma_i) \succsim^* \mathcal{I}(t_i\sigma_i) \succsim^* \mathcal{I}(s_{i+1}\sigma_{i+1}) \succsim^* s_{i+1}\mathcal{I}(\sigma_{i+1}). \qquad (\ddagger)$$

By using $\mathcal{P} \subseteq \succ \cup \succsim$ we obtain a strict or weak decrease between every two terms $s_i\mathcal{I}(\sigma_i)$ and $s_{i+1}\mathcal{I}(\sigma_{i+1})$. Thus, by Lem. 7.7, the strictly decreasing pairs can only occur finitely often. This shows that there must be some $n$ such that for all $i$, $s_{i+n} \to t_{i+n} \in \mathcal{P} \setminus \succ$. Hence, there is an infinite minimal $(\mathcal{P} \setminus \succ, \mathcal{R})$-chain.

If additionally, $\mathcal{C}_\varepsilon \subseteq \succ$ and $\succ$ is monotone, we first prove that there is some $n$ with $t_{n+i}\sigma_{n+i} \to^*_{\mathcal{U}_{\neg\mathcal{F}}} s_{n+i+1}\sigma_{n+i+1}$ and $s_{n+i} \in \mathcal{T}(\mathcal{F},\mathcal{V})$ for all $i$. If this would not be the case, then infinitely often $t_i\sigma \to^*_{\mathcal{U}_{\neg\mathcal{F}}} s_{i+1}\sigma_{i+1}$ does not hold or infinitely often $s_i \notin \mathcal{T}(\mathcal{F},\mathcal{V})$. Hence, by Lem. 7.17(vi), for infinitely many $i$, $\mathcal{I}(t_i\sigma_i) \succsim^* \circ \to_{\mathcal{C}_\varepsilon} \circ \succsim^* \mathcal{I}(s_{i+1}\sigma_{i+1})$ or by Lem. 7.17(ii), for infinitely many $i$, $\mathcal{I}(s_i\sigma_i) \to^+_{\mathcal{C}_\varepsilon} s_i\mathcal{I}(\sigma_i)$. As $\succ$ contains $\mathcal{C}_\varepsilon$ and is monotone, we also have $\to_{\mathcal{C}_\varepsilon} \subseteq \succ$, and hence in both cases we obtain infinitely many strict decreases. Using the same reasoning as for (‡), we have infinitely many $i$ with $s_i\mathcal{I}(\sigma_i)$ $(\succ \cup \succsim) \circ \succsim^* \circ \succ \circ \succsim^* s_{i+1}\mathcal{I}(\sigma_{i+1})$ and for the remaining $i$s we can use the previous results showing $s_i\mathcal{I}(\sigma)$ $(\succ \cup \succsim) \circ \succsim^* s_{i+1}\mathcal{I}(\sigma_{i+1})$. Then, using Lem. 7.7 where $N = (\succ \cup \succsim)^*$ and $S = N \circ \succ \circ N$, yields a contradiction.

Hence, for all $i$ we obtain $s_{n+i}\sigma_{n+i} \to_{\mathcal{P}_{\neg\mathcal{F}}} t_{n+1}\sigma_{n+i} \to^*_{\mathcal{U}_{\neg\mathcal{F}}} s_{n+i+1}\sigma_{n+i+1}$. Since $\mathcal{P} \cup \mathcal{U} \subseteq \succ \cup \succsim$ and since both $\succ$ and $\succsim$ are monotone and stable, we conclude (again by Lem. 7.7) that from some point onwards only rules from $(\mathcal{P}_{\neg\mathcal{F}} \cup \mathcal{U}_{\neg\mathcal{F}}) \setminus \succ$ are used. Hence, we have constructed a $(\mathcal{P}_{\neg\mathcal{F}} \setminus \succ, \mathcal{U}_{\neg\mathcal{F}} \setminus \succ)$-chain which is also minimal since $\mathsf{SN}_{\mathcal{R}}(t_i\sigma_i)$ implies $\mathsf{SN}_{\mathcal{U}_{\neg\mathcal{F}}\setminus\succ}(t_i\sigma_i)$ as $\mathcal{U}_{\neg\mathcal{F}} \subseteq \mathcal{R}$ by the requirement $\mathcal{U} \subseteq \mathcal{R}$. □

To obtain an executable function which checks for correct applications of Thm. 7.14 it is only demanded that the input and output DP problem, the reduction pair (without the details of the fresh symbol $\mathsf{c}$), and the usable rules are given. The corresponding interpretation / precedence / ... for $\mathsf{c}$ is then added automatically. Moreover, the argument filter is constructed from the reduction pair. For example, for polynomial interpretations one always defines $\pi$ in a way that $i \in \pi(f)$ iff $x_i$ occurs within $\mathcal{P}ol(f(\vec{x}_n))$. In this way, $\succsim$ is always $\pi$-compatible and $\mathcal{C}_\varepsilon \subseteq \succsim$. Hence, for the automation of Thm. 7.14 where $\succ$ is not monotone, one only has to check $\mathcal{P} \subseteq \succ \cup \succsim$, $\mathcal{U} \subseteq \succsim$, and $\mathsf{urClosed}^{\pi}_{\mathcal{U}}(\mathcal{P},\mathcal{R})$ since the remaining requirements are satisfied by construction.

For the other case, where also rules of $\mathcal{R}$ are deleted, it is additionally checked that $\succ$ is monotone and that $\mathcal{U} \subseteq \mathcal{R}$. To achieve the former for polynomials, it is ensured that the coefficients of all variables are larger than zero and that all remaining parts of the

polynomial are non-negative. For path orders in combination with argument filters, it is ensured that no argument is dropped, i.e., the argument filter may only permute and duplicate arguments.

## 7.5. Experiments

In the end, what we want to have is a workflow which automatically certifies or rejects a given termination proof in CPF-format (a common format for termination proofs that is supported by all certifiers).[5] To this end, we have to parse the proof, detect which processors have been applied on which DP problems, and ensure that the preconditions of every processor are met. We achieved this goal by writing a CPF parser and for each processor an executable function which checks the preconditions. If a processor application cannot be certified, the function rejects, providing an informative error message. As the parser and the check-functions are written in Isabelle, we just invoke Isabelle's code-generator [49] to obtain the executable program CeTA from IsaFoR.

This is in contrast to the other two certifiers CiME/Coccinelle and Rainbow/CoLoR.[6] Both of them provide a parser (as part of CiME and Rainbow) that takes a termination proof and produces a Coq-script as output. The resulting script refers to facts proven in Coccinelle and CoLoR, respectively, which can then be checked by Coq.

For more details on this difference and the architecture of the overall proof-checking function in CeTA we refer to [104].

To measure the impact of our results we used 5 strategies for the two termination tools AProVE [39] and T$_{\text{T}}$T$_2$ [67].

- In the ***basic*** strategy the termination tools only use the dependency graph processor and the reduction pair processor without usable rules. (These are the only techniques that have been described in [104].)
- The ***sc*** strategy is an extension of ***basic*** by the subterm criterion processor.
- Similarly, ***ur*** is like ***basic*** except that usable rules may be used.
- The fourth strategy, ***sc+ur***, is a combination of the previous three.
- Finally, ***full***, is a strategy where all CeTA-certifiable processors may be used. We refer to `http://cl-informatik.uibk.ac.at/software/ceta/versions.php` for the details where all techniques are listed. (Our experiments have been performed using CeTA version 1.10.)

Note that for ***basic***, ***sc***, ***ur***, and ***sc+ur***, we only take linear polynomial interpretations as reduction pairs, whereas in ***full*** also other reduction pairs are used.

For each of the 2132 standard TRSs in the Termination Problem Database (version 7.0.2)[7] and for each strategy, we first ran both termination tools for at most one minute and then tried to certify all successful proofs with CeTA. The experiments were performed on a machine with 8 Dual Core AMD Opteron 885 processors and 64 GB RAM running Linux. An overview of the results is given in the following table where the column labels A, C, and T, refer to AProVE, CeTA, and T$_{\text{T}}$T$_2$, respectively.

[5] `http://cl-informatik.uibk.ac.at/software/cpf/`

[6] Note that for a restricted set of techniques, CoLoR also features code-generation.

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

| | ***basic*** | | ***sc*** | | ***ur*** | | ***sc+ur*** | | ***full*** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | A | C | A | C | A | C | A | C | A | C |
| YES | 453 | 453 | 566 | 566 | 681 | 681 | 684 | 684 | 1242 | 1242 |
| avg. time | | 0.063 | | 0.051 | | 0.064 | | 0.061 | | 0.051 |
| | T | C | T | C | T | C | T | C | T | C |
| YES | 439 | 439 | 553 | 553 | 663 | 663 | 669 | 669 | 1223 | 1223 |
| avg. time | | 0.059 | | 0.048 | | 0.065 | | 0.062 | | 0.074 |

The table rows show successful termination proofs / certificates for termination proofs (YES), and the average time for certifying (in seconds). All details on the experiments are available on CeTA's website.

When comparing ***basic*** with ***sc+ur*** one can observe that adding the subterm criterion and usable rules helps to increase the number of certified termination proofs by over 50% for both termination tools. Moreover, checking the additional application conditions of the new techniques—where $\mathsf{urClosed}^{\pi}_{\mathcal{U}}(\mathcal{P},\mathcal{R})$ is the most expensive one—does not have any measurable impact on the certification time. That checking AProVE's proofs is slightly faster is explained by the fact that TTT2 always produces proofs with polynomial orders over the rationals, even if all coefficients are naturals. And thus, for TTT2's proofs, CeTA always has to perform computations over the rationals.

Our results also helped to win the annual termination competition for certified termination of TRSs in 2009.[8] First several termination tools were run to generate proofs on a random subset of 365 TRSs from the TPDB. For this, the tools where usually configured for a specific certifier in mind by restricting the set of termination techniques correspondingly. Then, all certifiers were run on all proofs. The following table summarizes the results, where the bold entries correspond to those proofs which were constructed specifically for that certifier.

| tool | AProVE | TTT2 | AProVE | CiME | AProVE | total |
|---|---|---|---|---|---|---|
| intended certifier | CeTA | | Coccinelle | | CoLoR | |
| proofs | 259 | 264 | 165 | 56 | 220 | 964 |
| CeTA | **259** | **264** | 94 | 50 | 107 | 774 |
| Coccinelle | 12 | 2 | **104** | **55** | 30 | 203 |
| CoLoR | 67 | 53 | 92 | 9 | **178** | 399 |

We observe that many proofs generated for CeTA cannot be handled by the other certifiers—only between 1 % and 26 % of these proofs have been certified—where one major reason is that the other certifiers do not incorporate usable rules.

Looking at the other direction we see, that even if the termination tool produced proofs for another certifier, CeTA (version 1.09) achieved between 60 % and 90 % of the score of the intended certifier.

In total, only 190 proofs have been rejected by CeTAin the competition. Of these proofs, 65 are supported in the meantime (the competition version did not feature monotone matrix interpretations [32], which are supported by version 1.10), 117 contain unsupported techniques (non-linear polynomial orders and RPO [27]), 6 are obviously buggy (e.g., the subterm criterion is applied with a projection that maps a binary symbol to its third argument), and 2 are faulty (some LPO was wrongly applied and some argument filter

[8] http://termcomp.uibk.ac.at/termcomp/competition/certificationResults.seam?cat=10235&comp=101722

delivers an unsolvable constraint). (At least for one of these proofs we know that this was due to an output bug of the proof producing tool.)

## 7.6. Conclusion

We have presented the first formalization of two important termination techniques within the theorem prover Isabelle/HOL: the subterm criterion and the reduction pair processor with usable rules, where we combined the improvements of [41] and [42]. The integration of these techniques into our termination proof certifier CeTA allowed to certify significantly more termination proofs.

However, there are several termination techniques that have not been certified by now. To change this, in the future we aim at certifying several techniques for innermost termination like narrowing, rewriting, and instantiation [3, 42], or estimations of innermost dependency graphs [3, 41, 50].

# 8. Signature Extensions Preserve Termination – An Alternative Proof via Dependency Pairs

### Publication details

Christian Sternagel and René Thiemann. Signature Extensions Preserve Termination – An Alternative Proof via Dependency Pairs. In *Proceedings of the 19th Annual Conference of the EACSL on Computer Science Logic*, volume 6247 of LNCS, pages 514–528. Springer, 2010.

### Abstract

We give the first mechanized proof of the fact that for showing termination of a term rewrite system, we may restrict to well-formed terms using just the function symbols actually occurring in the rules of the system. Or equivalently, termination of a term rewrite system is preserved under signature extensions. We did not directly formalize the existing proofs for this well-known result, but developed a new and more elegant proof by reusing facts about dependency pairs.

We also investigate signature extensions for termination proofs that use dependency pairs. Here, we were able to develop counterexamples which demonstrate that signature extensions are unsound in general. We further give two conditions where signature extensions are still possible.

## 8.1. Introduction

Our main objective is to formally show that the termination behavior of (first-order) term rewrite systems (TRSs) [5] does not change under signature extensions. This is an important part of a bigger development inside IsaFoR (an **Isa**belle **Fo**rmalization of **R**ewriting) which is used to generate CeTA (a tool for **Ce**rtified **T**ermination **A**nalysis) [104].[1] All our results have been formalized and machine-checked in the interactive proof assistant Isabelle/HOL [84]. In the following, whenever we speak about *formalizing* something, we mean a machine-checked formalization using Isabelle/HOL.

In the literature, termination of $\mathcal{R}$ (denoted by $\mathsf{SN}(\mathcal{R})$), is usually only defined for terms that do exclusively incorporate function symbols from the *signature* $\mathcal{F}$ of $\mathcal{R}$. Often, it is implicitly assumed that this is equivalent to termination for terms over arbitrary extensions $\mathcal{F}' \supseteq \mathcal{F}$. This is legitimate, since it has been shown that termination is modular under certain conditions (see [79, 85] for details) and signature extensions satisfy these conditions. A property $P$ is called *modular*, whenever $P\ \mathcal{R}$ and $P\ \mathcal{S}$, for TRSs $\mathcal{R}$ and $\mathcal{S}$ over disjoint signatures $\mathcal{F}$ and $\mathcal{G}$, implies $P\ (\mathcal{R} \cup \mathcal{S})$. (Note that $P\ x$ is Isabelle/HOL's

[1] `http://cl-informatik.uibk.ac.at/software/ceta`

way of writing a function or predicate $P$ applied to an argument $x$.) Now, to use modularity of termination to achieve $\mathsf{SN}(\mathcal{R})$ over the signature $\mathcal{F}' \supseteq \mathcal{F}$, we choose $\mathcal{S} = \varnothing$ and $\mathcal{G} = \mathcal{F}' - \mathcal{F}$. Then, the above mentioned conditions are trivially satisfied and we obtain $\mathsf{SN}(\mathcal{R}) \Longrightarrow \mathsf{SN}(\mathcal{R} \cup \mathcal{S})$, where the latter system has the same rules as $\mathcal{R}$, but the signature $\mathcal{F}'$.

In this way, the two aforementioned proofs (which both use rather similar proof techniques), can be used to obtain termination preservation under signature extensions. However, the first proof [79] is quite long and complicated even on paper (10 pages, neglecting preliminaries). Concerning the second proof [85]—although short on paper—there are two reasons for not going that way:

(i) This proof would require to formalize concepts that are currently not available in our library but are assumed as preliminaries in the paper proof (which is the only reason that the proof is short). This includes, e.g., multi-hole contexts, and functions like *rank*, *top*, etc. Furthermore, some of those concepts seem bulky to formalize, e.g., multi-hole contexts would require that a context having $n$ holes is always applied to exactly $n$ terms. This cannot be guaranteed on the type level without having dependent types and would lead to side-conditions that had to be added to all proofs using multi-hole contexts.

(ii) We do already have a formalization of many term rewriting related concepts. Thus, it seems only natural to build on top of those available results.

Hence, we take a different road (that may seem as a detour in the beginning). We use $\mathcal{F}(\mathcal{R})$ to denote the signature just containing function symbols that do actually occur in some rule of $\mathcal{R}$. By $\Rightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}}$ we denote the rewrite relation induced by $\mathcal{R}$ just using terms over $\mathcal{F}(\mathcal{R})$ and by $\rightarrow_{\mathcal{R}}$ the same relation but for terms over arbitrary extensions of $\mathcal{F}(\mathcal{R})$ (i.e., $\Rightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}}$ is a restriction of $\rightarrow_{\mathcal{R}}$). Hence, our first main result can be written as

**Theorem 8.1.** $\mathsf{SN}(\Rightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}}) \longleftrightarrow \mathsf{SN}(\rightarrow_{\mathcal{R}})$

In the proof, we concentrate on the direction from left to right, since the converse trivially holds. Before we give our general proof idea, we want to show why "the direct approach" is difficult. By "direct" we mean:

> *Assume that there is an infinite sequence in $\rightarrow_{\mathcal{R}}$ and construct an infinite sequence in $\Rightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}}$ out of it.*

For this purpose we would have to provide a function $f$ such that $f\,s \Rightarrow\!\!\!\!\!\rightarrow_{\mathcal{R}} f\,t$ is implied by $s \rightarrow_{\mathcal{R}} t$ for arbitrary terms $s$ and $t$. This requires that $f$ somehow removes all function symbols that are not in $\mathcal{F}(\mathcal{R})$ from its argument but still preserves any redexes (i.e., subterms where rules are applicable). For example the simple idea to *clean* terms by replacing all subterms $f(\ldots)$ where $f \notin \mathcal{F}(\mathcal{R})$ by the same variable, does not work. The reason is that a given infinite $\rightarrow_{\mathcal{R}}$-derivation might take place strictly below a symbol $f \notin \mathcal{F}(\mathcal{R})$ and then, after turning $f(\ldots)$ into a variable, those reductions can no longer be simulated. To cut a long story short, we stopped at some point to investigate this direction further, since all our approaches became awfully complicated (especially for mechanizing the proof).

Our salvation appeared in the form of *dependency pairs* (DPs). By redirecting the course of our proof into the DP setting [3] and back again, we were able to give a short and (in our opinion) elegant proof of Theorem 8.1, using the simple technique of cleaning.

The reason is that by using DPs we obtain a derivation which contains infinitely many reductions at the root position. And all these root reductions are still possible after cleaning. Note that this also shows that signature extensions are sound for termination problems in the DP setting (Lemma 8.8)—our second main result.

However, after trying to extend our proof to the DP setting with *minimal chains*, we discovered a counterexample demonstrating that signature extensions are unsound for non-left-linear TRSs. A small modification of this counterexample also shows that the technique of root-labeling [95] in the DP setting with minimal chains—which relies on signature extensions—is also only sound for left-linear TRSs. This refutes the corresponding result in [95] which does not demand left-linearity. (As the modularity results of [79, 85] do not consider minimal chains, these results are not affected by our counterexample.)

In total, in this paper we show that signature extensions are possible for termination of TRSs and that they can be used in the DP setting for left-linear TRSs or for non-minimal chains. We also show that the soundness proofs of root-labeling can be repaired by additionally demanding left-linearity.

The structure of our discourse is as follows: In Section 8.2 we recall some necessary definitions of term rewriting (as used in our formalization). Afterwards, in Section 8.3, we give some results on DPs. Two of our main results are given in Section 8.4, where we also formally prove completeness of DPs. Then, in Section 8.5 we show some applications—including root-labeling—and limitations of our results. Here, we also discuss the problem of signature extensions in combination with minimal chains and show that there is no problem in the left-linear case. We finally conclude in Section 8.6.

Since all facts we are using have been machine-checked, we do not give any proofs for results from Sections 8.2 and 8.3 and refer the interested reader to the IsaFoR sources (freely available from its website). Our formalization of Theorem 8.1 can be found under the name `SN_wfrstep_SN_rstep_conv` in the theory `DpFramework`. Also in Section 8.4 we try to skip technical details and give a high-level overview of our proofs.

## 8.2. Preliminaries

In IsaFoR we are concerned with first-order *terms* defined by the data type:

**datatype** $(\alpha, \beta)$ *term* $=$ *Var* $\beta$ | *Fun* $\alpha$ $((\alpha, \beta)$ *term list*$)$

Hence, a term is either a *variable*, or a *function symbol* applied to a list of argument terms. Note that this definition does not incorporate any well-formedness conditions. In particular, there is no signature that terms are restricted to. We identify a function symbol by its representation together with its arity. Hence, the function symbol $f$ in the term *Fun* $f$ [] is different from the function symbol $f$ in the term *Fun* $f$ [*Var* $x$] (the former has arity 0 and the latter arity 1). To increase readability we write terms like the previous two as $f$ (a constant without arguments) and $f(x)$, respectively. A *(rewrite) rule* is a pair of terms and a *TRS* is a set of rules. A TRS is *well-formed* iff all left-hand sides of rules are non-variable terms and for each rule every variable occurring in the right-hand side also occurs in the left-hand side. We write $\mathsf{wf\_trs}\ \mathcal{R}$ to indicate that the TRS $\mathcal{R}$ is well-formed.

**Example 8.2.** The TRS $\{\, \mathsf{add}(\mathsf{0}, y) \to y,\ \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \,\}$, encoding addition on Peano numbers, is well-formed.

The *rewrite relation induced by a TRS* $\mathcal{R}$ is obtained by closing $\mathcal{R}$ under substitutions and contexts, i.e., $\to_{\mathcal{R}}$ is defined inductively by the rules:

$$\frac{(l,\, r) \in \mathcal{R}}{l \to_{\mathcal{R}} r} \qquad \frac{s \to_{\mathcal{R}} t}{s\sigma \to_{\mathcal{R}} t\sigma} \qquad \frac{s \to_{\mathcal{R}} t}{C[s] \to_{\mathcal{R}} C[t]}$$

Here, $t\sigma$ denotes the application of a substitution $\sigma$ to a term $t$ and $C[t]$ denotes substituting the hole in the context $C$ by the term $t$. Whenever $s \to_{\mathcal{R}} t$, we say that $s$ rewrites (in one step) to $t$.

A TRS is *terminating/strongly normalizing* iff the rewrite relation $\to_{\mathcal{R}}$ induced by $\mathcal{R}$ is well-founded—denoted by $\mathsf{SN}(\to_{\mathcal{R}})$. (We sometimes write $\mathsf{SN}(\mathcal{R})$ instead of $\mathsf{SN}(\to_{\mathcal{R}})$ to stress that termination is a property depending merely on $\mathcal{R}$.) Termination of a specific term is written as $\mathsf{SN}_{\mathcal{R}}(t)$, i.e., there is no infinite $\to_{\mathcal{R}}$-derivation starting from $t$.

Using the definition of $\to_{\mathcal{R}}$, termination is formalized as $\mathsf{SN}(\to_{\mathcal{R}}) \equiv \nexists\, \mathbf{t}.\ \forall\, i.\ \mathbf{t}_i \to_{\mathcal{R}} \mathbf{t}_{i+1}$. Here, we use functions from natural numbers to some type $\tau$, to encode infinite sequences over elements of type $\tau$, which are written by $\mathbf{t}$ in contrast to terms $t$. We use subscripts to indicate positions in such infinite sequences, i.e., we write $\mathbf{t}_i$ to denote the $i$-th element in the infinite sequence $\mathbf{t}$.

Remember that by $\mathcal{F}(\mathcal{R})$ we denote the signature of function symbols actually occurring in some rule of $\mathcal{R}$. Using the function

$$\begin{aligned} \mathcal{F}(x) &= \varnothing, \\ \mathcal{F}(f(\vec{ts})) &= \{(f,\, |\vec{ts}|)\} \cup \bigcup \{\mathcal{F}(t) \mid t \in \vec{ts}\}. \end{aligned}$$

$\mathcal{F}(\mathcal{R})$ is obtained by extending $\mathcal{F}(\cdot)$ to TRSs in the obvious way.

**Example 8.3.** The signature of the TRS from Example 8.2 is $\{(\mathsf{add}, 2), (\mathsf{s}, 1), (\mathsf{0}, 0)\}$.

## 8.3. Dependency Pairs

To get hold of the *(recursive) function calls* in a TRS, the so called *dependency pairs* are used [3, 28].

**Definition 8.4.** *The DPs of a TRS* $\mathcal{R}$ *are defined by*

$$\mathsf{DP}(\mathcal{R}) = \{(l^\sharp,\, f^\sharp(\vec{ts})) \mid \exists\, r.\ (l,\, r) \in \mathcal{R} \wedge f \in \mathcal{D}(\mathcal{R}) \wedge r \trianglerighteq f(\vec{ts}) \wedge l \not\rhd f(\vec{ts})\}$$

*where* $(\rhd)$ $\trianglerighteq$ *denotes the (proper) subterm relation on terms and* $\mathcal{D}(\mathcal{R})$ *is the set of defined function symbols in* $\mathcal{R}$.[2] *By* $.^\sharp$ *we denote the operation of marking the root symbol of a term with the special marker* $\sharp$. *In examples we use capitalization and hence write* $F$ *instead of* $f^\sharp$.

**Example 8.5.** Since the TRS of Example 8.2 contains just one "recursive call," we get the single DP $\mathsf{ADD}(\mathsf{s}(x), y) \to \mathsf{ADD}(x, y)$.

Note how DPs get rid of context information. This is exactly what makes them so useful in our proof.

Having DPs, we can use an alternative characterization of nontermination using DP problems and chains. A *DP problem* $(\mathcal{P}, \mathcal{R})$ just consists of two TRSs $\mathcal{P}$ and $\mathcal{R}$. Then a $(\mathcal{P}, \mathcal{R})$*-chain* is an infinite sequence of the following shape:

[2] A function symbol is defined in a TRS, if it occurs as the root of a left-hand side.

$$\forall\, i.\ (\mathbf{s}_i,\, \mathbf{t}_i) \in \mathcal{P} \wedge \mathbf{t}_i\sigma_i \to_{\mathcal{R}}^{*} \mathbf{s}_{i+1}\sigma_{i+1}.$$

We use the abbreviation ichain $(\mathcal{P},\, \mathcal{R})$ $\mathbf{s}$ $\mathbf{t}$ $\sigma$ for such a sequence. The soundness result of DPs then states that a (well-formed) TRS $\mathcal{R}$ is terminating if there is no infinite $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$-chain where the formalization was described in [104].

**Lemma 8.6.** wf_trs $\mathcal{R} \Longrightarrow \neg$ SN$(\to_{\mathcal{R}}) \Longrightarrow \exists \mathbf{s}\ \mathbf{t}\ \sigma.$ ichain $(\mathsf{DP}(\mathcal{R}),\, \mathcal{R})$ $\mathbf{s}$ $\mathbf{t}$ $\sigma$

Sometimes, we are interested in *minimal* $(\mathcal{P}, \mathcal{R})$-chains. The only difference between min_ichain $(\mathcal{P},\, \mathcal{R})$ $\mathbf{s}$ $\mathbf{t}$ $\sigma$ and ichain $(\mathcal{P},\, \mathcal{R})$ $\mathbf{s}$ $\mathbf{t}$ $\sigma$, is the additional requirement in minimal chains that $\mathsf{SN}_{\mathcal{R}}(\mathbf{t}_i\sigma_i)$ for all $i$.

## 8.4. Main Results

Since our term data type does not take care of building only terms corresponding to a specific signature, by default any rewrite relation $\to_{\mathcal{R}}$ in our formalization is defined over terms containing arbitrary function symbols. Our first goal is to show that once we have shown termination for terms using only function symbols from $\mathcal{F}(\mathcal{R})$, this implies that $\to_{\mathcal{R}}$ does terminate for arbitrary terms. Before doing that, we need means to identify well-formed terms. To this end we use the inductively defined set $\mathcal{T}(\mathcal{F})$, containing all terms that are well-formed with respect to the signature $\mathcal{F}$.

**Definition 8.7** (Well-Formed Terms)**.**

$$\frac{}{x \in \mathcal{T}(\mathcal{F})} \qquad \frac{(f,\, |\vec{ts}|) \in \mathcal{F} \quad \forall\, t{\in}\vec{ts}.\ t \in \mathcal{T}(\mathcal{F})}{f(\vec{ts}) \in \mathcal{T}(\mathcal{F})}$$

Using this definition we can define the well-formed rewrite relation induced by a TRS $\mathcal{R}$:

$$\twoheadrightarrow_{\mathcal{R}} \equiv \{(s,\, t) \mid s \to_{\mathcal{R}} t \wedge s \in \mathcal{T}(\mathcal{F}(\mathcal{R})) \wedge t \in \mathcal{T}(\mathcal{F}(\mathcal{R}))\}.$$

Further, let $\mathcal{C}(\mathcal{F})$ denote the set of well-formed contexts with respect to the signature $\mathcal{F}$. What we want to show is $\mathsf{SN}(\twoheadrightarrow_{\mathcal{R}}) \Longrightarrow \mathsf{SN}(\to_{\mathcal{R}})$. For the proof we need a way to remove unwanted function symbols from terms. This is the purpose of the following *cleaning* function:

$$\begin{aligned} [\![y]\!]_{\mathcal{F}} &= y \\ [\![f(\vec{ts})]\!]_{\mathcal{F}} &= \mathit{if}\ (f,\, |\vec{ts}|) \in \mathcal{F}\ \mathit{then}\ f(\mathit{map}\ [\![\cdot]\!]_{\mathcal{F}}\ \vec{ts})\ \mathit{else}\ z \end{aligned}$$

where $z$ denotes an arbitrary but fixed variable. Intuitively, every subterm of a term whose root is not in the given signature, is replaced by $z$. Having this, the proof of $\mathsf{SN}(\twoheadrightarrow_{\mathcal{R}}) \Longrightarrow \mathsf{SN}(\to_{\mathcal{R}})$ (actually we prove its contrapositive) is done in three stages:

(i) First, we assume $\neg\, \mathsf{SN}(\to_{\mathcal{R}})$. Then by the soundness of DPs (Lemma 8.6) we obtain an infinite $(\mathsf{DP}(\mathcal{R}),\, \mathcal{R})$-chain.

(ii) Next, we show that every infinite chain can be transformed into an infinite *clean* chain.

(iii) And finally, we show completeness of the DP-transformation for well-formed terms, i.e., that an infinite clean $(\mathsf{DP}(\mathcal{R}),\, \mathcal{R})$-chain can be transformed into an infinite derivation w.r.t. $\twoheadrightarrow_{\mathcal{R}}$. Hence, $\neg\, \mathsf{SN}(\twoheadrightarrow_{\mathcal{R}})$, concluding the proof.

In total we get $\mathsf{wf\_trs}\ \mathcal{R} \Longrightarrow \mathsf{SN}(\Rightarrow\!\!\!\!\!\!\cdot_{\mathcal{R}}) \Longrightarrow \mathsf{SN}(\to_{\mathcal{R}})$ and since every non-well-formed TRS is nonterminating, we finally have a proof of Theorem 8.1. Note that the second step also shows the second main result: signature extensions are valid when performing termination proofs using DPs (without minimality).

It remains to prove the following two lemmas where we use the abbreviations $\mathcal{F}(\mathcal{P}, \mathcal{R}) \equiv \mathcal{F}(\mathcal{P}) \cup \mathcal{F}(\mathcal{R})$ and $\sharp(\mathcal{R}) \equiv \mathcal{F}(\mathcal{R}) \cup \mathcal{F}^{\sharp}(\mathcal{R})$ with $\mathcal{F}^{\sharp}(\mathcal{R}) \equiv \{(f^{\sharp}, n) \mid (f, n) \in \mathcal{F}(\mathcal{R})\}$, and the cleaning function is extended to sequences of substitutions in the obvious way.

**Lemma 8.8** (Signature Restrictions for Chains)**.**
$$\mathcal{F}(\mathcal{P}, \mathcal{R}) \subseteq \mathcal{F} \Longrightarrow \mathsf{ichain}\ (\mathcal{P}, \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ \sigma \Longrightarrow \mathsf{ichain}\ (\mathcal{P}, \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ [\![\sigma]\!]_{\mathcal{F}}$$

**Lemma 8.9** (Completeness of DPs for $\Rightarrow\!\!\!\!\!\!\cdot_{\mathcal{R}}$)**.**
$$\mathsf{ichain}\ (\mathsf{DP}(\mathcal{R}), \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ [\![\sigma]\!]_{\sharp(\mathcal{R})} \Longrightarrow \neg\ \mathsf{SN}(\Rightarrow\!\!\!\!\!\!\cdot_{\mathcal{R}})$$

Note that by applying first Lemma 8.8 and then Lemma 8.9, we also obtain the classical completeness result of DPs.

**Lemma 8.10** (Completeness of DPs)**.** $\mathsf{ichain}\ (\mathsf{DP}(\mathcal{R}), \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ \sigma \Longrightarrow \neg\ \mathsf{SN}(\to_{\mathcal{R}})$

*Proof.* Obviously, we have $\mathcal{F}(\mathsf{DP}(\mathcal{R}), \mathcal{R}) \subseteq \sharp(\mathcal{R})$. Together with the assumption $\mathsf{ichain}\ (\mathsf{DP}(\mathcal{R}), \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ \sigma$, this yields $\mathsf{ichain}\ (\mathsf{DP}(\mathcal{R}), \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ [\![\sigma]\!]_{\sharp(\mathcal{R})}$, using Lemma 8.8. Then, from Lemma 8.9, we obtain $\neg\ \mathsf{SN}(\Rightarrow\!\!\!\!\!\!\cdot_{\mathcal{R}})$ and thus $\neg\ \mathsf{SN}(\to_{\mathcal{R}})$. □

*of Lemma 8.8.* From the assumptions of Lemma 8.8 we obtain

$$\forall i.\, \mathbf{s}_i \in \mathcal{T}(\mathcal{F}) \wedge \mathbf{t}_i \in \mathcal{T}(\mathcal{F}), \tag{1}$$
$$\forall i.\, \mathbf{t}_i\sigma_i \to_{\mathcal{R}}^{*} \mathbf{s}_{i+1}\sigma_{i+1}, \tag{2}$$
$$\forall i.\, (\mathbf{s}_i, \mathbf{t}_i) \in \mathcal{P}. \tag{3}$$

Further note that whenever there is an $\mathcal{R}$-step from $s$ to $t$, then either this step is also possible in the cleaned versions of $s$ and $t$, or the cleaned versions are equal, i.e.,

$$s \to_{\mathcal{R}} t \Longrightarrow [\![s]\!]_{\mathcal{F}(\mathcal{R})} \to_{\mathcal{R}}^{=} [\![t]\!]_{\mathcal{F}(\mathcal{R})}.$$

From this and (2) we may conclude

$$\forall i.\, [\![\mathbf{t}_i\sigma_i]\!]_{\mathcal{F}} \to_{\mathcal{R}}^{*} [\![\mathbf{s}_{i+1}\sigma_{i+1}]\!]_{\mathcal{F}}$$

by induction over the length of the rewrite sequence (remember that $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}$). Using (1) we may push the applications of the clean function inside, resulting in

$$\forall i.\, [\![\mathbf{t}_i]\!]_{\mathcal{F}}[\![\sigma_i]\!]_{\mathcal{F}} \to_{\mathcal{R}}^{*} [\![\mathbf{s}_{i+1}]\!]_{\mathcal{F}}[\![\sigma_{i+1}]\!]_{\mathcal{F}}.$$

Together with (3) we obtain the desired clean infinite chain as (1) shows $[\![\mathbf{s}_i]\!]_{\mathcal{F}} = \mathbf{s}_i$ and $[\![\mathbf{t}_i]\!]_{\mathcal{F}} = \mathbf{t}_i$ for all $i$. □

*of Lemma 8.9.* Again, we show the lemma in its contrapositive form. Thus, we assume $\mathsf{SN}(\Rightarrow\!\!\!\!\!\!\cdot_{\mathcal{R}})$. Now, let $\mathcal{F}$ denote the signature of $\mathcal{R}$ and $\mathsf{u}(\cdot)$ the operation of 'unsharping,' i.e., removing $\sharp$s from terms:

$$\mathsf{u}(t) = \begin{cases} f(\mathit{map}\ \mathsf{u}(\cdot)\ \vec{ts}) & \text{if } t = f^{\sharp}(\vec{ts}) \text{ or } t = f(\vec{ts}), \text{ and} \\ t & \text{otherwise.} \end{cases}$$

The extension of $\mathsf{u}$ to substitutions is defined as $\mathsf{u}(\sigma)(x) = \mathsf{u}(\sigma(x))$. For the sake of a contradiction, assume that there is an infinite $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$-chain over $\mathbf{s}$, $\mathbf{t}$, and $[\![\sigma]\!]_{\sharp(\mathcal{R})}$. Since cleaning does not affect $\mathbf{s}$ and $\mathbf{t}$, this implies an infinite $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$-chain over $[\![\mathbf{s}]\!]_{\sharp(\mathcal{R})}$, $[\![\mathbf{t}]\!]_{\sharp(\mathcal{R})}$, and $[\![\sigma]\!]_{\sharp(\mathcal{R})}$, i.e.,

$$\forall i.\, ([\![\mathbf{s}_i]\!]_{\sharp(\mathcal{R})}, [\![\mathbf{t}_i]\!]_{\sharp(\mathcal{R})}) \in \mathsf{DP}(\mathcal{R}) \tag{4}$$

$$\forall i.\, [\![\mathbf{t}_i]\!]_{\sharp(\mathcal{R})}[\![\sigma_i]\!]_{\sharp(\mathcal{R})} \to_{\mathcal{R}}^{*} [\![\mathbf{s}_{i+1}]\!]_{\sharp(\mathcal{R})}[\![\sigma_{i+1}]\!]_{\sharp(\mathcal{R})} \tag{5}$$

Then from (4) we obtain

$$\forall i.\, \exists C.\, C \in \mathcal{C}(\mathcal{F}) \wedge [\![\mathsf{u}(\mathbf{s}_i)]\!]_{\mathcal{F}} \to_{\mathcal{R}} C[[\![\mathsf{u}(\mathbf{t}_i)]\!]_{\mathcal{F}}]$$

by construction of $\mathsf{DP}(\mathcal{R})$. Using the *Axiom of Choice* we hence obtain a sequence of contexts $\mathbf{C}$, such that $\mathbf{C}_i$ is the context employed in the $i$-th step of (4), i.e.,

$$\forall i.\, \mathbf{C}_i \in \mathcal{C}(\mathcal{F}) \wedge [\![\mathsf{u}(\mathbf{s}_i)]\!]_{\mathcal{F}} \to_{\mathcal{R}} \mathbf{C}_i[[\![\mathsf{u}(\mathbf{t}_i)]\!]_{\mathcal{F}}] \tag{6}$$

Let $\mathbf{D}$ denote the following sequence:

$$\mathbf{D}_i = \begin{cases} \Box & \text{if } i = 0, \\ \mathbf{D}_{i-1} \circ (\mathbf{C}_i[\![\mathsf{u}(\sigma_i)]\!]_{\mathcal{F}}) & \text{otherwise.} \end{cases}$$

Where $\circ$ denotes the composition of contexts, i.e., the right context replaces the hole of the left one. This function gives for the $i$-th $\mathsf{DP}$-step in the infinite chain, all the contexts that have been lost due to using $\mathsf{DP}(\mathcal{R})$ instead of $\mathcal{R}$ and additionally applies all the necessary substitutions. For the sake of brevity we define:

$$\mathbf{s}'_i = \mathbf{D}_i[[\![\mathsf{u}(\mathbf{s}_i)]\!]_{\mathcal{F}}[\![\mathsf{u}(\sigma_i)]\!]_{\mathcal{F}}]$$
$$\mathbf{t}'_i = \mathbf{D}_{i+1}[[\![\mathsf{u}(\mathbf{t}_i)]\!]_{\mathcal{F}}[\![\mathsf{u}(\sigma_i)]\!]_{\mathcal{F}}]$$

Then by (6) we have $\mathbf{s}'_i \to_{\mathcal{R}} \mathbf{t}'_i$, since rewriting is closed under contexts and substitutions. From (5) we conclude $[\![\mathsf{u}(\mathbf{t}_i)]\!]_{\mathcal{F}}[\![\mathsf{u}(\sigma_i)]\!]_{\mathcal{F}} \to_{\mathcal{R}}^{*} [\![\mathsf{u}(\mathbf{s}_{i+1})]\!]_{\mathcal{F}}[\![\mathsf{u}(\sigma_{i+1})]\!]_{\mathcal{F}}$, since removing $\sharp$s does not destroy any redexes of $\mathcal{R}$. By wrapping this derivation in the context $\mathbf{D}_{i+1}$ we obtain $\mathbf{t}'_i \to_{\mathcal{R}}^{*} \mathbf{s}'_{i+1}$. Combining this with $\mathbf{s}'_i \to_{\mathcal{R}} \mathbf{t}'_i$ yields

$$\mathbf{s}'_i \to_{\mathcal{R}}^{+} \mathbf{s}'_{i+1}$$

From our assumption $\mathsf{SN}(\Rightarrow\!\!\!\!\!\!\cdot\;\;_{\mathcal{R}})$ we conclude that $\mathcal{R}$ is well-formed. Moreover, it is apparent from the definitions of $[\![\cdot]\!]_{\mathcal{F}}$ and $\mathbf{D}_i$, together with (6) that all the $\mathbf{s}'_i$s are well-formed, i.e., $\mathbf{s}'_i \in \mathcal{T}(\mathcal{F})$. Together with the well-formedness of $\mathcal{R}$ one can prove that also all intermediate terms in all derivations $\mathbf{s}'_i \to_{\mathcal{R}}^{+} \mathbf{s}'_{i+1}$ are in $\mathcal{T}(\mathcal{F})$. Thus we have an infinite $\Rightarrow\!\!\!\!\!\!\cdot\;\;_{\mathcal{R}}$-sequence which contradicts our assumption $\mathsf{SN}(\Rightarrow\!\!\!\!\!\!\cdot\;\;_{\mathcal{R}})$. □

## 8.5. Applications

In most termination tools, termination techniques are freely combined within a complex termination proof. For example, it is a standard procedure to first remove some rules from $\mathcal{R}$, resulting in $\mathcal{R}'$, and then prove $\mathsf{SN}(\mathcal{R}')$ without caring about any changes in the signature. I.e., proving termination of $\mathsf{SN}(\mathcal{R}')$ is performed as if the signature were

$\mathcal{F}(\mathcal{R}')$ and not the original signature $\mathcal{F}(\mathcal{R})$. The soundness of this approach relies upon Theorem 8.1.

At first view, Theorem 8.1 might not seem important, as there are several termination techniques which do not rely upon the signature. For example, when using polynomial interpretations, it always suffices to give the interpretations for the function symbols occurring in the TRS, no matter if the signature contains other symbols. The reason is that the interpretation of any other symbol has no impact when computing the polynomials for the left-hand sides and right-hand sides of the rules. Similar situations occur for other reduction orders and other termination techniques, like semantic labeling [110].

However, we are aware of at least two termination techniques where the signature is essential.

**String Reversal.** If we restrict terms in rewriting to employ only unary function symbols, we are in the setting of *string rewriting*. For notational convenience we write $\mathsf{abc}$ instead of $\mathsf{a}(\mathsf{b}(\mathsf{c}(x)))$, where the variable $x$ is implicit. There are several termination techniques that work only/better for strings. One of them is string reversal. This technique uses the fact that a string rewrite system (SRS) $\mathcal{S}$ is terminating iff $\mathsf{rev}(\mathcal{S})$ is terminating. Here, $\mathsf{rev}(\mathcal{S})$ denotes the mapping of the function

$$\mathsf{rev}(t) = \begin{cases} \mathsf{rev}(t')a & \text{if } t = at', \\ t & \text{otherwise,} \end{cases}$$

over all left-hand sides and right-hand sides of $\mathcal{S}$. In practice, this often helps to automatically find a termination proof.

**Example 8.11.** Consider the following TRS

$$\begin{aligned} \mathsf{a}(\mathsf{b}(\mathsf{b}(x))) &\to \mathsf{a}(\mathsf{b}(\mathsf{a}(\mathsf{a}(\mathsf{a}(x))))) \\ \mathsf{f}(x, y) &\to x \end{aligned}$$

which is not an SRS. One can remove the second rule by a polynomial order which maps $\mathsf{a}(x)$ and $\mathsf{b}(x)$ to $x$, and $\mathsf{f}(x, y)$ to $x + y + 1$. Then the SRS

$$\mathsf{abb} \to \mathsf{abaaaa}$$

remains, where the signature still contains the binary symbol $\mathsf{f}$. As string reversal is only defined for unary symbols, the presence of $\mathsf{f}$ forbids the application of string reversal. But after applying the signature restriction to $\mathsf{a}$ and $\mathsf{b}$ we are allowed to forget about $\mathsf{f}$ and apply string reversal to obtain the following SRS:

$$\mathsf{bba} \to \mathsf{aaaaba}$$

Note that in this reversed SRS there are no dependency pairs as $\mathsf{ba}$ is a proper subterm of $\mathsf{bba}$. Therefore, termination is now trivially proven.

**Root-Labeling.** Root-labeling [95] is a special version of semantic labeling [110]. We start with a short description of semantic labeling. We interpret a TRS $\mathcal{R}$ by an $\mathcal{F}$-algebra $\mathcal{M} = (M, \{f_{\mathcal{M}}\}_{f \in \mathcal{F}})$. That is, we interpret every function symbol $f$ of arity $n$, by a function $f_{\mathcal{M}} \colon M^n \to M$, over the carrier $M$. Then, the interpretation of a term with respect to a given variable assignment $\mu$, is given by: $[\mu]_{\mathcal{M}}(x) = \mu(x)$ and

$[\mu]_{\mathcal{M}}(f(t_1,\ldots,t_n)) = f_{\mathcal{M}}([\mu]_{\mathcal{M}}(t_1),\ldots,[\mu]_{\mathcal{M}}(t_n))$. We say that $\mathcal{M}$ is a *model* of $\mathcal{R}$ iff for all assignments $\mu$ and all rules $l \to r \in \mathcal{R}$, we have $[\mu]_{\mathcal{M}}(l) = [\mu]_{\mathcal{M}}(r)$.

Now, we can label the function symbols of $\mathcal{R}$ according to the interpretation of their arguments. For every $n$-ary function symbol $f$, we choose a set of labels $L_f \neq \varnothing$ in combination with a mapping $\pi_f \colon M^n \to L_f$. The labeling is extended to terms as follows: $\mathsf{lab}_\mu(x) = x$ and $\mathsf{lab}_\mu(f(t_1,\ldots,t_n)) = f_m(\mathsf{lab}_\mu(t_1),\ldots,\mathsf{lab}_\mu(t_n))$ with $m = \pi_f([\mu]_{\mathcal{M}}(t_1),\ldots,[\mu]_{\mathcal{M}}(t_n))$. Then, the labeled TRS $\mathcal{R}_{\mathsf{lab}}$ consists of the rules $\mathsf{lab}_\mu(l) \to \mathsf{lab}_\mu(r)$ for all $l \to r \in \mathcal{R}$ and assignments $\mu$. Zantema [110] has shown that for every model of $\mathcal{R}$, the TRS $\mathcal{R}$ is terminating iff the TRS $\mathcal{R}_{\mathsf{lab}}$ is terminating.

The difficult part of applying semantic labeling for proving termination, is to find a proper model. This is solved in the special case of root-labeling by fixing the interpretation. Every function symbol is interpreted by itself (i.e., $f_{\mathcal{M}}(x_1,\ldots,x_n) = f$) and the labeling is fixed to tuples of function symbols (i.e., $\pi_f(x_1,\ldots,x_n) = (x_1,\ldots,x_n)$). Now, to satisfy the necessary *model condition*, we close the rules of a TRS under the so called *flat contexts* before labeling. This makes sure that for every resulting rule $l \to r$, the root symbol of $l$ is the same as the root symbol of $r$ and thus, $[\mu]_{\mathcal{M}}(l) = [\mu]_{\mathcal{M}}(r)$ for every assignment $\mu$. Here, the flat contexts are determined solely by the signature. Again, Theorem 8.1 shows that one can reduce the possibly infinite set of flat contexts (if the signature $\mathcal{F}$ is infinite) to a finite set of flat contexts (if $\mathcal{F}(\mathcal{R})$ is finite).

**Example 8.12.** Consider the TRS $\{\mathsf{a}(\mathsf{b}(x)) \to \mathsf{b}(\mathsf{a}(\mathsf{a}(x)))\}$. This yields the set of flat contexts $\{\mathsf{a}(\Box), \mathsf{b}(\Box)\}$. After closing the TRS under flat contexts we obtain the two rules $\{\mathsf{a}(\mathsf{a}(\mathsf{b}(x))) \to \mathsf{a}(\mathsf{b}(\mathsf{a}(\mathsf{a}(x)))), \mathsf{b}(\mathsf{a}(\mathsf{b}(x))) \to \mathsf{b}(\mathsf{b}(\mathsf{a}(\mathsf{a}(x))))\}$.[3] Now, root-labeling results in the following labeled TRS:

$$\begin{aligned}
\mathsf{a}_{\mathsf{a}}(\mathsf{a}_{\mathsf{b}}(\mathsf{b}_{\mathsf{a}}(x))) &\to \mathsf{a}_{\mathsf{b}}(\mathsf{b}_{\mathsf{a}}(\mathsf{a}_{\mathsf{a}}(\mathsf{a}_{\mathsf{a}}(x))))\\
\mathsf{a}_{\mathsf{a}}(\mathsf{a}_{\mathsf{b}}(\mathsf{b}_{\mathsf{b}}(x))) &\to \mathsf{a}_{\mathsf{b}}(\mathsf{b}_{\mathsf{a}}(\mathsf{a}_{\mathsf{a}}(\mathsf{a}_{\mathsf{b}}(x))))\\
\mathsf{b}_{\mathsf{a}}(\mathsf{a}_{\mathsf{b}}(\mathsf{b}_{\mathsf{a}}(x))) &\to \mathsf{b}_{\mathsf{b}}(\mathsf{b}_{\mathsf{a}}(\mathsf{a}_{\mathsf{a}}(\mathsf{a}_{\mathsf{a}}(x))))\\
\mathsf{b}_{\mathsf{a}}(\mathsf{a}_{\mathsf{b}}(\mathsf{b}_{\mathsf{b}}(x))) &\to \mathsf{b}_{\mathsf{b}}(\mathsf{b}_{\mathsf{a}}(\mathsf{a}_{\mathsf{a}}(\mathsf{a}_{\mathsf{b}}(x))))
\end{aligned}$$

The advantage of root-labeling or semantic labeling is that afterwards one can distinguish different occurrences of symbols as they might have different labels. For example, the last but one symbols of the left- and right-hand sides are $\mathsf{a}_{\mathsf{b}}$ and $\mathsf{a}_{\mathsf{a}}$, respectively, whereas in the original TRS these symbols were just $\mathsf{a}$ and could not be distinguished. That such a distinction can be helpful is demonstrated in several examples [95, 110].

Note that root-labeling is also applied in the DP setting. Here, Lemma 8.8 can be used to show that it suffices to build the flat contexts w.r.t. the signature of the given DP problem.

However, many termination tools base their termination analysis on DPs where always minimal chains are considered. The reason to work with minimal chains is that many powerful termination techniques are only sound when regarding minimal chains [42, 101].

Unfortunately, when trying to lift Lemma 8.8 to minimal chains, we figured out that this is impossible. It is easy to show that cleaning terms might introduce nontermination if non-left-linear rules are present. For example if $\mathsf{a}$ and $\mathsf{b}$ are not in the signature and

[3] If the signature would be larger, e.g., if there would be an additional ternary symbol $\mathsf{c}$, then the flat contexts would include $\{\mathsf{c}(\Box, x_2, x_3), \mathsf{c}(x_1, \Box, x_3), \mathsf{c}(x_1, x_2, \Box)\}$ and for each of these contexts one would get another rule. Hence, the signature restriction is essential to get few flat contexts and therefore small systems.

there is a rule $\mathsf{f}(x,x) \to \mathsf{f}(x,x)$, then this rule cannot be applied on $\mathsf{f}(\mathsf{a},\mathsf{b})$. However, it is applicable on the cleaned term $\mathsf{f}(z,z)$.

Moreover, we even found a counter-example where there is an infinite minimal $(\mathcal{P},\mathcal{R})$-chain, but no infinite minimal $(\mathcal{P},\mathcal{R})$-chain if only terms over $\mathcal{F}(\mathcal{P}\cup\mathcal{R})$ may be used. Hence, there cannot be any function that transforms an infinite minimal $(\mathcal{P},\mathcal{R})$-chain over an arbitrary signature into an infinite minimal chain which only contains terms over $\mathcal{F}(\mathcal{P}\cup\mathcal{R})$. In other words, Lemma 8.8 does not hold if one would replace infinite chains by minimal infinite chains.

**Example 8.13** (Restricting the signature to $\mathcal{F}(\mathcal{P}\cup\mathcal{R})$ is unsound for minimal chains)**.** To present a counter-example we give a "termination proof" for a nonterminating TRS where the only unsound step is the signature restriction to the signature of the current DP problem. Here, we make use of the DP-framework [42] in which one proves termination by simplifying the initial DP problems by termination techniques until one obtains a DP problem that does not admit an infinite minimal chain. For soundness it is only required that whenever $(\mathcal{P},\mathcal{R})$ is simplified to $(\mathcal{P}',\mathcal{R}')$ then an infinite minimal $(\mathcal{P},\mathcal{R})$-chain must imply the existence of an infinite minimal $(\mathcal{P}',\mathcal{R}')$-chain.

So, let $\mathcal{R}$ be the following nonterminating TRS.

$$\begin{aligned}
\mathsf{g}(\mathsf{f}(x,y,x',z,z,u)) &\to \mathsf{g}(\mathsf{f}(x,y,x,x,y,\mathsf{h}(y,x')))\\
\mathsf{a} &\to \mathsf{b}\\
\mathsf{a} &\to \mathsf{c}\\
\mathsf{h}(x,x) &\to \mathsf{h}(x,x)\\
\mathsf{h}(\mathsf{a},x) &\to \mathsf{h}(x,x)\\
\mathsf{h}(\mathsf{b},x) &\to \mathsf{h}(x,x)\\
\mathsf{h}(\mathsf{c},x) &\to \mathsf{h}(x,x)\\
\mathsf{h}(\mathsf{h}(x_1,x_2),x) &\to \mathsf{h}(x,x)\\
\mathsf{h}(\mathsf{f}(x_1,\ldots,x_6),x) &\to \mathsf{h}(x,x)
\end{aligned}$$

The initial DP problem $(\mathsf{DP}(\mathcal{R}),\mathcal{R})$ can be simplified to $(\mathcal{P},\mathcal{R})$ where $\mathcal{P} = \{\mathsf{G}(\mathsf{f}(x,y,x',z,z,u)) \to \mathsf{G}(\mathsf{f}(x,y,x,x,y,\mathsf{h}(y,x')))\}$. The reason is that there is a minimal infinite $(\mathcal{P},\mathcal{R})$-chain: choose every $\mathbf{s}_i$ and $\mathbf{t}_i$ to be the left-hand side and right-hand side of the only rule in $\mathcal{P}$, respectively. Further, choose $\sigma_i = \sigma$ for $\sigma(x) = \mathsf{g}(\mathsf{a})$, $\sigma(y) = \sigma(z) = \mathsf{g}(\mathsf{b})$, $\sigma(x') = \mathsf{g}(\mathsf{c})$, and $\sigma(u) = \mathsf{h}(\mathsf{g}(\mathsf{b}),\mathsf{g}(\mathsf{c}))$.

Note that this chain is also a minimal $(\mathcal{P},\mathcal{R}')$-chain where $\mathcal{R}'$ is like $\mathcal{R}$ but without the $\mathsf{g}(\ldots)\to\mathsf{g}(\ldots)$-rule. Thus, $(\mathcal{P},\mathcal{R})$ can be simplified to $(\mathcal{P},\mathcal{R}')$.

Using the argument filter processor [101, Theorem 4.37], it is shown that there also is an infinite minimal chain when collapsing $\mathsf{G}$ to its first argument. Hence, the same substitution $\sigma$ can be used to obtain an infinite minimal chain for the DP problem $(\mathcal{P}',\mathcal{R}')$ where $\mathcal{P}' = \{\mathsf{f}(x,y,x',z,z,u) \to \mathsf{f}(x,y,x,x,y,\mathsf{h}(y,x'))\}$.

Now, if it would be sound to restrict the signature of $(\mathcal{P}',\mathcal{R}')$ to $\mathcal{F}(\mathcal{P}'\cup\mathcal{R}') = \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{f},\mathsf{h}\}$ then we can conclude termination. The reason is that over this signature there are no infinite minimal $(\mathcal{P}',\mathcal{R}')$-chains anymore.

We prove this last statement by contradiction. Suppose there is an infinite minimal $(\mathcal{P}',\mathcal{R}')$-chain over $\mathbf{s}$, $\mathbf{t}$, and $\delta$, where $\delta_i$ instantiates all variables by terms over $\mathcal{F}(\mathcal{P}'\cup\mathcal{R}')$, $\mathbf{s}_i = \mathsf{f}(x,y,x',z,z,u)$, and $\mathbf{t}_i = \mathsf{f}(x,y,x,x,y,\mathsf{h}(y,x'))$, for all $i$. $\delta_i$. Then all $\mathbf{t}_i\delta_i$ are terminating w.r.t. $\mathcal{R}'$. Hence, $\delta_1(y)$ must be a variable (otherwise, $\mathsf{h}(y,x')\delta_1$ would not be terminating due to the six $\mathsf{h}$-rules of $\mathcal{R}'$). Moreover, by using that $\delta_1(y)$ is a variable,

the derivation $\mathbf{t}_1\delta_1 \to^*_{\mathcal{R}'} \mathbf{s}_2\delta_2$ shows that $\delta_1(y) = \delta_2(y)$ and $\delta_1(y) = \delta_2(z)$. Note that since $\mathcal{R}'$ is not collapsing, whenever a term rewrites to a variable then the term must be identical to the variable. Hence, since $\delta_2(z)$ is a variable and $\delta_1(x) \to^*_{\mathcal{R}'} \delta_2(z)$ we obtain $\delta_1(x) = \delta_2(z)$ and for a similar reason we obtain $\delta_1(x) = \delta_2(x')$. In total, we can conclude $\delta_2(y) = \delta_2(x')$. This finally yields a contradiction as there is the nonterminating subterm $\mathsf{h}(y, x')\delta_2 = \mathsf{h}(\delta_2(y), \delta_2(y))$.

The consequences are severe: termination proofs relying upon techniques that require minimal chains and also use signature restrictions are unsound without further restrictions.

And indeed, for the technique of root-labeling—which performs a signature restriction within the soundness proof—we were able to construct a counter-example which refutes the main theorem for root-labeling with DPs.

**Example 8.14** (Root-labeling is unsound for minimal chains)**.** We use a similar TRS as in Example 8.13 to show the problem of root-labeling with minimal chains. Let $\mathcal{R}$ consist of the following rules.

$$\begin{aligned}
\mathsf{g}(\mathsf{f}(x, y, x', z, z, u)) &\to \mathsf{g}(\mathsf{f}(x, y, x, x, y, \mathsf{h}(y, x'))) \\
\mathsf{a} &\to \mathsf{b} \\
\mathsf{a} &\to \mathsf{c} \\
\mathsf{h}(x, x) &\to \mathsf{h}(x, x) \\
\mathsf{h}(\mathsf{a}, x) &\to \mathsf{h}(x, x) \\
\mathsf{h}(x, \mathsf{a}) &\to \mathsf{h}(x, x) \\
\mathsf{f}(x_1, \dots, \mathsf{a}, \dots, x_5) &\to \mathsf{f}(x_1, \dots, \mathsf{a}, \dots, x_5)
\end{aligned}$$

Here, the last rule represents 6 rules where the $\mathsf{a}$ can be at any position.

We again can simplify the initial DP-problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$ to $(\mathcal{P}, \mathcal{R})$ for the same $\mathcal{P} = \{\mathsf{G}(\mathsf{f}(x, y, x', z, z, u)) \to \mathsf{G}(\mathsf{f}(x, y, x, x, y, \mathsf{h}(y, x')))\}$ that we had in the previous example. The reason is again that there is an infinite minimal chain by choosing $\sigma_i = \sigma$ for $\sigma(x) = \mathsf{g}(\mathsf{a})$, $\sigma(y) = \sigma(z) = \mathsf{g}(\mathsf{b})$, $\sigma(x') = \mathsf{g}(\mathsf{c})$, and $\sigma(u) = \mathsf{h}(\mathsf{g}(\mathsf{b}), \mathsf{g}(\mathsf{c}))$.

Note that by using this substitution we also get an infinite minimal $(\mathcal{P}, \mathcal{R}')$-chain where $\mathcal{R}' = \mathcal{R} \setminus \{\mathsf{g}(\dots) \to \mathsf{g}(\dots)\}$. Hence, it is sound to simplify $(\mathcal{P}, \mathcal{R})$ to $(\mathcal{P}, \mathcal{R}')$.

Now, in [95, proofs of Lemmas 13 and 17] it is wrongly stated[4] that for this example, *w.l.o.g. one can restrict to substitutions over the signature* $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{h}\}$: With a similar reasoning as in Example 8.13 one can prove that there is no infinite minimal $(\mathcal{P}, \mathcal{R}')$-chain using this restricted class of substitutions. We further show in detail that Lemma 17 of [95] itself is wrong, not only its proof.

The result of Lemma 17 states that if there is an infinite minimal $(\mathcal{P}, \mathcal{R}')$-chain then there also is an infinite minimal chain for the DP problem $(\mathcal{P}', \mathcal{R}'')$ that is obtained by the flat context closure. In our example we obtain $\mathcal{P}' = \mathcal{P} \cup \{\mathsf{G}(\mathsf{a}) \to \mathsf{G}(\mathsf{b}), \mathsf{G}(\mathsf{a}) \to \mathsf{G}(\mathsf{c})\}$ and $\mathcal{R}'' = (\mathcal{R}' \setminus \{\mathsf{a} \to \mathsf{b}, \mathsf{a} \to \mathsf{c}\}) \cup \mathcal{R}'''$ where $\mathcal{R}'''$ consists of the following rules:

$$\begin{aligned}
\mathsf{h}(\mathsf{a}, x) &\to \mathsf{h}(\mathsf{b}, x) \\
\mathsf{h}(\mathsf{a}, x) &\to \mathsf{h}(\mathsf{c}, x) \\
\mathsf{h}(x, \mathsf{a}) &\to \mathsf{h}(x, \mathsf{b}) \\
\mathsf{h}(x, \mathsf{a}) &\to \mathsf{h}(x, \mathsf{c})
\end{aligned}$$

[4] In detail, in [95] our upcoming Lemma 8.17 is used without the requirement of left-linearity.

$$\mathsf{f}(x_1, \ldots, \mathsf{a}, \ldots, x_5) \to \mathsf{f}(x_1, \ldots, \mathsf{b}, \ldots, x_5)$$
$$\mathsf{f}(x_1, \ldots, \mathsf{a}, \ldots, x_5) \to \mathsf{f}(x_1, \ldots, \mathsf{c}, \ldots, x_5)$$

We show that for this DP problem $(\mathcal{P}', \mathcal{R}'')$ there are no infinite minimal chains anymore. So, if Lemma 17 of [95] would be sound, we could wrongly "prove" termination of $\mathcal{R}$. Again, we assume there is an infinite minimal $(\mathcal{P}', \mathcal{R}'')$-chain where $\delta_i$ are the corresponding substitutions and where we do not even restrict the signature of any $\delta_i$. Obviously, all $(\mathbf{s}_i, \mathbf{t}_i)$ are taken from $\mathcal{P}$ and not from one of the additional rules in $\mathcal{P}'$. Since every left-hand side of $\mathcal{R}''$ also is a left-hand side of a nonterminating rule in $\mathcal{R}''$, we know that every terminating term w.r.t. $\mathcal{R}''$ is also a normal form w.r.t. $\mathcal{R}''$. Hence, from $\mathbf{t}_1\delta_1 \to^*_{\mathcal{R}''} \mathbf{s}_2\delta_2$ we conclude $\mathbf{t}_1\delta_1 = \mathbf{s}_2\delta_2$. Thus, $\delta_2(x') = \delta_1(x) = \delta_2(z) = \delta_1(y) = \delta_2(y)$. Therefore, we obtain the nonterminating subterm $\mathsf{h}(y, x')\delta_2 = \mathsf{h}(\delta_2(y), \delta_2(y))$ which is a contradiction to the minimality of the chain.

To conclude, the current applications of root-labeling in termination tools which rely upon DPs with minimal chains are wrong for two reasons: first, one cannot restrict the signature to the implicit signature of the given DP-problem, and second, root-labeling is unsound in the DP setting with minimal chains.

However, for signature restrictions in combination with minimal chains we were able to prove soundness, provided that the TRS $\mathcal{R}$ of a DP problem $(\mathcal{P}, \mathcal{R})$ is left-linear.

**Lemma 8.15** (Signature Restrictions for Minimal Chains)**.** $\mathsf{left\_linear}\ \mathcal{R} \Longrightarrow$
$\mathcal{F}(\mathcal{P}, \mathcal{R}) \subseteq \mathcal{F} \Longrightarrow \mathsf{min\_ichain}\ (\mathcal{P}, \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ \sigma \Longrightarrow \mathsf{min\_ichain}\ (\mathcal{P}, \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ [\![\sigma]\!]_{\mathcal{F}}$

The proof of Lemma 8.15 is similar to the proof of Lemma 8.8. The only missing step is to prove that left-linearity ensures that cleaning does not introduce nontermination.

**Lemma 8.16** (Cleaning of Left-Linear TRSs Preservers SN)**.** *(i)* $\mathsf{left\_linear}\ \mathcal{R} \Longrightarrow$
$\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F} \Longrightarrow \mathsf{SN}_{\mathcal{R}}(s) \Longrightarrow [\![s]\!]_{\mathcal{F}} \to_{\mathcal{R}} t \Longrightarrow \exists\, u.\ [\![u]\!]_{\mathcal{F}} = t \wedge s \to_{\mathcal{R}} u$

*(ii)* $\mathsf{left\_linear}\ \mathcal{R} \Longrightarrow \mathcal{F}(\mathcal{R}) \subseteq \mathcal{F} \Longrightarrow \mathsf{SN}_{\mathcal{R}}(s) \Longrightarrow \mathsf{SN}_{\mathcal{R}}([\![s]\!]_{\mathcal{F}})$

*Proof.* (i) We prove this fact via induction over $s$. In the base case, $s$ is a variable. Then we have the rewrite step $s \to_{\mathcal{R}} t$, since cleaning does not change variables. But then, there is a variable left-hand side, implying that $\mathcal{R}$ is not terminating and thus contradicting $\mathsf{SN}_{\mathcal{R}}(s)$.

In the step case we have $s = f(\vec{ss})$. Now, we proceed by a case distinction. If $(f, |\vec{ss}|) \notin \mathcal{F}$ then cleaning will transform $s$ into the variable $z$. Again, there would be a variable left-hand side, contradicting strong normalization of $s$. Thus, $(f, |\vec{ss}|) \in \mathcal{F}$. Hence, $f(map\ [\![\cdot]\!]_{\mathcal{F}}\ \vec{ss}) \to_{\mathcal{R}} t$. If this is a non-root step, the result follows from the induction hypothesis. Otherwise, this is a root rewrite step. Thus we obtain a rule $(l, r) \in \mathcal{R}$ and a substitution $\sigma$, such that, $[\![f(\vec{ss})]\!]_{\mathcal{F}} = l\sigma$ and $r\sigma = t$. Additionally, we know that this rule is left-linear and that its left-hand side is well-formed. It can be shown that this implies the existence of a substitution $\tau$, such that, $[\![\tau_{\mathcal{V}\mathsf{ar}(l)}]\!]_{\mathcal{F}} = \sigma|_{\mathcal{V}\mathsf{ar}(l)}$ and $f(\vec{ss}) = l\tau$ (we omit the rather technical proof). Here, $\sigma|_V$ denotes the restriction of a substitution $\sigma$ to a set of variables $V$, i.e., all variables that are not in $V$, are no longer modified by the restricted substitution. Then $[\![r\tau]\!]_{\mathcal{F}} = [\![r]\!]_{\mathcal{F}}[\![\tau]\!]_{\mathcal{F}} = r[\![\tau_{\mathcal{V}\mathsf{ar}(l)}]\!]_{\mathcal{F}} = r\sigma|_{\mathcal{V}\mathsf{ar}(l)} = r\sigma = t$ and $s = f(\vec{ss}) = l\tau \to_{\mathcal{R}} r\tau$. Here, we needed to use the property $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$, which must be valid since otherwise $\mathsf{SN}_{\mathcal{R}}(s)$ does not hold.

(ii) Assume that $[\![s]\!]_{\mathcal{F}}$ is not terminating. Thus, there is an infinite sequence of $\mathcal{R}$-steps, starting from $[\![s]\!]_{\mathcal{F}}$. By iteratively applying the previous result, we obtain an infinite $\mathcal{R}$-sequence starting at $s$. □

We were also able to formally show that the signature restriction that is done in root-labeling (which is exactly the upcoming Lemma 8.17 without the requirement of left-linearity) is sound for minimal chains with the requirement of left-linearity. Hence, with the following lemma one can repair the paper proofs of [95, Lemmas 13 and 17] by demanding left-linearity. Essentially, the lemma states that one can restrict to the symbols that occur below the root in $\mathcal{P}$ ($\mathcal{F}_{>\epsilon}(\mathcal{P})$), together with the symbols of $\mathcal{R}$, under the additional assumption that neither left-hand sides nor right-hand sides of $\mathcal{P}$ are variables and the roots of $\mathcal{P}$ are not defined in $\mathcal{R}$.

**Lemma 8.17** (Signature Restrictions Ignoring Roots)**.** $\mathsf{left\_linear}\ \mathcal{R} \Longrightarrow$
$\mathcal{F}_{>\epsilon}(\mathcal{P}) \cup \mathcal{F}(\mathcal{R}) \subseteq \mathcal{F} \Longrightarrow$
$\forall\, s\ t.\ (s,\ t) \in \mathcal{P} \longrightarrow s \notin \mathcal{V}\mathsf{ar} \wedge t \notin \mathcal{V}\mathsf{ar} \wedge \neg\ \mathsf{root}(t) \in \mathcal{D}(\mathcal{R}) \Longrightarrow$
$\mathsf{min\_ichain}\ (\mathcal{P},\ \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ \sigma \Longrightarrow \mathsf{min\_ichain}\ (\mathcal{P},\ \mathcal{R})\ \mathbf{s}\ \mathbf{t}\ [\![\sigma]\!]_{\mathcal{F}}$

The lemma is proven in the same way as Lemma 8.15, except that one only applies cleaning strictly below the root. By cleaning below the root one can also proof a variant of Lemma 8.17 where minimal chains are replaced by arbitrary chains, and where left-linearity is no longer required.[5]

Using Lemma 8.17 and the original proofs of [95] it is shown that root-labeling is sound in combination with minimal chains if we restrict to left-linear $\mathcal{R}$-components. Hence, the main example of [95, Touzet's SRS] is still working, since it applies root-labeling on a DP problem with left-linear $\mathcal{R}$.

## 8.6. Conclusion

We presented an alternative, and more importantly, the first mechanized proof of the fact that termination is preserved under signature extensions. We have also shown that signature extensions are possible when using DPs, but only if one considers arbitrary chains or left-linear TRSs. For minimal chains we have given a counterexample which shows that for non-left-linear TRSs one cannot restrict to the signature of the current DP problem.

We believe these results to be interesting in their own. However, we developed these results with a certain goal in mind. In the end we want to apply our main positive results to be able to certify termination proofs which rely upon techniques where the signature is essential: string reversal and root-labeling. If one applies these techniques directly on a TRS, then both techniques can now be certified in the way they are used in current termination tools. For root-labeling in the DP setting with minimal chains, we have shown that it is unsound for arbitrary DP problems. We have further shown how to repair the existing proofs by demanding left-linearity. It remains as future work, to also formalize the remaining proof for root-labeling in the DP setting.

[5] However, one needs the additional requirement that left-hand sides of $\mathcal{R}$ are not variables, which in Lemma 8.17 follows from the minimality of the chain.

# 9. Modular and Certified Semantic Labeling and Unlabeling

### Publication details

Christian Sternagel and René Thiemann. Modular and Certified Semantic Labeling and Unlabeling. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications*, volume 10 of LIPIcs, pages 329–344. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011.

### Abstract

Semantic labeling is a powerful transformation technique to prove termination of term rewrite systems. The dual technique is unlabeling. For unlabeling it is essential to drop the so called decreasing rules which sometimes have to be added when applying semantic labeling. We indicate two problems concerning unlabeling and present our solutions.

The first problem is that currently unlabeling cannot be applied as a modular step, since the decreasing rules are determined by a semantic labeling step which may have taken place much earlier. To this end, we give an implicit definition of decreasing rules that does not depend on any knowledge about preceding labelings.

The second problem is that unlabeling is in general unsound. To solve this issue, we introduce the notion of extended termination problems. Moreover, we show how existing termination techniques can be lifted to operate on extended termination problems.

All our proofs have been formalized in Isabelle/HOL as part of the IsaFoR/CeTA project.

## 9.1. Introduction

In recent years, termination provers for term rewrite systems (TRSs) became more and more powerful. Nowadays, we do no longer have to prove termination by embedding all rules of a TRS into a single reduction order. Instead, most provers construct multi-step proofs by combining different termination techniques[1] resulting in tree-like termination proofs. As a result, termination provers became more complex and thus, more error-prone. It is regularly demonstrated that we cannot blindly trust termination provers. Every now and then, some prover delivers a faulty proof. Most of the time, this is only detected if there is another prover giving a contradictory answer. Furthermore, it just is too much work to check a generated proof by hand. (Besides, checking by hand is not very reliable.)

To solve this issue, recent interest is in the automatic certification of termination proofs [12, 23, 104]. To this end, we formalized many termination techniques in our Isabelle/HOL [84] library IsaFoR [104] (in the remainder we just write *Isabelle*, instead of *Isabelle/HOL*). Using IsaFoR, we obtain CeTA, an automatic certifier for termination proofs.

[1] Several termination techniques are based upon reduction orders, but there are also techniques which do not generate orders. Hence, the multi-step proofs are not just a lexicographic combination of orders.

In this paper, we present our formalization of semantic labeling and unlabeling [110], two important termination techniques. Semantic labeling introduces differently labeled variants of the same symbol, allowing a distinction in orders, etc. Semantic labeling typically produces large TRSs. Hence, unlabeling is important to keep the number of symbols and rules small.

**Example 9.1.** Consider the small TRS `Secret_05/teparla3` from the termination problem database (TPDB) which only consists of two rules, has two different symbols, and two variables. We just describe the structure of the proof that has been generated by the termination prover AProVE [39] in 21 seconds during the 2008 termination competition.[2]

After applying the dependency pair transformation [3] and some standard techniques, a termination problem containing three rules and three different symbols is obtained. Then, semantic labeling is applied. The result after simplification, is a system with five rules and seven different symbols. Unlabeling yields a problem with three rules and three symbols. Another labeling produces a new termination problem with 12 rules. This is finally proven to be terminating using a matrix interpretation [32] of dimension two.

Note that without the unlabeling step, the second labeling would have returned a system with 5025 rules instead of 12—for this huge termination problem no suitable matrix interpretation of dimension two is detected.

Whereas the previous example shows that unlabeling is essential to keep systems small, we also found examples where unlabeling was the key to get a successful termination proof at all, cf. Example 9.24 for details.

Unfortunately, unlabeling is not sound in general. In order to allow nested labeling and unlabeling and turn unlabeling into a sound and modular technique (not relying on context information), we have designed a new framework. All existing termination techniques are easily integrated in this framework. In fact, CeTA uses the new framework for certification.

Note that all the proofs that are presented (or omitted) in the following, have been formalized as part of IsaFoR. Hence, we merely give sketches of our "real" proofs. Our goal is to show the general proof outlines and help to understand the full proofs. The library IsaFoR with all formalized proofs, the executable certifier CeTA, and all details about our experiments are available at CeTA's website:

`http://cl-informatik.uibk.ac.at/software/ceta`

The paper is structured as follows. In Section 9.2, we recapitulate some required notions of term rewriting as well as the basic definitions of semantic labeling. Afterwards, in Section 9.3, we give some challenges for modular labeling and unlabeling. Then, in Section 9.4, we extend the previous results to the dependency pair framework. We discuss challenges for the certification in Section 9.5. Our experiments are presented in Section 9.6 before we conclude in Section 9.7.

## 9.2. Preliminaries

### 9.2.1. Term Rewriting

We assume familiarity with term rewriting [5]. Still, we recall the most important notions that are used later on. A *(first-order) term* $t$ over a set of *variables* $\mathcal{V}$ and a set of *function*

[2] See `http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=35708`

*symbols* $\mathcal{F}$ is either a variable $x \in \mathcal{V}$ or an $n$-ary function symbol $f \in \mathcal{F}$ applied to $n$ argument terms $f(\vec{t}_n)$. For brevity we write $\vec{t}_n$ to denote a sequence of $n$ elements $t_1, \ldots, t_n$ and $(h(\vec{t}_n))$ (note the additional pair of parentheses) for $(h(t_1), \ldots, h(t_n))$, i.e., mapping a function $h$ over the elements of a sequence $\vec{t}_n$. A *context* $C$ is a term containing exactly one hole ($\Box$). Replacing $\Box$ in a context $C$ by a term $t$ is denoted by $C[t]$. A *(rewrite) rule* is a pair of terms $\ell \to r$ and a TRS $\mathcal{R}$ is a set of such rules. The *rewrite relation (induced by $\mathcal{R}$)* $\to_{\mathcal{R}}$ is the closure under substitutions and contexts of $\mathcal{R}$, i.e., $s \to_{\mathcal{R}} t$ iff there is a context $C$, a rule $\ell \to r \in \mathcal{R}$, and a substitution $\sigma$ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$.

We say that an element $t$ is *terminating / strongly normalizing (w.r.t. some binary relation $S$)*, and write $\mathsf{SN}_S(t)$, if it cannot start an infinite sequence $t = t_1\ S\ t_2\ S\ t_3\ S \cdots$. The whole relation is terminating, written $\mathsf{SN}(S)$, if all elements are terminating w.r.t. it. For a TRS $\mathcal{R}$ and a term $t$, we write $\mathsf{SN}(\mathcal{R})$ and $\mathsf{SN}_{\mathcal{R}}(t)$ instead of $\mathsf{SN}(\to_{\mathcal{R}})$ and $\mathsf{SN}_{\to_{\mathcal{R}}}(t)$. We write $S^+$ ($S^*$) for the (reflexive and) transitive closure of $S$.

**Definition 9.2** (Termination Technique)**.** *A* termination technique *is a mapping $TT$ from TRSs to TRSs. It is* sound *if termination of $TT(\mathcal{R})$ implies termination of $\mathcal{R}$.*

Using sound termination techniques one tries to modify a given TRS $\mathcal{R}$ until the empty TRS is reached. If this succeeds, one obtains a proof tree showing termination of $\mathcal{R}$.

## 9.2.2. Semantic Labeling

An algebra $\mathcal{A}$ over $\mathcal{F}$ is a pair $(A, \{f_{\mathcal{A}}\}_{f \in \mathcal{F}})$ consisting of a non-empty carrier $A$ and an interpretation function $f_{\mathcal{A}} \colon A^n \to A$ for every $n$-ary function symbol $f \in \mathcal{F}$. Given an assignment $\alpha \colon \mathcal{V} \to A$, we write $[\alpha]_{\mathcal{A}}(t)$ for the interpretation of the term $t$. An algebra $\mathcal{A}$ is a model of a rewrite system $\mathcal{R}$, if $[\alpha]_{\mathcal{A}}(\ell) = [\alpha]_{\mathcal{A}}(r)$ for all rules $\ell \to r \in \mathcal{R}$ and all assignments $\alpha$. If the carrier $A$ is equipped with a well-founded order $>_A$ such that $[\alpha]_{\mathcal{A}}(\ell) \geqslant_A [\alpha]_{\mathcal{A}}(r)$ for all $\ell \to r \in \mathcal{R}$ and all assignments $\alpha$, then $\mathcal{A}$ is a *quasi-model* of $\mathcal{R}$.

For each function symbol $f$ there also is a corresponding non-empty set $L_f$ of labels for $f$ and a labeling function $\ell_f \colon A^n \to L_f$. The labeled signature $\mathcal{F}_{\mathsf{lab}}$ consists of $n$-ary function symbols $f_a$ for every $n$-ary function symbol $f \in \mathcal{F}$ and label $a \in L_f$. The labeling function $\ell_f$ determines the label of the root symbol $f$ of a term $f(\vec{t}_n)$ based on the values of the arguments $\vec{t}_n$. For every assignment $\alpha \colon \mathcal{V} \to A$ the mapping $\mathsf{lab}_\alpha \colon \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}_{\mathsf{lab}}, \mathcal{V})$ is inductively defined by

$$\mathsf{lab}_\alpha(t) = \begin{cases} f_{\ell_f([\alpha]_{\mathcal{A}}(\vec{t}_n))}(\mathsf{lab}_\alpha(\vec{t}_n)) & \text{if } t = f(\vec{t}_n), \\ t & \text{otherwise.} \end{cases}$$

The labeled TRS $\mathsf{lab}(\mathcal{R})$ over the signature $\mathcal{F}_{\mathsf{lab}}$ consists of the rules $\mathsf{lab}_\alpha(\ell) \to \mathsf{lab}_\alpha(r)$ for all $\ell \to r \in \mathcal{R}$ and $\alpha \colon \mathcal{V} \to A$.

For quasi-models, every set of labels $L_f$ needs to be equipped with a well-founded order $>_{L_f}$, giving rise to the set $\mathcal{D}\mathrm{ec}$ of *decreasing rules*:

$$\mathcal{D}\mathrm{ec} = \{f_a(\vec{x}_n) \to f_b(\vec{x}_n) \mid a, b \in L_f, a >_{L_f} b, n\text{-ary } f \in \mathcal{F}\}$$

Furthermore, every interpretation function $f_{\mathcal{A}}$ and every labeling function $\ell_f$ has to be weakly monotone, i.e., if $a \geqslant_A a'$ then $f_{\mathcal{A}}(a_1, \ldots, a, \ldots, a_n) \geqslant_A f_{\mathcal{A}}(a_1, \ldots, a', \ldots, a_n)$ and $\ell_f(a_1, \ldots, a, \ldots, a_n) \geqslant_{L_f} \ell_f(a_1, \ldots, a', \ldots, a_n)$.

Unlabeling a symbol is defined via the following function, removing one layer of labels. Then, the function is extended homomorphically to terms, rules, and TRSs.

$$\mathsf{unlab}(f) = \begin{cases} g & \text{if } f = g_a, \\ f & \text{if } f \text{ is not labeled.} \end{cases}$$

In [110], Zantema showed that labeled TRSs can simulate their unlabeled counterparts (corresponding to **i** and **ii** in the following lemma; **iii** and **iv** are obvious).

**Lemma 9.3.** *Let $\mathcal{R}$ be a TRS,$\mathcal{A}$ an algebra, and $\alpha$ an arbitrary assignment.*

*(i) If $\mathcal{A}$ is a model of $\mathcal{R}$ then $t \to_{\mathcal{R}} u$ implies $\mathsf{lab}_\alpha(t) \to_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_\alpha(u)$.*

*(ii) If $\mathcal{A}$ is a quasi-model of $\mathcal{R}$ then $t \to_{\mathcal{R}} u$ implies $\mathsf{lab}_\alpha(t) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \mathsf{lab}_\alpha(u)$.*

*(iii) $t \to_{\mathsf{lab}(\mathcal{R})} u$ implies $\mathsf{unlab}(t) \to_{\mathcal{R}} \mathsf{unlab}(u)$.*

*(iv) $t \to_{\mathcal{D}\mathrm{ec}} u$ implies $\mathsf{unlab}(t) = \mathsf{unlab}(u)$.*

From Lemma 9.3 we obtain that $\mathcal{R}$ is terminating if and only if $\mathsf{lab}(\mathcal{R})\,(\cup\,\mathcal{D}\mathrm{ec})$ is terminating when $\mathcal{A}$ is a (quasi-)model of $\mathcal{R}$. Completeness is achieved by unlabeling all terms in a possible infinite rewrite sequence of the labeled TRS. Soundness is proved by transforming a presupposed infinite rewrite sequence in $\mathcal{R}$ into an infinite rewrite sequence in $\mathsf{lab}(\mathcal{R})\,(\cup\,\mathcal{D}\mathrm{ec})$. This is done by applying the labeling function $\mathsf{lab}_\alpha(\cdot)$ (for an arbitrary assignment $\alpha$) to all terms in the infinite rewrite sequence of $\mathcal{R}$. Hence, semantic labeling is sound and complete for termination (using models and quasi-models, respectively).

## 9.3. Modular Semantic Labeling and Unlabeling

One problem with semantic labeling is that the labeled system is usually large. Hence, termination provers such as AProVE [39], Jambox [31], Torpa [111], and TPA [62] perform labeling, then try to simplify the resulting TRS by sound termination techniques, and afterwards *unlabel* the TRS again, to continue on a small system. This poses two challenges:

(i) If labeling was performed using a quasi-model, then the decreasing rules are added. However, unlabeling a decreasing rule $f_a(\vec{x}_n) \to f_b(\vec{x}_n)$ leads to the nonterminating rule $f(\vec{x}_n) \to f(\vec{x}_n)$. Hence, one has to remove the decreasing rules before unlabeling.

(ii) Between labeling and unlabeling, arbitrary (sound) termination techniques may be applied. However, for unlabeling we want to remove the decreasing rules that are determined by the corresponding labeling step. Hence, unlabeling is not a modular technique that only takes a TRS as input. Instead, it relies on context information, namely the decreasing rules that have been used in the corresponding labeling step (which may occur several steps upwards in the termination proof).

Solving the first challenge is technically easy: just remove the decreasing rules before unlabeling. The only question is, whether it is always sound to remove the decreasing rules.

To handle the second challenge, we propose an implicit definition of decreasing rules.

**Definition 9.4** (Decreasing rules of a TRS)**.** *We define the* decreasing rules of a TRS $\mathcal{R}$ *as* $\mathcal{D}(\mathcal{R}) = \{\ell \to r \in \mathcal{R} \mid \mathsf{unlab}(\ell) = \mathsf{unlab}(r) \wedge \ell \neq r\}$*. We further define the* unlabeled version of a TRS *as* $\mathcal{U}(\mathcal{R}) = \mathsf{unlab}(\mathcal{R} \setminus \mathcal{D}(\mathcal{R}))$*.*

The condition $\ell \neq r$ ensures that a labeled variant of an original rule is never decreasing. For example, if $f(\vec{x}_n) \to f(\vec{x}_n)$ is a rule (and hence the original TRS is not terminating), then each labeled variant has the form $f_a(\vec{x}_n) \to f_a(\vec{x}_n)$ for some $a \in L_f$. If we would consider such a rule as decreasing, we could transform a nonterminating TRS into a terminating one, using labeling and unlabeling.

**Lemma 9.5.** *Let $L_f$ and $>_{L_f}$ be given for each symbol $f$ to determine $\mathcal{D}$ec. Then $\mathcal{D}(\mathcal{D}\mathrm{ec}) = \mathcal{D}\mathrm{ec}$, $\mathcal{D}(\mathsf{lab}(\mathcal{R})) = \varnothing$, $\mathcal{D}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}) = \mathcal{D}\mathrm{ec}$, and $\mathcal{U}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}) = \mathcal{R}$.*

Now it is easy to define a modular version of unlabeling which does not require external knowledge about what the decreasing rules are.

**Definition 9.6** (Unlabeling as modular termination technique)**.** *The* unlabeling termination technique *replaces a TRS $\mathcal{R}$ by $\mathcal{U}(\mathcal{R})$.*

Hence, we solved the second challenge and made unlabeling into an independent technique which does not need any knowledge on the previous application of semantic labeling that introduced the decreasing rules. Thus, termination proofs can now use the following structure where no global information has to be passed around:

(i) Switch from $\mathcal{R}$ to $\mathsf{lab}(\mathcal{R})$ ($\cup\, \mathcal{D}\mathrm{ec}$).

(ii) Modify $\mathsf{lab}(\mathcal{R})$ ($\cup\, \mathcal{D}\mathrm{ec}$) by sound termination techniques resulting in $\mathcal{R}'$.

(iii) Unlabel $\mathcal{R}'$ resulting in $\mathcal{U}(\mathcal{R}')$.

Although this approach is used in termination provers, it is unsound in general as not every sound termination technique may be used between labeling and unlabeling. This is illustrated by the following example.

**Example 9.7.** We start with the nonterminating TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{b}), \mathsf{b} \to \mathsf{a}\}$. Then, we apply semantic labeling using the algebra $\mathcal{A}$ with $A = \{0, 1\}$, interpretations $\mathsf{f}_\mathcal{A}(x) = 0$, $\mathsf{a}_\mathcal{A} = 0$, $\mathsf{b}_\mathcal{A} = 1$, $L_\mathsf{f} = A$, $\ell_\mathsf{f}(x) = x$, and the standard order on the naturals. Note that $\mathcal{A}$ is a quasi-model of $\mathcal{R}$. The resulting labeled TRS is $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec} = \{\mathsf{f}_0(\mathsf{a}) \to \mathsf{f}_1(\mathsf{b}), \mathsf{b} \to \mathsf{a}, \mathsf{f}_1(x) \to \mathsf{f}_0(x)\}$. It is sound to replace $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$ by the (nonterminating) TRS $\mathcal{R}' = \{\mathsf{f}_1(x) \to \mathsf{f}_0(x), \mathsf{f}_0(x) \to \mathsf{f}_1(x)\}$. However, unlabeling $\mathcal{R}'$ yields $\mathcal{U}(\mathcal{R}') = \varnothing$ as both rules in $\mathcal{R}'$ are decreasing according to Definition 9.4. Hence, some of the performed deductions were not sound. Since semantic labeling and the switch from $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$ to $\mathcal{R}'$ are sound, we obtain that unlabeling via $\mathcal{U}$ is unsound.

The problematic step when unlabeling, i.e., when switching from $\mathcal{R}$ to $\mathcal{U}(\mathcal{R}) = \mathsf{unlab}(\mathcal{R} \setminus \mathcal{D}(\mathcal{R}))$, is the removal of the decreasing rules. If the decreasing rules are the only source of nontermination, then this removal is unsound. However, the decreasing rules $\mathcal{D}\mathrm{ec}$ that are obtained from semantic labeling are always terminating. Thus, after labeling we have to prove termination of the labeled system including the decreasing rules, but we may assume that the decreasing rules are terminating. If we know that the decreasing rules are terminating, then unlabeling by $\mathcal{U}$ is sound. We obtain the following structure of termination proofs:

(i) Initially we have to prove $\mathsf{SN}(\mathcal{R})$.

(ii) After labeling, we have to prove $\mathsf{SN}(\mathcal{D}(\mathcal{R}')) \Longrightarrow \mathsf{SN}(\mathcal{R}')$ for $\mathcal{R}' = \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\text{ec}$.

(iii) Then, we modify $\mathcal{R}'$ to $\mathcal{R}''$ with $\mathsf{SN}(\mathcal{D}(\mathcal{R}'')) \Longrightarrow \mathsf{SN}(\mathcal{R}'')$ implies $\mathsf{SN}(\mathcal{D}(\mathcal{R}')) \Longrightarrow \mathsf{SN}(\mathcal{R}')$.

(iv) Finally, we unlabel $\mathcal{R}''$ resulting in $\mathcal{U}(\mathcal{R}'')$ and have to prove $\mathsf{SN}(\mathcal{U}(\mathcal{R}''))$.

This approach works fine for termination proofs where semantic labeling is not nested. However, we are aware of termination proofs where labeling is applied in a nested way.

**Example 9.8.** Consider the TRS `Gebhardt_06/16` from the TPDB. During the 2008 termination competition, Jambox proved termination of this TRS, applying the following steps: labeling - labeling - labeling - polynomial order - unlabeling - four applications of polynomial orders - unlabeling - unlabeling.[3]

To support this kind of proof we define the following variant of strong normalization.

**Definition 9.9.** *An* extended termination problem *is a pair* $(\mathcal{R}, n)$ *consisting of a TRS* $\mathcal{R}$ *and a number* $n \in \mathbb{N}$. *An extended problem* $(\mathcal{R}, n)$ *is* strongly normalizing *(*$\mathsf{SN}(\mathcal{R}, n)$*) iff*

$$(\forall m < n.\, \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))) \quad \Longrightarrow \quad \mathsf{SN}(\mathcal{R}).$$

*An* extended termination technique *is a mapping xTT from extended termination problems to extended termination problems. It is* sound *iff* $\mathsf{SN}(xTT(\mathcal{R}, n))$ *implies* $\mathsf{SN}(\mathcal{R}, n)$.

The number $n$ in an extended termination problem $(\mathcal{R}, n)$ describes how often we can assume that the decreasing rules are terminating, and hence, it tells us how often we can delete the decreasing rules during unlabeling. The following lemma provides the link between both variants of strong normalization.

**Lemma 9.10.** *(i)* $\mathsf{SN}(\mathcal{R})$ *iff* $\mathsf{SN}(\mathcal{R}, 0)$.

*(ii) If* $\mathsf{SN}(\mathcal{R})$ *then* $\mathsf{SN}(\mathcal{R}, n)$.

**Lemma 9.11** (Extended Unlabeling)**.** *Extended unlabeling is sound where*

$$\mathcal{U}(\mathcal{R}, n) = \begin{cases} (\mathcal{U}(\mathcal{R}), n-1) & \text{if } n > 0, \\ (\mathsf{unlab}(\mathcal{R}), 0) & \text{otherwise.} \end{cases}$$

*Proof.* We only consider the interesting case where $n > 0$. So, we have to show $\mathsf{SN}(\mathcal{R}, n)$ under the first assumption $\mathsf{SN}(\mathcal{U}(\mathcal{R}), n-1)$. To prove $\mathsf{SN}(\mathcal{R}, n)$, we have to prove $\mathsf{SN}(\mathcal{R})$ under the second assumption $\forall m < n.\, \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$. Since $n > 0$ we can choose $m = 0$ and obtain $\mathsf{SN}(\mathcal{D}(\mathcal{R}))$.

To show $\mathsf{SN}(\mathcal{R})$ we assume that there is an infinite $\to_{\mathcal{R}}$-derivation $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$ and obtain a contradiction. The infinite derivation is also an infinite $\to_{\mathcal{R}\setminus\mathcal{D}(\mathcal{R})} \cup \to_{\mathcal{D}(\mathcal{R})}$-derivation. Since $\mathcal{D}(\mathcal{R})$ is terminating, we know that there are infinitely many $i$ with $t_i \to_{\mathcal{R}\setminus\mathcal{D}(\mathcal{R})} t_{i+1}$. Hence $\mathsf{unlab}(t_i) \to_{\mathcal{U}(\mathcal{R})} \mathsf{unlab}(t_{i+1})$ for all these $i$ as $\mathcal{U}(\mathcal{R}) = \mathsf{unlab}(\mathcal{R} \setminus \mathcal{D}(\mathcal{R}))$. Moreover, for all $i$ where $t_i \to_{\mathcal{D}(\mathcal{R})} t_{i+1}$, we know that $\mathsf{unlab}(t_i) \to_{\mathsf{unlab}(\mathcal{D}(\mathcal{R}))} \mathsf{unlab}(t_{i+1})$ and hence, $\mathsf{unlab}(t_i) = \mathsf{unlab}(t_{i+1})$ since every rule in $\mathsf{unlab}(\mathcal{D}(\mathcal{R}))$ has the

[3] See `http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=27220`

same left- and right-hand side. Thus, we have constructed an infinite derivation for $\mathcal{U}(\mathcal{R})$ proving that $\mathsf{SN}(\mathcal{U}(\mathcal{R}))$ does not hold. Together with the assumption $\mathsf{SN}(\mathcal{U}(\mathcal{R}), n-1)$, we obtain that $\forall m < n-1.\, \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{U}(\mathcal{R}))))$ does not hold (by Definition 9.9). Hence, there is some $m < n-1$ such that $\mathsf{SN}(\mathcal{D}(\mathcal{U}^{m+1}(\mathcal{R})))$ does not hold. Now, using the second assumption and $m+1 < n$ we obtain $\mathsf{SN}(\mathcal{D}(\mathcal{U}^{m+1}(\mathcal{R})))$, providing the required contradiction. □

**Lemma 9.12** (Extended Semantic Labeling)**.** *Semantic labeling is sound as extended termination technique: Whenever we can switch from $\mathcal{R}$ to $\mathsf{lab}(\mathcal{R})$ ($\cup\, \mathcal{D}\mathrm{ec}$) via semantic labeling, then it is sound to switch from $(\mathcal{R}, n)$ to $(\mathsf{lab}(\mathcal{R})$ ($\cup\, \mathcal{D}\mathrm{ec}), n+1)$.*

*Proof.* Note that models are just a special case of quasi-models as already observed in [110]. Hence, we only consider quasi-models in the proof. So, assuming $\mathsf{SN}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}, n+1)$ we have to prove $\mathsf{SN}(\mathcal{R}, n)$. To show the latter, we may assume $\forall m < n.\, \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$ and have to prove $\mathsf{SN}(\mathcal{R})$. We do so by assuming that there is an infinite $\mathcal{R}$-derivation $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$ and deriving a contradiction. As we have a quasi-model we know that $\mathsf{lab}_\alpha(t_1) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \mathsf{lab}_\alpha(t_2) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \cdots$ is an infinite $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$-derivation, showing that $\mathsf{SN}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec})$ does not hold. By the conditions of semantic labeling, we further know $\mathsf{SN}(\mathcal{D}\mathrm{ec})$. Using $\mathsf{SN}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}, n+1)$ we conclude that $\forall m < n+1.\, \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec})))$ does not hold. Hence there is some $m < n+1$ such that $\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec})))$ does not hold. If $m = 0$ then by Lemma 9.5 we know that $\mathcal{D}(\mathcal{U}^m(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec})) = \mathcal{D}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}) = \mathcal{D}\mathrm{ec}$, and thus $\mathsf{SN}(\mathcal{D}\mathrm{ec})$ does not hold, a contradiction. Otherwise, $m = m' + 1$ for some $m'$ where $m' < n$. Together with $\forall m < n.\, \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$, we obtain $\mathsf{SN}(\mathcal{D}(\mathcal{U}^{m'}(\mathcal{R})))$. On the other hand, we know that $\mathsf{SN}(\mathcal{D}(\mathcal{U}^{m'+1}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec})))$ does not hold. This again leads to a contradiction since $\mathcal{D}(\mathcal{U}^{m'+1}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec})) = \mathcal{D}(\mathcal{U}^{m'}(\mathcal{U}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}))) = \mathcal{D}(\mathcal{U}^{m'}(\mathcal{R}))$ by Lemma 9.5. □

The previous two lemmas show that labeling and unlabeling can be performed as independent techniques on extended termination problems.

The question remains how to integrate other existing termination techniques, i.e., which techniques may be applied between labeling and unlabeling. Here, we consider two variants.

**Definition 9.13** (Lift)**.** *Let $TT$ be some termination technique. Then $\mathsf{lift}(TT)$ and $\mathsf{lift}_0(TT)$ are extended termination techniques where $\mathsf{lift}(TT)(\mathcal{R}, n) = (TT(\mathcal{R}), n)$ and $\mathsf{lift}_0(TT)(\mathcal{R}, n) = (TT(\mathcal{R}), 0)$.*

In principle $\mathsf{lift}(TT)$ is preferable, since it does not change $n$, allowing to remove the decreasing rules when unlabeling (which is not possible using $\mathsf{lift}_0(TT)$). However, in general the fact that $TT$ is sound does not imply that $\mathsf{lift}(TT)$ is sound. This can easily be seen by reusing Example 9.7 where the extended termination problem $(\mathcal{R}, 0)$ is transformed to $(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}, 1)$ by semantic labeling, then to $(\mathcal{R}', 1)$ using $\mathsf{lift}(TT)$ for the unnamed sound termination technique $TT$ in Example 9.7, and then to $(\varnothing, 0)$ by unlabeling. Since this establishes a complete termination proof for the nonterminating TRS $\mathcal{R}$, and since labeling and unlabeling are sound, we know that $\mathsf{lift}(TT)$ is unsound.

Since we cannot always use $\mathsf{lift}(TT)$, we give three different approaches to use termination techniques as extended termination techniques (in order of preference):

(i) Identify a (hopefully large) class of termination techniques $TT$ for which soundness of $TT$ implies soundness of $\mathsf{lift}(TT)$.

(ii) Perform a direct proof that $\mathsf{lift}(TT)$ is sound as extended termination technique.

(iii) Use $\mathsf{lift}_0(TT)$ for any sound termination technique $TT$.

We first prove soundness of approach **iii**.

**Lemma 9.14.** *If $TT$ is sound then $\mathsf{lift}_0(TT)$ is sound.*

*Proof.* We have to prove that $\mathsf{SN}(TT(\mathcal{R}),0)$ implies $\mathsf{SN}(\mathcal{R},n)$. So, assume $\mathsf{SN}(TT(\mathcal{R}),0)$. Hence, $\mathsf{SN}(TT(\mathcal{R}))$ using Lemma 9.10(**i**). As $TT$ is sound, we conclude $\mathsf{SN}(\mathcal{R})$ and this implies $\mathsf{SN}(\mathcal{R},n)$ by Lemma 9.10(**ii**). □

We start to prove soundness of $\mathsf{lift}(TT)$ for some sound termination technique $TT$ in order to detect where the problem is. To prove soundness, we have to show that $\mathsf{SN}(TT(\mathcal{R}),n)$ implies $\mathsf{SN}(\mathcal{R},n)$. Thus, assume $\mathsf{SN}(TT(\mathcal{R}),n)$. To prove $\mathsf{SN}(\mathcal{R},n)$ we may assume that $\forall m<n.\,\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$ and have to prove $\mathsf{SN}(\mathcal{R})$. Since $TT$ is sound, it suffices to prove $\mathsf{SN}(TT(\mathcal{R}))$. To this end, it suffices to show $\forall m<n.\,\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R}))))$ by using $\mathsf{SN}(TT(\mathcal{R}),n)$. Hence, the only missing step is to conclude

$$\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R}))) \quad\Longrightarrow\quad \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R})))). \tag{$\star$}$$

**Lemma 9.15.** *If $TT$ is sound and if ($\star$) is satisfied for all $m$, then $\mathsf{lift}(TT)$ is sound.*

A sufficient condition to ensure ($\star$) is to demand that $TT(\mathcal{R})\subseteq\mathcal{R}$ as $\mathsf{unlab}$, $\mathcal{D}$, and $\mathcal{U}$ are monotone w.r.t. set inclusion. Hence, all techniques that remove rules like rule removal via reduction pairs, or (RFC) matchbounds [36, 66] can safely be used between labeling and unlabeling. However, this excludes techniques like the flat context closure which is required for root-labeling.

**Definition 9.16** (Root-Labeling)**.** *Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$. Let $\mathcal{A}_\mathcal{F}$ be an algebra with carrier $\mathcal{F}$. Moreover, for every $n$-ary $f\in\mathcal{F}$, we fix the interpretation function $f_{\mathcal{A}_\mathcal{F}}(\vec{x}_n)=f$, the set of labels $L_f=\mathcal{F}^n$, and the labeling function $\ell_f(\vec{x}_n)=(\vec{x}_n)$.*

Note that root-labeling is just a specific instantiation of general semantic labeling with models. Hence, it is sound whenever $\mathcal{A}_\mathcal{F}$ is a model of $\mathcal{R}$. However, in general $\mathcal{A}_\mathcal{F}$ does not constitute a model of $\mathcal{R}$. Hence, a transformation technique was introduced that modifies $\mathcal{R}$ in a way that $\mathcal{A}_\mathcal{F}$ always is a model of the result: the *closure under flat contexts*.

**Definition 9.17** (Flat Context Closure)**.** *For an $n$-ary symbol $f$, the* flat context for the $i$-th argument *is $\mathcal{FC}^i(f)=f(x_1,\ldots,x_{i-1},\Box,x_{i+1},\ldots,x_n)$, where all the $x_j$ are fresh variables. The set of* flat contexts over $\mathcal{F}$ *is defined by $\mathcal{FC}(\mathcal{F})=\{\mathcal{FC}^i(f)\mid n\text{-ary } f\in\mathcal{F}, 1\leqslant i\leqslant n\}$. The* closure under flat contexts of a TRS $\mathcal{R}$ w.r.t. the signature $\mathcal{F}$ *is given by*

$$\mathcal{FC}_\mathcal{F}(\mathcal{R})=\{C[\ell]\to C[r]\mid C\in\mathcal{FC}(\mathcal{F}),\ell\to r\in\mathcal{R}_\mathsf{a}\}\cup(\mathcal{R}\setminus\mathcal{R}_\mathsf{a})$$

*where $\mathcal{R}_\mathsf{a}$ denotes those rules of $\mathcal{R}$, for which the root of the left-hand side and the root of the right-hand side differ.*

Since Jambox applies root-labeling recursively (the labeling in Example 9.8 is root-labeling), we definitely would like to aim at a larger class of termination techniques than those which satisfy $TT(\mathcal{R})\subseteq\mathcal{R}$. A natural extension would be to use the weaker condition $\to_{TT(\mathcal{R})}\subseteq\to_\mathcal{R}$. Then, also root-labeling together with the closure under flat contexts would be supported. Unfortunately, $\to_{TT(\mathcal{R})}\subseteq\to_\mathcal{R}$ does not imply $\to_{\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R})))}\subseteq\to_{\mathcal{D}(\mathcal{U}^m(\mathcal{R}))}$ and thus, does not imply ($\star$). Moreover, in the following example we show that even if $TT$ is sound and $\to_{TT(\mathcal{R})}\subseteq\to_\mathcal{R}$ then soundness of $\mathsf{lift}(TT)$ cannot be guaranteed.

**Example 9.18.** Consider the TRS $\mathcal{R} = \{\mathsf{f}_1(x) \to \mathsf{f}_0(\mathsf{a}), \mathsf{f}_0(x) \to \mathsf{f}_1(x)\}$. Let $TT$ be the termination technique that replaces $\mathcal{R}$ by $\mathcal{R}' = \{\mathsf{f}_1(\mathsf{a}) \to \mathsf{f}_0(\mathsf{a}), \mathsf{f}_0(x) \to \mathsf{f}_1(x)\}$. Then, $TT$ is sound as $\mathcal{R}'$ is not terminating. Moreover, $\to_{\mathcal{R}'} \subseteq \to_{\mathcal{R}}$. Nevertheless, $\mathsf{lift}(TT)$ is unsound, since it would replace $(\mathcal{R}, 1)$ by $(\mathcal{R}', 1)$. That this replacement is unsound can be seen as follows: $\mathsf{SN}(\mathcal{R}, 1)$ does not hold since $\mathcal{R}$ is not terminating but the decreasing rules of $\mathcal{R}$ (i.e., $\mathcal{D}(\mathcal{R}) = \{\mathsf{f}_0(x) \to \mathsf{f}_1(x)\}$) are terminating. However, $\mathsf{SN}(\mathcal{R}', 1)$ is satisfied as $\mathcal{D}(\mathcal{R}') = \mathcal{R}'$ and hence termination of $\mathcal{D}(\mathcal{R}')$ implies termination of $\mathcal{R}'$.

We have seen that requiring $TT(\mathcal{R}) \subseteq \mathcal{R}$ is too restrictive to allow root-labeling. But only requiring $\to_{TT(\mathcal{R})} \subseteq \to_{\mathcal{R}}$ is unsound. However, there is another condition which is weaker than set inclusion, implies soundness, *and* allows the application of flat context closures.

**Definition 9.19.** *The* context subset relation $\subseteq_c$ *is defined as*

$$\mathcal{R} \subseteq_c \mathcal{S} \text{ iff } \forall \ell \to r \in \mathcal{R}.\, \exists C, \ell' \to r' \in \mathcal{S}.\, \ell = C[\ell'] \wedge r = C[r'].$$

**Lemma 9.20.** *(i)* $\mathcal{R} \subseteq \mathcal{S}$ *implies* $\mathcal{R} \subseteq_c \mathcal{S}$

*(ii)* $\mathcal{R} \subseteq_c \mathcal{S}$ *implies* $\to_{\mathcal{R}} \subseteq \to_{\mathcal{S}}$

*(iii)* $\mathcal{R} \subseteq_c \mathcal{S}$ *implies* $\mathcal{D}(\mathcal{R}) \subseteq_c \mathcal{D}(\mathcal{S})$ *and* $\mathcal{U}(\mathcal{R}) \subseteq_c \mathcal{U}(\mathcal{S})$

*(iv) If* $TT$ *is sound and* $\forall \mathcal{R}.\, TT(\mathcal{R}) \subseteq_c \mathcal{R}$ *then* $\mathsf{lift}(TT)$ *is sound*

*Proof.* (i) To show $\mathcal{R} \subseteq_c \mathcal{S}$, let $\ell \to r \in \mathcal{R}$. Using $\mathcal{R} \subseteq \mathcal{S}$ we know that $\ell \to r \in \mathcal{S}$. Hence, $\exists C, \ell' \to r' \in \mathcal{S}.\, \ell = C[\ell'] \wedge r = C[r']$ by choosing $C = \Box$ and $\ell' \to r' = \ell \to r$.

(ii) Assume $t = D[\ell\sigma] \to_{\mathcal{R}} D[r\sigma] = s$ using some rule $\ell \to r \in \mathcal{R}$. As $\mathcal{R} \subseteq_c \mathcal{S}$, we obtain $C$ and $\ell' \to r' \in \mathcal{S}$ such that $\ell = C[\ell']$ and $r = C[r']$. Hence, $t = D[\ell\sigma] = D[C[\ell']\sigma] = D[C\sigma[\ell'\sigma]] \to_{\mathcal{S}} D[C\sigma[r'\sigma]] = D[C[r']\sigma] = D[r\sigma] = s$.

(iii) We first show $\mathcal{D}(\mathcal{R}) \subseteq_c \mathcal{D}(\mathcal{S})$. So, let $\ell \to r \in \mathcal{D}(\mathcal{R})$. Hence, $\ell \to r \in \mathcal{R}$, $\mathsf{unlab}(\ell) = \mathsf{unlab}(r)$ and $\ell \neq r$. Using $\mathcal{R} \subseteq_c \mathcal{S}$ we obtain $C$ and $\ell' \to r' \in \mathcal{S}$ such that $\ell = C[\ell']$ and $r = C[r']$. Thus, $\mathsf{unlab}(C)[\mathsf{unlab}(\ell')] = \mathsf{unlab}(C[\ell']) = \mathsf{unlab}(\ell) = \mathsf{unlab}(r) = \mathsf{unlab}(C[r']) = \mathsf{unlab}(C)[\mathsf{unlab}(r')]$ shows that $\mathsf{unlab}(\ell') = \mathsf{unlab}(r')$. Similarly, $C[\ell'] = \ell \neq r = C[r']$ implies $\ell' \neq r'$. So, $\ell' \to r' \in \mathcal{D}(\mathcal{S})$ and thus, $\exists C, \ell' \to r' \in \mathcal{D}(\mathcal{S}).\, \ell = C[\ell'] \wedge r = C[r']$.

Now let us show $\mathcal{U}(\mathcal{R}) = \mathsf{unlab}(\mathcal{R} \setminus \mathcal{D}(\mathcal{R})) \subseteq_c \mathsf{unlab}(\mathcal{S} \setminus \mathcal{D}(\mathcal{S})) = \mathcal{U}(\mathcal{S})$. This property is the crucial part, since potentially we remove less rules from $\mathcal{R}$ than from $\mathcal{S}$. Assume $\mathsf{unlab}(\ell) \to \mathsf{unlab}(r) \in \mathcal{U}(\mathcal{R})$, i.e., $\ell \to r \in \mathcal{R}$ and $\mathsf{unlab}(\ell) \neq \mathsf{unlab}(r) \vee \ell = r$. As $\mathcal{R} \subseteq_c \mathcal{S}$ we obtain $C$ and $\ell' \to r' \in \mathcal{S}$ such that $\ell = C[\ell']$ and $r = C[r']$. Hence, $\mathsf{unlab}(\ell) = \mathsf{unlab}(C[\ell']) = \mathsf{unlab}(C)[\mathsf{unlab}(\ell')]$ and $\mathsf{unlab}(r) = \mathsf{unlab}(C[r']) = \mathsf{unlab}(C)[\mathsf{unlab}(r')]$. Thus, we can simplify $\mathsf{unlab}(\ell) \neq \mathsf{unlab}(r) \vee \ell = r$ to $\mathsf{unlab}(C)[\mathsf{unlab}(\ell')] \neq \mathsf{unlab}(C)[\mathsf{unlab}(r')] \vee C[\ell'] = C[r']$ and further to $\mathsf{unlab}(\ell') \neq \mathsf{unlab}(r') \vee \ell' = r'$. Using $\ell' \to r' \in \mathcal{S}$ this shows that $\ell' \to r' \in \mathcal{S} \setminus \mathcal{D}(\mathcal{S})$ and thus, $\mathsf{unlab}(\ell') \to \mathsf{unlab}(r') \in \mathcal{U}(\mathcal{S})$. By choosing the context $\mathsf{unlab}(C)$ and the rule $\mathsf{unlab}(\ell') \to \mathsf{unlab}(r')$ we have finally shown that $\exists C, \ell' \to r' \in \mathcal{U}(\mathcal{S}).\, \mathsf{unlab}(\ell) = C[\ell'] \wedge \mathsf{unlab}(r) = C[r']$.

(iv) By Lemma 9.15 we only have to prove ($\star$). Using $TT(\mathcal{R}) \subseteq_c \mathcal{R}$ and **iii** one can show that $\mathcal{U}^m(TT(\mathcal{R})) \subseteq_c \mathcal{U}^m(\mathcal{R})$ by induction on $m$. Using **iii** again, we conclude $\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R}))) \subseteq_c \mathcal{D}(\mathcal{U}^m(\mathcal{R}))$ and thus, $\to_{\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R})))} \subseteq \to_{\mathcal{D}(\mathcal{U}^m(\mathcal{R}))}$ by **ii**. Then ($\star$) immediately follows. □

**Corollary 9.21.** *Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$. Then $\mathsf{lift}(\mathcal{FC}_\mathcal{F})$ is sound.*

*Proof.* It was shown in [95] that $\mathcal{FC}_\mathcal{F}$ is sound for TRSs. Furthermore, $\mathcal{FC}_\mathcal{F}(\mathcal{R}) \subseteq_c \mathcal{R}$ by definition of $\mathcal{FC}(\mathcal{F})$ and thus, by Lemma 9.20(**iv**), $\mathsf{lift}(\mathcal{FC}_\mathcal{F})$ is sound, too. □

Note that several termination techniques $TT$ satisfy $TT(\mathcal{R}) \subseteq_c \mathcal{R}$ and hence, can be used between labeling and unlabeling. However, there are still some techniques which do not satisfy this requirement. Examples would be string reversal and uncurrying [53].

Of course, it is possible to use $\mathsf{lift}_0(TT)$, however, for string reversal also a direct soundness proof can be performed to show that lifting string reversal is sound.

**Theorem 9.22.** *Let $TT$ be the technique of string reversal where $TT(\mathcal{R}) = \mathsf{rev}(\mathcal{R})$, if $\mathcal{R}$ is a string rewrite system, and $TT(\mathcal{R}) = \mathcal{R}$, otherwise. Then $\mathsf{lift}(TT)$ is sound.*

*Proof.* By Lemma 9.15 we just have to prove ($\star$), i.e., we have to show for all $m$ that $\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$ implies $\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathsf{rev}(\mathcal{R}))))$. To this end, we have proven that reversing can be commuted with both $\mathcal{D}$ and $\mathcal{U}$: $\mathsf{rev}(\mathcal{D}(\mathcal{R})) = \mathcal{D}(\mathsf{rev}(\mathcal{R}))$ and $\mathsf{rev}(\mathcal{U}(\mathcal{R})) = \mathcal{U}(\mathsf{rev}(\mathcal{R})$. Hence, $\mathsf{rev}(\mathcal{D}(\mathcal{U}^m(\mathcal{R}))) = \mathcal{D}(\mathcal{U}^m(\mathsf{rev}(\mathcal{R})))$. This completes the proof: since string reversal is complete, we know that termination of $\mathcal{D}(\mathcal{U}^m(\mathcal{R}))$ implies termination of $\mathsf{rev}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$ and therefore, also of $\mathcal{D}(\mathcal{U}^m(\mathsf{rev}(\mathcal{R})))$. □

To summarize, we can now certify termination proofs where labeling and unlabeling are modular techniques (and hence, can be applied recursively), and where all supported techniques of CeTA (except uncurrying) can be used between labeling and unlabeling.

An easy alternative to our extended termination techniques would be the use of relative rewriting. The obvious idea is to add the decreasing rules as relative rules when performing semantic labeling. In this way, unlabeling would directly be modular and sound, since one can always remove relative rules where both sides of the rule are identical. This alternative is used in the independent and unpublished formalization of semantic labeling in the CoLoR library. The main problem with this alternative is that some techniques like RFC matchbounds can be used in our framework, but not in combination with relative rewriting in general (during the termination competition in 2010 a tool has been disqualified for giving a wrong answer for a relative termination problem; the reason was the use of RFC matchbounds). For a further discussion on matchbounds and relative rewriting we refer to [54].

## 9.4. Dependency Pair Framework

The DP framework [42] is a way to modularize termination proofs. Instead of TRSs one investigates so called DP problems, consisting of two TRSs. The *initial DP problem* for a TRS $\mathcal{R}$ is $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$ where $\mathsf{DP}(\mathcal{R})$ denotes the *dependency pairs* of $\mathcal{R}$ [3]. A $(\mathcal{P}, \mathcal{R})$-*chain* is a possibly infinite derivation of the form:

$$s_1\sigma_1 \to_\mathcal{P} t_1\sigma_1 \to^*_\mathcal{R} s_2\sigma_2 \to_\mathcal{P} t_2\sigma_2 \to^*_\mathcal{R} s_3\sigma_3 \to_\mathcal{P} \cdots \qquad (\star)$$

where $s_i \to t_i \in \mathcal{P}$ for all $i > 0$. If additionally every $t_i\sigma_i$ is terminating w.r.t. $\mathcal{R}$, then the chain is *minimal*. A DP problem $(\mathcal{P}, \mathcal{R})$ is called *finite* [42], if there is no minimal infinite $(\mathcal{P}, \mathcal{R})$-chain. Proving finiteness of a DP problem is done by simplifying $(\mathcal{P}, \mathcal{R})$ using so called *processors* recursively. A processor transforms a DP problem into a new DP problem. The aim is to reach a DP problem where the $\mathcal{P}$-component is empty (such DP problems are trivially finite). To conclude finiteness of the initial DP problem, the applied processors need to be *sound*. A processor $\mathcal{P}\mathrm{roc}$ is sound whenever for all DP problems $(\mathcal{P}, \mathcal{R})$ we have that finiteness of $\mathcal{P}\mathrm{roc}(\mathcal{P}, \mathcal{R})$ implies finiteness of $(\mathcal{P}, \mathcal{R})$.

Semantic labeling can easily be lifted to DP problems. Soundness of the following processor is an immediate consequence of [110].

**Theorem 9.23.** *Let $(\mathcal{P}, \mathcal{R})$ be a DP problem and $\mathcal{A}$ be an algebra. If $\mathcal{A}$ is a quasi-model of $\mathcal{R}$, then it is sound to return the DP problem* $(\mathsf{lab}(\mathcal{P}), \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec})$.

The following example shows that unlabeling is not only necessary for efficiency, but that unlabeling is required to apply other techniques.

**Example 9.24.** We consider the TRS `Secret_07/4` from the TPDB.

$$\begin{array}{ll} 1\colon \mathsf{g}(\mathsf{c}, \mathsf{g}(\mathsf{c}, x)) \to \mathsf{g}(\mathsf{e}, \mathsf{g}(\mathsf{d}, x)) & 4\colon \mathsf{g}(x, \mathsf{g}(y, \mathsf{g}(x, y))) \to \mathsf{g}(\mathsf{a}, \mathsf{g}(x, \mathsf{g}(y, \mathsf{b}))) \\ 2\colon \mathsf{g}(\mathsf{d}, \mathsf{g}(\mathsf{d}, x)) \to \mathsf{g}(\mathsf{c}, \mathsf{g}(\mathsf{e}, x)) & 5\colon \quad \mathsf{f}(\mathsf{g}(x, y)) \to \mathsf{g}(y, \mathsf{g}(\mathsf{f}(\mathsf{f}(x)), \mathsf{a})) \\ 3\colon \mathsf{g}(\mathsf{e}, \mathsf{g}(\mathsf{e}, x)) \to \mathsf{g}(\mathsf{d}, \mathsf{g}(\mathsf{c}, x)) & \end{array}$$

In the 2008 termination competition AProVE found a termination proof of the following structure (we present a simplified version, missing some unnecessary steps that have been applied in the original proof).[4] First, the initial DP problem is transformed into $(\mathcal{P}, \{1\text{–}4\})$ where $\mathcal{P}$ consists of the pairs $\mathsf{G}(\mathsf{c}, \mathsf{g}(\mathsf{c}, x)) \to \mathsf{G}(\mathsf{e}, \mathsf{g}(\mathsf{d}, x))$, $\mathsf{G}(\mathsf{d}, \mathsf{g}(\mathsf{d}, x)) \to \mathsf{G}(\mathsf{c}, \mathsf{g}(\mathsf{e}, x))$, and $\mathsf{G}(\mathsf{e}, \mathsf{g}(\mathsf{e}, x)) \to \mathsf{G}(\mathsf{d}, \mathsf{g}(\mathsf{c}, x))$. Then, labeling and further processing yields the DP problem $(\mathcal{P}', \mathcal{R}')$ where $\mathcal{P}'$ contains the pairs

$$\begin{array}{ll} \mathsf{G}_{00}(\mathsf{c}, \mathsf{g}_{00}(\mathsf{c}, x)) \to \mathsf{G}_{00}(\mathsf{e}, \mathsf{g}_{00}(\mathsf{d}, x)) & \mathsf{G}_{00}(\mathsf{e}, \mathsf{g}_{00}(\mathsf{e}, x)) \to \mathsf{G}_{00}(\mathsf{d}, \mathsf{g}_{00}(\mathsf{c}, x)) \\ \mathsf{G}_{00}(\mathsf{d}, \mathsf{g}_{00}(\mathsf{d}, x)) \to \mathsf{G}_{00}(\mathsf{c}, \mathsf{g}_{00}(\mathsf{e}, x)) & \end{array}$$

and $\mathcal{R}'$ is the following TRS.

$$\begin{array}{ll} \mathsf{g}_{00}(\mathsf{c}, \mathsf{g}_{00}(\mathsf{c}, x)) \to \mathsf{g}_{00}(\mathsf{e}, \mathsf{g}_{00}(\mathsf{d}, x)) & \mathsf{g}_{00}(\mathsf{c}, \mathsf{g}_{01}(\mathsf{c}, x)) \to \mathsf{g}_{00}(\mathsf{e}, \mathsf{g}_{01}(\mathsf{d}, x)) \\ \mathsf{g}_{00}(\mathsf{d}, \mathsf{g}_{00}(\mathsf{d}, x)) \to \mathsf{g}_{00}(\mathsf{c}, \mathsf{g}_{00}(\mathsf{e}, x)) & \mathsf{g}_{00}(\mathsf{d}, \mathsf{g}_{01}(\mathsf{d}, x)) \to \mathsf{g}_{00}(\mathsf{c}, \mathsf{g}_{01}(\mathsf{e}, x)) \\ \mathsf{g}_{00}(\mathsf{e}, \mathsf{g}_{00}(\mathsf{e}, x)) \to \mathsf{g}_{00}(\mathsf{d}, \mathsf{g}_{00}(\mathsf{c}, x)) & \mathsf{g}_{00}(\mathsf{e}, \mathsf{g}_{01}(\mathsf{e}, x)) \to \mathsf{g}_{00}(\mathsf{d}, \mathsf{g}_{01}(\mathsf{c}, x)) \end{array}$$

Hence, all labeled versions of Rule 4 have been deleted, and unlabeling yields the DP problem $(\mathcal{P}, \{1\text{–}3\})$. This DP problem is applicative. Hence, we may apply the $\mathcal{A}$-transformation [41] to obtain the DP problem having the pairs

$$\mathsf{C}(\mathsf{c}(x)) \to \mathsf{E}(\mathsf{d}(x)) \qquad \mathsf{D}(\mathsf{d}(x)) \to \mathsf{C}(\mathsf{e}(x)) \qquad \mathsf{E}(\mathsf{e}(x)) \to \mathsf{D}(\mathsf{c}(x))$$

and the rules

$$\mathsf{c}(\mathsf{c}(x)) \to \mathsf{e}(\mathsf{d}(x)) \qquad \mathsf{d}(\mathsf{d}(x)) \to \mathsf{c}(\mathsf{e}(x)) \qquad \mathsf{e}(\mathsf{e}(x)) \to \mathsf{d}(\mathsf{c}(x))$$

This DP problem is solved using standard techniques. Note that for the $\mathcal{A}$-transformation it was essential that unlabeling was performed, as the DP problem $(\mathcal{P}', \mathcal{R}')$ is not applicative.

[4] See `http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=35909`

Unfortunately, unlabeling as processor is in general unsound. In contrast to unlabeling on TRSs, here a problem already arises when using the model-version of semantic labeling without decreasing rules. The main reason is that unlabeling might introduce nontermination. Hence, minimality of an unlabeled infinite chain cannot be guaranteed.[5]

**Example 9.25.** Consider the DP problem $(\mathcal{P}, \varnothing)$ where $\mathcal{P} = \{\mathsf{F}(x) \to \mathsf{F}(\mathsf{g}(\mathsf{a}))\}$. This DP problem is obviously not finite. Applying semantic labeling is trivially possible since there are no rules which have to satisfy the (quasi-)model condition. We choose $A = \{0, 1\}$, and for each $f$ we define $f_{\mathcal{A}}(\ldots) = 0$ and $\ell_f(\vec{x}_n) = (\vec{x}_n)$. We obtain the labeled pairs $\mathsf{lab}(\mathcal{P}) = \{\mathsf{F}_0(x) \to \mathsf{F}_0(\mathsf{g}_0(\mathsf{a})), \mathsf{F}_1(x) \to \mathsf{F}_0(\mathsf{g}_0(\mathsf{a}))\}$ and by Theorem 9.23 we know that the DP problem $(\mathsf{lab}(\mathcal{P}), \varnothing)$ is again not finite. We can further modify the DP problem by replacing it with $(\mathsf{lab}(\mathcal{P}), \mathcal{R})$ where $\mathcal{R} = \{\mathsf{g}_1(x) \to \mathsf{g}_1(x)\}$. Note that this modification is sound since $(\mathsf{lab}(\mathcal{P}), \mathcal{R})$ still allows a minimal infinite chain and is therefore not finite.

However, applying unlabeling we obtain the DP problem $(\mathcal{P}, \mathsf{unlab}(\mathcal{R}))$ which is finite as now the right-hand side $\mathsf{F}(\mathsf{g}(\mathsf{a}))$ of the only pair in $\mathcal{P}$ is not terminating w.r.t. $\mathcal{U}(\mathcal{R}) = \{\mathsf{g}(x) \to \mathsf{g}(x)\}$. Hence, unlabeling is unsound in general. The main problem is again that the notion of soundness is too weak. It allows the application of processors between labeling and unlabeling which may replace $(\mathsf{lab}(\mathcal{P}), \varnothing)$ by $(\mathsf{lab}(\mathcal{P}), \mathcal{R})$.

To solve this problem, we again add a counter $n$ which tells us how often we may unlabel.

**Definition 9.26.** *An* extended DP problem *is a triple $(\mathcal{P}, \mathcal{R}, n)$ where $(\mathcal{P}, \mathcal{R})$ is a DP problem and $n \in \mathbb{N}$. An extended DP problem $(\mathcal{P}, \mathcal{R}, n)$ is* finite *iff there is no infinite chain*

$$s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to^*_{\mathcal{R}} s_3\sigma_3 \to_{\mathcal{P}} t_3\sigma_3 \to^*_{\mathcal{R}} \cdots$$

*such that for all $i$: $\forall m \leqslant n.\, \mathsf{SN}_{\mathcal{U}^m(\mathcal{R})}(\mathsf{unlab}^m(t_i\sigma_i))$.*

Hence, the only difference between finiteness of DP problems and extended DP problems is the minimality condition ($\mathsf{SN}_{\mathcal{R}}(t_i\sigma_i)$ versus $\forall m \leqslant n.\, \mathsf{SN}_{\mathcal{U}^m(\mathcal{R})}(\mathsf{unlab}^m(t_i\sigma_i))$). We therefore obtain a similar lemma to Lemma 9.10, but now for DP problems.

**Lemma 9.27.** *(i) $(\mathcal{P}, \mathcal{R})$ is finite iff $(\mathcal{P}, \mathcal{R}, 0)$ is finite.*

*(ii) If $(\mathcal{P}, \mathcal{R})$ is finite then $(\mathcal{P}, \mathcal{R}, n)$ is finite.*

As for termination techniques we can lift every processor to an extended processor.

**Definition 9.28** (Lift)**.** *Let $\mathcal{P}$roc be a processor with $\mathcal{P}\mathrm{roc}(\mathcal{P}, \mathcal{R}) = (\mathcal{P}', \mathcal{R}')$. Then $\mathsf{lift}(\mathcal{P}\mathrm{roc})$ and $\mathsf{lift}_0(\mathcal{P}\mathrm{roc})$ are extended processors where $\mathsf{lift}(\mathcal{P}\mathrm{roc})(\mathcal{P}, \mathcal{R}, n) = (\mathcal{P}', \mathcal{R}', n)$ and $\mathsf{lift}_0(\mathcal{P}\mathrm{roc})(\mathcal{P}, \mathcal{R}, n) = (\mathcal{P}', \mathcal{R}', 0)$.*

We obtain similar results for $\mathsf{lift}_0$ as for termination techniques: whenever $\mathcal{P}$roc is sound then $\mathsf{lift}_0(\mathcal{P}\mathrm{roc})$ is sound. However, additionally demanding that $\mathcal{R}' \subseteq_c \mathcal{R}$ or even $\mathcal{P}' \subseteq \mathcal{P} \wedge \mathcal{R}' = \mathcal{R}$ where $\mathcal{P}\mathrm{roc}(\mathcal{P}, \mathcal{R}) = (\mathcal{P}', \mathcal{R}')$ does not suffice to ensure soundness of $\mathsf{lift}(\mathcal{P}\mathrm{roc})$. This is demonstrated in the upcoming example.

[5] There is no problem in the formalization of semantic labeling in CoLoR at this point, as it does not feature *minimal* chains.

**Example 9.29.** Let $\mathcal{P} = \{\mathsf{F}_0(x) \to \mathsf{F}_0(\mathsf{b})\}$, $\mathcal{P}' = \{\mathsf{F}_0(x) \to \mathsf{F}_0(\mathsf{g}_0(\mathsf{b}))\}$, and $\mathcal{R} = \{\mathsf{g}_1(x) \to \mathsf{g}_0(\mathsf{h}_1(x))\}$. Then $(\mathcal{P}, \mathcal{R}, 1)$ is not finite as obviously there is an infinite $(\mathcal{P}, \mathcal{R})$-chain where all terms in the chain are $\mathsf{F}_0(\mathsf{b})$ and moreover, $\mathsf{F}_0(\mathsf{b})$ is terminating w.r.t. $\mathcal{R}$ and $\mathsf{unlab}(\mathsf{F}_0(\mathsf{b})) = \mathsf{F}(\mathsf{b})$ is terminating w.r.t. $\mathcal{U}(\mathcal{R}) = \{\mathsf{g}(x) \to \mathsf{g}(\mathsf{h}(x))\}$. Hence, also $(\mathcal{P} \cup \mathcal{P}', \mathcal{R}, 1)$ is not finite by constructing the same chain.

Note that the processor $\mathcal{P}\mathrm{roc}$ which replaces $(\mathcal{P} \cup \mathcal{P}', \mathcal{R})$ by $(\mathcal{P}', \mathcal{R})$ is sound, since $(\mathcal{P}', \mathcal{R})$ is not finite: again, there is an infinite $(\mathcal{P}', \mathcal{R})$-chain, and every chain is also minimal since $\mathcal{R}$ is terminating. However, $\mathsf{lift}(\mathcal{P}\mathrm{roc})$ is unsound as $(\mathcal{P}', \mathcal{R}, 1)$ is finite: otherwise, there would be an infinite chain where $\mathsf{F}_0(\mathsf{g}_0(\mathsf{b}))$ is terminating w.r.t. $\mathcal{R}$ and $\mathsf{unlab}(\mathsf{F}_0(\mathsf{g}_0(\mathsf{b}))) = \mathsf{F}(\mathsf{g}(\mathsf{b}))$ is terminating w.r.t. $\mathcal{U}(\mathcal{R})$. But it is easy to see that $\mathsf{F}(\mathsf{g}(\mathsf{b}))$ is not terminating w.r.t. $\mathcal{U}(\mathcal{R})$.

Since requiring just $\mathcal{R}' \subseteq_c \mathcal{R}$ (or even $\mathcal{P}' \subseteq \mathcal{P} \wedge \mathcal{R}' = \mathcal{R}$) does not suffice to ensure soundness of $\mathsf{lift}(\mathcal{P}\mathrm{roc})$ we demand a slightly stronger property than soundness.

**Definition 9.30.** *A processor* $\mathcal{P}\mathrm{roc}$ *is* chain-identifying *iff whenever* $\mathcal{P}\mathrm{roc}(\mathcal{P}, \mathcal{R}) = (\mathcal{P}', \mathcal{R}')$ *and there is some minimal infinite* $(\mathcal{P}, \mathcal{R})$*-chain*

$$s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to^*_{\mathcal{R}} s_3\sigma_3 \to_{\mathcal{P}} t_3\sigma_3 \to^*_{\mathcal{R}} \cdots$$

*then* $\mathcal{R}' \subseteq_c \mathcal{R}$ *and there is some* $k$ *such that*

$$s_k\sigma_k \to_{\mathcal{P}'} t_k\sigma_k \to^*_{\mathcal{R}'} s_{k+1}\sigma_{k+1} \to_{\mathcal{P}'} t_{k+1}\sigma_{k+1} \to^*_{\mathcal{R}'} s_{k+2}\sigma_{k+2} \to_{\mathcal{P}'} t_{k+2}\sigma_{k+2} \to^*_{\mathcal{R}'} \cdots$$

*is an infinite* $(\mathcal{P}', \mathcal{R}')$*-chain.*

Chain-identifying processors ensure that every minimal infinite chain of $(\mathcal{P}, \mathcal{R})$ has an infinite tail where $\mathcal{R}^*$-steps can be replaced by $\mathcal{R}'^*$-steps and all pairs are from $\mathcal{P}'$. Note that every chain-identifying processor is sound. Moreover, several processors are indeed chain-identifying. Some examples are the reduction pair processor, the dependency graph processor, and all standard processors which just remove pairs and rules. The following lemma shows that chain-identifying processors can be used as extended processors via $\mathsf{lift}$.

**Lemma 9.31.** *(i) If* $\mathcal{P}\mathrm{roc}$ *is sound, then* $\mathsf{lift}_0(\mathcal{P}\mathrm{roc})$ *is sound.*

*(ii) If* $\mathcal{P}\mathrm{roc}$ *is chain-identifying then* $\mathsf{lift}(\mathcal{P}\mathrm{roc})$ *is sound.*

*Proof.* Let $\mathcal{P}$, $\mathcal{R}$, $\mathcal{P}'$, and $\mathcal{R}'$ be given such that $\mathcal{P}\mathrm{roc}(\mathcal{P}, \mathcal{R}) = (\mathcal{P}', \mathcal{R}')$.

(i) We assume that $(\mathcal{P}', \mathcal{R}', 0)$ is finite and have to show that $(\mathcal{P}, \mathcal{R}, n)$ is finite. By Lemma 9.27(**i**) and the assumption we know that $(\mathcal{P}', \mathcal{R}')$ is finite. Thus, also $(\mathcal{P}, \mathcal{R})$ is finite using the soundness of $\mathcal{P}\mathrm{roc}$. By Lemma 9.27(**ii**) we conclude finiteness of $(\mathcal{P}, \mathcal{R}, n)$.

(ii) Here, we may assume that $(\mathcal{P}', \mathcal{R}', n)$ is finite and have to show that $(\mathcal{P}, \mathcal{R}, n)$ is finite. We show finiteness of $(\mathcal{P}, \mathcal{R}, n)$ via contraposition. So, assume $(\mathcal{P}, \mathcal{R}, n)$ is not finite. This shows that there is an infinite $(\mathcal{P}, \mathcal{R})$-chain

$$s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to^*_{\mathcal{R}} s_3\sigma_3 \to_{\mathcal{P}} t_3\sigma_3 \to^*_{\mathcal{R}} \cdots$$

such that for all $i$ we have $\forall m \leqslant n.\, \mathsf{SN}_{\mathcal{U}^m(\mathcal{R})}(\mathsf{unlab}^m(t_i\sigma_i))$. By choosing $m = 0$ we also have $\mathsf{SN}_{\mathcal{R}}(t_i\sigma_i)$ for all $i$. Hence, the chain is also a minimal infinite $(\mathcal{P}, \mathcal{R})$-chain. Since $\mathcal{P}\mathrm{roc}$ is chain-identifying we know that $\mathcal{R}' \subseteq_c \mathcal{R}$ and there is some $k$ such that

$$s_k\sigma_k \to_{\mathcal{P}'} t_k\sigma_k \to^*_{\mathcal{R}'} s_{k+1}\sigma_{k+1} \to_{\mathcal{P}'} t_{k+1}\sigma_{k+1} \to^*_{\mathcal{R}'} s_{k+2}\sigma_{k+2} \to_{\mathcal{P}'} t_{k+2}\sigma_{k+2} \to^*_{\mathcal{R}'} \cdots$$

is an infinite $(\mathcal{P}', \mathcal{R}')$-chain. We continue to prove that for every $i$ and every $m \leqslant n$ we have $\mathsf{SN}_{\mathcal{U}^m(\mathcal{R}')}(\mathsf{unlab}^m(t_i\sigma_i))$. This leads to the desired contradiction, since then we have shown that $(\mathcal{P}', \mathcal{R}', n)$ is not finite.

To prove $\mathsf{SN}_{\mathcal{U}^m(\mathcal{R}')}(\mathsf{unlab}^m(t_i\sigma_i))$ we first use minimality of the $(\mathcal{P}, \mathcal{R})$-chain to conclude $\mathsf{SN}_{\mathcal{U}^m(\mathcal{R})}(\mathsf{unlab}^m(t_i\sigma_i))$. Then the result immediately follows since the rewrite relation of $\mathcal{U}^m(\mathcal{R}')$ is a subset of the rewrite relation of $\mathcal{U}^m(\mathcal{R})$ by Lemma 9.20, **ii** and **iii**. □

Using these results allowed us to develop the first certified proof of the TRS in Example 9.24. We only had to change the given proof such that uncurrying [53] is used instead of the $\mathcal{A}$-transformation, since we have only formalized the former technique. The detailed proof is provided in the IsaFoR/CeTA repository.[6]

However, unlike for TRSs, root-labeling is not directly supported as root-labeling on DP problems [95, 97] is not a chain-identifying processor. Here again, root-labeling itself is not the problem, but making sure that the fixed algebra is a model of $\mathcal{R}$, which is again done by closing under flat contexts. In the DP framework we need the auxiliary function $\mathsf{block}_\triangle$, given by the equations $\mathsf{block}_\triangle(f(\vec{t}_n)) = f(\triangle(\vec{t}_n))$ and $\mathsf{block}_\triangle(x) = x$.

**Definition 9.32** (Flat Context Closure)**.** *Let $(\mathcal{P}, \mathcal{R})$ be a DP problem such that $\mathcal{R}$ is left-linear and $\mathcal{F}$ is a superset of the signature of $\mathcal{R}$ combined with the non-root symbols of $\mathcal{P}$. Furthermore, let $\triangle$ be a function symbol not in $\mathcal{F}$. Then the closure under flat contexts of $(\mathcal{P}, \mathcal{R})$ is given by $\mathcal{FC}_\mathcal{F}(\mathcal{P}, \mathcal{R}) = (\mathsf{block}_\triangle(\mathcal{P}), \mathcal{FC}_{\{\triangle\}\cup\mathcal{F}}(\mathcal{R}))$.*

As the pairs of a DP problem are modified, we do not get soundness of $\mathsf{lift}(\mathcal{FC}_\mathcal{F})$ via Lemma 9.31. Nevertheless, by using the definition of finiteness of extended DP problems and providing a manual proof one can show that $\mathsf{lift}(\mathcal{FC}_\mathcal{F})$ is indeed sound.

## 9.5. Problems in Certification

We present three problems that arose when trying to certify proofs with semantic labeling.

The first problem for the certifier is that internally it only works on extended termination/DP problems, whereas in the provided proofs just TRSs and DP problems are given without the additional numbers. However, this problem is fixed by computing the number during certification. This is easy and seems to be a safe solution: the format for termination proofs remains unchanged, and so far no termination proof was refused with the reason that the internal computation of the number was wrong.

The second and third problem are concerned with how semantic labeling is applied, since usually variations of Lemma 9.12 and Theorem 9.23 are used in termination provers.

The second problem occurs for TRSs as well as DP problems. The theory about semantic labeling demands that $\mathcal{D}\mathrm{ec}$ is added to the new TRS when using quasi-models. However, termination provers typically reduce the set of rules and "optimize" semantic labeling by only adding rules $\mathcal{D}\mathrm{ec}'$ such that $\to_{\mathcal{D}\mathrm{ec}} \subseteq \to^+_{\mathcal{D}\mathrm{ec}'}$.

For example, if $L_f = \{0, 1, 2\}$ and the order is the standard order on the naturals, then $\mathcal{D}\mathrm{ec} = \{f_2(x) \to f_1(x), f_1(x) \to f_0(x), f_2(x) \to f_0(x)\}$. However, the last rule is often omitted since it can be simulated by the previous two rules. To certify these termination proofs, we fist need to show that we may safely replace $\mathcal{D}\mathrm{ec}$ by any $\mathcal{D}\mathrm{ec}'$

[6] See `http://cl2-informatik.uibk.ac.at/rewriting/mercurial.cgi/IsaFoR/raw-file/v1.16/examples/secret_07_trs_4_top.proof.xml`

where $\to_{\mathcal{D}\mathrm{ec}} \subseteq \to^+_{\mathcal{D}\mathrm{ec}'}$. Moreover, we have to provide a certified algorithm which for a given TRS $\mathcal{D}\mathrm{ec}'$ and a given order can ensure that the condition $\to_{\mathcal{D}\mathrm{ec}} \subseteq \to^+_{\mathcal{D}\mathrm{ec}'}$ is satisfied. Furthermore, the algorithm should accept *all* $\mathcal{D}\mathrm{ec}'$ where the condition is satisfied.

The third problem only occurs when dealing with quasi-models in the DP framework. Note that in standard DP problems the roots of $\mathcal{P}$ are special symbols (tuple symbols) which do not occur in the remaining DP problem. However, when applying Theorem 9.23 as it is, this invariant is destroyed since the decreasing rules for tuple symbols are added as new rules. We illustrate the problem and two possible solutions in the following example.

**Example 9.33.** Consider a DP problem $(\mathcal{P}, \mathcal{R})$ where $\mathsf{F}(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}(x, \mathsf{b}) \in \mathcal{P}$ and $\mathsf{b}$ is not defined in $\mathcal{R}$. Then, the dependency graph estimation EDG [3] can detect that there is no connection from the mentioned pair to itself. However, when performing labeling with a quasi-model where $\mathsf{s}(x)$ is interpreted as $\min(x+1, 2)$ and where $\ell_{\mathsf{F}}(x, y) = x$ then for the mentioned pair we get all three rules $\{6, 8, 10\}$ in the labeled pairs $\mathcal{P}'$ and the decreasing rules for $\mathsf{F}$ are $\mathcal{D}\mathrm{ec}_{\mathsf{F}} = \{7, 9, 11\}$.

| | | |
|---|---|---|
| 6: $\mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_2(x, \mathsf{b})$ | 8: $\mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_1(x, \mathsf{b})$ | 10: $\mathsf{F}_1(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_0(x, \mathsf{b})$ |
| 7: $\mathsf{F}_2(x, y) \to \mathsf{F}_1(x, y)$ | 9: $\mathsf{F}_2(x, y) \to \mathsf{F}_0(x, y)$ | 11: $\mathsf{F}_1(x, y) \to \mathsf{F}_0(x, y)$ |

Note that when adding $\mathcal{D}\mathrm{ec}_{\mathsf{F}}$ as new rules, then the EDG contains an edge from $\mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_1(x, \mathsf{b})$ to all other pairs since $\mathsf{F}_1$ is defined in $\mathcal{D}\mathrm{ec}_{\mathsf{F}}$. Hence, this is not the preferred way to add decreasing rules: not even the decrease in the labels is recognized.

One solution is to add $\mathcal{D}\mathrm{ec}_{\mathsf{F}}$ as new pairs. Then one obtains a standard DP problem and the decrease in the labels is reflected in the EDG. But there still is a path from $\mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_2(x, \mathsf{b})$ to $\mathsf{F}_1(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_0(x, \mathsf{b})$ via the pair $\mathsf{F}_2(x, y) \to \mathsf{F}_1(x, y)$, since the information that the second argument of $\mathsf{F}_n$ is $\mathsf{b}$ is lost when passing the pair $\mathsf{F}_2(x, y) \to \mathsf{F}_1(x, y)$.

To encounter this problem, there is another solution where $\mathcal{D}\mathrm{ec}_{\mathsf{F}}$ is not produced at all, but where the labels of all tuple-symbols in right-hand sides of $\mathcal{P}'$ are decreased. In this example, one would have to add the additional pair $\mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_0(x, \mathsf{b})$ to $\mathcal{P}'$.

Hence, termination proofs might have used one of the two variants instead of Theorem 9.23. Here, the first variant returns the problem $(\mathsf{lab}(\mathcal{P}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}^\sharp}, \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}})$ and the second variant returns $(\mathsf{lab}(\mathcal{P})^{\geq}, \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}})$ where $\mathcal{D}\mathrm{ec}_{\mathcal{F}^\sharp}$ are the decreasing rules for all tuple symbols, $\mathcal{D}\mathrm{ec}_{\mathcal{F}}$ are the decreasing rules for the remaining symbols, and $\mathsf{lab}(\mathcal{P})^{\geq} = \{s \to f_{\ell'}(\vec{t}) \mid s \to f_\ell(\vec{t}) \in \mathsf{lab}(\mathcal{P}), \ell \geq_{L_f} \ell'\}$.

To certify these termination proofs the problem was mainly in formalizing that these variants of Theorem 9.23 are indeed sound.

**Theorem 9.34.** *Both variants of Theorem 9.23 are sound, provided that they are applied on DP problems $(\mathcal{P}, \mathcal{R})$ where neither left- nor right-hand sides of $\mathcal{P}$ are variables and the roots of $\mathcal{P}$ are distinguished tuple symbols which do not occur in the remaining DP problem.*

We shortly describe the proof idea. The main problem is that we cannot w.l.o.g. restrict the substitutions in a chain such that they do not contain tuple symbols [97]. Thus, we may have to apply rules in $\mathcal{D}\mathrm{ec}_{\mathcal{F}^\sharp}$ also below the root, in order to simulate a reduction $t_i\sigma_i \to^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$. The trick is to introduce a second set of labels and labeling functions for the tuple symbols. The new labeling functions label all tuple symbols by the same element. Hence, no decreasing rules are required for them (w.r.t. the second set of labeling functions) and on all other symbols the labeling functions coincide.

Afterwards, we use a combined labeling of terms: The root of the term is labeled according to the original function, and below the root it is labeled w.r.t. the second labeling function. In this way no decreasing rules for the tuple symbols have to be applied below the root and moreover, on all terms in the DP problem, the original and the combined labeling produce the same result. Thus, we can transform a given $(\mathcal{P}, \mathcal{R})$-chain into a $(\mathsf{lab}(\mathcal{P}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}^\sharp}, \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}})$-chain. Theorem 9.34 easily follows.

To summarize, we discussed some problems which occurred when trying to certify existing proofs which are mainly due to optimizations of the basic semantic labeling theorems. Of course, we also need to check the model condition, whether the orders are weakly monotone when using quasi-orders, etc. Whereas the general theorems about soundness of semantic labeling have been formalized for arbitrary carriers, for the certification we currently only support finite carriers. Then checking the required conditions is performed via enumerating all possible assignments.

In total, our formalization of pure semantic labeling consists of 3300 lines of Isabelle, where roughly half of it is about semantic labeling on generic algebras, and the other half contains executable functions for the certifier using algebras over finite carriers and soundness proofs for these functions. Moreover, the theory about the semantic labeling framework with extended termination techniques, extended DP problems, etc., consists of another 1000 lines.

## 9.6. Experiments

To test the impact of our formalization we ran AProVE on the TPDB (version 8.0), considering all 2795 TRSs. We used two different strategies which are similar to the strategy CERT that was used during the 2010 termination competition in the certified termination category: −SL is like CERT but with semantic labeling removed, and +SL is like CERT including all three variants of semantic labeling that are supported by AProVE (root-labeling, semantic-labeling on finite carriers with models and quasi-models).

We performed all our experiments on a machine with two 2.8 GHz Quad-Core Intel Xeon processors and 6 GB of main memory. The following results where obtained using a 60 seconds timeout.

| | −SL | +SL | total |
|---|---|---|---|
| termination proofs | 1137 | 1207 | 1227 |
| nontermination proofs | 225 | 218 | 227 |
| total time (in minutes) | 1186 | 1219 | |
| certification time (in minutes) | 1 | 3 | |

CeTA (version 1.17) certified all but two proofs. On one TRS, both −SL and +SL delivered a faulty proof, caused by a bug in the LPO output of AProVE (which will be fixed soonish).

The results show that by using semantic labeling we obtain 90 new certified termination proofs. This is an increase of nearly 8 %. Note that +SL has not solved all TRSs where −SL was successful. This is due to timing issues in the strategy.

## 9.7. Conclusion

During our formalization of semantic labeling we have detected that unlabeling is unsound when using the current semantics of termination problems. We solved the problem

by extending termination problems and the DP framework such that recursive labeling and unlabeling are supported, as well as all other existing termination techniques. This framework forms the semantic basis of our certifier CeTA which now fully supports semantic labeling.

**Acknowledgments** We thank Christian Kuknat and Carsten Fuhs for their support in providing certifiable proofs with semantic labeling generated by AProVE.

# 10. Termination of Isabelle Functions via Termination of Rewriting

### Publication details

Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle Functions via Termination of Rewriting. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving*, volume 6898 of LNCS, pages 152–167. Springer, 2011.

### Abstract

We show how to automate termination proofs for recursive functions in (a first-order subset of) Isabelle/HOL by encoding them as term rewrite systems and invoking an external termination prover. Our link to the external prover includes full proof reconstruction, where all necessary properties are derived inside Isabelle/HOL without oracles. Apart from the certification of the imported proof, the main challenge is the formal reduction of the proof obligation produced by Isabelle/HOL to the termination of the corresponding term rewrite system. We automate this reduction via suitable tactics which we added to the IsaFoR library.

## 10.1. Introduction

In a proof assistant based on higher-order logic (HOL), such as Isabelle/HOL [84], recursive function definitions typically require a termination proof. To release the user from finding suitable termination arguments manually, it is desirable to automate these termination proofs as much as possible.

There have already been successful approaches to port and adapt existing termination techniques from term rewriting and other areas to Isabelle [18, 68]. They indeed increase the degree of automation for termination proofs of HOL functions. However, these approaches do not cover all powerful techniques that have been developed in term rewriting, e.g., [32, 110]. These techniques are implemented in a number of termination tools (e.g., AProVE [39], T$_{\text{T}}$T$_2$ [67] and many others) that can show termination of (first-order) term rewrite systems (TRSs) automatically. (In the remainder we use 'termination tool' exclusively to refer to such fully automatic and external provers.) Instead of porting further proof techniques to Isabelle, we prefer to use the existing termination tools, giving direct access to an abundance of methods and their efficient implementations.

Using termination tools inside proof assistants has been an open problem for some time and is often mentioned as future work when discussing certification of termination proofs [13, 21]. However, this requires more than a communication interface between two programs. In LCF-style proof assistants [44] such as Isabelle, all proofs must be checked by a small trusted kernel. Thus, integrating external tools as unverified oracles is

unsatisfactory: any error in the external tool or in the integration code would compromise the overall soundness. Instead, the external tool must provide a certificate that can be checked by the proof assistant.

Our approach involves the following steps.

(i) Generate the definition of a TRS $\mathcal{R}^f$ which corresponds to the function $f$.

(ii) Prove that termination of $\mathcal{R}^f$ indeed implies the termination goal for $f$.

(iii) Run the termination tool on $\mathcal{R}^f$ and obtain a certificate.

(iv) Replay the certificate using a formally verified checker.

While steps 1 and 3 are not hard, and the ground work for step 4 is already available in the IsaFoR library [94, 104], which formalizes term rewriting and several termination techniques,[1] this paper is concerned with the missing piece, the reduction of termination proof obligations for HOL functions to the termination of a TRS. This is non-trivial, as the languages differ considerably. Termination of a TRS expresses the well-foundedness of a relation over terms, i.e., of type $(term \times term)\ set$, where *term*s are first-order terms. In contrast, the termination proof obligation for a HOL function states the well-foundedness of its call relation, which has the type $(\alpha \times \alpha)\ set$, where $\alpha$ is the argument type of the function. In essence, we must move from a shallow embedding (the functional programming fragment of Isabelle/HOL) to a deep embedding (the formalization of term rewriting in IsaFoR).

The goal of this paper is to provide this formal relationship between termination of first-order HOL functions and termination of TRSs. More precisely, we develop a tactic that automatically reduces the termination proof obligation of a HOL function to the termination problem of a TRS. This allows us to use arbitrary termination tools for fully automated termination proofs inside Isabelle. Thus, powerful termination tools become available to the Isabelle user, while retaining the strong soundness guarantees of an LCF-style proof assistant. Since our approach is generic, it automatically benefits from future improvements to termination tools and the termination techniques within IsaFoR. Our implementation is available as part of IsaFoR.

**Outline of this paper.** We give a short introduction on term rewriting, HOL and HOL functions in §10.2. Then we show our main result in §10.3 on how to systematically discharge the termination proof obligation of a HOL function via proving termination of a TRS. In §10.4 we present some examples which show the strengths and limitations of our technique. How to extend our approach to support more HOL functions is discussed in §10.5. We conclude in §10.6.

## 10.2. Preliminaries

### 10.2.1. Higher-Order Logic

We consider classical HOL, which is based on simply-typed lambda-calculus, enriched with a simple form of ML-like polymorphism. Among its basic types are a type *bool* of truth values and a function space type constructor $\Rightarrow$ (where $\alpha \Rightarrow \beta$ denotes the type of

[1] See http://cl-informatik.uibk.ac.at/software/ceta for a list of supported techniques.

total functions mapping values of type $\alpha$ to values of type $\beta$). Sets are modeled by a type $\alpha$ *set*, which just abbreviates $\alpha \Rightarrow$ *bool*.

By an add-on tool, HOL supports algebraic datatypes, which includes the types *nat* (with constructors 0 and *Suc*) and *list* (with constructors [] and #).

Another add-on tool, the *function package* [69], completes the functional programming layer by allowing recursive function definitions, which are not covered by the primitives of the logic. Since it internally employs a well-founded recursion principle, it requires the user to prove well-foundedness of a certain relation, extracted automatically from the function definition (cf. §10.2.3). This proof obligation, by its construction, directly corresponds to the termination of the function being defined. It is the proof of this goal that we want to automate.

As opposed to functional programming languages, there is no operational semantics for HOL; the meaning of its expressions is instead given by a set-theoretic denotational semantics. As a consequence, there is no direct notion of evaluation or termination of an expression. Thus, when we informally say that we prove "termination of a HOL function," this simply means that we discharge the proof obligation produced by the function package.

## 10.2.2. Supported Fragment

Isabelle supports a wide spectrum of specifications, using various forms of inductive, coinductive and recursive definitions, as well as quantifiers and Hilbert's choice operator. Clearly, not all of them can be easily expressed using TRSs. Thus, we must limit ourselves to a subset which is sufficiently close to rewriting, and consider only algebraic datatypes, given by a set of constructors together with their types, and recursive functions, given by their defining equations with pattern matching. Additionally, we impose the following restrictions:

(i) Functions and constructors must be first-order (no functions as arguments).

(ii) Patterns are constructor terms and must be linear and non-overlapping.

(iii) Patterns must be complete.

(iv) Expressions consist of variables, function applications, and case-expressions only. In particular, partial applications and $\lambda$-abstractions are excluded.

Linearity is always satisfied by function definitions that are accepted by Isabelle's function package, and pattern overlaps are eliminated automatically. For ease of presentation, we assume that there is no mutual recursion ($f$ calls $g$ and $g$ calls $f$) and no nested recursion (arguments of a recursive call contain other recursive calls; they may of course contain calls to other defined functions).

Most of the above restrictions are not fundamental, and we discuss in §10.5 how some of them can be removed. Our chosen fragment of HOL rather represents a compromise between expressive power and a reasonably simple presentation and implementation of our reduction technique. Note that case-expressions encompass the simpler if-expressions, which can be seen as case-expressions on type *bool*. Isabelle's (non-recursive and monomorphic) let-expressions can simply be inlined or replaced by case-expressions if patterns are involved.

The functions *half* and *log* below (*log* computes the logarithm) illustrate our supported fragment and will be used as running examples throughout this paper.

$half\ 0 = 0$
$half\ (Suc\ 0) = 0$
$half\ (Suc\ (Suc\ n)) = Suc\ (half\ n)$
$log\ n = (case\ half\ n\ of\ 0 \Rightarrow 0 \mid Suc\ m \Rightarrow Suc\ (log\ (Suc\ m)))$

### 10.2.3. Function Definitions by Well-Founded Recursion

When the user writes a recursive definition, the function package analyzes the equations and extracts the recursive calls. This information is then used to synthesize the termination proof obligation.

Formally, we define the operation $\text{CALLS}_f$ that computes the set of calls to $f$ inside an expression, each together with a condition under which it occurs.

- $\text{CALLS}_f(g\ e_1\ \ldots\ e_k) \equiv \text{CALLS}_f(e_1) \cup \ldots \cup \text{CALLS}_f(e_k)$ if $g$ is a constructor or a defined function other than $f$,
- $\text{CALLS}_f(f\ e_1\ \ldots\ e_n) \equiv \text{CALLS}_f(e_1) \cup \ldots \cup \text{CALLS}_f(e_n) \cup \{(e_1, \ldots, e_n, True)\}$,
- $\text{CALLS}_f(x) \equiv \varnothing$ for all variables $x$, and
- $\text{CALLS}_f(case\ e\ of\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) \equiv \text{CALLS}_f(e) \cup (\text{CALLS}_f(e_1) \wedge e = p_1) \cup \ldots \cup (\text{CALLS}_f(e_k) \wedge e = p_k)$ where $\text{CALLS}_f(e_i) \wedge e = p_i$ is like $\text{CALLS}_f(e_i)$, but every $(t_1, \ldots, t_m, \varphi) \in \text{CALLS}_f(e_i)$ is replaced by $(t_1, \ldots, t_m, \varphi \wedge e = p_i)$.

The termination proof obligation requires us to exhibit a strongly normalizing relation $\succ$ such that for each defining equation $f\ p_1\ \ldots\ p_n = e$ and each $(r_1, \ldots, r_n, \phi) \in \text{CALLS}_f(e)$ we can prove $\phi \implies (p_1, \ldots, p_n) \succ (r_1, \ldots, r_n)$.

Consider for example the definition of *half*, where we have $\text{CALLS}_{half}(0) \equiv \varnothing$ and $\text{CALLS}_{half}(Suc\ (half\ n)) \equiv \{(n, True)\}$. We obtain the following obligation.

1. $SN\ ?R$
2. $\forall n.\ (Suc\ (Suc\ n),\ n) \in ?R$

The variable $?R :: (nat \times nat)\ set$ is a *schematic variable*, which can be instantiated during the proof, i.e., it can be seen as existentially quantified.

For *log*, we have $\text{CALLS}_{log}(case\ half\ n\ of\ 0 \Rightarrow 0 \mid Suc\ m \Rightarrow Suc\ (log\ (Suc\ m))) \equiv \{(Suc\ m,\ half\ n = Suc\ m)\}$, and the following proof obligation is produced.

1. $SN\ ?R$
2. $\forall n\ m.\ half\ n = Suc\ m \implies (n,\ Suc\ m) \in ?R$

Two things should be noted here. First, the fact that the recursive call is guarded by a case-expression is reflected by a condition in the corresponding subgoal. Without this condition, which models the usual evaluation behavior of *case*, the goal would be unprovable. Second, the goal may refer to previously defined functions. To prove it, we must refer to properties of these functions, either through their definitions, or through other lemmas about them.

When the proof obligation is discharged, the function package can use the result to derive the recursive equations as theorems (previously, they were just conjectures—consider the recursive equation $f\ x = Suc\ (f\ x)$, which is inconsistent). Additionally, an induction rule is provided, which expresses "induction along the computation." The induction rules for *half* and *log* are shown below.

$$P\ 0 \Longrightarrow P\ (Suc\ 0) \Longrightarrow (\forall n.\ P\ n \Longrightarrow P\ (Suc\ (Suc\ n))) \Longrightarrow \forall n.\ P\ n$$
$$(\forall n.\ (\forall m.\ half\ n = Suc\ m \Longrightarrow P\ (Suc\ m)) \Longrightarrow P\ n) \Longrightarrow \forall n.\ P\ n$$

### 10.2.4. IsaFoR - Term Rewriting Formalized in Isabelle/HOL

In the following, we assume that the reader is familiar with the basics of term rewriting [5]. Many notions and facts from rewriting have been formalized in the Isabelle library IsaFoR [104]. Before we can give the reduction from termination of HOL functions to termination of corresponding TRSs in §10.3, we need some more details on IsaFoR. Terms are represented straightforwardly by the datatype:

$$\textbf{datatype}\ (\alpha,\ \beta)\ term = Var\ \beta \mid Fun\ \alpha\ ((\alpha,\ \beta)\ term\ list)$$

The type variables $\alpha$ and $\beta$, which represent function and variable symbols, respectively, are always instantiated with the type *string* in our setting. Hence, we abbreviate (*string*, *string*) *term* by *term* in the following. For example, the term $\mathsf{f}(\mathsf{x}, \mathsf{y})$ is represented by *Fun* "*f*" [*Var* "*x*", *Var* "*y*"]. A TRS is represented by a value of type (*term* × *term*) *set*.

The semantics of a TRS is given by its rewrite relation $\rightarrow_{\mathcal{R}}$, defined by closing $\mathcal{R}$ under contexts and substitutions. Termination of $\mathcal{R}$ is formalized as $SN\ (\rightarrow_{\mathcal{R}})$.

IsaFoR formalizes many criteria commonly used in automated termination proofs. Ultimately, it contains an executable and terminating function

$$\textit{check-proof} :: (term \times term)\ list \Rightarrow proof \Rightarrow bool$$

and a proof of the following soundness theorem:

**Theorem 10.1** (Soundness of Check)**.** *check-proof* $\mathcal{R}$ *prf* $\Longrightarrow$ $SN\ (\rightarrow_{\mathcal{R}})$

Here, *prf* is a certificate (i.e., a termination proof of $\mathcal{R}$) from some external source, encoded as a value of a suitable datatype, and $\mathcal{R}$ is the TRS under consideration.[2] Whenever *check-proof* returns *True* for some given TRS $\mathcal{R}$ and certificate *prf*, we have established (inside Isabelle) that *prf* is a valid termination proof for $\mathcal{R}$. Thus, we can prove termination of concrete TRSs inside Isabelle.

The technical details on the supported termination techniques and the structure of the certificate (i.e., the type *proof*) are orthogonal to our use of the check function, which only relies on Theorem 10.1.

### 10.2.5. Terminology and Notation

The layered nature of our setting requires that we carefully distinguish three levels of discourse. Primarily, there is higher-order logic (implemented in Isabelle/HOL), in which all mechanized reasoning takes place. The termination goals we ultimately want to solve are formulated on this level. Of course, the syntax of HOL consists of terms, but to distinguish them from the embedded term language of term rewriting, we refer to them as *expressions*. They are uniformly written in *italics* and follow the conventions of the lambda-calculus (in particular, function application is denoted by juxtaposition). HOL equality is denoted by =. For example, the definition of *half* above is a HOL expression.

[2] To be executable, *check-proof* demands that $\mathcal{R}$ is given as a list of rules and not as a set. We ignore this difference, since it is irrelevant for this paper.

The second level is the "sub-language" of first-order terms, which is deeply embedded into HOL by the datatype *term*. When we speak of a *term*, we always refer to a value of that type, not an arbitrary HOL expression. While this embedding is simple and adequate, the concrete syntax with the *Fun* and *Var* constructors and string literals is rather unwieldy. Hence, for readability, we use sans-serif font to abbreviate the constructors and the quotes: Instead of *Var* "$v$" we write $\mathsf{v}$, and instead of *Fun* "$f$" $[\ldots]$ we write $\mathsf{f}(\ldots)$, omitting the parentheses () for nullary functions. This recovers the well-known concrete syntax of term rewriting, but we must keep in mind that the constructors and strings are still present, although they are not written as such.

Finally, we must relate the two languages with each other, and describe the proof procedures that derive the relevant properties. While the properties themselves can be stated in HOL for each concrete instance, the general schema cannot, as it must talk about "all HOL expressions." Thus, we use a meta-language as another level above HOL, in which we express the transformations and tactics. This level corresponds to our implementation (in ML). Functions of the meta-language are written in SMALL CAPITALS (e.g., $\textsc{calls}_f$), and variables of the meta-language, which typically range over arbitrary HOL expressions or patterns, are written $e$ or $p$, possibly with annotations. For HOL expressions that are arguments of recursive calls we also use $r$. Equality of the meta-language is written $\equiv$ and denotes syntactic equality of HOL expressions. In particular, $e \equiv e'$ implies $e = e'$, since HOL's equality is reflexive.

Both embeddings are deep, that is, each level can talk about the syntax of the lower levels. As a simple example, the concept of a ground term can be defined as a recursive HOL function $\mathit{ground} :: \mathit{term} \Rightarrow \mathit{bool}$:

$$\mathit{ground}\ (\mathit{Var}\ x) = \mathit{False}$$
$$\mathit{ground}\ (\mathit{Fun}\ f\ ts) = (\forall t{\in}\mathit{set}(ts).\ \mathit{ground}\ t)$$

Then we can immediately deduce that $\mathit{ground}\ (\mathsf{f}(\mathsf{x})) = \mathit{False}$, due to the presence of $\mathsf{x}$. Note however that the similar-looking statement $\mathit{ground}\ (\mathsf{f}(x)) = \mathit{False}$ is not uniformly true. More precisely, its universal closure $\forall x.\ \mathit{ground}\ (\mathsf{f}(x)) = \mathit{False}$ does not hold, since we could instantiate $x$ with the term $\mathsf{c}$ (i.e., *Fun* "$c$" $[\,]$). Thus, we must not confuse variables of the different levels. Obviously, we cannot quantify over a variable $\mathsf{x}$, which is just the *Var* constructor applied to a string.

Similarly, the meta-language can talk about the syntax of HOL, as in the definition of $\textsc{calls}_f$, which is recursive over the structure of HOL expressions.

# 10.3. The Reduction to Rewriting

## 10.3.1. Encoding Expressions and Defining Equations

We define a straightforward encoding of HOL expressions as terms, denoted by the meta-level operation ENC. For case-free expressions, we turn variables into term variables and (curried) applications into applications on the term level:

$$\textsc{enc}(x) \equiv \mathsf{x}$$
$$\textsc{enc}(f\ e_1\ \ldots\ e_n) \equiv \mathsf{f}(\textsc{enc}(e_1), \ldots, \textsc{enc}(e_n))$$

Each case-expression is replaced by a new function symbol, for which we will include additional rules below. To simplify bookkeeping, we assume that each occurrence of a

case-expression is annotated with a unique integer $j$.

$$\begin{aligned}&\textsc{enc}(\mathit{case}_j\ e\ \mathit{of}\ p_1 \Rightarrow e_1\ \mid \ldots \mid p_k \Rightarrow e_k)\\ &\quad\equiv \mathsf{case_j}(\textsc{enc}(e), \textsc{enc}(y_1), \ldots, \textsc{enc}(y_m))\end{aligned}$$

where $y_1, \ldots, y_m$ are all variables that occur free in some $e_i$ but not in $p_i$.

The operation RULES yields the rewrite rules for a function or case-expression. For a function $f$ with defining equations $\ell_1 = r_1,\ \ldots,\ \ell_k = r_k$, they are

$$\textsc{rules}(f) \equiv \{\ \textsc{enc}(\ell_1) \to \textsc{enc}(r_1),\ \ldots,\ \textsc{enc}(\ell_k) \to \textsc{enc}(r_k)\ \}\,.$$

For the case-expression $\mathit{case}_j\ e\ \mathit{of}\ p_1 \Rightarrow e_1\ \mid \ldots \mid p_k \Rightarrow e_k$ we have

$$\begin{aligned}\textsc{rules}(\mathit{case}_j) \equiv \{\ &\mathsf{case_j}(\textsc{enc}(p_1), \textsc{enc}(y_1), \ldots, \textsc{enc}(y_m)) \to \textsc{enc}(e_1),\\ &\ldots,\\ &\mathsf{case_j}(\textsc{enc}(p_k)), \textsc{enc}(y_1), \ldots, \textsc{enc}(y_m)) \to \textsc{enc}(e_k)\ \}\,.\end{aligned}$$

We define the TRS for $f$ as $\mathcal{R}^f = \textsc{rules}(f) \cup \bigcup_{g \in \mathcal{S}_f} \textsc{rules}(g)$ where $\mathcal{S}_f$ is the set of all functions that are used (directly or indirectly) by $f$. Our encoding is similar to the well known technique of unraveling which transforms conditional into unconditional TRSs [77, 86].[3]

For example, $\mathcal{R}^{\mathit{log}}$ is defined as follows and completely contains $\mathcal{R}^{\mathit{half}}$.

$$\begin{aligned}\mathsf{half}(\mathsf{0}) &\to \mathsf{0} & \mathsf{log}(\mathsf{n}) &\to \mathsf{case_0}(\mathsf{half}(\mathsf{n}))\\ \mathsf{half}(\mathsf{Suc}(\mathsf{0})) &\to \mathsf{0} & \mathsf{case_0}(\mathsf{0}) &\to \mathsf{0}\\ \mathsf{half}(\mathsf{Suc}(\mathsf{Suc}(\mathsf{n}))) &\to \mathsf{Suc}(\mathsf{half}(\mathsf{n})) & \mathsf{case_0}(\mathsf{Suc}(\mathsf{m})) &\to \mathsf{Suc}(\mathsf{log}(\mathsf{Suc}(\mathsf{m})))\end{aligned}$$

### 10.3.2. Embedding Functions

At this point, we have defined a translation, but we cannot reason about it in Isabelle, since ENC is only an extra-logical concept, defined on the meta-level. In fact, it is easy to see that it cannot be defined in HOL: If we had a HOL function *enc* satisfying $\mathit{enc}\ 0 = \mathsf{0}$ and $\mathit{enc}\ (\mathit{half}\ 0) = \mathsf{half}(\mathsf{0})$, we would immediately have a contradiction, since $\mathit{half}\ 0 = 0$, and $\mathsf{half}(\mathsf{0}) \neq \mathsf{0}$, but a function must always yield the same result on the same input.

In a typical reflection scenario, we would proceed by defining an interpretation for *term*. For example, if we were modeling the syntax of integer arithmetic expressions, then we could define a function $\mathit{eval} :: \mathit{term} \Rightarrow \mathit{int}$ (possibly also depending on a variable assignment) which interprets terms as integers. However, in our setting, the result type of such a function is not fixed, as our terms represent HOL expressions of arbitrary types. Thus, the result type of *eval* would depend on the actual term it is applied to. This cannot be expressed in a logic without dependent types, which means we cannot use this approach here.

Instead, we take the opposite route: For all relevant types $\sigma$, we define a function $\mathit{emb}_\sigma :: \sigma \Rightarrow \mathit{term}$, mapping values of type $\sigma$ to their canonical term representation.

Using Isabelle's type classes, we use a single overloaded function *emb*, which belongs to a type class *embeddable*. Concrete datatypes can be declared to be instances of this class by defining *emb*, usually by structural recursion w.r.t. the datatype. For example, here are the definitions for the types *nat* and *list*:

[3] It would be possible to directly generate dependency pair problems. However, techniques like [96] and several termination tools rely on the notion of "minimal chains," which is not ensured by our approach.

$$\begin{array}{ll} emb\ 0 = \mathsf{0} & emb\ [\,] = \mathsf{Nil} \\ emb\ (Suc\ n) = \mathsf{Suc}(emb\ n) & emb\ (x\ \#\ xs) = \mathsf{Cons}(emb\ x,\ emb\ xs) \end{array}$$

This form of definition is canonical for all algebraic datatypes, and suitable definitions of *emb* can be automatically generated for all user-defined datatypes, turning them into instances of the class *embeddable*. This is analogous to the instances generated automatically by Haskell's "deriving" statement. It is also possible to manually provide the definition of *emb* for other types if they behave like datatypes like the predefined type *bool* for the Booleans.

Note that by construction, the result of *emb* is always a constructor ground term. For a HOL expression $e$ that consists only of datatype constructors, (e.g., $Suc\ (Suc\ 0)$), we have $emb\ e = \textsc{enc}(e)$. For other expressions this is not the case, e.g., $emb\ (half\ 0) = emb\ 0 = \mathsf{0}$, but $\textsc{enc}(half\ 0) \equiv \mathsf{half}(\mathsf{0})$.

To formulate our proofs, we need another encoding of expressions as terms: The operation GENC is a slight variant of ENC, which treats variables differently, mapping them to their embeddings instead of term variables.

$$\begin{array}{rl} \textsc{genc}(x) & \equiv emb\ x \\ \textsc{genc}(f\ e_1\ \ldots\ e_n) & \equiv \mathsf{f}(\textsc{genc}(e_1), \ldots, \textsc{genc}(e_n)) \\ \textsc{genc}(case_j\ e\ of\ p_1 \Rightarrow e_1\ \mid \ldots \mid p_k \Rightarrow e_k) & \\ \quad \equiv \mathsf{case_j}(\textsc{genc}(e), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & \end{array}$$

where $y_1, \ldots, y_m$ are all variables that occur free in some $e_i$ but not in $p_i$.

Hence, $\textsc{genc}(e)$ never contains term variables. However, it contains the same HOL variables as $e$. For example, $\textsc{genc}(half\ (Suc\ n)) \equiv \mathsf{half}(\mathsf{Suc}(emb\ n))$.

### 10.3.3. Rewrite Lemmas

The definitions of $\mathcal{R}^{half}$ and $\mathcal{R}^{log}$ above are straightforward, but reasoning with them is clumsy and low-level: To establish a single rewrite step, we must extract the correct rule (that is, prove that it is in the set $\mathcal{R}^{half}$ or $\mathcal{R}^{log}$), invoke closure under substitution, and construct the correct substitution explicitly as a function of type $string \Rightarrow term$.

To avoid such repetitive reasoning, we automatically derive an individual lemma for each rewrite rule. From the definition of $\mathcal{R}^{half}$, we obtain the following rules, which we call *rewrite lemmas*:

$$\begin{array}{ll} \mathsf{half}(\mathsf{0}) \to_{\mathcal{R}^{half}} \mathsf{0} & \mathsf{half}(\mathsf{Suc}(\mathsf{0})) \to_{\mathcal{R}^{half}} \mathsf{0} \\ \forall t.\ \mathsf{half}(\mathsf{Suc}(\mathsf{Suc}(t))) \to_{\mathcal{R}^{half}} \mathsf{Suc}(\mathsf{half}(t)) & \end{array}$$

Note that the term variable $\mathsf{n}$ in the last rule has been turned into a universally-quantified HOL variable by applying the "generic substitution" $\{\mathsf{n} \mapsto t\}$. The advantage of this format is that applying a rewrite rule merely involves instantiating a universal quantifier, for which we can use the matching facilities of Isabelle. In particular, we can instantiate $t$ with $emb\ n$, which in general results in a rewrite lemma of the form $\textsc{genc}(f\ p_1\ \ldots\ p_n) \to_{\mathcal{R}} \textsc{genc}(e)$ for a defining equation $f\ p_1\ \ldots\ p_n = e$.

### 10.3.4. The Simulation Property

The following property connects our generated TRSs with HOL expressions.

**Definition 10.2** (Simulation Property)**.** *For every expression $e$ and $\mathcal{R} = \bigcup\{\mathcal{R}^f \mid f$ occurs in $e\}$, the* simulation property for $e$ *is the statement*

$$\textsc{genc}(e) \to^*_{\mathcal{R}} \mathit{emb}\ e.$$

As we cannot quantify over all HOL expressions within HOL itself, we cannot formalize that the simulation property holds for all $e$.

However, we will devise a tactic that derives this property for any given concrete expression. The basic building blocks of such proofs are lemmas of the following form, which are derived for each function symbol and can be composed to show the simulation property for a given expression.

**Definition 10.3** (Simulation Lemma)**.** *The* simulation lemma *for a function $f$ of arity $n$ is the statement*

$$\forall x_1 \ldots x_n.\ \mathsf{f}(\mathit{emb}\ x_1, \ldots, \mathit{emb}\ x_n) \to^*_{\mathcal{R}^f} \mathit{emb}\ (f\ x_1\ \ldots\ x_n)\ .$$

E.g., the simulation lemma for *half* is $\forall n.\ \mathsf{half}(\mathit{emb}\ n) \to^*_{\mathcal{R}^{half}} \mathit{emb}\ (\mathit{half}\ n)$.

The lemma claims that the rules that we produced for $f$ can indeed be used to reduce a function application to the (embedding of) the value of the function. Of course, this way of saying "$\mathcal{R}^f$ computes $f$" admits the possibility that there are other $\mathcal{R}^f$-reductions that lead to different normal forms or that do not terminate, since we are not requiring confluence or strong normalization. But this form of simulation lemma is sufficient for our purpose.

We show in §10.3.6 how simulation lemmas are proved automatically.

## 10.3.5. Reduction of Termination Goals

After having proved termination of $\mathcal{R}^f$ using a termination tool in combination with IsaFoR and Theorem 10.1, we now show how to use this result to solve the termination goal for the HOL function $f$. Recall from §10.2.3 that we must exhibit a strongly normalizing relation $\succ$ such that $\phi \implies (p_1, \ldots, p_n) \succ (r_1, \ldots, r_n)$ for all $(r_1, \ldots, r_n, \phi) \in \textsc{calls}_f(e)$ for each defining equation $f\ p_1\ \ldots\ p_n = e$.

To this end, we first define $\rightsquigarrow$ as $\to_{\mathcal{R}^f} \cup \rhd$ where $\rhd$ is the strict subterm relation. The addition of $\rhd$ is required to strip off constructors and non-recursive function applications that are wrapped around recursive calls in right-hand sides of $\mathcal{R}^f$. Since $\to_{\mathcal{R}^f}$ is strongly normalizing and closed under contexts, also $\rightsquigarrow$ is strongly normalizing. This allows us to finally choose $\succ$ as the following relation.

$$(x_1, \ldots, x_n) \succ (y_1, \ldots, y_n) \text{ iff } \mathsf{f}(\mathit{emb}\ x_1, \ldots, \mathit{emb}\ x_n) \rightsquigarrow^+ \mathsf{f}(\mathit{emb}\ y_1, \ldots, \mathit{emb}\ y_n)$$

It remains to show that the arguments of recursive calls decrease w.r.t. $\succ$. That is, for each recursive call we have a goal of the form

$$\phi \implies \mathsf{f}(\mathit{emb}\ p_1, \ldots, \mathit{emb}\ p_n) \rightsquigarrow^+ \mathsf{f}(\mathit{emb}\ r_1, \ldots, \mathit{emb}\ r_n)$$

where $f\ p_1\ \ldots\ p_n = e$ is a defining equation of $f$ and $(r_1, \ldots, r_n, \phi) \in \textsc{calls}_f(e)$. In the following, we illustrate the automated proof of this goal.

Note that since the $p_i$'s are patterns, we have $emb\ p_i = \textsc{genc}(p_i)$, and hence

$$\begin{aligned}
& \mathsf{f}(emb\ p_1, \ldots, emb\ p_n) & \\
=\ & \mathsf{f}(\textsc{genc}(p_1), \ldots, \textsc{genc}(p_n)) & (p_i \text{ are patterns}) \\
\equiv\ & \textsc{genc}(f\ p_1\ \ldots\ p_n) & (\text{definition of } \textsc{genc}) \\
\to_{\mathcal{R}^f}\ & \textsc{genc}(e) & (\text{rewrite lemma})
\end{aligned}$$

Thus, it remains to construct a sequence $\textsc{genc}(e) \rightsquigarrow^* \mathsf{f}(emb\ r_1, \ldots, emb\ r_n)$, which reduces the right-hand side of the definition to a particular recursive call, eliminating any surrounding context. We proceed recursively over $e$.

- If $e \equiv g\ e_1\ \ldots\ e_m$ for a constructor $g$ or a defined function symbol $g \not\equiv f$, then $(r_1, \ldots, r_n, \phi) \in \textsc{calls}(e_i)$ for some particular $i$. Hence, we have

$$\begin{aligned}
& \textsc{genc}(e) & \\
\equiv\ & \mathsf{g}(\textsc{genc}(e_1), \ldots, \textsc{genc}(e_m)) & (\text{definition of } \textsc{genc}) \\
\rhd\ & \textsc{genc}(e_i) & (\text{definition of } \rhd) \\
\rightsquigarrow^*\ & \mathsf{f}(emb\ r_1, \ldots, emb\ r_n) & (\text{apply tactic recursively})
\end{aligned}$$

- If $e \equiv f\ e_1\ \ldots\ e_n$ then (since we excluded nested recursion) we have $e_i = r_i$ for all $i$. Hence, we have

$$\begin{aligned}
& \textsc{genc}(e) & \\
\equiv\ & \mathsf{f}(\textsc{genc}(r_1), \ldots, \textsc{genc}(r_n)) & (\text{definition of } \textsc{genc}) \\
\to^*_{\mathcal{R}^f}\ & \mathsf{f}(emb\ r_1, \ldots, emb\ r_n) & (\text{simulation property})
\end{aligned}$$

- If $e \equiv\ case_j\ e_0\ of\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$ then we distinguish where the recursive call is located. If $(r_1, \ldots, r_n, \phi) \in \textsc{calls}_f(e_0)$, then we have

$$\begin{aligned}
& \textsc{genc}(e) & \\
\equiv\ & \mathsf{case_j}(\textsc{genc}(e_0), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & (\text{definition of } \textsc{genc}) \\
\rhd\ & \textsc{genc}(e_0) & (\text{definition of } \rhd) \\
\rightsquigarrow^*\ & \mathsf{f}(emb\ r_1, \ldots, emb\ r_n) & (\text{apply tactic recursively})
\end{aligned}$$

  Otherwise, $\phi \equiv (\chi \wedge e_0 = p_i)$ for some $\chi$ and $1 \leqslant i \leqslant k$, and $(r_1, \ldots, r_n, \chi) \in \textsc{calls}(e_i)$. We may therefore use the assumption $e_0 = p_i$ and proceed with

$$\begin{aligned}
& \textsc{genc}(e) & \\
\equiv\ & \mathsf{case_j}(\textsc{genc}(e_0), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & (\text{definition of } \textsc{genc}) \\
\to^*_{\mathcal{R}^f}\ & \mathsf{case_j}(emb\ e_0, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & (\text{simulation property}) \\
=\ & \mathsf{case_j}(emb\ p_i, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & (\text{assumption } e_0 = p_i) \\
=\ & \mathsf{case_j}(\textsc{genc}(p_i), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) & (\text{since } p_i \text{ is a pattern}) \\
\to_{\mathcal{R}^f}\ & \textsc{genc}(e_i) & (\text{rewrite lemma}) \\
\rightsquigarrow^*\ & \mathsf{f}(emb\ r_1, \ldots, emb\ r_n) & (\text{apply tactic recursively})
\end{aligned}$$

### 10.3.6. Proof of the Simulation Property

We have seen that for the reduction of termination goals it is essential to use the simulation property $\textsc{genc}(e) \to^*_{\mathcal{R}^f} \mathit{emb}\ e$ for expressions $e$ that occur below recursive calls or within conditions that guard a recursive call. Below, we show how this property is derived for an individual expression, assuming that we already have simulation lemmas for all functions that occur in it. We again proceed recursively over $e$.

- If $e$ is a HOL variable $x$ then $\textsc{genc}(e) \equiv \textsc{genc}(x) \equiv \mathit{emb}\ x \equiv \mathit{emb}\ e$ and thus, the result follows by reflexivity of $\to^*_{\mathcal{R}^f}$.
- If $e \equiv g\ e_1\ \ldots\ e_k$ for a function symbol $g$ then

$$\begin{aligned}
&\textsc{genc}(e) \\
&\equiv \mathsf{g}(\textsc{genc}(e_1), \ldots, \textsc{genc}(e_k)) && \text{(definition of GENC)} \\
&\to^*_{\mathcal{R}^f} \mathsf{g}(\mathit{emb}\ e_1, \ldots, \mathit{emb}\ e_k) && \text{(apply tactic recursively)} \\
&\to^*_{\mathcal{R}^f} \mathit{emb}\ (g\ e_1\ \ldots\ e_k) && \text{(simulation lemma for } g\text{)} \\
&\equiv \mathit{emb}\ e
\end{aligned}$$

- If $e \equiv \mathit{case}_j\ e_0\ \mathit{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$ then we construct the following rewrite sequence:

$$\begin{aligned}
&\textsc{genc}(e) \\
&\equiv \mathsf{case_j}(\textsc{genc}(e_0), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) && \text{(definition of GENC)} \\
&\to^*_{\mathcal{R}^f} \mathsf{case_j}(\mathit{emb}\ e_0, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) && \text{(apply tactic recursively)}
\end{aligned}$$

  Now we apply a case analysis on $e_0$, which must be equal (in HOL, not syntactically) to one of the patterns. In each particular case we may assume $e_0 = p_i$. Then we continue:

$$\begin{aligned}
&\mathsf{case_j}(\mathit{emb}\ e_0, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) \\
&= \mathsf{case_j}(\mathit{emb}\ p_i, \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) && \text{(assumption } e_0 = p_i\text{)} \\
&= \mathsf{case_j}(\textsc{genc}(p_i), \textsc{genc}(y_1), \ldots, \textsc{genc}(y_m)) && \text{(since } p_i \text{ is a pattern)} \\
&\to_{\mathcal{R}^f} \textsc{genc}(e_i) && \text{(rewrite lemma)} \\
&\to^*_{\mathcal{R}^f} \mathit{emb}\ e_i && \text{(apply tactic recursively)} \\
&= \mathit{emb}\ e && \text{(assumption } e_0 = p_i\text{)}
\end{aligned}$$

The tactic above assumes that simulation lemmas for all functions in $e$ are already present. Note the simulation lemma is trivial to prove if $f$ is a constructor, since $\mathsf{f}(\mathit{emb}\ x_1, \ldots, \mathit{emb}\ x_n) = \mathit{emb}\ (f\ x_1\ \ldots\ x_n)$ by definition of *emb*.

For defined symbols of non-recursive functions the simulation lemmas are derived by unfolding the definition of the function and applying the tactic above. Thus, simulation lemmas are proved bottom-up in the order of function dependencies. When a function is recursive, the proof of its simulation lemma proceeds by induction using the induction principle from the function definition.

**Example 10.4.** We show how the simulation lemma for *log* is proved, assuming that the simulation lemmas for 0, *Suc*, and *half* are already available.

So our goal is to show $\mathsf{log}(emb\ n) \to^*_{\mathcal{R}^{log}} emb\ (log\ n)$ for any $n :: nat$. We apply the induction rule of *log* and obtain the following induction hypothesis.

$$\forall m.\ half\ n = Suc\ m \implies \mathsf{log}(emb\ (Suc\ m)) \to^*_{\mathcal{R}^{log}} emb\ (log\ (Suc\ m))$$

Let $c$ abbreviate $case\ half\ n\ of\ 0 \Rightarrow 0 \mid Suc\ m \Rightarrow Suc\ (log\ (Suc\ m))$. Then

$$\begin{array}{lll}
& \mathsf{log}(emb\ n) & \\
\to_{\mathcal{R}^{log}} & \mathsf{case_0}(\mathsf{half}(emb\ n)) & \text{(rewrite lemma)} \\
\to^*_{\mathcal{R}^{log}} & \mathsf{case_0}(emb\ (half\ n)) & \text{(simulation lemma of } half\text{)}
\end{array}$$

We continue by case analysis on $half\ n$. We only present the more interesting case $half\ n = Suc\ m$ (the other case $half\ n = 0$ is similar):

$$\begin{array}{lll}
& \mathsf{case_0}(emb\ (half\ n)) & \\
= & \mathsf{case_0}(emb\ (Suc\ m)) & \text{(assumption } half\ n = Suc\ m\text{)} \\
= & \mathsf{case_0}(\mathsf{Suc}(emb\ m)) & \text{(def. of } emb\text{)} \\
\to_{\mathcal{R}^{log}} & \mathsf{Suc}(\mathsf{log}(\mathsf{Suc}(emb\ m))) & \text{(rewrite lemma)} \\
\to^*_{\mathcal{R}^{log}} & \mathsf{Suc}(\mathsf{log}(emb\ (Suc\ m))) & \text{(simulation lemma of } Suc\text{)} \\
\to^*_{\mathcal{R}^{log}} & \mathsf{Suc}(emb\ (log\ (Suc\ m))) & \text{(induction hypothesis)} \\
\to^*_{\mathcal{R}^{log}} & emb\ (Suc\ (log\ (Suc\ m))) & \text{(simulation lemma of } Suc\text{)} \\
= & emb\ c & \text{(assumption } half\ n = Suc\ m\text{)} \\
= & emb\ (log\ n) & \text{(def. of } log\text{)}
\end{array}$$

## 10.4. Examples

We show some characteristic examples that illustrate the strengths and weaknesses of our approach. Each example is representative for several similar ones that occur in the Isabelle distribution.

**Example 10.5.** Consider binary trees defined by the type

**datatype** *tree* = *E* | *N tree nat tree*

and a function *remdups* that removes duplicates from a tree. The function is defined by the following equations (the auxiliary function *del* removes all occurrences of an element from a tree; we omit its straightforward definition here):

*remdups E* = *E*
*remdups* (*N l x r*) = *N* (*remdups* (*del x l*)) *x* (*remdups* (*del x r*))

The termination argument for *remdups* relies on a property of *del*: the result of *del* is smaller than its argument. In Isabelle, the user must manually state and prove (by induction) the lemma $size\ (del\ x\ t) \leq size\ t$, before termination can be shown. Here, *size* is an overloaded function generated automatically for every algebraic datatype.

For a termination tool, termination of the related TRS is easily proved using standard techniques, eliminating the need for finding and proving the lemma.

**Example 10.6.** The following function (originally due to Boyer and Moore [14]) normalizes conditional expressions consisting of atoms ($AT$) and if-expressions ($IF$).

$$\begin{array}{l} norm\ (AT\ a) = AT\ a \\ norm\ (IF\ (AT\ a)\ y\ z) = IF\ (AT\ a)\ (norm\ y)\ (norm\ z) \\ norm\ (IF\ (IF\ u\ v\ w)\ y\ z) = norm\ (IF\ u\ (IF\ v\ y\ z)\ (IF\ w\ y\ z)) \end{array}$$

Isabelle's standard size measure is not sufficient to prove termination of *norm*, and a custom measure function must be specified by the user. Using a termination tool, the proof is fully automatic and no measure function is required.

**Example 10.7.** The Isabelle distribution contains the following implementation of the merge sort algorithm (transformed into non-overlapping rules internally):

$$\begin{array}{l} msort\ [\,] = [\,] \\ msort\ [x] = [x] \\ msort\ xs = merge\ (msort\ (take\ (length\ xs\ div\ 2)\ xs))\ (msort\ (drop\ (length\ xs \\ div\ 2)\ xs)) \end{array}$$

The situation is similar to Example 10.5, as we must know how *take* and *drop* affect the length of the list. However, in this case, Isabelle's list theory already provides the necessary lemmas, e.g., $length\ (take\ n\ xs) = min\ n\ (length\ xs)$. Together with the built-in arithmetic decision procedures (which know about *div* and *min*), the termination proof works fully automatically.

For termination tools, the proof is a bit more challenging and requires techniques that are not yet formalized in IsaFoR (in particular, the technique of *rewriting dependency pairs* [37]). Thus, our connection to termination tools cannot handle *msort* yet. However, when this technique is added to IsaFoR in the future, no change will be required in our implementation to benefit from it.

These examples show the main strength of our reduction to rewriting: absolutely no user input in the form of lemmas or measure functions is required. On the other hand, Isabelle's ability to pick up previously established results can make the built-in termination prover surprisingly strong in the presence of a good library, as the *msort* example shows. Even though that example can be solved by termination tools (and only the formalization lags behind), it shows an intrinsic weakness of the approach, since existing facts are not used and must be rediscovered by the termination tool if necessary.

## 10.5. Extensions

In this section, we reconsider the restrictions imposed in §10.2.2.

**Nested Recursion.** So far, we excluded nested recursion like $f\ (Suc\ n) = f\ (f\ n)$. The problem is that to prove termination of $f$ we need its simulation lemma to reduce the inner call in the proof of the outer call, cf. §10.3.5. But proving the simulation lemma uses the induction rule of $f$, which in turn requires termination.

To solve this problem, we can use the partial induction rule that is generated by the function package even before a termination proof [69]. This rule, which is similar to the one used previously, contains extra domain conditions of the form $dom_f\ x$. It allows us to derive the restricted simulation lemma $dom_f\ n \implies \mathsf{f}(emb\ n) \to^*_{\mathcal{R}^f} emb\ (f\ n)$. In

the termination proof obligation for the outer recursive call, we may assume this domain condition for the inner call (a convenience provided by the function package), so that this restricted form of simulation lemma suffices. Hence, dealing with nested recursion simply requires a certain amount of additional bookkeeping.

**Underspecification.** So far, we require functions to be completely defined, i.e., no cases are missing in left-hand sides or case-expressions. However, $head\,(x \# xs) = x$ is a common definition. It is internally completed by $head\,[\,] = undefined$ in Isabelle, where $undefined :: \alpha$ is an arbitrary but unknown value of type $\alpha$.

For such functions, we cannot derive the simulation lemma, since this would require $\mathsf{head}(\mathsf{Nil})$ to be equal to *emb undefined*, which is an unknown term of the form $\mathsf{Suc}^k(\mathsf{0})$. The obvious idea of adding the rule $\mathsf{head}(\mathsf{Nil}) \to \mathsf{undefined}$ to the TRS does not work, since $\mathsf{undefined}$ cannot be equal to *emb undefined*.

We can solve the problem by using fresh variables for unspecified cases, e.g., by adding the rule $\mathsf{head}(\mathsf{Nil}) \to \mathsf{x}$. Then, the simulation lemma holds. However, the resulting TRS is no longer terminating. This new problem can be solved by using a variant of innermost rewriting, which would require support by IsaFoR as well as the termination tool.

**Non-Representable Types and Polymorphism.** Clearly, our embedding is limited to types that admit a term representation. This excludes uncountable types such as real numbers and most function types. However, even if such types occur in HOL functions, they may not be relevant for termination. Then, we can simply map all such values to a fixed constant by defining, e.g., $emb\ (r :: real) = \mathsf{real}$. Hence, the simulation lemmas for functions returning real numbers are trivial to prove. Furthermore, a termination proof that does not rely on these values works without problems. Like for underspecified functions, the generated TRS no longer models the original function completely, but is only an abstraction that is sufficient to prove termination.

A similar issue arises with polymorphic functions: To handle a function of type $\alpha\ list \Rightarrow \alpha\ list$ we need a definition of *emb* on type $\alpha$. Mapping values of type $\alpha$ to a constant is unproblematic, since the definition is irrelevant for the proof. However, a class instance $\alpha :: embeddable$ would violate the type class discipline. This can be solved by either replacing the use of type classes by explicit dictionary constructions (where $emb_{list}$ would take the embedding function to use for the list elements as an argument), or by restricting $\alpha$ to class *embeddable*. Since the type class does not carry any axioms, the system allows us to remove the class constraint from the final theorem, so no generality is lost.

**Higher-Order Functions.** Higher-order functions pose new difficulties. First, we cannot hope to define *emb* on function types. In particular, this means that we cannot even state the simulation lemma for a function like *map*. Second, the termination conditions for functions with higher-order recursion depend on user-provided congruence rules of a certain format [69]. These congruence rules then influence the form of the premise $\phi$ in the termination condition.

A partial solution could be to create a first-order function $map_f$ for each invocation of *map* on a concrete function $f$. Commonly used combinators like *map*, *filter* and *fold* could be supported in this way.

## 10.6. Conclusion

We have presented a generic approach to discharge termination goals of HOL functions by proving termination of a corresponding generated TRS. Hence, where before a manual termination proof might have been required, now external termination tools can be used. Since our approach is not tied to any particular termination proof technique, its power scales up as the capabilities of termination tools increase and more techniques are formalized in IsaFoR.

A complete prototype of our implementation is available in the IsaFoR/CeTA distribution (version 1.18, http://cl-informatik.uibk.ac.at/software/ceta), which also includes usage examples. It remains as future work to extend our approach to a larger class of HOL functions. Moreover, the implementation has to be more smoothly embedded into the Isabelle system such that a user can easily access the provided functionality. The general approach is not limited to Isabelle, and could be ported to other theorem provers like Coq, which has similar recursive definition facilities (e.g., [6]) and rewriting libraries similar to IsaFoR [13, 21].

**Acknowledgment.** Jasmin Blanchette gave helpful feedback on a draft of this paper.

# 11. Generalized and Formalized Uncurrying

### Publication details

Christian Sternagel and René Thiemann. Generalized and Formalized Uncurrying. In *Proceedings of the 8th International Symposium on the Frontiers of Combining Systems*, volume 6989 of LNCS, pages 243–258. Springer, 2011.

### Abstract

Uncurrying is a termination technique for applicative term rewrite systems. During our formalization of uncurrying in the theorem prover Isabelle, we detected a gap in the original pen-and-paper proof which cannot directly be filled without further preconditions. Our final formalization does not demand additional preconditions, and generalizes the existing techniques since it allows to uncurry non-applicative term rewrite systems. Furthermore, we provide new results on uncurrying for relative termination and for dependency pairs.

## 11.1. Introduction

In recent years, the way to prove termination of term rewrite systems (TRSs) has changed. Current termination tools no longer search for a single reduction order containing the rewrite relation. Instead, they combine various termination techniques in a modular way, resulting in large and tree-like termination proofs, where at each node a specific technique is applied.

On the one hand, this combination makes termination tools more powerful. On the other hand, it makes them more complex and error-prone. It is regularly demonstrated that we cannot blindly trust the output of termination provers. Every now and then, some prover delivers a faulty proof. Often, this is only detected if there is another prover giving a contradictory answer for the same input, as a manual inspection of proofs is infeasible due to their size.

The problem is solved by combining two systems. For a given TRS, we first use a termination tool to automatically detect a termination proof (which may contain errors). Then, we use a highly trusted certifier which checks whether the detected proof is indeed correct. In total, the combination yields a powerful and trustable workflow to prove termination.

To obtain a highly trustable certifier, a common approach is to first formalize the desired termination techniques once and for all (thereby ensuring their soundness) and then, for a given proof, check that the used techniques are applied correctly [12, 23, 104]. We formalized the dependency pair framework (DP framework) [42] and many termination techniques in our Isabelle/HOL [84] library IsaFoR [104] (in the remainder

we just write *Isabelle*, instead of *Isabelle/HOL*). From IsaFoR, we code-extract CeTA, an automatic certifier for termination proofs.

In this paper, we present one of the latest additions to IsaFoR: the formalization of uncurrying, as described in [53]. However, we did not only formalize uncurrying, but also generalized it. Furthermore, we found a gap in one of the original proofs, which we could fortunately close.

Note that all the proofs that are presented (or omitted) in the following, have been formalized as part of IsaFoR. Hence, in this paper, we merely give sketches of our "real" proofs. Our goal is to show the general proof outlines and help to understand the full proofs. The library IsaFoR with all formalized proofs, the executable certifier CeTA, and all details about our experiments are available at CeTA's website: `http://cl-informatik.uibk.ac.at/software/ceta`.

The paper is structured as follows. In Sect. 11.2, we shortly recapitulate some required notions of term rewriting. Afterwards, in Sect. 11.3, we describe applicative rewriting, give an overview of approaches using *uncurrying* for proving termination, and present our generalization of uncurrying for TRSs. Then, in Sect. 11.4, we show how to lift uncurrying to the DP framework. We present heuristics and our experiments in Sect. 11.5, before we conclude in Sect. 11.6.

## 11.2. Preliminaries

We assume familiarity with term rewriting [5]. Still, we recall the most important notions that are used later on. A *(first-order) term* $t$ over a set of *variables* $\mathcal{V}$ and a set of *(function) symbols* $\mathcal{F}$ is either a variable $x \in \mathcal{V}$ or a function symbol $f \in \mathcal{F}$ applied to argument terms $f(t_1, \ldots, t_n)$ where the *arity* of $f$ is $\mathsf{ar}(f) = n$. A *context* $C$ is a term containing exactly one hole $\Box$. Replacing $\Box$ in a context $C$ by a term $t$ is denoted by $C[t]$.

A *rewrite rule* is a pair of terms $\ell \to r$ and a TRS $\mathcal{R}$ is a set of rewrite rules. The set of *defined symbols* (of $\mathcal{R}$) is $\mathcal{D}_\mathcal{R} = \{f \mid f(\ldots) \to r \in \mathcal{R}\}$. The *rewrite relation (induced by $\mathcal{R}$)* $\to_\mathcal{R}$ is the closure under substitutions and contexts of $\mathcal{R}$, i.e., $s \to_\mathcal{R} t$ iff there is a context $C$, a rewrite rule $\ell \to r \in \mathcal{R}$, and a substitution $\sigma$ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. A term $t$ is *root-stable w.r.t.* $\mathcal{R}$ iff there is no derivation $t \to^*_\mathcal{R} \ell\sigma$ for some $\ell \to r \in \mathcal{R}$ and substitution $\sigma$.

We say that a term $t$ is *terminating w.r.t.* $\mathcal{R}$ ($\mathsf{SN}_\mathcal{R}(t)$) if it cannot start an infinite derivation $t = t_1 \to_\mathcal{R} t_2 \to_\mathcal{R} t_3 \to_\mathcal{R} \cdots$. A TRS is terminating ($\mathsf{SN}(\mathcal{R})$) iff all terms are terminating w.r.t. $\mathcal{R}$. A TRS $\mathcal{R}$ is *terminating relative to* a TRS $\mathcal{S}$ iff there is no infinite $\mathcal{R} \cup \mathcal{S}$-derivation with infinitely many $\mathcal{R}$-steps.

## 11.3. Applicative Rewriting and Uncurrying

An applicative term rewrite system (ATRS) is a TRS over an *applicative signature* $\mathcal{F} = \{\circ\} \cup \mathcal{C}$, where $\circ$ is a unique binary symbol (the application symbol) and all symbols in $\mathcal{C}$ are constants. ATRSs can be used to encode many higher-order functions without explicit abstraction as first-order TRSs. In the remainder we use $\circ$ as an infix-symbol which associates to the left ($s \circ t \circ u = (s \circ t) \circ u$). In examples we omit $\circ$ whenever this increases readability.

**Example 11.1.** The following ATRS $\mathcal{R}$ (a variant of [53, Example 7], replacing addition by subtraction) contains the $\mathsf{map}$ function (which applies a function to all arguments of a list) and subtraction on Peano numbers in applicative form.

$$\begin{array}{lll}
1: & \mathsf{sub}\ \mathsf{0} \to \mathsf{K}\ \mathsf{0} & 5: \quad \mathsf{K}\ x\ y \to x \\
2: & \mathsf{sub}\ x\ \mathsf{0} \to x & 6: \quad \mathsf{map}\ z\ \mathsf{nil} \to \mathsf{nil} \\
3: & \mathsf{sub}\ x\ x \to \mathsf{0} & 7: \quad \mathsf{map}\ z\ (\mathsf{cons}\ x\ xs) \to \mathsf{cons}\ (z\ x)\ (\mathsf{map}\ z\ xs) \\
4: & \mathsf{sub}\ (\mathsf{s}\ x)\ (\mathsf{s}\ y) \to \mathsf{sub}\ x\ y &
\end{array}$$

Proving termination of ATRSs is challenging without dedicated termination techniques (e.g., for reduction orders, we cannot interpret $\mathsf{sub} \circ x \circ y$ as $x$, since $\mathsf{sub}$ is a constant and not binary).

Until now, there have at least been three approaches to tackle this problem. All of them try to uncurry a TRS such that, for example, Rule 4 from above becomes $\mathsf{sub}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{sub}(x, y)$.

To distinguish the three approaches, we need the following definitions:

**Definition 11.2.** *A term $t$ is* head variable free *iff $t$ does not contain a subterm of the form $x \circ s$ where $x$ is a variable. The* applicative arity *of a constant $f$ in an ATRS $\mathcal{R}$ ($\mathsf{aa}_{\mathcal{R}}(f)$) is the maximal number $n$, such that $f \circ t_1 \circ \cdots \circ t_n$ occurs as a subterm in $\mathcal{R}$.* Uncurrying *an application $f \circ t_1 \circ \cdots \circ t_n$ with $\mathsf{aa}_{\mathcal{R}}(f) = n$ yields the term $f(t_1, \ldots, t_n)$. A term $t$ is* proper w.r.t. $\mathsf{aa}_{\mathcal{R}}$ *iff $t$ is a variable or $t = f \circ t_1 \circ \cdots \circ t_n$ where $\mathsf{aa}_{\mathcal{R}}(f) = n$ and each $t_i$ is proper.*

The oldest of the three approaches is from [59]. It requires that all terms in a TRS are proper w.r.t. $\mathsf{aa}_{\mathcal{R}}$, and shows that then termination of $\mathcal{R}$ is equivalent to termination of the TRS obtained by uncurrying all terms of $\mathcal{R}$. Since proper terms do not contain any partial applications, the application symbol is completely eliminated by uncurrying. However, requiring proper terms is rather restrictive: Essentially, it is demanded that the TRS under consideration is a standard first-order TRS which is just written in applicative form. For example, the approach is not applicable to Example 11.1, since there is a head variable in the right-hand side of Rule 7 ($z \circ x$) and $\mathsf{sub}$ as well as $\mathsf{K}$ are applied to a single argument in Rule 1, even though $\mathsf{aa}_{\mathcal{R}}(\mathsf{sub}) = 2$ and $\mathsf{aa}_{\mathcal{R}}(\mathsf{K}) = 2$.

The second approach was given in [41, 101]. Here, the same preconditions as in [59] apply, but the results are extended to innermost rewriting and to the DP framework. The latter has the advantage, that only the current subproblem has to satisfy the preconditions. For example, when treating the dependency pair

$$\mathsf{map} \circ z \circ^{\sharp} (\mathsf{cons} \circ x \circ xs) \to \mathsf{map} \circ z \circ^{\sharp} xs \tag{1}$$

for the recursive call of $\mathsf{map}$, we can perform uncurrying (since there are no usable rules and (1) satisfies the preconditions). Moreover, in [101] uncurrying is combined with the reduction pair processor to further relax the preconditions.

The third approach is given in [53]. Here, the preconditions for uncurrying have been reduced drastically as only the left-hand sides of the TRS $\mathcal{R}$ must be head variable free. In return, we have to $\eta$-saturate $\mathcal{R}$ and add uncurrying rules. Moreover, for each constant $f$ with $\mathsf{aa}_{\mathcal{R}}(f) = n$ we obtain $n$ new function symbols $f_1, \ldots, f_n$ of arities $1, 2, \ldots, n$ which handle partial applications.

**Example 11.3.** When $\eta$-saturating the TRS $\mathcal{R}$ of Example 11.1, we have to add the rule $\mathsf{sub} \circ \mathsf{0} \circ y \to \mathsf{K} \circ \mathsf{0} \circ y$. The uncurried TRS consists of the following rules:

$$
\begin{array}{rrrr}
8: & \mathsf{sub}_1(\mathsf{0}) & \to \mathsf{K}_1(\mathsf{0}) & 12: \quad \mathsf{sub}_2(\mathsf{s}_1(x), \mathsf{s}_1(y)) \to \mathsf{sub}_2(x, y) \\
9: & \mathsf{sub}_2(\mathsf{0}, y) & \to \mathsf{K}_2(\mathsf{0}, y) & 13: \quad \mathsf{K}_2(x, y) \to x \\
10: & \mathsf{sub}_2(x, \mathsf{0}) & \to x & 14: \quad \mathsf{map}_2(z, \mathsf{nil}) \to \mathsf{nil} \\
11: & \mathsf{sub}_2(x, x) & \to \mathsf{0} & 15: \quad \mathsf{map}_2(z, \mathsf{cons}_2(x, xs)) \to \mathsf{cons}_2(z \circ x, \mathsf{map}_2(z, xs))
\end{array}
$$

Moreover, we have to add the following uncurrying rules:

$$
\begin{array}{rrrr}
16: & \mathsf{s} \circ x & \to \mathsf{s}_1(x) & 21: \quad \mathsf{cons} \circ x \to \mathsf{cons}_1(x) \\
17: & \mathsf{K} \circ x & \to \mathsf{K}_1(x) & 22: \quad \mathsf{cons}_1(x) \circ y \to \mathsf{cons}_2(x, y) \\
18: & \mathsf{K}_1(x) \circ y & \to \mathsf{K}_2(x, y) & 23: \quad \mathsf{map} \circ x \to \mathsf{map}_1(x) \\
19: & \mathsf{sub} \circ x & \to \mathsf{sub}_1(x) & 24: \quad \mathsf{map}_1(x) \circ y \to \mathsf{map}_2(x, y) \\
20: & \mathsf{sub}_1(x) \circ y & \to \mathsf{sub}_2(x, y) &
\end{array}
$$

Also [53] gives an extension to the DP framework.

To summarize, the traditional technique of uncurrying of [59] is completely subsumed by [41, 101], but [41, 101] and [53] are incomparable. The advantage of [41, 101] is that the generated TRSs and DP problems are smaller, and that uncurrying is also available in a combination with the reduction pair processor, whereas [53] supports head variables (see [101, Chap. 6] for a more detailed comparison).

Since [53] is used in more termination tools (it is used in at least Jambox [31] and T$_{\text{T}}$T$_2$ [67] whereas we only know of AProVE [39] that implements all uncurrying techniques of [41, 101]), we incorporated the techniques of [53] in our certifier CeTA.

During our formalization we have

- detected a gap in a proof of [53] which could not directly be closed without adding further preconditions to one of the main results,
- generalized the technique of uncurrying which now entails the result of [53] even without adding any additional precondition, and
- generalized the technique of *freezing* from [53].

The structure in [53] is as follows. First, uncurrying is developed for TRSs over applicative signatures $\{\circ\} \cup \mathcal{C}$. Then, it is extended to DP problems, introducing a second application symbol $\circ^\sharp$ that may only occur at root-positions of $\mathcal{P}$ and is not uncurried at all. Finally, *freezing* is applied to uncurry applications of $\circ^\sharp$.

Following this structure, we first fully formalized uncurrying on TRSs. However, in the extension to DP problems there is a missing step which is illustrated in more detail in Example 11.14 on page 119. The main problem is that signature restrictions on DP problems are in general unsound.

To fill the gap, one option is to use the results of [97] about signature restrictions, which can however only be applied if $\mathcal{R}$ is left-linear. This clearly weakens the applicability of uncurrying, e.g., Example 11.1 is not left-linear.

Alternatively, one can try to perform uncurrying without restricting to applicative signatures. This is what we did. All uncurrying techniques that we formalized work on terms and TRSs over arbitrary signatures.

The major complication is the increase of complexity in the cases that have to be considered. For example, using an applicative signature, we can assume that every term is of the form $x \circ t_1 \circ \cdots \circ t_n$ or $f \circ t_1 \circ \cdots \circ t_n$ where $n \in \mathbb{N}$, $x$ is a variable, and $f$ is a constant. Generalizing this to arbitrary signatures we have to consider the two cases $x \circ t_1 \circ \cdots \circ t_n$ and $f(s_1, \ldots, s_m) \circ t_1 \circ \cdots \circ t_n$ instead, where $f$ is an $m$-ary symbol. Hence,

when considering a possible rewrite step, we also have to consider the new case that the step is performed in some $s_i$.

We not only generalized uncurrying to work for arbitrary signatures and relative rewriting, but also to a free choice of the applicative arity $\mathsf{aa}(f)$. This is in contrast to [53], where the applicative arity is fixed by Definition 11.2. We will elaborate on this difference after presenting our main theorem.

**Definition 11.4** (Symbol maps and applicative arity)**.** *Let $\mathcal{F}$ be a signature. A* symbol map *is a mapping $\pi : \mathcal{F} \to [\mathcal{F}]$ from symbols to non-empty lists of symbols. It is* injective *if for all $f$ and $g$, $\pi(f)$ contains no duplicates, $\pi(f)$ does not contain $\circ$, and whenever $f \neq g$ then $\pi(f)$ and $\pi(g)$ do not share symbols. If $\pi(f) = [f_0, \ldots, f_n]$, then the* applicative arity of $f$ w.r.t. $\pi$ *is $\mathsf{aa}_\pi(f) = n$. The* applicative arity of a term *is defined by $\mathsf{aa}_\pi(t) = \mathsf{aa}_\pi(f) \dot{-} n$, where $x \dot{-} y = \max(x - y, 0)$, for $t = f(s_1, \ldots, s_m) \circ t_1 \circ \cdots \circ t_n$ and is undefined otherwise.*

Intuitively, if $\pi(f) = [f_0, \ldots, f_n]$ then every application of $f$ on $i \leqslant n$ arguments $t_1, \ldots, t_i$ will be fully uncurried to $f_i(t_1, \ldots, t_i)$. If more than $n$ arguments are applied, then we obtain $f_n(t_1, \ldots, t_n) \circ t_{n+1} \circ \ldots \circ t_i$. A symbol map containing an entry for $f$, uniquely determines the applicative arity $n$ as well as the names of the (partial) applications $f_0, \ldots, f_n$ of $f$.

In the following we assume a fixed symbol map $\pi$ and just write $\mathsf{aa}(f)$ and $\mathsf{aa}(t)$ instead of $\mathsf{aa}_\pi(f)$ and $\mathsf{aa}_\pi(t)$, respectively. Additionally, we assume that $\pi(f) = [f_0, \ldots, f_{\mathsf{aa}(f)}]$ where in examples we write $f$ instead of $f_0$. Now we can define the uncurrying TRS w.r.t. $\pi$.

**Definition 11.5.** *The* uncurrying TRS $\mathcal{U}$ *contains the rule*

$$f_k(x_1, \ldots, x_m, y_1, \ldots, y_k) \circ y_{k+1} \to f_{k+1}(x_1, \ldots, x_m, y_1, \ldots, y_{k+1})$$

*for every $f \in \mathcal{F}$ with $\mathsf{ar}(f) = m$ and $\mathsf{aa}(f) = n$, and every $k < n$. The variables $x_1$, …, $x_m$, $y_1$, …, $y_{k+1}$ are pairwise distinct.*

In [53], terms are uncurried by computing the unique normal form w.r.t. $\mathcal{U}$. For our formalization we instead used the upcoming uncurrying function for the following two reasons: First, we do not have any results about confluence of TRSs. Hence, to even define *the* normal form w.r.t. $\mathcal{U}$ would require to formalize several additional lemmas which show that every term has exactly one normal form. This would be quite some effort which we prefer to avoid. The second reason is efficiency. When certifying the application of uncurrying in large termination proofs, we have to compute the uncurried version of a term. It is just more efficient to use a function which performs uncurrying directly, than to compute a normal form w.r.t. a TRS where possible redexes have to be searched, etc.

**Definition 11.6.** *The uncurrying function $\llcorner\cdot\lrcorner$ on terms is defined as*

- $\llcorner x \circ t_1 \circ \cdots \circ t_n \lrcorner = x \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner$
- $\llcorner f(s_1, \ldots, s_m) \circ t_1 \circ \cdots \circ t_n \lrcorner = f_k(\llcorner s_1 \lrcorner, \ldots, \llcorner s_m \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_k \lrcorner) \circ \llcorner t_{k+1} \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner$ *where $k = \min(n, \mathsf{aa}(f))$*

*It is homomorphically extended to operate on rules, TRSs, and substitutions.*

We establish the link between $\mathcal{U}$ and $\llcorner\cdot\lrcorner$ in the following lemma which generalizes the corresponding results in [53].

**Lemma 11.7.**

- *if* $k < \mathsf{aa}(f)$ *and* $\mathsf{ar}(f) = m$ *then* $f_k(s_1, \ldots, s_{m+k}) \circ t \to_{\mathcal{U}} f_{k+1}(s_1, \ldots, s_{m+k}, t)$
- *if* $k + n \leqslant \mathsf{aa}(f)$ *and* $\mathsf{ar}(f) = m$ *then* $f_k(s_1, \ldots, s_{m+k}) \circ t_1 \circ \ldots \circ t_n \to^*_{\mathcal{U}} f_{k+n}(s_1, \ldots, s_{m+k}, t_1, \ldots, t_n)$
- $\llcorner s \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \to^*_{\mathcal{U}} \llcorner s \circ t_1 \circ \cdots \circ t_n \lrcorner$
- *if* $\mathsf{aa}(s) = 0$ *or* $\mathsf{aa}(s)$ *is undefined then* $\llcorner s \circ t_1 \circ \cdots \circ t_n \lrcorner = \llcorner s \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner$
- $\llcorner s \lrcorner \cdot \llcorner \sigma \lrcorner \to^*_{\mathcal{U}} \llcorner s \cdot \sigma \lrcorner$
- *if* $t$ *is head variable free then* $\llcorner s \cdot \sigma \lrcorner = \llcorner s \lrcorner \cdot \llcorner \sigma \lrcorner$

The last two results show how uncurrying can be exchanged with applying substitutions. As we will often need the equality $\llcorner \ell \cdot \sigma \lrcorner = \llcorner \ell \lrcorner \cdot \llcorner \sigma \lrcorner$ for left-hand sides $\ell$, it is naturally to demand that left-hand sides are head variable free.

**Definition 11.8.** *A TRS is* left head variable free *if all left-hand sides are head variable free.*

Termination of $\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}$ does not suffice to conclude termination of $\mathcal{R}$, cf. [53, Example 13]. The reason is that first we have to $\eta$-saturate $\mathcal{R}$.

**Definition 11.9.** *A TRS* $\mathcal{R}$ *is* $\eta$-closed *iff for every rule* $\ell \to r$ *with* $\mathsf{aa}(\ell) > 0$ *there is a rule* $\ell \circ x \to r \circ x \in \mathcal{R}$ *where* $x$ *is fresh w.r.t.* $\ell \to r$*. The* $\eta$-saturation $\mathcal{R}_\eta$ of $\mathcal{R}$ *is obtained by adding new rules* $\ell \circ x \to r \circ x$ *until the result is* $\eta$*-closed.*

The upcoming theorem is the key to use uncurrying for termination proofs. It allows to simulate one $\mathcal{R}$-step by many steps in the uncurried system.

**Theorem 11.10.** *Let* $\mathcal{R}$ *be* $\eta$*-closed and left head variable free. Let there be no left-hand side of* $\mathcal{R}$ *which is a variable. If* $s \to_{\mathcal{R}} t$ *then* $\llcorner s \lrcorner \to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner t \lrcorner$*.*

*Proof.* Let $s = C[\ell\sigma] \to_{\mathcal{R}} C[r\sigma] = t$ where $\ell \to r \in \mathcal{R}$. We show $\llcorner s \lrcorner \to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner t \lrcorner$ by induction on the size of $C$.

- If $C = f(s_1, \ldots, D, \ldots, s_m) \circ t_1 \circ \cdots \circ t_n$ for some $f \neq \circ$ then by the induction hypothesis we know that $\llcorner D[\ell\sigma] \lrcorner \to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner D[r\sigma] \lrcorner$. Moreover,

$$\begin{aligned}
\llcorner s \lrcorner &= \llcorner f(s_1, \ldots, D[\ell\sigma], \ldots, s_m) \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= f_k(\llcorner s_1 \lrcorner, \ldots, \llcorner D[\ell\sigma] \lrcorner, \ldots, \llcorner s_m \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_k \lrcorner) \circ \llcorner t_{k+1} \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} f_k(\ldots, \llcorner D[r\sigma] \lrcorner, \ldots, \llcorner s_m \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_k \lrcorner) \circ \llcorner t_{k+1} \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&= \llcorner f(s_1, \ldots, D[r\sigma], \ldots, s_m) \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t \lrcorner
\end{aligned}$$

  where $k = \min(n, \mathsf{aa}(f))$.

- If $C = t_0 \circ D \circ t_1 \circ \cdots \circ t_n$ then by the induction hypothesis we know $\llcorner D[\ell\sigma] \lrcorner \to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner D[r\sigma] \lrcorner$. If $\mathsf{aa}(t_0) = 0$ or $\mathsf{aa}(t_0)$ is undefined then

$$\begin{aligned}
\llcorner s \lrcorner &= \llcorner t_0 \circ D[\ell\sigma] \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t_0 \lrcorner \circ \llcorner D[\ell\sigma] \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner t_0 \lrcorner \circ \llcorner D[r\sigma] \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&= \llcorner t_0 \circ D[r\sigma] \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t \lrcorner
\end{aligned}$$

  Otherwise, $\mathsf{aa}(t_0) > 0$ and hence, $t_0 = f(s_1, \ldots, s_m) \circ s_{m+1} \circ \cdots \circ s_{m+k}$ where $k < \mathsf{aa}(f)$. It follows that

$$\begin{aligned}
\llcorner s \lrcorner &= \llcorner f(s_1, \ldots, s_m) \circ s_{m+1} \circ \cdots \circ s_{m+k} \circ D[\ell\sigma] \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= f_{k+1+n'}(\ldots, \llcorner s_{m+k} \lrcorner, \llcorner D[\ell\sigma] \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_{n'} \lrcorner) \circ \llcorner t_{n'+1} \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} f_{k+1+n'}(\ldots, \llcorner s_{m+k} \lrcorner, \llcorner D[r\sigma] \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_{n'} \lrcorner) \circ \llcorner t_{n'+1} \lrcorner \circ \cdots \\
&= \llcorner f(s_1, \ldots, s_m) \circ s_{m+1} \circ \cdots \circ s_{m+k} \circ D[r\sigma] \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t \lrcorner
\end{aligned}$$

  where $n' = \min(\mathsf{aa}(f) - k - 1, n)$.

- If $C = \Box \circ t_1 \circ \cdots \circ t_n$ and $n = 0 \vee \mathsf{aa}(\ell) = 0$ then

$$\begin{aligned}
\llcorner s \lrcorner &= \llcorner \ell \cdot \sigma \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner \ell \cdot \sigma \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&= \llcorner \ell \lrcorner \cdot \llcorner \sigma \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to_{\llcorner \mathcal{R} \lrcorner} \llcorner r \lrcorner \cdot \llcorner \sigma \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^*_{\mathcal{U}} \llcorner r \cdot \sigma \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^*_{\mathcal{U}} \llcorner r \cdot \sigma \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t \lrcorner
\end{aligned}$$

  since $\ell$ is head variable free and if $n \neq 0$ then $\mathsf{aa}(\ell\sigma) = \mathsf{aa}(\ell) = 0$.

- If $C = \Box \circ t_1 \circ \cdots \circ t_n$ with $n > 0$ and $\mathsf{aa}(\ell) > 0$ then $\ell' \to r' = \ell \circ x \to r \circ x \in \mathcal{R}$ since $\mathcal{R}$ is $\eta$-closed. Moreover, by changing $\sigma$ to $\delta = \sigma \uplus \{x/t_1\}$ we achieve $s = \ell \cdot \sigma \circ t_1 \circ \cdots \circ t_n = \ell'\delta \circ t_2 \circ \cdots \circ t_n = D[\ell'\delta]$ and $r = r \cdot \sigma \circ t_1 \circ \cdots \circ t_n = r'\delta \circ t_2 \circ \cdots \circ t_n = D[r'\delta]$ for the context $D = \Box \circ t_2 \circ \cdots t_n$ which is strictly smaller than $C$. Hence, the result follows by the induction hypothesis.

- If $C = \Box \circ t_1 \circ \cdots \circ t_n$ with $n > 0$ and $\mathsf{aa}(\ell)$ is undefined then $\ell = x \circ \ell_1 \circ \cdots \circ \ell_k$ with $k \geq 0$. But if $k > 0$ then $\ell$ is not head variable free and if $k = 0$ then $\mathcal{R}$ contains a variable as left-hand side. In both cases we get a contradiction to the preconditions in the theorem. □

Note that the condition that the left-hand sides of $\mathcal{R}$ are not variables is new in comparison to [53]. Nevertheless, in the following corollary we can drop this condition, since otherwise $\llcorner \mathcal{R} \lrcorner$ is not terminating anyway.

**Corollary 11.11.** *If $\mathcal{R}_\eta$ is left head variable free then termination of $\llcorner \mathcal{R}_\eta \lrcorner \cup \mathcal{U}$ implies termination of $\mathcal{R}$.*

When using uncurrying for relative termination of $\mathcal{R}/\mathcal{S}$, it turns out that the new condition on the left-hand sides can only be ignored for $\mathcal{R}$ – since otherwise relative termination of $\llcorner\mathcal{R}\lrcorner/\llcorner\mathcal{S}\lrcorner \cup \mathcal{U}$ does not hold – but not for $\mathcal{S}$.

**Corollary 11.12.** *If $\mathcal{R}_\eta \cup \mathcal{S}_\eta$ is left head variable free and the left-hand sides of $\mathcal{S}$ are not variables, then relative termination of $\llcorner\mathcal{R}_\eta\lrcorner/\llcorner\mathcal{S}_\eta\lrcorner \cup \mathcal{U}$ implies relative termination of $\mathcal{R}/\mathcal{S}$.*

**Example 11.13.** Let $\mathcal{R} = \{\mathsf{f} \circ \mathsf{f} \circ x \to \mathsf{f} \circ x\}$ and $\mathcal{S} = \{x \to \mathsf{f} \circ x\}$. Then $\mathcal{R}/\mathcal{S}$ is not relative terminating since $\mathsf{f}\circ\mathsf{f}\circ x \to_\mathcal{R} \mathsf{f}\circ x \to_\mathcal{S} \mathsf{f}\circ\mathsf{f}\circ x \to_\mathcal{R} \dots$ is an infinite $\mathcal{R}\cup\mathcal{S}$-derivation with infinitely many $\mathcal{R}$-steps.

For $\pi(\mathsf{f}) = [\mathsf{f}, \mathsf{f}_1, \mathsf{f}_2]$ we have $\mathcal{R}_\eta = \mathcal{R}$, $\mathcal{S}_\eta = \mathcal{S}$, $\llcorner\mathcal{R}_\eta\lrcorner = \{\mathsf{f}_2(\mathsf{f}, x) \to \mathsf{f}_1(x)\}$, $\llcorner\mathcal{S}_\eta\lrcorner = \{x \to \mathsf{f}_1(x)\}$, and $\mathcal{U} = \{\mathsf{f} \circ x \to \mathsf{f}_1(x), \mathsf{f}_1(x) \circ y \to \mathsf{f}_2(x, y)\}$. It is easy to see that $\llcorner\mathcal{R}_\eta\lrcorner/\llcorner\mathcal{S}_\eta\lrcorner \cup \mathcal{U}$ is relative terminating by counting the number of $\mathsf{f}$-symbols. Since both $\mathcal{R}_\eta$ and $\mathcal{S}_\eta$ are head variable free, we have shown that Corollary 11.12 does not hold if one would drop the new variable condition on $\mathcal{S}$.

As already mentioned, Corollary 11.11 generalizes the similar result of [53, Theorem 16] in two ways: first, there is no restriction to applicative signatures, and second, one can freely choose the applicative arities. Since in principle the choice of $\pi$ does not matter (uncurrying preserves termination for every choice of $\pi$), we can only heuristically determine whether the additional freedom increases termination proving power and therefore refer to our experiments in Sect. 11.5.

## 11.4. Uncurrying in the Dependency Pair Framework

The DP framework [42] facilitates modular termination proofs. Instead of single TRSs, we consider *DP problems* $(\mathcal{P}, \mathcal{R})$, consisting of two TRSs $\mathcal{P}$ and $\mathcal{R}$ where elements of $\mathcal{P}$ are often called *pairs* to distinguish them from the *rules* of $\mathcal{R}$. The *initial DP problem* for a TRS $\mathcal{R}$ is $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$, where $\mathsf{DP}(\mathcal{R}) = \{f^\sharp(s_1, \dots, s_n) \to g^\sharp(t_1, \dots, t_m) \mid f(s_1, \dots, s_n) \to C[g(t_1, \dots, t_m)] \in \mathcal{R}, g \in \mathcal{D}_\mathcal{R}\}$ is the set of *dependency pairs* of $\mathcal{R}$, incorporating a fresh *tuple symbol* $f^\sharp$ for each defined symbol $f$. The initial DP problem is also a *standard* DP problem, i.e., root symbols of pairs do not occur elsewhere in $\mathcal{P}$ or $\mathcal{R}$.[1] A $(\mathcal{P}, \mathcal{R})$-*chain* is a possibly infinite derivation of the form:

$$s_1\sigma_1 \to_\mathcal{P} t_1\sigma_1 \to^*_\mathcal{R} s_2\sigma_2 \to_\mathcal{P} t_2\sigma_2 \to^*_\mathcal{R} s_3\sigma_3 \to_\mathcal{P} \cdots \qquad (\star)$$

where $s_i \to t_i \in \mathcal{P}$ for all $i > 0$. If additionally every $t_i\sigma_i$ is terminating w.r.t. $\mathcal{R}$, then the chain is *minimal*. A DP problem $(\mathcal{P}, \mathcal{R})$ is called *finite* [42], if there is no minimal infinite $(\mathcal{P}, \mathcal{R})$-chain. Proving finiteness of a DP problem is done by simplifying $(\mathcal{P}, \mathcal{R})$ using so called *processors* recursively. A processor transforms a DP problem into a new DP problem. The aim is to reach a DP problem with empty $\mathcal{P}$-component (such DP problems are trivially finite). To conclude finiteness of the initial DP problem, the applied processors need to be *sound*. A processor *Proc* is sound whenever for all DP problems $(\mathcal{P}, \mathcal{R})$ we have that finiteness of $\mathit{Proc}(\mathcal{P}, \mathcal{R})$ implies finiteness of $(\mathcal{P}, \mathcal{R})$.

In the following we explain how uncurrying is used as processor in the DP framework. In Sect. 11.3 it was already mentioned that in [53] the notion of applicative TRS was lifted to applicative DP problem by allowing a new binary application symbol $\circ^\sharp$ (where

[1] Several termination provers only work on standard DP problems.

we sometimes just write $^\sharp$ in examples). The symbol $\circ^\sharp$ naturally occurs as tuple symbol of $\circ$.

To prove soundness of the uncurrying processor, in [53] it is assumed that there is a minimal $(\mathcal{P},\mathcal{R})$-chain $s_1\sigma_1 \to_\mathcal{P} t_1\sigma_1 \to^*_\mathcal{R} s_2\sigma_2 \to_\mathcal{P} \cdots$, which is converted into a minimal $(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$-chain by reusing the results for TRSs. However, there is a gap in this reasoning. Right in the beginning it is silently assumed that all terms $s_i\sigma_i$ and $t_i\sigma_i$ have tuple symbols as roots and that their arguments are applicative terms, i.e., terms over an applicative signature $\{\circ\} \cup \mathcal{C}$. Without this assumption it is not possible to apply the results of uncurrying for TRSs, since those are only available for applicative terms in [53].

The following variant of [97, Example 14] shows that in general restricting substitutions in chains to an applicative signature $\{\circ\} \cup \mathcal{C}$ is unsound.

**Example 11.14.** Consider the applicative and standard DP problem $(\mathcal{P},\mathcal{R})$ where $\mathcal{P} = \{\mathsf{g}^\sharp\ (\mathsf{f}\ x\ y\ z\ z\ u\ v) \to \mathsf{g}^\sharp\ (\mathsf{f}\ x\ y\ x\ y\ x\ (\mathsf{h}\ y\ u))\}$ and $\mathcal{R}$ contains the rules:

$$\begin{aligned} \mathsf{a} &\to \mathsf{b} & \mathsf{f}\ \mathsf{a}\ x_2\ x_3\ x_4\ x_5 &\to \mathsf{f}\ \mathsf{a}\ x_2\ x_3\ x_4\ x_5 \\ \mathsf{a} &\to \mathsf{c} & \mathsf{f}\ x_1\ \mathsf{a}\ x_3\ x_4\ x_5 &\to \mathsf{f}\ x_1\ \mathsf{a}\ x_3\ x_4\ x_5 \\ \mathsf{h}\ x\ x &\to \mathsf{h}\ x\ x & \mathsf{f}\ (y\ z)\ x_2\ x_3\ x_4\ x_5 &\to \mathsf{f}\ (y\ z)\ x_2\ x_3\ x_4\ x_5 \\ & & \mathsf{f}\ x_1\ (y\ z)\ x_3\ x_4\ x_5 &\to \mathsf{f}\ x_1\ (y\ z)\ x_3\ x_4\ x_5 \end{aligned}$$

There is a minimal $(\mathcal{P},\mathcal{R})$-chain taking $s_i = \mathsf{g}^\sharp(\mathsf{f}\ x\ y\ z\ z\ u\ v)$, $t_i = \mathsf{g}^\sharp\ (\mathsf{f}\ x\ y\ x\ y\ x\ (\mathsf{h}\ y\ u))$, and $\sigma_i = \{x/\mathsf{k}(\mathsf{a}), y/\mathsf{k}(\mathsf{b}), z/\mathsf{k}(\mathsf{b}), u/\mathsf{k}(\mathsf{c}), v/\mathsf{h}\ (\mathsf{k}(\mathsf{b}))\ (\mathsf{k}(\mathsf{c}))\}$ where $\mathsf{k}$ is a unary symbol. However, there is no minimal $(\mathcal{P},\mathcal{R})$-chain using substitutions over the signature $\{\circ\} \cup \mathcal{C}$, regardless of the choice of constants $\mathcal{C}$.

Since our generalizations in Sect. 11.3 do not have any restrictions on the signature, we were able to correct the corresponding proofs in [53] such that the major theorems are still valid.[2] It follows the generalization of [53, Theorem 33].

**Theorem 11.15.** *The uncurrying processor $\mathcal{U}'_1$ is sound where* $\mathcal{U}'_1(\mathcal{P},\mathcal{R}) =$

$$\begin{cases} (\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U}) & \text{if } \mathcal{P} \cup \mathcal{R}_\eta \text{ is left head variable free and } \pi \text{ is injective,} \\ (\mathcal{P},\mathcal{R}) & \text{otherwise.} \end{cases}$$

*Proof.* The proof mainly uses the results from the previous section. We assume an infinite minimal $(\mathcal{P},\mathcal{R})$-chain $s_1\sigma_1 \to_\mathcal{P} t_1\sigma_1 \to^*_\mathcal{R} s_2\sigma_2 \to_\mathcal{P} t_2\sigma_2 \to^*_\mathcal{R} \cdots$ and construct an infinite minimal $(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$-chain as follows.

We achieve $\llcorner s_i\sigma_i\lrcorner = \llcorner s_i\lrcorner \cdot \llcorner\sigma_i\lrcorner$ and $\llcorner t_i\lrcorner \cdot \llcorner\sigma_i\lrcorner \to^*_\mathcal{U} \llcorner t_i\sigma_i\lrcorner$ since $\mathcal{P}$ is left head variable free. Moreover, using $t_i\sigma_i \to^*_\mathcal{R} s_{i+1}\sigma_{i+1}$ and Theorem 11.10 we conclude $\llcorner t_i\sigma_i\lrcorner \to^*_{\llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U}} \llcorner s_{i+1}\sigma_{i+1}\lrcorner$. Here, the condition that $\mathcal{R}_\eta$ must not contain variables as left-hand sides is ensured by the minimality of the chain: if $x \to r \in \mathcal{R}_\eta$ then $\mathsf{SN}_\mathcal{R}(t_i\sigma)$ does not hold. Hence, we constructed a $(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$-chain as

$$\llcorner s_i\sigma_i\lrcorner = \llcorner s_i\lrcorner \cdot \llcorner\sigma_i\lrcorner \to_{\llcorner\mathcal{P}\lrcorner} \llcorner t_i\lrcorner \cdot \llcorner\sigma_i\lrcorner \to^*_\mathcal{U} \llcorner t_i\sigma_i\lrcorner \to^*_{\llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U}} \llcorner s_{i+1}\sigma_{i+1}\lrcorner$$

for all $i$. To ensure that the chain is minimal it is demanded that $\pi$ is injective. Otherwise, two different symbols can be mapped to the same new symbol which clearly can introduce nontermination. The structure of the proof that minimality is preserved is similar to the one in [53] and we just refer to IsaFoR for details. □

[2] After the authors of [53] where informed of the gap, they independently developed an alternative fix, which is part of an extended, but not yet published version of [53].

The uncurrying processor of Theorem 11.15 generalizes [53, Theorem 33] in three ways: the signature does not have to be applicative, we can freely choose the applicative arity via $\pi$, and we can freely choose the application symbol. The last generalization lets Theorem 11.15 almost subsume the technique of freezing [53, Corollary 40] which is used to uncurry $\circ^\sharp$.

**Definition 11.16** (Freezing [53]). *A* simple freeze $\maltese$ *is a subset of* $\mathcal{F}$.[3] Freezing *is applied on non-variable terms as follows*

$$\maltese(f(t_1,\dots,t_n)) = \begin{cases} f(t_1,\dots,t_n) & \text{if } n = 0 \text{ or } f \notin \maltese \\ f^g(s_1,\dots,s_m,t_2,\dots,t_m) & \text{if } t_1 = g(s_1,\dots,s_m) \text{ and } f \in \maltese \end{cases}$$

*where each* $f^g$ *is a new symbol. It is homomorphically extended to rules and TRSs. The freezing DP processor is defined as* $\maltese(\mathcal{P},\mathcal{R}) =$

$$\begin{cases} (\maltese(\mathcal{P}),\mathcal{R}) & \text{if } (\mathcal{P},\mathcal{R}) \text{ is a standard DP problem where for all } s \to f(t_1,\dots,t_n) \\ & \in \mathcal{P} \text{ with } f \in \maltese, \text{ both } t_1 \notin \mathcal{V} \text{ and all instances of } t_1 \text{ are root-stable,} \\ (\mathcal{P},\mathcal{R}) & \text{otherwise.} \end{cases}$$

In [53, Theorem 39], it is shown that freezing is sound.

**Example 11.17.** In the following we use numbers to refer to rules from previous examples. We consider the DP problem $(\mathcal{P},\mathcal{R})$ where $\mathcal{P} = \{\mathsf{sub}\ (\mathsf{s}\ x)^\sharp\ (\mathsf{s}\ y) \to \mathsf{sub}\ x^\sharp\ y\}$ and $\mathcal{R} = \{2\text{–}4\}$. Uncurrying $\circ$ with $\pi(\mathsf{s}) = [\mathsf{s},\mathsf{s}_1]$, $\pi(\mathsf{sub}) = [\mathsf{sub},\mathsf{sub}_1,\mathsf{sub}_2]$, $\pi(\mathsf{0}) = [\mathsf{0}]$, and $\pi(^\sharp) = [^\sharp]$ yields the DP problem $(\llcorner\mathcal{P}\lrcorner,\llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$ where $\llcorner\mathcal{P}\lrcorner = \{\mathsf{sub}_1(\mathsf{s}_1(x))^\sharp\ \mathsf{s}_1(y) \to \mathsf{sub}_1(x)^\sharp\ y\}$ and $\llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U}$ consists of $\{10\text{–}12, 16, 19, 20\}$.

Afterwards we uncurry the resulting DP problem using $\circ^\sharp$ as application symbol and $\pi$ where $\pi(\mathsf{sub}_1) = [\mathsf{sub}_1,-^\sharp]$ and $\pi(f) = [f]$ for all other symbols. We obtain $(\mathcal{P}',\mathcal{R}')$ where $\mathcal{P}' = \{\mathsf{s}_1(x)\ -^\sharp\ \mathsf{s}_1(x)) \to x\ -^\sharp\ y\}$ and $\mathcal{R}' = \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U} \cup \{\mathsf{sub}_1(x)^\sharp\ y \to -^\sharp(x,y)\}$. Note that freezing returns nearly the same DP problem. The only difference is that uncurrying produces the additional rule $\mathsf{sub}_1(x)^\sharp\ y \to x\ -^\sharp\ y$ which we do not obtain via freezing. However, since this rule is not usable it also does not harm that much.

Moreover, uncurrying sometimes is applicable where freezing is not. If we would have started with the DP problem $(\mathcal{P},\mathcal{R}'')$ where $\mathcal{R}'' = \{1\text{–}5\}$ then uncurrying would result in $(\llcorner\mathcal{P}\lrcorner,\llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}')$ where $\llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}' = \{8\text{–}13, 16\text{–}20\}$. On this DP problem freezing is not applicable (the instances of $\mathsf{sub}_1(x)$ in the right-hand side of the only pair in $\llcorner\mathcal{P}\lrcorner$ are not root-stable due to Rule 8). Nevertheless, one can uncurry $\circ^\sharp$, resulting in $(\mathcal{P}',\llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}' \cup \mathcal{R}_{\mathsf{new}})$ where $\mathcal{R}_{\mathsf{new}} = \{\mathsf{sub}_1(x)^\sharp\ y \to x\ -^\sharp\ y, \mathsf{0}\ -^\sharp\ y \to \mathsf{K}_1(\mathsf{0})^\sharp\ y\}$. Note that the uncurrying of $\circ^\sharp$ transformed a standard DP problem into a non-standard one, as $-^\sharp$ occurs as root of a term in $\mathcal{P}'$, but also within $\mathcal{R}_{\mathsf{new}}$.

Whenever freezing with $\maltese = \{\circ^\sharp\}$ is applicable, then also uncurrying of $\circ^\sharp$ is possible: the condition $t_1 \notin \mathcal{V}$ in Definition 11.16 implies that $\mathcal{P} \cup \mathcal{R}_\eta$ is left head variable free. The only difference is that uncurrying produces more rules than freezing, namely the uncurrying rules and the uncurried rules of those rules which have to be added for the $\eta$-saturation. However, if freezing is applicable then none of these additional rules are

[3] In [53] one can also specify an argument position for each symbol. This can be simulated by permuting the arguments accordingly.

usable.[4] Hence, all techniques which only consider the usable rules (like the reduction pair processor) perform equally well, no matter whether one has applied freezing or uncurrying. Still, one wants to get rid of the additional rules, especially since they are also the reason why standard DP problems are transformed into non-standard ones.

In Example 11.17 we have seen that sometimes uncurrying of tuple symbols is applicable where freezing is not. Thus, to have the best of both techniques we devised a special uncurrying technique for tuple symbols which fully subsumes freezing without the disadvantage of $\mathcal{U}'_1$: if freezing is applicable then standard DP problems are transformed into standard DP problems by the new technique.

Before we describe the new uncurrying processor formally, we shortly list the differences to the uncurrying processor of Theorem 11.15:

- Since the task is to uncurry tuple symbols, we restrict the applicative arities to be at most one. Moreover, uncurrying is performed only on the top-level. Finally, the application symbol may be of arbitrary non-zero arity.
- Rules that have to be added for the $\eta$-saturation and the uncurrying rules are added as pairs (to the $\mathcal{P}$-component), and not as rules (to the $\mathcal{R}$-component).
- If freezing is applicable, we do neither add the uncurrying rules nor do we apply $\eta$-saturation.

**Example 11.18.** We continue with the DP problems of Example 11.17.

If one applies the special uncurrying processor on $(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$ then one obtains $(\mathcal{P}', \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$ which is the same as $\text{❄}(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$ for $\text{❄} = \{\circ^\sharp\}$.

And if one applies the special uncurrying processor on $(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}')$ then one obtains the standard DP problem $(\mathcal{P}' \cup \mathcal{R}_{\mathsf{new}}, \llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}')$.

**Definition 11.19.** *Let $\circ^\sharp$ be an $n$-ary application symbol where $n > 0$. Let $\pi$ be an injective symbol map where $\pi(f) \in \{[f], [f, f^\sharp]\}$ for all $f$. The top-uncurrying function $\ulcorner . \urcorner$ maps terms to terms. It is defined as $\ulcorner t \urcorner =$*

$$\begin{cases} f^\sharp(s_1, \ldots, s_m, t_2, \ldots, t_n) & \text{if } t = \circ^\sharp(f(s_1, \ldots, s_m), t_2, \ldots, t_n) \text{ and } \pi(f) = [f, f^\sharp] \\ t & \text{otherwise} \end{cases}$$

*and is homomorphically extended to pairs, rules, and substitutions. The* top-uncurrying rules *are defined as*

$$\mathcal{U}^t = \{\circ^\sharp(f(x_1, \ldots, x_m), y_2, \ldots, y_n) \to f^\sharp(x_1, \ldots, x_m, y_2, \ldots, y_n) \mid \pi(f) = [f, f^\sharp]\}$$

*Then the* top-uncurrying processor *is defined as* $\mathsf{top}(\mathcal{P}, \mathcal{R}) =$

$$\begin{cases} (\ulcorner\mathcal{P}_\eta\urcorner \cup \mathcal{U}^t_?, \mathcal{R}) & \text{if } \circ^\sharp \text{ is not defined w.r.t. } \mathcal{R} \text{ and for all } s \to t \in \mathcal{P}_\eta : s, t \notin \mathcal{V}, \\ & s \neq \circ^\sharp(x, s_2, \ldots, s_n), \text{ and the root of } t \text{ is not defined w.r.t. } \mathcal{R} \\ (\mathcal{P}, \mathcal{R}) & \text{otherwise} \end{cases}$$

*where $\mathcal{U}^t_? = \emptyset$ and $\mathcal{P}_\eta = \mathcal{P}$ if for all $s \to \circ^\sharp(t_1, \ldots, t_n) \in \mathcal{P}$ and $\sigma$ the term $t_1\sigma$ is root-stable, and $\mathcal{U}^t_? = \mathcal{U}^t$ and $\mathcal{P}_\eta = \mathcal{P} \cup \{\circ^\sharp(\ell, x_2, \ldots, x_n) \to \circ^\sharp(r, x_2, \ldots, x_n) \mid \ell \to r \in \mathcal{R}, \mathrm{root}(\ell) = g, \pi(g) = [g, g^\sharp]\}$, otherwise. Here, $x_2, \ldots, x_n$ are distinct fresh variables that do not occur in $\ell \to r$.*

[4] In detail: a technique that can detect that instances of a subterm of a right-hand side of $\mathcal{P}$ are root-stable can also detect that the additional rules are not usable.

**Theorem 11.20.** *The top-uncurrying processor* top *is sound.*

*Proof.* The crucial part is to prove that whenever $t = f(t_1, \dots, t_m) \to_{\mathcal{R}}^* s$, $f \notin \mathcal{D}_{\mathcal{R}}$, $t$ is an instance of a right-hand side of $\mathcal{P}$, and $\mathsf{SN}_{\mathcal{R}}(t)$, then $\ulcorner t \urcorner \to_{\ulcorner \mathcal{P}_\eta \urcorner \cup \mathcal{U}_?^t \cup \mathcal{R}}^* \ulcorner s \urcorner$ where $\ulcorner \mathcal{P}_\eta \urcorner \cup \mathcal{U}_?^t$-steps are root steps and all terms in this derivation are terminating w.r.t. $\mathcal{R}$.

Using this result, the main result is established as follows. Assume there is an infinite minimal $(\mathcal{P}, \mathcal{R})$-chain. Then every step $s\sigma \to_{\mathcal{P}} t\sigma \to_{\mathcal{R}}^* s'\sigma'$ in the chain is transformed as follows. Since $s \to t \in \mathcal{P}$, we conclude that $t\sigma = f(t_1\sigma, \dots, t_m\sigma)$ where $f \notin \mathcal{D}_{\mathcal{R}}$ and $\mathsf{SN}_{\mathcal{R}}(t\sigma)$. Hence, using the crucial step we know that $\ulcorner t\sigma \urcorner \to_{\ulcorner \mathcal{P}_\eta \urcorner \cup \mathcal{U}_?^t \cup \mathcal{R}}^* \ulcorner s'\sigma' \urcorner$. Moreover, by case analysis on $t$ one can show that $\ulcorner t \urcorner \sigma \to_{\mathcal{U}_?^t}^* \ulcorner t\sigma \urcorner$ via root reductions, and similarly, by additionally using the restrictions on $s$ one derives $\ulcorner s\sigma \urcorner = \ulcorner s \urcorner \sigma$. Hence,

$$\ulcorner s\sigma \urcorner = \ulcorner s \urcorner \sigma \to_{\ulcorner \mathcal{P} \urcorner} \ulcorner t \urcorner \sigma \to_{\mathcal{U}_?^t}^* \ulcorner t\sigma \urcorner \to_{\ulcorner \mathcal{P}_\eta \urcorner \cup \mathcal{U}_?^t \cup \mathcal{R}}^* \ulcorner s'\sigma' \urcorner$$

where all terms in this derivation right of $\to_{\ulcorner \mathcal{P} \urcorner}$ are terminating w.r.t. $\mathcal{R}$ and where all $\ulcorner \mathcal{P}_\eta \urcorner \cup \mathcal{U}_?^t$-steps are root reductions. Thus, we can turn the root reductions into pairs, resulting in an infinite minimal $(\ulcorner \mathcal{P}_\eta \urcorner \cup \mathcal{U}_?^t, \mathcal{R})$-chain.

To prove the crucial part we perform induction on the number of steps where the base case – no reductions – is trivial. Otherwise, $t = f(t_1, \dots, t_m) \to_{\mathcal{R}}^* u \to_{\mathcal{R}} s$. Using $\mathsf{SN}_{\mathcal{R}}(t)$ we also know $\mathsf{SN}_{\mathcal{R}}(u)$ and since $f \notin \mathcal{D}_{\mathcal{R}}$ we know that $u = f(u_1, \dots, u_m)$ and $t_i \to_{\mathcal{R}}^* u_i$ for all $1 \leqslant i \leqslant m$. Moreover, $s = f(s_1, \dots, s_m)$ and $s$ is obtained from $u$ by a reduction $u_i \to_{\mathcal{R}} s_i$ for some $1 \leqslant i \leqslant m$. Hence, we may apply the induction hypothesis and conclude $\ulcorner t \urcorner \to_{\ulcorner \mathcal{P}_\eta \urcorner \cup \mathcal{U}_?^t \cup \mathcal{R}}^* \ulcorner u \urcorner$.

It remains to simulate the reduction $u \to_{\mathcal{R}} s$. The simulation is easy if $f \neq \circ^\sharp$, since then $\ulcorner u \urcorner = u \to_{\mathcal{R}} s = \ulcorner s \urcorner$. Otherwise, $f = \circ^\sharp$ and $m = n$. We again first consider the easy case where $u_i \to_{\mathcal{R}} s_i$ for some $i > 1$. Then an easy case analysis on $u_1$ yields $\ulcorner u \urcorner \to_{\mathcal{R}} \ulcorner s \urcorner$ since $u$ and $s$ are uncurried in the same way (since $u_1 = s_1$). Otherwise, $u = \circ^\sharp(u_1, \dots, u_m)$, $s = \circ^\sharp(s_1, u_2, \dots, u_m)$ and $u_1 \to_{\mathcal{R}} s_1$. If $u_1 \to_{\mathcal{R}} s_1$ is a reduction below the root then both $s$ and $t$ are uncurried in the same way and again $\ulcorner u \urcorner \to_{\mathcal{R}} \ulcorner s \urcorner$ is easily established. If however $u_1 = \ell\sigma \to r\sigma = s_1$ for some rule $\ell \to r \in \mathcal{R}$ then we know that $u_1$ is not root-stable and hence also $t_1$ is not root-stable. As $t = \circ^\sharp(t_1, \dots, t_n)$ is an instance of a right-hand side of $\mathcal{P}$ we further know that there is a pair $s' \to \circ^\sharp(t'_1, \dots, t'_n) \in \mathcal{P}$ where $t_1 = t'_1\sigma$. Since $t'_1\sigma$ is not root-stable $\mathcal{U}_?^t = \mathcal{U}^t$ and $\mathcal{P}_\eta \supseteq \{\circ^\sharp(\ell, x_2, \dots, x_n) \to \circ^\sharp(r, x_2, \dots, x_n) \mid \ell \to r \in \mathcal{R}, \mathrm{root}(\ell) = g, \pi(g) = [g, g^\sharp]\}$. Let $\ell = g(\ell_1, \dots, \ell_k)$. If $\pi(g) = [g]$ then

$$\begin{aligned}
\ulcorner u \urcorner &= \ulcorner \circ^\sharp(g(\ell_1, \dots, \ell_k)\sigma, u_2, \dots, u_n) \urcorner \\
&= \circ^\sharp(g(\ell_1, \dots, \ell_k)\sigma, u_2, \dots, u_n) \\
&\to_{\mathcal{R}} \circ^\sharp(r\sigma, u_2, \dots, u_n) \\
&\to_{\mathcal{U}_?^t}^* \ulcorner \circ^\sharp(r\sigma, u_2, \dots, u_n) \urcorner \\
&= \ulcorner s \urcorner.
\end{aligned}$$

And otherwise, $\pi(g) = [g, g^\sharp]$. Hence, $\circ^\sharp(\ell, x_2, \dots, x_n) \to \circ^\sharp(r, x_2, \dots, x_n) \in \mathcal{P}_\eta$. We define $\delta = \sigma \uplus \{x_2/u_2, \dots, x_n/u_n\}$ and achieve

$$\begin{aligned}
\ulcorner u \urcorner &= \ulcorner \circ^\sharp(g(\ell_1, \dots, \ell_k)\sigma, u_2, \dots, u_n) \urcorner \\
&= g^\sharp(\ell_1\sigma, \dots, \ell_k\sigma, u_2, \dots, u_n) \\
&= g^\sharp(\ell_1, \dots, \ell_k, x_2, \dots, x_n)\delta \\
&= \ulcorner \circ^\sharp(g(\ell_1, \dots, \ell_k), x_2, \dots, x_n) \urcorner \delta
\end{aligned}$$

$$
\begin{aligned}
&= \ulcorner \circ^\sharp(\ell, x_2, \ldots, x_n)\urcorner\delta \\
&\to_{\ulcorner \mathcal{P}_\eta \urcorner} \ulcorner \circ^\sharp(r, x_2, \ldots, x_n)\urcorner\delta \\
&\to^*_{\mathcal{U}^t_?} \ulcorner \circ^\sharp(r, x_2, \ldots, x_n)\delta\urcorner \\
&= \ulcorner \circ^\sharp(r\sigma, u_2, \ldots, u_n)\urcorner \\
&= \ulcorner s \urcorner.
\end{aligned}
$$

Using that $\pi$ is injective one can also show that termination of all terms in the derivation is guaranteed where we refer to our library IsaFoR for details. □

Note that the top-uncurrying processor fully subsumes freezing since the step from $(\mathcal{P}, \mathcal{R})$ to $(❄(\mathcal{P}), \mathcal{R})$ using $❄ = \{f_1, \ldots, f_n\}$ can be simulated by $n$ applications of top where in each iteration one chooses $f_i$ as application symbol and defines $\pi(g) = [g, f_i^g]$ for all $g \neq f_i$. The following example shows that top is also useful where freezing is not applicable.

**Example 11.21.** Consider the TRS $\mathcal{R}$ where $x \div y$ computes $\lceil \frac{x}{2y} \rceil$.

$$
\begin{aligned}
\mathsf{s}(x) - \mathsf{s}(y) &\to x - y & \qquad \mathsf{double}(x) &\to x + x \\
0 - y &\to 0 & \mathsf{double}(0) &\to 0 \\
x - 0 &\to x & \mathsf{double}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \\
0 + y &\to y & 0 \div \mathsf{s}(y) &\to 0 \\
\mathsf{s}(x) + y &\to \mathsf{s}(x + y) & \mathsf{s}(x) \div \mathsf{s}(y) &\to \mathsf{s}((\mathsf{s}(x) - \mathsf{double}(\mathsf{s}(y))) \div \mathsf{s}(y))
\end{aligned}
$$

Proving termination is hard for current termination provers. Let us consider the interesting DP problem $(\mathcal{P}, \mathcal{R})$ where $\mathcal{P} = \{\mathsf{s}(x) \div^\sharp \mathsf{s}(y) \to (\mathsf{s}(x) - \mathsf{double}(\mathsf{s}(y))) \div^\sharp \mathsf{s}(y)\}$. The problem is that one cannot use standard reduction pairs with argument filters since one has to keep the first argument of $-$, and then the filtered term of $\mathsf{s}(x)$ is embedded in the filtered term of $\mathsf{s}(x) - \mathsf{double}(\mathsf{s}(y))$. Consequently, powerful termination provers such as AProVE and TTT2 fail on this TRS.

However, one can uncurry the tuple symbol $\div^\sharp$ where $\pi(-) = [-, -^\sharp]$, $\pi(\mathsf{s}) = [\mathsf{s}, \mathsf{s}^\sharp]$, and $\pi(f) = [f]$, otherwise. Then the new DP problem $(\mathcal{P}', \mathcal{R})$ is created where $\mathcal{P}'$ consists of the following pairs

$$
\begin{aligned}
(x - y) \div^\sharp z &\to -^\sharp(x, y, z) & \qquad -^\sharp(\mathsf{s}(x), \mathsf{s}(y), z) &\to -^\sharp(x, y, z) \\
\mathsf{s}(x) \div^\sharp y &\to \mathsf{s}^\sharp(x, y) & -^\sharp(0, y, z) &\to 0 \div^\sharp z \\
\mathsf{s}^\sharp(x, \mathsf{s}(y)) &\to -^\sharp(\mathsf{s}(x), \mathsf{double}(\mathsf{s}(y)), \mathsf{s}(y)) & -^\sharp(x, 0, z) &\to x \div^\sharp z
\end{aligned}
$$

where the subtraction is computed via the new pairs, and not via the rules anymore. The right column consists of the uncurried and $\eta$-saturated $-$-rules, and the left column contains the two uncurrying rules followed by the uncurried pair of $\mathcal{P}$. Proving finiteness of this DP problem is possible using standard techniques: linear 0/1-polynomial interpretations and the dependency graph suffice. Therefore, termination of the whole example can be proven fully automatically by using a new version of TTT2 where top-uncurrying is integrated.

## 11.5. Heuristics and Experiments

The generalizations for uncurrying described in this paper are implemented in TTT2 [67]. To fix the symbol map we used the following three heuristics:

- $\pi_+$ corresponds to the definition of applicative arity of [53]. More formally, $\pi_+(f) = [f_0, \ldots, f_n]$ where $n$ is maximal w.r.t. all $f(\ldots) \circ t_1 \circ \cdots \circ t_n$ occurring in $\mathcal{R}$. The advantage of $\pi_+$ is that all uncurryings are performed.

- $\pi_\pm$ is like $\pi_+$, except that the applicative arity is reduced whenever we would have to add a rule during $\eta$-saturation. Formally, $\pi_\pm(f) = [f_0, \ldots, f_n]$ where $n = \min(\mathsf{aa}_{\pi_+}(f), \min\{k \mid f(\ldots) \circ t_1 \circ \cdots \circ t_k \to r \in \mathcal{R}\})$.

- $\pi_-$ is almost dual to $\pi_+$. Formally, $\pi_-(f) = [f_0, \ldots, f_n]$ where $n$ is minimal w.r.t. all maximal subterms of the shape $f(\ldots) \circ t_1 \circ \cdots \circ t_n$ occurring in $\mathcal{R}$. The idea is to reduce the number of uncurrying rules.

We conducted two sets of experiments to evaluate our work. Note that all proofs generated during our experiments are certified by CeTA (version 1.18). Our experiments were performed on a server with eight dual-core AMD Opteron® processors 885, running at a clock rate of 2.6 GHz and on 64 GB of main memory. The time limit for the first set of experiments was 10 s (as in [53]), whereas the time limit for the second set was 5 s (T$_\text{T}$T$_2$'s time limit in the termination competition).

The first set of experiments was run with a setup similar to [53]. Accordingly, as input we took the same 195 ATRSs from the termination problem database (TPDB). For proving termination, we switch from the input TRS to the initial DP problem and then repeat the following as often as possible: compute the estimated dependency graph, split it into its strongly connected components and apply the "main processor." Here, as "main processor" we incorporated the subterm criterion and matrix interpretations (of dimensions one and two). Concerning uncurrying, the following approaches were tested: no uncurrying (none), uncurry the given TRS before computing the initial DP problem (trs), apply $\mathcal{U}'_1/\mathcal{U}'_2$ as soon as all other processors fail (where $\mathcal{U}'_2$ is the composition of $\mathcal{U}'_1$ and top). The results can be found in Table 11.1. Since on ATRSs, our generalization of uncurrying corresponds to standard uncurrying, it is not surprising that the numbers of the first three columns coincide with those of [53] (modulo *mirroring* and a slight difference in the used strategy for trs). They are merely included to see the relative gain when using uncurrying on ATRSs.

With the second set of experiments, we tried to evaluate the total gain in certified termination proofs. Therefore, we took a restricted version of T$_\text{T}$T$_2$'s competition strategy that was used in the July 2010 issue of the international termination competition[5] (called *base strategy* in the following). The restriction was to use only those termination techniques that where certifiable by CeTA before our formalization of uncurrying. Then, we used this base strategy to filter the TRSs (we did ignore all SRSs) of the TPDB (version 8.0). The result were 511 TRSs for which T$_\text{T}$T$_2$ did neither generate a termination proof nor a nontermination proof using the base strategy. For our experiments we extended the base strategy by the generalized uncurrying techniques using different heuristics for the applicative arity. The results can be found in Table 11.2. It turned out, that the $\pi_-$ heuristic is rather weak. Concerning $\pi_\pm$, there is at least one TRS that could not be proven using $\pi_+$, but with $\pi_\pm$. The total of 35 in the first row of Table 11.2 is already reached without taking $\mathcal{U}'_1$ into account. This indicates that in practice a combination of uncurrying as initial step (trs) and the processor $\mathcal{U}'_2$, gives the best results. Finally, note that in comparison to the July 2010 termination competition (where T$_\text{T}$T$_2$ could generate 262 certifiable proofs), the number of certifiable proofs of T$_\text{T}$T$_2$ is increased by over

[5] http://termcomp.uibk.ac.at

Table 11.1.: Experiments as in [53]

| | direct | | processor | |
|---|---|---|---|---|
| | none | trs | $\mathcal{U}'_1$ | $\mathcal{U}'_2$ |
| subterm criterion | 41 | 53 | 41 | 66 |
| matrix (dimension 1) | 66 | 98 | 95 | 114 |
| matrix (dimension 2) | 108 | 137 | 133 | 138 |

Table 11.2.: Newly certified proofs

| | direct | processor | | total |
|---|---|---|---|---|
| | trs | $\mathcal{U}'_1$ | $\mathcal{U}'_2$ | |
| $\pi_+$ | 26 | 16 | 22 | 35 |
| $\pi_\pm$ | 28 | 15 | 17 | 29 |
| $\pi_-$ | 24 | 14 | 14 | 24 |
| total | 28 | 16 | 24 | 36 |

10 % using the new techniques. In these experiments, termination has been proven for 10 *non-applicative* TRSs where our generalizations of uncurrying have been the key to success.

## 11.6. Conclusions

This paper describes the first formalization of uncurrying, an important technique to prove termination of higher-order functions which are encoded as first-order TRSs. The formalization revealed a gap in the original proof which is now fixed. Adding the newly developed generalization of uncurrying to our certifier CeTA, increased the number of certifiable proofs on the TPDB by 10 %.

# 12. On the Formalization of Termination Techniques Based on Multiset Orderings


### Publication details

René Thiemann, Guillaume Allais, and Julian Nagele. On the Formalization of Termination Techniques Based on Multiset Orderings. In *Proceedings of the 23rd International Conference on Rewriting Techniques and Applications*, volume 15 of LIPIcs, pages 339–354. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.



### Abstract

Multiset orderings are a key ingredient in certain termination techniques like the recursive path ordering and a variant of size-change termination. In order to integrate these techniques in a certifier for termination proofs, we have added them to the *Isabelle Formalization of Rewriting*. To this end, it was required to extend the existing formalization on multiset orderings towards a generalized multiset ordering. Afterwards, the soundness proofs of both techniques have been established, although only after fixing some definitions.

Concerning efficiency, it is known that the search for suitable parameters for both techniques is NP-hard. We show that checking the correct application of the techniques—where all parameters are provided—is also NP-hard, since the problem of deciding the generalized multiset ordering is NP-hard.


## 12.1. Introduction

The multiset ordering has been invented in the '70s to prove termination of programs [29]. It is an ingredient for important termination techniques like the multiset path ordering (MPO) [26], the recursive path ordering (RPO) [27], or recently [7,20] it has been used in combination with the size-change principle of [74], in the form of SCNP reduction pairs.

The original version of the multiset ordering w.r.t. some base ordering $\succ$ can be defined as $M \succ_{ms} N$ iff it is possible to obtain $N$ from $M$ by replacing at least one element of $M$ by several strictly smaller elements in $N$.

In other papers—like [7, 20, 92]—a generalization of the multiset ordering is used (denoted by $\succ_{gms}$). To define $\succ_{gms}$ one assumes that in addition to $\succ$ there is a compatible non-strict ordering $\succsim$. Then $\succ_{gms}$ is like $\succ_{ms}$, but in the multiset comparison it is additionally allowed to replace each element by a smaller one (w.r.t. $\succsim$). To illustrate the difference between $\succ_{ms}$ and $\succ_{gms}$, we take $\succ$ and $\succsim$ as the standard orderings on polynomials over the naturals. Then $\succ_{gms}$ is strictly more powerful than $\succ_{ms}$: for example,

$\{\{x+1, 2y\}\} \succ_{gms} \{\{y, x\}\}$ since $x+1 \succ x$ and $2y \succsim y$, but $\{\{x+1, 2y\}\} \succ_{ms} \{\{y, x\}\}$ does not hold, since $2y \neq y$ and also $2y \not\succ y$ as $y$ can be instantiated by 0.

Note that $\succ_{gms}$ is indeed used in powerful termination tools like AProVE [39]. Hence, for the certification of termination proofs which make use of SCNP reduction pairs or RPOs which are defined via $\succ_{gms}$, we need a formalization of this multiset ordering.

However, in the literature and in formalizations, often only $\succ_{ms}$ is considered. And even those papers that use $\succ_{gms}$ only shortly list the differences between $\succ_{ms}$ and $\succ_{gms}$—if at all—and afterwards just assume that both multiset orderings have similar properties.

To change this situation, as one new contribution of this paper, we give the first formalization of $\succ_{gms}$, and could indeed show that $\succ_{ms}$ and $\succ_{gms}$ behave quite similar. However, we also found one essential difference between both orderings. It is well known that deciding $\succ_{ms}$ is easy, one just removes identical elements and afterwards has to find for each element in the one set a larger element in the other. In contrast, we detected and proved that deciding $\succ_{gms}$ is an NP-complete problem.

Note that the original definition of RPO misses a feature that is present in other orderings like polynomial interpretations where it is possible to compare variables against a least element: $x \succsim 0$. This feature was already integrated in other orderings like KBO [65], and it can also be integrated into RPO where one allows to compare $x \succsim c$ if $c$ is a constant of least precedence. For example, the internal definition of RPO in AProVE is the one of [92]—which is based on $\succ_{gms}$—with the additional inference rule of "$x \succsim c$". As a second new contribution we formalized this RPO variant and fixed its definition, as it turned out that it is not stable. Moreover, we show that the change from $\succ_{ms}$ to $\succ_{gms}$ increases the complexity of RPO: From [71] it is known that deciding whether two terms are in relation w.r.t. a given RPO or MPO is in P. However, if one uses the definitions of MPO and RPO in this paper which are based on $\succ_{gms}$, then the same decision problem becomes NP-complete.

As third new contribution we also give the first formalization of SCNP reduction pairs where we could establish the main soundness theorem, although several definitions had to be fixed. Again, we have shown that the usage of $\succ_{gms}$ makes certification for SCNP reduction pairs an NP-complete problem. As a consequence, the search for suitable SCNP reduction pairs and the problem whether two terms are in relation for a given SCNP reduction pair belong to the same complexity class, as they are both NP-complete.

All our formalizations are using the proof assistant Isabelle/HOL [84]. They are available within the IsaFoR library (Isabelle Formalization of Rewriting, [104]). The new parts of the corresponding proof checker CeTA—which is just obtained by applying the code generator of Isabelle [49] on IsaFoR—have been tested on the examples of the experiments performed in [92]. After fixing some output bugs, eventually all proofs could be certified. Both IsaFoR and CeTA are freely available at:

`http://cl-informatik.uibk.ac.at/software/ceta`

The paper is structured as follows. We give preliminaries and the exact definitions for $\succ_{ms}$ and $\succ_{gms}$ in Sec. 12.2. Afterwards we discuss the formalization of $\succ_{gms}$ in Sec. 12.3. Different variants of RPO are discussed in Sec. 12.4. Here, we also show that certifying constraints for the RPO variant used in AProVE is NP-complete. In Sec. 12.5 we discuss the formalization of SCNP reduction pairs. Finally, in Sec. 12.6 we elaborate on our certified algorithms for checking whether two terms are in relation w.r.t. RPO or the orderings from an SCNP reduction pair, and we report on our experimental results.

## 12.2. Preliminaries

We refer to [5] for basic notions and notations of rewriting. A *signature* is a set of symbols $\mathcal{F} = \{f, g, F, G, \dots\}$, each associated with an *arity*. We write $\mathcal{T}(\mathcal{F}, \mathcal{V})$ for the set of terms over signature $\mathcal{F}$ and set of variables $\mathcal{V}$. We write $\mathcal{V}(t)$ for the set of variables occurring in $t$. A relation $\succ$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is *stable* iff it is closed under substitutions, and it is *monotone*, iff it is closed under contexts. A *term rewrite system* (TRS) is a set of *rules* $\ell \to r$. The rewrite relation $\to_{\mathcal{R}}$ of a TRS $\mathcal{R}$ is the smallest stable and monotone relation containing $\mathcal{R}$.

We write $Id$ for the identity relation, and for each relation $\succ$, let $\succeq$ be its reflexive closure.

**Definition 12.1** (Ordering pair, reduction pair)**.** *The pair $(\succsim, \succ)$ is an* ordering pair *(over carrier $A$) iff $\succsim$ is a quasi-ordering over $A$, $\succ$ is a transitive and well-founded relation over $A$, and $\succ$ and $\succsim$ are compatible, i.e., $\succ \circ \succsim \subseteq \succ$ and $\succsim \circ \succ \subseteq \succ$.*

*The pair $(\succsim, \succ)$ is a* non-monotone reduction pair *iff $(\succsim, \succ)$ is an ordering pair over $\mathcal{T}(\mathcal{F}, \mathcal{V})$ where both $\succ$ and $\succsim$ are stable. If additionally $\succsim$ is monotone, then $(\succsim, \succ)$ is a* reduction pair*, and if both $\succ$ and $\succsim$ are monotone, then $(\succsim, \succ)$ is a* monotone reduction pair*.*

Throughout this paper we only consider finite multisets $\{\{x_1, \dots, x_n\}\}$ and we write $\mathfrak{P}(A)$ for the set of all multisets with elements from $A$. For every function $f : A \to A$ and every multiset $M \in \mathfrak{P}(A)$ we define the image of $f$ on $M$ as $f[M] = \{\{f(x) \mid x \in M\}\}$.

**Definition 12.2** (Multiset orderings)**.** *Let $\succ$ and $\succsim$ be relations over $A$. We define the* multiset ordering *($\succ_{ms}$), the* generalized multiset ordering *($\succ_{gms}$), and the corresponding non-strict ordering ($\succsim_{gms}$) over $\mathfrak{P}(A)$ as follows: $M_1 \succ_{ms} / \succ_{gms} / \succsim_{gms} M_2$ iff there are $S_i$ and $E_i$ such that $M_1 = S_1 \cup E_1$, $M_2 = S_2 \cup E_2$, and*

- *conditions i, ii, and iv are satisfied: $M \succ_{ms} N$*
- *conditions i, iii, and iv are satisfied: $M \succ_{gms} N$*
- *conditions i and iii are satisfied: $M \succsim_{gms} N$*

*where conditions i–iv are defined by:*

*(i)* *for each $y \in S_2$ there is some $x \in S_1$ with $x \succ y$*

*(ii)* *$E_1 = E_2$*

*(iii)* *$E_1 = \{\{x_1, \dots, x_n\}\}$, $E_2 = \{\{y_1, \dots, y_n\}\}$, and $x_i \succsim y_i$ for all $1 \le i \le n$*

*(iv)* *$S_1 \neq \emptyset$*

*Whenever $M_i$ is split into $S_i \cup E_i$ we call $S_i$ the* strict part *and $E_i$ the* non-strict part*.*

Note that $\succ_{gms}$ indeed generalizes $\succ_{ms}$, since $\succ_{gms} = \succ_{ms}$ if $\succsim = Id$.

## 12.3. Formalization of the Generalized Multiset Ordering

Replacing condition ii by iii makes the formalization of $\succ_{gms}$ a bit more involved than the one of $\succ_{ms}$: the simple operation of (multiset) equality $E_1 = E_2$ is replaced by demanding that both $E_1$ and $E_2$ can be enumerated in such a way that $x_i \succsim y_i$ for all $1 \leq i \leq n$. Note that instead of enumerations one can equivalently demand that there is a bijection $f : E_1 \to E_2$ such that for all $x \in E_1$ we have $x \succsim f(x)$.

There is one main advantage of formalizing condition iii via enumerations instead of bijections. It will be rather easy to develop an algorithm deciding $\succ_{gms}$. However, using enumerations we also observe a drawback: the proof that $\succ_{gms}$ and $\succsim_{gms}$ are compatible and transitive orderings will be harder than using bijections since composing bijections is easier than combining enumerations.

The formalization of the following (expected) properties for $\succ_{gms}$ and $\succsim_{gms}$ has been rather simple.

**Lemma 12.3.** *$\succ_{gms}$ and $\succsim_{gms}$ have the following properties:*

*(i) the empty set is the unique minimum of $\succ_{gms}$ and $\succsim_{gms}$*

*(ii) if $\succsim$ is reflexive then so is $\succsim_{gms}$*

*(iii) if $\succ$ is irreflexive and compatible with $\succsim$ then $\succ_{gms}$ is irreflexive*

*(iv) if $\succ$ and $\succsim$ are closed under an operation op then $\succ_{gms}$ and $\succsim_{gms}$ are closed under op$[\cdot]$.*

In contrast, the formalization of transitivity and compatibility was more tedious.

**Lemma 12.4.** *If $\succ$ and $\succsim$ are compatible and transitive then so are $\succ_{gms}$ and $\succsim_{gms}$.*

*Proof.* For the sake of simplicity, we will work here with the definition of $(\succ_{gms}, \succsim_{gms})$ using bijections rather than enumerations: all technical details when using the representation with enumerations are available in IsaFoR, theory *Multiset-Extension*. Given that this lemma states four results that are pretty similar, we will only prove transitivity of $\succsim_{gms}$ and let the reader see how the proof could be slightly modified in the other cases.

Let $M, N$ and $P$ be three multisets such that $M \succsim_{gms} N$ (1) and $N \succsim_{gms} P$ (2). From (1) we get the partitions $M = M' \cup E$ and $N = N' \cup F$ and the bijection $f_{MN} : E \to F$. From (2) we get the partitions $N = N'' \cup F'$ and $P = P' \cup G$ and the bijection $f_{NP} : F' \to G$. We define the multiset $I$ as $I = F \cap F'$ and claim that the partitions $M = (M \setminus f_{MN}^{-1}[I]) \cup f_{MN}^{-1}[I]$ and $P = (P \setminus f_{NP}[I]) \cup f_{NP}[I]$ and the bijection $f_{NP} \circ f_{MN}$ are the ones needed to prove $M \succsim_{gms} P$.

If $p \in P \setminus f_{NP}[I]$ then we have to find some $m \in M \setminus m \notin f_{MN}^{-1}[I]$ satisfying $m \succ p$. We distinguish two cases. In the first case, $p \in P'$ and thus there is an $n \in N''$ such that $n \succ p$ and $n \notin I$. If $n \in N'$ then by (1) there is some $m \in M'$ with $m \succ n$ and hence, $m \succ p$ by transitivity of $\succ$. Moreover, since $M' \subseteq M \setminus f_{MN}^{-1}[I]$ we found the desired element $m$. Otherwise, $n \in F$ and hence, for $m = f_{MN}^{-1}(n)$ we know $m \succsim n$ using (1). By compatibility, also $m \succ p$. Again, $m \in M \setminus f_{MN}^{-1}[I]$ since $n \notin I$. In the second case, $p \in G$ and thus for $n = f_{NP}^{-1}(p)$ we conclude $n \in F'' \setminus I$ and $n \succsim p$, and hence $n \notin F$. Thus, there is an element $m \in M'$ which satisfies $m \succ n$. By compatibility we again achieve $m \succ p$.

If $p \in f_{NP}[I]$, we have an element $n \in F'$ such that $n \succsim p$; $n$ is also in $F$ and therefore we have an element $m \in f_{MN}^{-1}[I]$ such that $m \succsim n \succsim p$ and hence, $m \succsim p$. □

Here lies the main difference to the formalization of $\succ_{ms}$ in [63]. While just transitivity needs to be shown for $\succ_{ms}$, we had to show transitivity of both $\succ_{gms}$ and $\succsim_{gms}$ as well as compatibility from both sides. Moreover the formalized proofs of these facts get more complicated since we cannot simply use bijections, set intersections, and differences, but have to deal with enumerations.

After having established Lem. 12.3 and Lem. 12.4 it remains to prove the most interesting and also most complicated property of $\succ_{gms}$, namely strong normalization. In the remainder of this section we assume that $\succ$ and $\succsim$ are compatible and transitive, and that $\succ$ is strongly normalizing. Our proof is closely related to the one for $\succ_{ms}$ [83] which is due to Buchholz: we first introduce a more atomic relation $\succ_{gms\text{-}step}$ which has $\succ_{gms}$ as its transitive closure. Then it suffices to prove that $\succ_{gms\text{-}step}$ is well-founded.

**Definition 12.5.** *We define $\succ_{gms\text{-}step}$ as $M \succ_{gms\text{-}step} N$ iff $M \succ_{gms} N$ where in the split $M = S \cup E$ the size of $S$ is exactly one.*

**Lemma 12.6.** *$\succ_{gms}$ is the transitive closure of $\succ_{gms\text{-}step}$.*

For strong normalization we perform an accessibility-style proof, i.e., we define $\mathcal{A}$ as the set of strongly normalizing elements w.r.t. $\succ_{gms\text{-}step}$ and show that $\mathcal{A}$ contains all multisets. Note that to show $M \in \mathcal{A}$ it suffices to prove strong normalization for all $N$ with $M \succ_{gms\text{-}step} N$. Moreover, since $\succ_{gms\text{-}step}$ is strongly normalizing on $\mathcal{A}$, one can use the following induction principle to prove some property $P$ for all elements in $\mathcal{A}$.

$$(\forall M.(\forall N \in \mathcal{A}.M \succ_{gms\text{-}step} N \to P(N)) \to P(M)) \to (\forall M \in \mathcal{A}.P(M)) \qquad (\star)$$

We will later on apply this induction scheme using the first of the following two predicates:

- $P(x)$ is defined as $\forall M.M \in \mathcal{A} \to M \cup \{\{x\}\} \in \mathcal{A}$
- $Q(M, x)$ is defined as $\forall b.x \succsim b \to M \cup \{\{b\}\} \in \mathcal{A}$

For showing that all multisets belong to $\mathcal{A}$, we will require the following technical lemma.

**Lemma 12.7.** *For all multisets $M \in \mathcal{A}$ and for all elements $a$, if $\forall N.M \succ_{gms\text{-}step} N \to Q(N, a)$ and $\forall b.a \succ b \to P(b)$ then $Q(M, a)$ holds.*

*Proof.* To prove $Q(M, a)$, let $b$ be an element such that $a \succsim b$ where we have to show $M \cup \{\{b\}\} \in \mathcal{A}$. To prove the latter, we consider an arbitrary $N$ with $M \cup \{\{b\}\} \succ_{gms\text{-}step} N$ and have to show $N \in \mathcal{A}$. From $M \cup \{\{b\}\} \succ_{gms\text{-}step} N$ we obtain suitable $m_1, \ldots, m_k$, $n_1, \ldots, n_k$, $m$, and $N'$ to perform the splits $M \cup \{\{b\}\} = \{\{m\}\} \cup \{\{m_1, \ldots, m_k\}\}$ and $N = N' \cup \{\{n_1, \ldots, n_k\}\}$. We distinguish two cases: either $b$ is part of $\{\{m_1, \ldots, m_k\}\}$ or equal to $m$.

If $b = m_i$ for some $i$ then $M \succ_{gms\text{-}step} N \setminus \{\{n_i\}\}$. Thanks to the first hypothesis and the fact that $a \succsim n_i$ (because $a \succsim b = m_i \succsim n_i$), we can conclude that $N \in \mathcal{A}$.

If $b = m$ then one can prove $\{\{n_1, \ldots, n_k\}\} \in \mathcal{A}$ using the fact that $\{\{m_1, \ldots, m_k\}\} = M \in \mathcal{A}$ and $\forall i.m_i \succsim n_i$. By induction on the size of $N'$ and thanks to the second hypothesis and the fact that for any $p \in N'$, $a \succsim b \succ p$, we deduce that $\{\{n_1, \ldots, n_k\}\} \cup N' \in \mathcal{A}$ which concludes the proof. □

**Lemma 12.8.** $\forall M.M \in \mathcal{A}$.

*Proof.* The proof of the lemma is done by induction on the size of the multiset $M$.

If $M$ is the empty multiset, then obviously, it is strongly normalizing (hence in $\mathcal{A}$).

Otherwise we have to prove $\forall a.\forall M \in \mathcal{A}.M \cup \{\{a\}\} \in \mathcal{A}$ which is the same as $\forall a.P(a)$. We perform a well-founded induction on $a$ (w.r.t. $\succ$) using the property $P$ and are left to prove $P(a)$ assuming that $\forall b.a \succ b \to P(b)$ holds. We pick a multiset $M \in \mathcal{A}$ and perform an induction on $M$ using $(\star)$ to prove the property $Q(M, a)$ (which entails $P(a)$ because $\succsim$ is reflexive): for any $M \in \mathcal{A}$ we have to prove $Q(M, a)$ given the induction hypothesis $\forall N.M \succ_{gms\text{-}step} N \to Q(N, a)$. But this result is a trivial application of Lem. 12.7 using the two induction hypotheses that we just generated. □

Using Lem. 12.6 and Lem. 12.8, strong normalization of $\succ_{gms}$ immediately follows.

**Theorem 12.9.** *$\succ_{gms}$ is strongly normalizing.*

## 12.4. Multiset and Recursive Path Ordering

In this section we study variations of MPO and RPO. To this end, throughout this section we assume that $\geqslant$ is some precedence for the signature $\mathcal{F}$. We write $> = \geqslant \setminus \leqslant$ for the strict part of the precedence, and $\approx = \geqslant \cap \leqslant$ for its equivalence relation.

A standard version of MPO allowing quasi-precedences can be defined by the following inference rules.

$$\frac{s_i \succeq^{mpo} t}{f(\vec{s}) \succ^{mpo} t} \qquad \frac{\{\{\vec{s}\}\} \succ^{mpo}_{ms} \{\{\vec{t}\}\}}{f(\vec{s}) \succ^{mpo} g(\vec{t})} f \approx g \qquad \frac{f(\vec{s}) \succ^{mpo} t_i \text{ for all } i}{f(\vec{s}) \succ^{mpo} g(\vec{t})} f > g$$

For example for the precedence where $\mathsf{g} \approx \mathsf{h}$ we conclude that $\mathsf{f}(y, \mathsf{s}(x)) \succ^{mpo} \mathsf{f}(x, y)$ and $\mathsf{g}(\mathsf{s}(x)) \succ^{mpo} \mathsf{h}(x)$, but $\mathsf{f}(\mathsf{g}(y), \mathsf{s}(x)) \not\succ^{mpo} \mathsf{f}(x, \mathsf{h}(y))$ since $\{\{\mathsf{g}(y), \mathsf{s}(x)\}\} \not\succ^{mpo}_{ms} \{\{x, \mathsf{h}(y)\}\}$ as $\mathsf{g}(y) \not\succeq^{mpo} \mathsf{h}(y)$.

To increase the power of MPO, the following inference rules are sometimes used which generalize MPO by defining two orderings $\succ^{gmpo}$ and $\succsim^{gmpo}$ where the multiset extension is done via $\succ_{gms}$. This variant of MPO is the one that is internally used within AProVE, and if one removes the last inference rule ($x \succsim^{gmpo} c$), then it is equivalent to the MPO definition of [92].

$$(1)\ \frac{s_i \succsim^{gmpo} t}{f(\vec{s}) \succ^{gmpo} t} \qquad (2)\ \frac{\{\{\vec{s}\}\} \succ^{gmpo}_{gms} \{\{\vec{t}\}\}}{f(\vec{s}) \succ^{gmpo} g(\vec{t})} f \approx g \qquad (3)\ \frac{f(\vec{s}) \succ^{gmpo} t_i \text{ for all } i}{f(\vec{s}) \succ^{gmpo} g(\vec{t})} f > g$$

$$(4)\ \frac{s \succeq^{gmpo} t}{s \succsim^{gmpo} t} \qquad (5)\ \frac{\{\{\vec{s}\}\} \succsim^{gmpo}_{gms} \{\{\vec{t}\}\}}{f(\vec{s}) \succsim^{gmpo} g(\vec{t})} f \approx g \qquad (6)\ \frac{}{x \succsim^{gmpo} c} \text{if } \forall f \in \mathcal{F} : f \geqslant c$$

Hence, there are two differences between $\succeq^{gmpo}$ (the reflexive closure of $\succ^{gmpo}$) and $\succsim^{gmpo}$: only in $\succsim^{gmpo}$ one can compare multisets using the non-strict multiset ordering, and one can compare variables against constants of least precedence. This latter feature is similar to polynomial orderings where $x \geqslant 0$, and it was also added to the Knuth-Bendix-Ordering [65].

The increase of power of $\succ^{gmpo}$ in comparison to $\succ^{mpo}$ is due to both new features in the non-strict relation. For example, using the same precedence as before, $\mathsf{f}(\mathsf{g}(y), \mathsf{s}(x)) \succ^{gmpo} \mathsf{f}(x, \mathsf{h}(y))$ as $\mathsf{g}(y) \succsim^{gmpo} \mathsf{h}(y)$ and $\mathsf{s}(x) \succ^{gmpo} x$. Moreover, $\mathsf{f}(\mathsf{f}(y, z), \mathsf{s}(x)) \succ^{gmpo} \mathsf{f}(x, \mathsf{f}(\mathsf{a}, z))$ if $\mathsf{a}$ has least precedence, where this decrease is only possible due to the comparison $y \succsim^{gmpo} \mathsf{a}$.

Note that $\succsim^{gmpo}$ is a strict superset of the equivalence relation where equality is defined modulo $\approx$ and modulo permutations. For this inclusion, indeed all three inference rules (4-6) of $\succsim^{gmpo}$ are essential. With (4) and (5) we completely subsume the equivalence relation, and (6) exceeds the equivalence relation. However, then also (5) adds additional power by using $\succsim_{gms}$ instead of multiset equality w.r.t. the equivalence relation. As an example, consider $\mathsf{g}(x) \succsim^{gmpo} \mathsf{g}(\mathsf{a})$.

Finally note that our definition does not require any explicit definition of an equivalence relation, one can just use $\succsim^{gmpo} \cap \precsim^{gmpo}$. This is in contrast to the RPO in related formalizations of CoLoR [13] and Coccinelle [21]. In Coccinelle first an equivalence relation for RPO is defined explicitly, before defining the strict ordering,[1] and the RPO of CoLoR currently just supports syntactic equality.[2]

The reason that AProVE uses $\succ^{gmpo}$ for its MPO implementation is easily understood, as $\succ^{gmpo}$ is more powerful than $\succ^{mpo}$, and the SAT/SMT-encodings of $\succ^{mpo}$ and $\succ^{gmpo}$ to find a suitable precedence are quite similar, so there is not much overhead. Hence, in order to be able to certify AProVE's MPO proofs—which then also allows to certify weaker variants of MPO—we have formally proved that $(\succsim^{gmpo}, \succ^{gmpo})$ is a monotone reduction pair. Concerning strong normalization of $\succ^{gmpo}$, we did not use Kruskal's tree theorem, but we performed a proof similar to the strong normalization proof of the (higher-order) recursive path ordering as in [13, 21, 57, 63] (which is based on reducibility predicates of Tait and Girard.) By following this proof and by using the results of Sec. 12.3, it was an easy—but tedious—task to formalize the following main result. Here—as for the generalized multiset ordering—the transitivity proof became more complex as one has to prove transitivity and compatibility of $\succsim^{gmpo}$ and $\succ^{gmpo}$ at the same time, i.e., within one large inductive proof.

**Theorem 12.10.** *The pair $(\succsim^{gmpo}, \succ^{gmpo})$ is a monotone reduction pair.*

Whereas Thm. 12.10 was to be expected—$\succ^{gmpo}$ is just an extension of $\succ^{mpo}$, and it is well known that $(\succeq^{mpo}, \succ^{mpo})$ is a monotone reduction pair—we detected a major difference when trying to certify existing proofs where one has to compute for a given precedence whether two terms are in relation. This problem is in P for $\succ^{mpo}$ but it turns out to be NP-complete for $\succ^{gmpo}$.

**Theorem 12.11.** *Let there be some fixed precedence. The problem of deciding $\ell \succ^{gmpo} r$ for two terms $\ell$ and $r$ is NP-complete.*

*Proof.* Membership in NP is easily proved. Since the size of every proof-tree for $\ell \succ^{mpo} r$ is bounded by $2 \cdot |\ell| \cdot |r|$, one can just guess how the inference rules for $\succ^{mpo}$ have to be applied; and for the multiset comparisons one can also just guess the splitting.

To show NP-hardness we perform a reduction from SAT. So let $\phi$ be some Boolean formula over variables $\{x_1, \ldots, x_n\}$ represented as a set of clauses $\{C_1, \ldots, C_m\}$ where each clause is a set of literals, and each literal $l$ is variable $x_i$ or a negated variable $\overline{x_i}$. W.l.o.g. we assume $n \geq 2$.

In the following we will construct one constraint $\ell \succ^{gmpo} r$ for terms $\ell, r \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ where $\mathcal{F} = \{\mathsf{a}, \mathsf{f}, \mathsf{g}, \mathsf{h}\}$ and $\mathcal{V} = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$. To this end, we define $s(l, C_j) = y_j$, if $l \in C_j$, and $s(l, C_j) = \mathsf{a}$, otherwise. Moreover, $t_x^+ = \mathsf{f}(x, s(x, C_1), \ldots, s(x, C_m))$, $t_x^- = \mathsf{f}(x, s(\overline{x}, C_1), \ldots, s(\overline{x}, C_m))$, $t_x = \mathsf{f}(x, \mathsf{a}, \ldots, \mathsf{a})$. We define $L = \{\{t_{x_1}^+, t_{x_1}^-, \ldots, t_{x_n}^+, t_{x_n}^-\}\}$

[1] See `http://www.lri.fr/~contejea/Coccinelle/doc/term_orderings.rpo.html`.
[2] See `http://color.inria.fr/doc/CoLoR.RPO.VRPO_Type.html`.

and $R = \{\{t_{x_1}, \ldots, t_{x_n}, y_1, \ldots, y_m\}\}$. Finally, we define $\ell = \mathsf{g}(L)$ and $r = \mathsf{h}(R)$—where here we abuse notation and interpret $L$ and $R$ as lists of terms.

We prove that $\phi$ is satisfiable iff $\ell \succ^{gmpo} r$ for the precedence where $\mathsf{a} \approx \mathsf{f} \approx \mathsf{g} \approx \mathsf{h}$. For this precedence, $\ell \succ^{gmpo} r$ iff $L \succ^{gmpo}_{gms} R$ since there is only one inference rule that can successfully be applied. The reason is that $\mathsf{f} \not> \mathsf{g}$ and for each term $t^{\pm}_{x_i}$ of $L$ we have $t^{\pm}_{x_i} \not\succsim^{gmpo} r$ where $t^{\pm}_{x_i}$ represents one of the terms $t^{+}_{x_i}$ or $t^{-}_{x_i}$. To see the latter, assume $t^{\pm}_{x_i} \succsim^{gmpo} r$ would hold. Then $\{x_i, y_1, \ldots, y_m\} \supseteq \mathcal{V}(t^{\pm}_{x_i}) \supseteq \mathcal{V}(r) \supseteq \{x_1, \ldots, x_n\}$ being a contradiction to $n \geq 2$.

To examine whether $L \succ^{gmpo}_{gms} R$ can hold, let us consider an arbitrary splitting of $R$ into a strict part $S'$ and a non-strict part $E'$. Notice that $t^{+}_{x} \succsim^{gmpo} t_x$ and $t^{-}_{x} \succsim^{gmpo} t_x$, but neither $t^{+}_{x} \succ^{gmpo} t_x$ nor $t^{-}_{x} \succ^{gmpo} t_x$: the reason is that for each $l$ and $C_j$ we get $s(l, C_j) \succsim^{gmpo} \mathsf{a}$ but not $s(l, C_j) \succ^{gmpo} \mathsf{a}$. Hence, each $t_x$ of $R$ must be contained in $E'$. As moreover, $t^{\pm}_{x_i} \succ^{gmpo} y_j$ iff $t^{\pm}_{x_i} \succsim^{gmpo} y_j$ we can w.l.o.g. assume that each $y_j \in S'$. In total, if $L \succ^{gmpo}_{gms} R$, then $R$ must be split into $S' \cup E'$ where $S' = \{\{y_1, \ldots, y_m\}\}$ and $E' = \{\{t_{x_1}, \ldots, t_{x_n}\}\}$ and $L$ must be split into $S \cup E$ such that all conditions of $\succ^{gmpo}_{gms}$ are satisfied.

At this point we consider both directions to show that $\phi$ is satisfiable iff $L \succ^{gmpo}_{gms} R$.

First assume that $\phi$ is satisfiable, so let $\alpha$ be some satisfying assignment. Then we choose $S = \{\{t^{+}_{x} \mid \alpha(x) = \top\}\} \cup \{\{t^{-}_{x} \mid \alpha(x) = \bot\}\}$ and $E = L \setminus S$. Notice that for each $x$ exactly one of $t^{+}_{x}$ and $t^{-}_{x}$ is in $S$, and the other is in $E$. Hence, for each $t_x \in E'$ there is a corresponding $t^{\pm}_{x} \in E$ with $t^{\pm}_{x} \succsim^{gmpo} t_x$. Next, we have to find for each $y_j \in S'$ some term in $S$ which is larger than $y_j$. To this end, notice that $\alpha$ is a satisfying assignment, thus there is some literal $x_i$ or $\overline{x_i}$ in $C_j$ which evaluates to true. If $x_i \in C_j$ then $\alpha(x_i) = \top$ and hence, $t^{+}_{x_i} \in S$ where $t^{+}_{x_i} = \mathsf{f}(\ldots, y_j, \ldots) \succ^{gmpo} y_j$. Otherwise, $\overline{x_i} \in C_j$ and $\alpha(x_i) = \bot$ and hence, $t^{-}_{x_i} \in S$ where $t^{-}_{x_i} = \mathsf{f}(\ldots, y_j, \ldots) \succ^{gmpo} y_j$. Thus, in both cases there is some term in $S$ being larger than $y_j$. Moreover, $S \neq \emptyset$ since $|S| = n \geq 2$.

For the other direction assume that $S$ and $E$ could be found such that $L = S \cup E$ and all conditions of $\succ^{gmpo}_{gms}$ are satisfied. Hence for each $t_{x_i} \in E'$ there is some term $t \in E$ satisfying $t \succsim^{gmpo} t_{x_i}$. Then, $t$ can only be $t^{+}_{x_i}$ or $t^{-}_{x_i}$ since $x_i \in \mathcal{V}(t_x)$ and each other term $t^{\pm}_{x_j}$ with $i \neq j$ does not contain the variable $x_i$. Thus, for each $i$ exactly one of the terms $t^{+}_{x_i}$ and $t^{-}_{x_i}$ is contained in $E$ and the other is contained in $S$. We define the assignment $\alpha$ where $\alpha(x_i) = \top$ iff $t^{+}_{x_i} \in S$. It remains to show that $\alpha$ is a satisfying assignment for $\phi$. So let $C_j$ be some clause of $\phi$. Since $y_j \in S'$ we know that there is some $t \in S$ with $t \succ^{gmpo} y_j$, i.e., $y_j \in \mathcal{V}(t)$. There are two cases. First, if $t = t^{+}_{x_i}$ for some $i$, then by the definition of $t^{+}$ we know that $y_j \in \mathcal{V}(t^{+}_{x_i})$ implies $x_i \in C_j$, and hence $C_j$ is evaluated to true, since $\alpha(x_i) = \top$ by definition of $\alpha$. Otherwise, $t = t^{-}_{x_i}$ for some $i$ where now $\overline{x_i} \in C_j$. Moreover, as $t^{-}_{x_i} \in S$, we know that $t^{+}_{x_i} \notin S$ and hence, $\alpha(x_i) = \bot$. Together with $\overline{x_i} \in C_j$ this again shows that $C_j$ is evaluated to true. Hence, all clauses evaluate to true using $\alpha$ which proves that $\phi$ is satisfiable. □

The following corollary states that NP-completeness is essentially due to fact that $\succ_{gms}$ and $\succsim_{gms}$ are hard to compute, even if all comparisons of the elements in the multiset are given. It can be seen within the previous proof, where the important part of the reduction from SAT was to define for each formula $\phi$ the multisets $L$ and $R$ such that $\phi$ is satisfiable iff $L \succ^{gmpo}_{gms} R$ (and also iff $L \succsim^{gmpo}_{gms} R$).

**Corollary 12.12.** *Given two orderings $\succ$ and $\succsim$, two multisets $M$ and $N$, and the set $\{(x, y, x \succ y, x \succsim y) \mid x \in M, y \in N\}$, deciding $M \succ_{gms} N$ and $M \succsim_{gms} N$ is NP-complete.*

Of course, if the splitting for the multiset comparison is given, then deciding $\succ_{gms}$ and $\succsim_{gms}$ becomes polynomial.

All our results have also been generalized to RPO where for every function symbol there is a status function $\tau$ which determines whether the arguments of each function $f$ should be compared lexicographically ($\tau(f) = lex$) or via multisets ($\tau(f) = mul$).[3] Here, the existing inference rules of $\succ^{gmpo}$ are modified that instead of $f \approx g$ it is additionally demanded that $\tau(f) = \tau(g) = mul$. Moreover, there are two additional inference rules for $f \approx g$ and $\tau(f) = \tau(g) = lex$, one for the strict ordering $\succ^{grpo}$ and one for the non-strict ordering $\succsim^{grpo}$.

Again, $\succ^{grpo}$ is used in AProVE instead of the standard definition of RPO ($\succ^{rpo}$,[27]). However, during our formalization we have detected that in contrast to $(\succsim^{gmpo}, \succ^{gmpo})$, the pair $(\succsim^{grpo}, \succ^{grpo})$ is not a reduction pair, as the orderings are not stable. To see this, consider a precedence where $\mathsf{a}$ and $\mathsf{b}$ have least precedence, and $\tau(\mathsf{a}) \neq \tau(\mathsf{b})$. Then $x \succsim^{grpo} \mathsf{a}$, but $\mathsf{b} \not\succsim^{grpo} \mathsf{a}$.

Our solution to this problem is to add the following further inference rules which allow comparisons of terms $f(\vec{s})$ with $g(\vec{t})$ where $\tau(f) \neq \tau(g)$. In detail, we require that $\vec{t}$ is empty and for a strict decrease, additionally $\vec{s}$ must be non-empty.

$$\frac{|\vec{s}| > 0 \quad |\vec{t}| = 0}{f(\vec{s}) \succ^{grpo} g(\vec{t})} f \approx g, \tau(f) \neq \tau(g) \qquad \frac{|\vec{t}| = 0}{f(\vec{s}) \succsim^{grpo} g(\vec{t})} f \approx g, \tau(f) \neq \tau(g)$$

If these inference rules are added, then indeed $(\succsim^{grpo}, \succ^{grpo})$ is a monotone reduction pair. As a consequence, one can interpret AProVE's version of RPO as a sound, but non-stable under-approximation of $\succ^{grpo}$.

We also tried to to relax the preconditions further, e.g. by allowing vectors $\vec{t}$ of length at most one. But no matter whether we also restrict the length of $\vec{s}$ in some way or not, and no matter whether we compared the arguments $\vec{s}$ and $\vec{t}$ lexicographically or via multisets, the outcome was always that transitivity or strong normalization are lost.

For example, if we add the inference rule that $\{\{\vec{s}\}\} \succ^{grpo}_{gms} \{\{\vec{t}\}\}$, $|\vec{t}| \leq 1$, $f \approx g$, and $\tau(f) \neq \tau(g)$ implies $f(s) \succ^{grpo} g(t)$, then strong normalization is lost: assume the precedence is defined by $\mathsf{f} \approx \mathsf{g} \approx \mathsf{h}$ and $3 > 2 > 1 > 0$, and the status is defined by $\tau(\mathsf{f}) = \tau(\mathsf{h}) = lex$ and $\tau(\mathsf{g}) = mul$. Then $\mathsf{f}(0,3) \succ^{grpo} \mathsf{g}(2) \succ^{grpo} \mathsf{h}(1) \succ^{grpo} \mathsf{f}(0,3)$ clearly shows that the resulting ordering is not strongly normalizing anymore.

To summarize, $\succ^{gmpo}$ and $\succ^{grpo}$ are strictly more powerful reduction orderings than the standard definitions of MPO and RPO ($\succ^{mpo}$ and $\succ^{rpo}$). The price for the increased power is that checking constraints for $\succ^{gmpo}$ and $\succ^{grpo}$ becomes NP-complete whereas it is in P for $\succ^{mpo}$ and $\succ^{rpo}$.

Note that if one would provide all required splittings for the multiset comparisons in $\succ^{gmpo}$ and $\succ^{grpo}$, then constraint checking again becomes polynomial. However, this would make certificates more bulky, and since in practice the arities of function symbols are rather small, certification can efficiently be done even without additional splitting information.

[3] We do not consider permutations for lexicographic comparisons in the definition of RPO as this feature can be simulated by generating reduction pairs using an RPO (without permutations) in combination with argument filterings as defined in [3]. In this way, we only have to formalize permutations once and we can reuse them for other orderings like KBO.

## 12.5. SCNP Reduction Pairs

The size-change criterion of [74] to prove termination of programs can be seen as a graph-theoretical problem: given a set of graphs—encoding for each recursive call the decrease in size of each argument—one tries to decide the *size-change termination condition* (SCT condition) on the graphs, namely that in every infinite sequence of graphs one can find an argument whose size is strictly decreased infinitely often. If the condition is satisfied, then termination is proved.

Concerning automation of the size-change principle, there are two problems: first, the base ordering (or size-measure or ranking function) must be provided to construct the graphs, and second, even for a given base ordering, deciding the SCT condition is PSPACE-complete.

To overcome these problems, in [7] and [20] sufficient criteria have been developed. They approximate the SCT condition in a way that can be encoded into SAT.

Another benefit of [20] is an integration of the approximated SCT condition as a reduction pair in the dependency pair framework (DP framework) [42], called *SCNP reduction pair*.

Although IsaFoR contains already a full formalization of size-change termination as it is used in [102], an integration of SCNP reduction pairs in the certification process might be beneficial for two reasons:

- since deciding the SCT condition is PSPACE hard, whereas the approximated condition can be encoded into SAT, the certificates might be easier to check
- the approximated SCT condition is not fully subsumed by the SCT condition as only the former allows incremental termination proofs in the DP framework

Before we describe our formalization of SCNP reduction pairs, we shortly recall some notions of the DP framework, a popular framework to perform modular termination proofs for TRSs. The main data structure are *DP problems* $(\mathcal{P}, \mathcal{R})$ consisting of two TRSs where all rules in $\mathcal{P}$ are of the form $F(\dots) \to G(\dots)$ where $F, G$ are symbols that do not occur in $\mathcal{R}$. The main task is to prove finiteness of the a given DP problem $(\mathcal{P}, \mathcal{R})$, i.e., absence of *infinite minimal chains* $s_1\sigma \to t_1\sigma \to^*_{\mathcal{R}} s_2\sigma \to t_2\sigma \dots$ where all $s_i \to t_i \in \mathcal{P}$ and all $t_i\sigma$ are terminating w.r.t. $\mathcal{R}$. To this end, one uses various *processors* to simplify the initial DP problem for a TRS until the $\mathcal{P}$-component is empty.

One of the most important processors is the reduction pair processor [42, 51]—where here we only present its basic version without other refinements like usable rules w.r.t. an argument filtering [42]. It can remove strictly decreasing rules from $\mathcal{P}$, provided that both $\mathcal{P}$ and $\mathcal{R}$ are at least weakly decreasing.

**Theorem 12.13** (Reduction pair processor)**.** *Let* $(\succsim, \succ)$ *be a reduction pair. If* $\mathcal{P} \subseteq \succ \cup \succsim$ *and* $\mathcal{R} \subseteq \succsim$ *then* $(\mathcal{P}, \mathcal{R})$ *is finite if* $(\mathcal{P} \setminus \succ, \mathcal{R})$ *is finite.*

Hence, to prove termination it suffices to find different reduction pairs to iteratively remove rules from $\mathcal{P}$ until all rules of $\mathcal{P}$ have been removed. Thus, with SCNP reduction pairs it is possible to choose different base orderings to remove different rules of $\mathcal{P}$.

In the following we report on details of SCNP reduction pairs as defined in [20] and on their formalization. Essentially, SCNP reduction pairs are generated from reduction pairs $(\succsim, \succ)$ via a multiset extension of a lexicographic combination of $\succ$ with the standard ordering on the naturals.

**Definition 12.14** (Multiset extension)**.** *We define that $\mu$ is a* multiset extension *iff for each ordering pair $(\succsim, \succ)$ over $A$, the pair $(\succsim_\mu, \succ_\mu)$ is an ordering pair over $\mathfrak{P}(A)$. Moreover, whenever $\succ$ and $\succsim$ are closed under an operator op ($x \,{}_{(}\!\succsim_{)}\, y$ implies $op(x) \,{}_{(}\!\succsim_{)}\, op(y)$ for all $x, y$), then $\succ_\mu$ and $\succsim_\mu$ must also be closed under the image of op on multisets ($M \,{}_{(}\!\succsim_{)_\mu}\, N$ implies $op[M] \,{}_{(}\!\succsim_{)_\mu}\, op[N]$).*

*We also call $(\succsim_\mu, \succ_\mu)$ the* multiset extension *of $(\succsim, \succ)$ w.r.t. $\mu$.*

For example, *gms* is a multiset extension and in [7] and [20] in total four extensions are listed to compare multisets (*gms*, *min*, *max*, and *dms*). The extension *dms* is the dual multiset extension [8] where our formulation is equivalent to the definition in [20].[4] The strict relation is defined as $M \succ_{dms} N$ iff $M = \{\{x_1, \ldots, x_m\}\} \cup \{\{z_1, \ldots, z_k\}\}$, $N = \{\{y_1, \ldots, y_n\}\} \cup \{\{z'_1, \ldots, z'_k\}\}$, $n > 0$, $\forall i. z_i \succsim z'_i$, and $\forall x_i. \exists y_j. x_i \succ y_j$; the non-strict relation $\succsim_{dms}$ is defined like $\succ_{dms}$ except that the condition $n > 0$ is omitted.

For SCNP reduction pairs, multisets are compared via one of the four multiset extensions. And to generate multisets from terms, the notion of a level mapping is used.

**Definition 12.15** (Level mapping [20])**.** *For each $f \in \mathcal{F}$ with arity $n$, let $\pi(f) \in \mathfrak{P}(\{1, \ldots, n\} \times \mathbb{N})$.[5] We define the* level-mapping *$\mathcal{L} : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathfrak{P}(\mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathbb{N})$ where $\mathcal{L}(f(s_1, \ldots, s_n)) = \{\{\langle s_i, k\rangle \mid (i, k) \in \pi(f)\}\}$.[6]*

**Definition 12.16.** *Let $(\succsim, \succ)$ be a reduction pair for terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Let $\mu$ be a multiset extension. The ordering pair $(\succsim^{\mathbb{N}}, \succ^{\mathbb{N}})$ over $\mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathbb{N}$ is defined as the lexicographic combination of $(\succsim, \succ)$ with the $>$-ordering on the naturals: $\langle s, n\rangle \succsim^{\mathbb{N}} \langle t, m\rangle$ iff $s \succ t \vee (s \succsim t \wedge n \geq m)$, and $\langle s, n\rangle \succ^{\mathbb{N}} \langle t, m\rangle$ iff $s \succ t \vee (s \succsim t \wedge n > m)$. The ordering pair $(\succsim^{\mathbb{N}}_\mu, \succ^{\mathbb{N}}_\mu)$ is defined as the multiset extension of $(\succsim^{\mathbb{N}}, \succ^{\mathbb{N}})$ w.r.t. $\mu$.*

In principle, $\succ^{\mathbb{N}}_\mu \cup \succsim^{\mathbb{N}}_\mu$ is the part of a SCNP reduction pair that should be used to compare left- and right-hand sides of $\mathcal{P}$ within the reduction pair processor. However, one also needs to orient the rules in $\mathcal{R}$ via $\succsim$. To this end, two types are introduced in [20] so that the final ordering incorporates both $\succ^{\mathbb{N}}_\mu \cup \succsim^{\mathbb{N}}_\mu$ for $\mathcal{P}$ and $\succsim$ for $\mathcal{R}$. In detail, the signature $\mathcal{F}$ is partitioned into $\mathcal{F}^b \uplus \mathcal{F}^\sharp$ consisting of base symbols $\mathcal{F}^b$ and tuple-symbols $\mathcal{F}^\sharp$. The set of *base terms* is $\mathcal{T}(\mathcal{F}^b, \mathcal{V})$, and a *tuple term* is a term of the form $F(t_1, \ldots, t_n)$ where $F \in \mathcal{F}^\sharp$ and each $t_i$ is a base term.

Notice that for a DP problem $(\mathcal{P}, \mathcal{R})$ all terms in $\mathcal{P}$ are tuple terms and all terms in $\mathcal{R}$ are base terms if one chooses $\mathcal{F}^\sharp$ to be the set of root symbols of terms in $\mathcal{P}$. Therefore, SCNP reduction pairs are defined in a way that the ordering depends on whether tuple terms or base terms are compared.

**Definition 12.17** (SCNP reduction pair [20])**.** *Let $\mu$ be a multiset extension and $\mathcal{L}$ be a level mapping. Let $(\succsim, \succ)$ be a reduction pair over $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The* SCNP reduction pair *is the pair $(\succsim^{\mathcal{L},\mu}, \succ^{\mathcal{L},\mu})$ where the relations $\succsim^{\mathcal{L},\mu}$ and $\succ^{\mathcal{L},\mu}$ over $\mathcal{T}(\mathcal{F}, \mathcal{V})$ are defined as follows. If $t$ and $s$ are tuple terms, then $t \succ^{\mathcal{L},\mu} s$ iff $\mathcal{L}(t) \succ^{\mathbb{N}}_\mu \mathcal{L}(s)$, and $t \succsim^{\mathcal{L},\mu} s$ iff $\mathcal{L}(t) \succsim^{\mathbb{N}}_\mu \mathcal{L}(s)$. Otherwise, if $t$ and $s$ are base terms, then $t \succsim^{\mathcal{L},\mu} s$ iff $t \succsim s$, and $t \succ^{\mathcal{L},\mu} s$ iff $t \succ s$.*

[4] There is a difference in the definition of the dual multiset extension in [7, 8] to the definition in [20] which is similar to the difference between $\succ_{ms}$ and $\succ_{gms}$.

[5] In [20] there was an additional condition that $\pi(f)$ may not contain two entries $\langle j, k_1\rangle$ and $\langle j, k_2\rangle$. It turned out that this condition is not required for soundness.

[6] We use the notation $\mathcal{L}$ instead of $\ell$ for level mappings as in this paper, $\ell$ are left-hand sides of rules.

Before stating the major theorem of [20] that SCNP reduction pairs are reduction pairs, we first have to clarify the notion of reduction pair: In [20, Sec. 2] a non standard definition of a reduction pair is used which differs from Def. 12.1. To distinguish between both kinds of reduction pairs we call the ones of [20] *typed reduction pairs.*

**Definition 12.18** (Typed reduction pair [20])**.** *A* typed reduction pair *is a reduction pair* $(\succsim, \succ)$ *over* $\mathcal{T}(\mathcal{F}, \mathcal{V})$ *with the additional condition that* $\succsim$ *compares either two tuple terms or two base terms.*

So, the major theorem of [20] states that whenever $(\succsim, \succ)$ is a typed reduction pair, then so is $(\succsim^{\mathcal{L},\mu}, \succ^{\mathcal{L},\mu})$ where $\mu$ is one of the four mentioned multiset extensions.

The major problem in the formalization of exactly this theorem required a link between the two notions of reduction pairs since all other theorems working with reduction pairs in IsaFoR are using Def. 12.1.

At this point, it turned out that there are problems with Def. 12.18. Of course, to use the major theorem of [20] one needs a typed reduction pair to start from. Unfortunately, common reduction pairs like RPO are not typed reduction pairs w.r.t. Def. 12.18. For example, if $F \in \mathcal{F}^\sharp$ then $F(F(x)) \succsim^{grpo} F(F(x))$, but then $\succsim$ does not satisfy the additional condition of Def. 12.18, since here two terms are in relation which are neither base terms nor tuple terms.

A possible solution might be to require that $(\succsim, \succ)$ is just a reduction pair and then show that $(\succsim^{\mathcal{L},\mu}, \succ^{\mathcal{L},\mu})$ is a typed reduction pair. However, even with this adaptation the problems remain, since Def. 12.18 itself is flawed: assume $(\succsim, \succ)$ is a typed reduction pair and $F$ is a non-constant tuple symbol. Since $\succsim$ is a quasi-ordering (on tuple-terms) it is reflexive, and thus $F(\vec{t}) \succsim F(\vec{t})$ for every list $\vec{t}$ of base terms. By monotonicity of $\succsim$ also $F(\dots, F(\vec{t}), \dots) \succsim F(\dots, F(\vec{t}), \dots)$ must hold, in contradiction to the condition that $\succsim$ only compares base terms or tuple terms. So, there is a severe problem in demanding both monotonicity of $\succsim$ and the additional condition of Def. 12.18.

Repairing Def. 12.18 by only requiring monotonicity w.r.t. $\mathcal{F}^b$ is also no solution, since for using reduction pairs in the reduction pair processor, it is essential that $\succsim$ is also closed under $\mathcal{F}^\sharp$-contexts.

For a proper fix of SCNP reduction pairs, note that the distinction between tuple terms and base terms in [20] is solely performed, to have two kinds of orderings: $\succsim$ for orienting rules in $\mathcal{R}$, and $\succsim^{\mathbb{N}}_{\mu}$ and $\succ^{\mathbb{N}}_{\mu}$ for orienting rules of $\mathcal{P}$.

However, there is already a notion which allows the usage of different orderings for orientation of $\mathcal{P}$ and $\mathcal{R}$, namely *reduction triples.* The advantage of this notion is the fact that it does not require any distinction between base and tuple terms.

**Definition 12.19** (Reduction triple, [51])**.** *A* reduction triple *is a triple* $(\succsim, \succsim_\top, \succ_\top)$ *such that* $(\succsim, \succ_\top)$ *is a reduction pair and* $(\succsim_\top, \succ_\top)$ *is a non-monotone reduction pair.*

Reduction triples can be used instead of reduction pairs in Thm. 12.13: in [51] it is shown that whenever $\mathcal{P} \subseteq \succ_\top \cup \succsim_\top$ and $\mathcal{R} \subseteq \succsim$ then $\mathcal{P}$ can be replaced by $\mathcal{P} \setminus \succ_\top$ in the DP problem $(\mathcal{P}, \mathcal{R})$. The proof is similar to the one of Thm. 12.13 and also in IsaFoR it was easy to switch from reduction pairs to reduction triples. Hence, the obvious attempt is to define SCNP reduction pairs as reduction triples. As there is no distinction of base terms and tuple terms, we will not run into problems that are caused by the required monotonicity of $\succsim$.

**Definition 12.20** (SCNP reduction triple)**.** *Let* $\mu$ *be a multiset extension and* $\mathcal{L}$ *be a level mapping. Let* $(\succsim, \succ)$ *be a reduction pair.*

*We define* $\succsim_\top$ *and* $\succ_\top$ *as* $t \succsim_\top s$ *iff* $\mathcal{L}(t) \succsim_\mu^{\mathbb{N}} \mathcal{L}(s)$*, and* $t \succ_\top s$ *iff* $\mathcal{L}(t) \succ_\mu^{\mathbb{N}} \mathcal{L}(s)$*. Then the* SCNP reduction triple *is defined as* $(\succsim, \succsim_\top, \succ_\top)$*.*

If we are able to prove that every SCNP reduction triple is indeed a reduction triple, then we are done. Unfortunately, it turns out that an SCNP reduction triple is not a reduction triple, where the new problem is that compatibility between $\succ_\top$ and $\succsim$ cannot be ensured. As an example, consider a reduction pair $(\succsim, \succ)$ which is defined via an RPO with precedence $b > a$ and status $\tau(F) = lex$. Moreover, let the level-mapping be defined via $\pi(F) = \{\{\langle 2, 0\rangle\}\}$ and take $\mu = gms$. Then $F(a, b) \succ_\top F(b, a)$ since $\mathcal{L}(F(a, b)) = \{\{\langle b, 0\rangle\}\} \succ_{gms}^{\mathbb{N}} \{\{\langle a, 0\rangle\}\} = \mathcal{L}(F(b, a))$. Furthermore, $F(b, a) \succsim F(a, b)$. However, if $\succ_\top$ and $\succsim$ were compatible, then we would be able to conclude $F(a, b) \succ_\top F(a, b)$, a contradiction.

To this end, we finally have defined a weaker notion than reduction triples which can still be used like reduction triples.

**Definition 12.21** (Root reduction triple)**.** *A* root reduction triple *is a triple* $(\succsim, \succsim_\top, \succ_\top)$ *such that* $(\succsim_\top, \succ_\top)$ *is a non-monotone reduction pair,* $\succsim$ *is a stable and monotone quasi-ordering, and whenever* $s \succsim t$ *then* $f(v_1, \ldots, v_{i-1}, s, v_{i+1}, \ldots, v_n) \succsim_\top f(v_1, \ldots, v_{i-1}, t, v_{i+1}, \ldots, v_n)$*.*

Note that every reduction triple $(\succsim, \succsim_\top, \succ_\top)$ is also a root reduction triple provided that $\succsim \subseteq \succsim_\top$—and as far as we know this condition is satisfied for all reduction triples that are currently used in termination tools. Moreover, root reduction triples can be used in the same way as reduction triples to remove pairs which has been proved in IsaFoR.

**Theorem 12.22** (Root reduction triple processor)**.** *Let* $(\succsim, \succsim_\top, \succ_\top)$ *be a root reduction triple. Whenever* $\mathcal{P} \subseteq \succ_\top \cup \succsim_\top$*,* $\mathcal{R} \subseteq \succsim$*, and* $(\mathcal{P} \setminus \succ_\top, \mathcal{R})$ *is finite, then* $(\mathcal{P}, \mathcal{R})$ *is finite.*

Hence, the notion of root reduction triple seems useful for termination proving. And indeed, it turns out that each SCNP reduction triple is a root reduction triple which finally shows that SCNP reduction triples can be used to remove pairs from DP problems.[7]

**Theorem 12.23.** *Every SCNP reduction triple is a root reduction triple.*

Thm. 12.23 is formally proved within IsaFoR, theory *SCNP*. Note that this formalization was straightforward, once the notion of root reduction triple was available: the whole formalization takes only 310 lines. It also includes the feature of $\epsilon$-arguments—an extension of size-change graphs which is mentioned in both [102] and [20]—and it contains results on $C_e$-compatibility and compatibility w.r.t. to argument filterings. The latter results are important when dealing with usable rules, cf. [42] for further details.

Regarding the formalization of multiset extensions—which is orthogonal to the formalization of SCNP reduction triple—we were able to integrate three of the four mentioned multiset extensions. However, for the multiset extension *dms* it is essential that the signature $\mathcal{F}$ is finite as otherwise strong normalization is not necessarily preserved, cf. Ex. 12.24. Here, the essential issue is that without an explicit bound on the sizes of the multiset, strong normalization is lost (such a bound is explicitly demanded in [7], but not in [20]).

[7] An alternative—but unpublished—fix to properly define SCNP reduction pairs has been developed by Carsten Fuhs. It is based on typed term rewriting. (private communication)

**Example 12.24.** In [20] it is assumed that the *initial* TRS is finite. Hence, also the initial signature is finite which in turn gives a bound on the sizes of the constructed multisets. However, for the soundness of SCNP reduction pairs it is essential that the signature of the *current* system is finite, since otherwise the following steps would be a valid termination proof for the TRS $\mathcal{R} = \{\mathsf{a} \to \mathsf{a}\}$ as there is no bound on the sizes of multisets.

- Build the initial DP problem $(\{\mathsf{A} \to \mathsf{A}\}, \mathcal{R})$.
- Replace this DP problem by the DP problem $\mathcal{D} = (\{\mathsf{A}_i(x, \ldots, x) \to \mathsf{A}_{i+1}(x, \ldots, x) \mid i \in \mathbb{N}\}, \emptyset)$ where the arity of each $\mathsf{A}_i$ is $i$. This step is sound, as $\mathcal{D}$ is not finite.
- Replace $\mathcal{D}$ by $(\emptyset, \emptyset)$. This can be done using the SCNP reduction pair where $\mu = dms$, $\pi(\mathsf{A}_i) = \{\langle j, 0\rangle \mid 1 \leq j \leq i\}$, and $(\succsim, \succ)$ is any reduction pair. The reason is that
$$\{\{\underbrace{\langle x, 0\rangle, \ldots, \langle x, 0\rangle}_{i \text{ times}}\}\} \succ^{dms} \{\{\underbrace{\langle x, 0\rangle, \ldots, \langle x, 0\rangle}_{i+1 \text{ times}}\}\}.$$

The demand for a bound on the signature for *dms* resulted in a small problem in our formalization, since in IsaFoR we never have finite signatures: for each symbol $f$ we directly include infinitely many $f$'s, one for each possible arity in $\mathbb{N}$. So, in principle there might not be any bound on the size of the multisets that are constructed by the level mapping. Hence, we use a slightly different version of *dms* which includes some fixed bound $n$ on the size of the multisets: we define $M \ {}_{(\succsim)\,dms\text{-}n} \ N$ iff $M \ {}_{(\succsim)\,dms} \ N$ and $|N| \leq n \vee |N| = |M|$. Hence, *dms-n* is a restriction of *dms* where either the sizes of the multisets are bounded by $n$ or where multisets of the same sizes are compared.

Note that the additional explicit restriction of $|N| \leq n$ ensures strong normalization even for infinite $\mathcal{F}$ or unbounded multisets. The other alternative $|N| = |M|$ is added, as otherwise reflexivity of $\succsim_{dms\text{-}n}$ is lost, for example $\{\{\underbrace{x, \ldots, x}_{n+1}\}\} \succsim_{dms\text{-}n} \{\{\underbrace{x, \ldots, x}_{n+1}\}\}$ would no longer hold.

In practice, the difference between *dms* and *dms-n* can be neglected. As for the certification we only consider finite TRSs, we just precompute a suitable large enough number $n$ such that for the resulting constraints *dms* and *dms-n* coincide.

We conclude this section by showing that certification of SCNP reduction triples is NP-hard in the case of $\mu \in \{gms, dms\}$ which also implies that deciding $\succ_{dms}$ is NP-hard.

**Theorem 12.25.** *Given a SCNP reduction triple $(\succsim, \succsim_\top, \succ_\top)$ and $\mu \in \{gms, dms\}$, the problem of deciding $\ell \succ_\top r$ for two terms $\ell$ and $r$ is NP-hard.*

*Proof.* For $\mu = gms$ we use nearly the identical reduction from SAT as we used in the proof of Thm. 12.11. To be more precise, for each formula $\phi$ we use the exactly the same terms $\ell$ and $r$ and the same multisets $L$ and $R$ as before. Moreover, we define the level-mapping by choosing $\pi(\mathsf{g}) = \{\langle i, 0\rangle \mid 1 \leq i \leq 2n\}$ and $\pi(\mathsf{h}) = \{\langle i, 0\rangle \mid 1 \leq i \leq n + m\}$. Then for $L' = \{\{\langle s, 0\rangle \mid s \in L\}\}$ and $R' = \{\{\langle s, 0\rangle \mid s \in R\}\}$ we obtain $\ell \succ_\top r$ iff $L' \succ^{\mathbb{N}}_{gms} R'$ iff $L \succ_{gms} R$. It remains to choose $\succ$ as the MPO within the proof of Thm. 12.11. Then $L \succ_{gms} R$ iff $\phi$ is satisfiable, so in total, $\ell \succ_\top r$ iff $\phi$ is satisfiable.

Furthermore, it can easily be argued that NP-hardness is not a result of our extended definition of MPO: we also achieve $L \succ_{gms} R$ iff $\phi$ is satisfiable if we define $\succ$ by a polynomial interpretation $\mathcal{P}ol$ with $\mathcal{P}ol(\mathsf{f}(x_0, \ldots, x_m)) = 1 + x_0 + \cdots + x_m$ and $\mathcal{P}ol(\mathsf{a}) = 0$.

For $\mu = dms$ one can use a similar reduction from SAT: satisfiability of $\phi$ is equivalent to $\ell \succ_\top r$ using the same level-mapping as before, but where now $\ell = \mathsf{h}(x_1 \vee \overline{x_1}, \ldots, x_n \vee$

$\overline{x_n}, \mathsf{s}(C'_1), \ldots, \mathsf{s}(C'_m))$, $r = \mathsf{g}(x_1, \overline{x_1}, \ldots, x_n, \overline{x_n})$, and $\succ$ is defined as the polynomial interpretation $\mathcal{P}ol$ where $\mathcal{P}ol(\mathsf{s}(x)) = 1 + x$ and $\mathcal{P}ol(x \vee y) = x + y$. Here, each $C'_i$ is a representation of clause $C_i$ as disjunction. □

## 12.6. Certification Algorithms

For the certification of existing proofs using RPO and SCNP reduction pairs there are in principle two possibilities, which we will explain using RPO.

The first solution for certifying $s \succ^{grpo} t$ is to use a shallow embedding. In this approach, some untrusted tool figures out how the inference rules of RPO have to be applied and generates a proof script that can then be checked by the proof assistant. This solution cannot be used in our case, since CeTA is obtained from IsaFoR via code generation.

The second solution is to use a deep embedding where an algorithm for deciding RPO is developed within the proof assistant in combination with a soundness proof. Then this algorithm is amenable for code generation, and such an algorithm is also used in related certifiers [13, 21] where it is accessible via reflection. Hence, we had to develop a function for deciding RPO constraints within Isabelle. In our case, we have written a function $grpo^\tau_>$ for RPO, which is parametrized by a precedence $>$ and a status $\tau$. It takes two terms $s$ and $t$ as input and returns the pair $(s \succ^{grpo} t, s \succsim^{grpo} t)$. In fact, we even defined RPO via $grpo^\tau_>$ and only later on derived the inference rules that have been presented in Sec. 12.4.

Since we proved several properties of RPO directly via $grpo^\tau_>$ (theory *RPO*), we implemented $grpo^\tau_>$ in a straightforward way as recursive function. As a result, $grpo^\tau_>$ has exponential runtime, since no sharing of identical subcalls is performed. To this end, we developed a second function for RPO, that has been proved to be equivalent to $grpo^\tau_>$, but where memoization is integrated—a well known technique where intermediate results are stored to avoid duplicate computations. In principle, this is an easy programming task, but since we use a deep embedding, we had to formally prove correctness of this optimization.

To stay as general as possible, the memoized function was implemented independent of the actual data structure used as memory. The interface we use for a memory is as follows. We call a memory valid w.r.t. to a function $f$ if all entries in the memory are results of $f$. Moreover we require functionality for looking up a result in the memory and for storing a new result with the obvious soundness properties: storing a correct entry in a valid memory yields a valid memory and looking up an entry in a valid memory returns a correct result, i.e., the same result that would have been computed by $f$.

Assuming we have such a memory at our disposal the idea of memoizing is straightforward: before computing a result we do a lookup in the memory and if an entry is found we return it and leave the memory unchanged. Otherwise we compute as usual, store the result in the memory and return both.

The main difficulty was that all recursive calls of $grpo^\tau_>$ are indirect via higher-order functions like the computation of the multiset- and lexicographic extension of a relation. Consequently results of recursive calls to RPO are not available directly, but only in these higher-order functions. Thus, they have to take care of storing results in the memory and of passing it on to the next RPO call. Hence, new memoizing versions of all these higher-order functions had to be implemented and their soundness had to be proved. Soundness meaning that, when given equivalent functions as arguments, they compute the same results as their counterparts without memory handling, and that given a valid memory as

input they return a valid memory in addition to their result. For further details we refer to theory *Efficient-RPO*.

Concerning the required decision procedures for *gms* and *dms* which have to find suitable splittings, we used a branch-and-bound approach.

We tested our certification algorithms by rerunning all experiments that have been performed in [20]. Here, AProVE tries to prove termination of 1,381 TRSs from the termination problem database (TPDB version 7.0) using 20 different strategies where in 12 cases SCNP reduction pairs are used and full size-change termination is tried 4 times. Here, in 16 strategies RPO or weaker orderings have been tried. As in [20] we used a timeout of 60 seconds and although we used a different computer than in [20], on the $20 \times 1,381$ termination problems, there are only 8 differences in both experiments—all due to a timeout.

By performing the experiments we were able to detect a bug in the proof output of AProVE—the usable rules have not been computed correctly. After a corresponding fix indeed all 9,025 generated proofs could be certified by CeTA (version 2.2).

The experiments contain 432 proofs where the usage of $\succ_{gms}$ and $\succsim_{gms}$ was essential, i.e., where the constraints could not be oriented by $\succ_{ms}$ and $\succsim_{ms}$. If one allows equality modulo the equivalence relation induced by the ordering, then still 39 proofs require $\succ_{gms}$ and $\succsim_{gms}$.

Regarding the time required for certification, although it is NP-complete, in our experiments, certification is much faster than proof search. The reason is that our certification algorithms are only exponential in the arity of the function symbols, and in the experiments the maximal arity was 7. In numbers: AProVE required more than 31 hours ($\approx$ 4 seconds per example) whereas CeTA was done in below 6 minutes ($\approx$ 0.04 seconds per example).

The experiments also show that proofs using SCNP reduction pairs can indeed be certified faster than proofs using full size-change termination. In average, the latter proofs require 50 % more time for certification than the former.

All details of our experiments are available at `http://cl-informatik.uibk.ac.at/software/ceta/experiments/multisets/`.

## 12.7. Summary

We have studied the generalization of the multiset ordering which is generated by two orderings: a strict one and a compatible non-strict one. Indeed this generalization preserves most properties of the standard multiset ordering, where we only detected one difference: the decision problem becomes NP-complete.

Concerning termination techniques that depend on multiset orderings, we formalized and corrected an extended variant of RPO that is used within AProVE, and we formalized and corrected SCNP reduction pairs. Certification of both techniques is NP-hard.

### Acknowledgments

We thank Carsten Fuhs for helpful discussions and his support in the experiments, and we thank the anonymous referees for their helpful comments.

# 13. Certification of Nontermination Proofs

### Publication details

Christian Sternagel and René Thiemann. Certification of Nontermination Proofs. In *Proceedings of the 3rd International Conference on Interactive Theorem Proving*, volume 7406 of LNCS, pages 266–282. Springer, 2012.

### Abstract

Automatic tools for proving (non)termination of term rewrite systems, if successful, deliver proofs as justification. In this work, we focus on how to certify nontermination proofs. Besides some techniques that allow to reduce the number of rules, the main way of showing nontermination is to find a loop, a finite derivation of a special shape that implies nontermination. For standard termination, certifying loops is easy. However, it is not at all trivial to certify whether a given loop also implies innermost nontermination. To this end, a complex decision procedure has been developed in [103]. We formalized this decision procedure in Isabelle/HOL and were able to simplify some parts considerably. Furthermore, from our formalized proofs it is easy to obtain a low complexity bound. Along the way of presenting our formalization, we report on generally applicable ideas that allow to reduce the formalization effort and improve the efficiency of our certifier.

## 13.1. Introduction

In program verification the focus is on proving that a function satisfies some property, e.g., termination. However, in presence of a *bug* it is more important to find a counterexample indicating the problem. In this way, we can save a lot of time by abandoning a verification attempt as soon as a counterexample is found. In term rewriting, a well known counterexample for termination is a loop, essentially giving some "input" on which a "program" does not terminate. As soon as specific evaluation strategies are considered it might not be easy to verify whether a given loop constitutes a proper counterexample. However, since many programming languages employ an eager evaluation strategy, methods for proving innermost nontermination are important. What is more, some very natural functions are not even expressible without evaluation strategy. Take for example equality on terms. There is no (finite) term rewrite system (TRS) that encodes equality on arbitrary terms (the problem is the case where the two given terms are different). Using innermost rewriting, encoding equality is possible by the following four rules, as shown by Daron Vroon (personal communication; he used this encoding to properly model the built-in equality of ACL2).

$$x == y \to \mathsf{chk}(\mathsf{eq}(x,y)) \quad (1) \qquad \mathsf{chk}(\mathsf{true}) \to \mathsf{true} \quad (3)$$
$$\mathsf{eq}(x,x) \to \mathsf{true} \quad (2) \qquad \mathsf{chk}(\mathsf{eq}(x,y)) \to \mathsf{false} \quad (4)$$

Current techniques for proving innermost nontermination of TRSs consist of preprocessing techniques (narrowing the search space by removing rules) followed by finding a loop, for which the complex decision procedure of [103] allows to decide whether it implies innermost nontermination. We formalized this decision procedure as part of our ***Isa**belle **F**ormalization **o**f **R**ewriting* (IsaFoR). The corresponding certifier CeTA can be obtained by Isabelle/HOL's code generator [49, 104]. Both IsaFoR and CeTA are freely available at `http://cl-informatik.uibk.ac.at/software/ceta/` (the relevant theories for this paper are `Innermost_Loops` and `Nontermination`, together with their respective implementation theories, indicated by the suffix `_Impl`).

During our formalization we were able to simplify some parts of the decision procedure considerably. Mostly, due to a new proof which, in contrast to the original proof, does not depend on Kruskal's tree theorem. As a result, we can replace the most complicated algorithm of [103] by a single line. Moreover, we report on how we managed to obtain efficient versions of other algorithms from [103] within Isabelle/HOL [84].

The remainder is structured as follows. In Sect. 13.2 we give preliminaries. Then, in Sect. 13.3, we describe the preprocessing techniques (narrowing the search space for finding a loop) that are supported by our certifier. Afterwards, we present details on loops w.r.t. the innermost strategy in Sect. 13.4. The main part of this paper is on our formalization of the decision procedure for innermost loops in Sect. 13.5, before we conclude in Sect. 13.6.

## 13.2. Preliminaries

We assume basic familiarity with term rewriting [5]. Nevertheless, we shortly recapitulate what is used later on. A *term* $t$ ($\ell$, $r$, $s$, $u$, $v$) is either a *variable* $x$ ($y$, $z$) from the set $\mathcal{V}$, or a *function symbol* $f$ ($g$) from the disjoint set $\mathcal{F}$ applied to some argument terms $f(t_1, \ldots, t_n)$. The *root* of a term is defined by $root(x) = x$ and $root(f(t_1, \ldots, t_n)) = f$. The set $args(t)$ of *arguments* of $t$ is defined by the equations $args(x) = \emptyset$ and $args(f(t_1, \ldots, t_n)) = \{t_1, \ldots, t_n\}$. The set of *variables occurring in a term* $t$ is denoted by $\mathcal{V}(t)$. A *context* $C$ ($D$) is a term containing exactly one occurrence of the special *hole* symbol $\Box$. Replacing the hole in a context $C$ by a term $t$ is written $C[t]$. The term $t$ is a *(proper) subterm* of the term $s$, written ($s \rhd t$) $s \trianglerighteq t$, iff there is a (non-hole) context $C$ such that $s = C[t]$, iff there is a (non-empty) position $p$ such that $s|_p = t$. We write $s \trianglerighteq_{\mathcal{F}} t$ iff $s \trianglerighteq t$ and $t \notin \mathcal{V}$. A *substitution* $\sigma$ ($\mu$) is a mapping from variables to terms whose *domain* $dom(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. The *range* of a substitution is $ran(\sigma) = \{\sigma(x) \mid x \in dom(\sigma)\}$. We represent concrete substitutions using the notation $\{x_1/t_1, \ldots, x_n/t_n\}$. We use $\sigma$ interchangeably with its homomorphic extension to terms, writing, e.g., $t\sigma$ to denote the application of the substitution $\sigma$ to the term $t$. A *(rewrite) rule* is a pair of terms $\ell \to r$ and a term rewrite system (TRS) $\mathcal{R}$ is a set of such rules. The *rewrite relation (induced by $\mathcal{R}$)* $\to_{\mathcal{R}}$ is defined by $s \to_{\mathcal{R}} t$ iff there is a context $C$, a rewrite rule $\ell \to r \in \mathcal{R}$, and a substitution $\sigma$ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. Here, we call $\ell\sigma$ a *redex* (short for *reducible expression*) and sometimes write $s \to_{\mathcal{R},\ell\sigma} t$ to make it explicit. A *normal form* is a term that does not contain any redexes. When a rewrite step $s \to_{\mathcal{R},\ell\sigma} t$ additionally satisfies that all arguments of $\ell\sigma$ are normal forms, it is called an *innermost (rewrite)*

*step*, written $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$. We freely drop $\mathcal{R}$ from $s \to_{\mathcal{R}} t$ if it is clear from the context.

A term $t$ is (innermost) nonterminating w.r.t. $\mathcal{R}$, iff there is an infinite (innermost) rewrite sequence starting at $t$, i.e., a derivation of the form

$$t = t_1 \xrightarrow{(\mathsf{i})}_{\mathcal{R}} t_2 \xrightarrow{(\mathsf{i})}_{\mathcal{R}} t_3 \xrightarrow{(\mathsf{i})}_{\mathcal{R}} \cdots$$

A TRS $\mathcal{R}$ is (innermost) nonterminating iff there is a term $t$ that is (innermost) nonterminating w.r.t. $\mathcal{R}$.

## 13.3. A Framework for Certifying Nontermination

As for termination, there are several techniques that may be combined in order to prove nontermination. On the one hand, there are basic techniques, i.e., those that immediately prove nontermination; and on the other hand, there are transformations, i.e., mappings that turn a given TRS $\mathcal{R}$ into a transformed TRS $\mathcal{R}'$ (for which, proving nontermination is hopefully easier). Such transformations are *complete* iff (innermost) nontermination of $\mathcal{R}'$ implies (innermost) nontermination of $\mathcal{R}$. In order to prove nontermination, arbitrary complete transformations can be applied, before finishing the proof by a basic technique.

In our development we formalized the following basic techniques and complete transformations. Except for innermost loops and string reversal, none of these techniques posed any difficulties in the formalization.

**Well-Formedness Check.** A TRS $\mathcal{R}$ is *(weakly) well-formed* iff no left-hand side is a variable and all (applicable) rules $\ell \to r$ satisfy $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$. Where a rule is *applicable* iff the arguments of its left-hand side are normal forms (otherwise the rule could never be used in the innermost case).

**Lemma 13.1.** *If $\mathcal{R}$ is not (weakly) well-formed, it is (innermost) nonterminating.*

Thus a basic technique is to check whether a TRS is (weakly) well-formed and conclude (innermost) nontermination, if it is not.

**Finding Loops.** The second basic technique is to find a loop and it is treated in more detail in Sect. 13.4.

**Rule Removal.** One way to narrow the search space when trying to prove nontermination, is to get rid of rules that cannot contribute to any infinite derivation. This can be done by employing the same techniques that are already known from termination, namely monotone reduction pairs [9, 73].

**String Reversal.** A special variant of TRSs are *string rewrite systems*, where all function symbols are fixed to be unary. For this special case, *string reversal* (see, e.g., [111] and [97] for its formalization) can be applied.

**Dependency Pair Transformation.** As for termination, also for nontermination, it is possible to switch from TRSs to *dependency pair problems* (DPPs) [3]. This is done by the so called *dependency pair transformation*, which intuitively, identifies the mutually

recursive dependencies of rewrite rules and makes them explicit in a second set of rewrite rules, the *dependency pairs*.

For nontermination of (innermost) DPPs, we support the following techniques:

**Finding Loops.** For DPPs $(\mathcal{P}, \mathcal{R})$ the search space for finding loops is further restricted by the fact that pairs from $\mathcal{P}$ are only applied at the root position.

**Rule Removal.** Also for DPPs it is possible to narrow the search space by employing reduction pairs to remove pairs and rules that do not contribute to any infinite derivation. Note that for nontermination analysis, also the dependency graph processor and the usable rules processor do just remove pairs and rules.

**Note.** Since $\mathcal{R}$ is (innermost) nonterminating (by the well-formedness check) whenever $\mathcal{R}$ contains a rule $x \to r$ for some $x \in \mathcal{V}$, we only consider TRSs where all left-hand sides of rules are not variables in the remainder.

## 13.4. Loops

Loops are derivations of the shape $t \to_{\mathcal{R}}^{+} C[t\mu]$. They always imply nontermination where the corresponding infinite reduction is

$$t \to_{\mathcal{R}}^{+} C[t\mu] \to_{\mathcal{R}}^{+} C[C\mu[t\mu^2]] \to_{\mathcal{R}}^{+} C[C\mu[C\mu^2[t\mu^3]]] \to_{\mathcal{R}}^{+} \cdots \tag{5}$$

A TRS which admits a loop is called *looping*.

Note that for innermost rewriting, loopingness does not necessarily imply nontermination, since the innermost rewrite relation is not closed under substitutions. More precisely, it is not enough to have an "innermost loop" of the form $t \overset{\mathsf{i}}{\to}_{\mathcal{R}}^{+} C[t\mu]$, since this does not necessarily imply an infinite sequence (5) when restricting to innermost rewriting. Therefore, in [103], the notion of an *innermost loop* was introduced. To facilitate the certification of innermost loops (i.e., to decide for a given loop, whether it is innermost or not), we need its constituting steps, i.e., a derivation of length $m > 0$ with redexes $\ell_i\sigma_i$:

$$t = t_1 \to_{\mathcal{R},\ell_1\sigma_1} t_2 \to_{\mathcal{R},\ell_2\sigma_2} \cdots \to_{\mathcal{R},\ell_m\sigma_m} t_{m+1} = C[t\mu] \tag{6}$$

**Definition 13.2** (Innermost Loops)**.** *A loop (6) is an* innermost loop *iff for all* $1 \leqslant i \leqslant m$ *and* $n \in \mathbb{N}$*, the term* $\ell_i\sigma_i\mu^n$ *is an innermost redex.*

That is, no matter how often $\mu$ is applied, all steps should be innermost.

**Lemma 13.3.** *A loop (6) is an innermost loop iff (5) is an innermost derivation.*

**Corollary 13.4.** *An innermost loop implies innermost nontermination.*

Note that for every loop (6) and all $n \in \mathbb{N}$, the term $\ell_i\sigma_i\mu^n$ is a redex. Hence, to make sure that those redexes are innermost, it suffices to check whether all arguments of $\ell_i\sigma_i\mu^n$ are normal forms for all $n \in \mathbb{N}$. Since $\ell_i\sigma_i$ is not a variable (we ruled out variables as left-hand sides of $\mathcal{R}$) this is equivalent to checking that for all arguments $t$ of $\ell_i\sigma_i$, the term $t\mu^n$ is a normal form for all $n \in \mathbb{N}$. Thus, to decide whether a loop is innermost, we can use the following characterization.

**Lemma 13.5.** *Let $\mathcal{R}$ be a TRS, (6) a loop, and $\mathcal{A} = \bigcup_{1 \leqslant i \leqslant m} args(\ell_i\sigma_i)$ the set of arguments of redexes in (6). Then, (6) is an innermost loop, iff for all $t \in \mathcal{A}$ and $n \in \mathbb{N}$ the term $t\mu^n$ is a normal form, iff for all $t \in \mathcal{A}$ and $\ell \to r \in \mathcal{R}$ the term $t\mu^n$ does not contain a redex $\ell\sigma$ for any $n \in \mathbb{N}$ and $\sigma$.*

Hence, we can easily check, whether a loop is innermost, whenever for two terms $t$ and $\ell$, and a substitution $\mu$, we can solve the problem whether there exist $n$ and $\sigma$, such that $t\mu^n$ contains a redex $\ell\sigma$. Such problems are called *redex problems* and a large part of [103] is devoted to develop a corresponding decision procedure.

**Example 13.6.** Consider a loop $t \to^+ C[t\mu]$ for a TRS $\mathcal{R}$ containing rules (1)-(4), where $\mu = \{x/\mathsf{cons}(z, y), y/\mathsf{cons}(z, x), z/\mathsf{0}\}$. Let $D[\mathsf{chk}(\mathsf{eq}(x, y))] \to D[\mathsf{false}]$ be a step of the loop. Then, for an innermost loop we must ensure that the term $\mathsf{eq}(x, y)\mu^n$ does not contain a redex w.r.t. $\mathcal{R}$, especially not w.r.t. rule (2).

The above decision procedure works in three phases: first, redex problems are simplified into a set of matching problems. Then, a modified matching algorithm is employed, where in the end identity problems have to be solved. Finally, a decision procedure for identity problems is applied.

In the remainder, let $\mu$ be an arbitrary but fixed substitution (usually originating from some loop $t \to^+_{\mathcal{R}} C[t\mu]$).

**Definition 13.7** (Redex, Matching, and Identity Problems)**.** *Let $s$, $t$, and $\ell$ be terms. Then a* redex problem *is a pair $t \mid\!\gtrdot \ell$, a* generalized matching problem *is a set of pairs $\{t_1 \gtrdot \ell_1, \ldots, t_k \gtrdot \ell_k\}$ (we call a generalized matching problem having only one pair, a* matching problem*, and drop the surrounding braces), and an* identity problem *is a pair $s \approxeq t$.*

*A redex problem $t \mid\!\gtrdot \ell$ is* solvable *iff there is a context $C$, a substitution $\sigma$, and an $n \in \mathbb{N}$ such that $t\mu^n = C[\ell\sigma]$. A (generalized) matching problem is* solvable *iff there is a substitution $\sigma$ and an $n \in \mathbb{N}$ such that $t_i\mu^n = \ell_i\sigma$ for all pairs $t_i \gtrdot \ell_i$. An identity problem is* solvable *iff there is an $n \in \mathbb{N}$ such that $s\mu^n = t\mu^n$. In those respective cases, we call $(C, \sigma, n)$, $(\sigma, n)$, and $n$, the* solution*.*

## 13.5. Formalization

In [109] a straightforward certification algorithm for loops is described which does nothing else than checking rewrite steps. We extend this result significantly by also formalizing the necessary machinery to decide whether a loop is innermost. In the following, we discuss the three phases of the decision procedure from [103].

**From Redex Problems to Matching Problems.** A redex problem $t \mid\!\gtrdot \ell$ with $\ell \in \mathcal{V}$ is trivially solvable using the solution $(\Box, \{\ell/t\}, 0)$. Thus, in the following we assume that $\ell \notin \mathcal{V}$. Then, solvability of $t \mid\!\gtrdot \ell$ is equivalent to the existence of a non-variable subterm $s$ of $t\mu^n$ such that $s = \ell\sigma$ (i.e., $\ell$ matches $s$). In order to simplify redex problems, we represent these subterms in a finite way and consequently generate only finitely many matching problems.

Either, $s$ starts inside $t$, so $s = u\mu^n$ for some $u \trianglelefteq_{\mathcal{F}} t$, or $s$ is completely inside $\mu^n$. But then, it must be of the form $u\mu^n$ for some $u \trianglelefteq_{\mathcal{F}} x\mu$ and $x$ in $\mathcal{W}(t) = \bigcup_n \mathcal{V}(t\mu^n)$, where $\mathcal{W}(t)$ collects all variables which can possibly occur in a term of the form $t\mu^n$. In both

cases, the equality $s = \ell\sigma$ can be reformulated to $u\mu^n = \ell\sigma$, i.e., solvability of the matching problem $u \trianglerighteq \ell$. In total, the redex problem is solvable iff one of the matching problems $u \trianglerighteq \ell$ is solvable for some $u \in \mathcal{U}(t)$, where $\mathcal{U}(t) = \{u \mid t \trianglerighteq_{\mathcal{F}} u \text{ or } x\mu \trianglerighteq_{\mathcal{F}} u \wedge x \in \mathcal{W}(t)\}$.

The following theorem (whose formalization was straightforward), corresponds to [103, Theorem 10].

**Theorem 13.8.** *Let $t \mid\trianglerighteq \ell$ be a redex problem. Let*

$$\mathcal{M}_{init}(t, \ell) = \mathit{if}\ \ell \in \mathcal{V}\ \mathit{then}\ \{t \trianglerighteq \ell\}\ \mathit{else}\ \{u \trianglerighteq \ell \mid u \in \mathcal{U}(t)\}$$

*be the set of initial matching problems. Then $t \mid\trianglerighteq \ell$ is solvable iff one of the matching problems in $\mathcal{M}_{init}(t, \ell)$ is solvable.*

**Example 13.9.** Continuing with Example 13.6, from each redex problem $\mathsf{eq}(x, y) \mid\trianglerighteq \ell$ we obtain the matching problems $\mathsf{eq}(x, y) \trianglerighteq \ell$, $\mathsf{cons}(z, y) \trianglerighteq \ell$, $\mathsf{cons}(z, x) \trianglerighteq \ell$, and $\mathsf{0} \trianglerighteq \ell$ where $\ell$ is an arbitrary left-hand side of the TRS.

Theorem 13.8 shows a way to convert redex problems into matching problems. However, for certification, it remains to develop an algorithm that actually computes $\mathcal{M}_{init}$. To this end, we need to compute $\mathcal{U}(t)$, which in turn requires to enumerate all subterms of a term and to compute $\mathcal{W}(t)$. Whereas the former is straightforward, computing $\mathcal{W}(t)$ is a bit more difficult: its original definition contains an infinite union.

Note that $\mathcal{W}(t)$ is only finite since we restrict to substitutions of finite domain and can be computed by a fixpoint algorithm: iteratively compute $\mathcal{V}(t)$, $\mathcal{V}(t\mu)$, $\mathcal{V}(t\mu^2)$, ..., until some $\mathcal{V}(t\mu^k)$ is reached where no new variables are detected. In principle, it is possible to formalize this algorithm directly, but we expect such a formalization to require tedious manual termination and soundness proofs. Thus instead, we characterize $\mathcal{W}(t)$ by the following reflexive transitive closure.

**Lemma 13.10.** *Let $R = \{(x, y) \mid x \neq y, x \in \mathcal{V}, y \in \mathcal{V}(x\mu)\}$. Then $\mathcal{W}(t) = \{y \mid \exists x \in \mathcal{V}(t), (x, y) \in R^*\}$.*

Note that $R$ in Lemma 13.10 can easily be computed since whenever $(x, y) \in R$ then $x \in dom(\mu)$. Moreover, $R$ is finite since we only consider substitutions of finite domain. Hence, the above characterization allows us to compute $\mathcal{W}$ by the algorithm of [98] (generating the reflexive transitive closure of finite relations).

Note that $\mathcal{W}(t)$ can also be defined inductively as the least set such that $\mathcal{V}(t) \subseteq \mathcal{W}(t)$ and $x \in \mathcal{W}(t) \Longrightarrow \mathcal{V}(x\mu) \subseteq \mathcal{W}(t)$. And whenever a finite set $S$ is defined inductively, instead of implementing an executable algorithm for $S$ manually, it might be easier to characterize $S$ via reflexive transitive closures and afterwards execute it via the algorithm of [98]. This approach is not restricted to $\mathcal{W}$: it has been applied in the next paragraph and also in other parts of IsaFoR.

An alternative might be Isabelle/HOL's predicate compiler [10]. It can be used to obtain executable functions for inductively defined predicates and sets. However, without manual tuning we were not able to obtain appropriate equations for the code generator. Furthermore, additional tuning is required to ensure termination of the resulting code in the target language. Ultimately, the current version of the predicate compiler provides a fixed execution model for predicates and sets (goal-oriented depth-first search) which might not yield the best performance for the desired application. Thus, for the time being we use our proposed solution via reflexive transitive closures, but perhaps in future versions of Isabelle/HOL, the predicate compiler will be a more convenient alternative.

**From Matching Problems to Identity Problems.** To decide solvability of a (generalized) matching problem $\{t_1 \gtrdot \ell_1, \ldots, t_k \gtrdot \ell_k\}$, in [103], a variant of a standard matching algorithm is used which simplifies (generalized) matching problems until they are in *solved form*, i.e., all right-hand sides $\ell_i$ are variables (or $\bot$ is obtained which represents a matching problem without solution).

**Definition 13.11** (Transformation of Matching Problems)**.** *In [103] the following transformation $\Rightarrow$ on general matching problems is defined. If $\mathcal{M}$ is a general matching problem with $\mathcal{M} = \{t \gtrdot \ell\} \uplus \mathcal{M}'$ where $\ell \notin \mathcal{V}$, then*

*(i)* $\mathcal{M} \Rightarrow \{t_1 \gtrdot \ell_1, \ldots, t_k \gtrdot \ell_k\} \cup \mathcal{M}'$, *if* $t = f(t_1, \ldots, t_k)$ *and* $\ell = f(\ell_1, \ldots, \ell_k)$

*(ii)* $\mathcal{M} \Rightarrow \bot$, *if* $t = f(\ldots)$, $\ell = g(\ldots)$, *and* $f \neq g$

*(iii)* $\mathcal{M} \Rightarrow \bot$, *if* $t \in \mathcal{V} \setminus \mathcal{V}_{\mathsf{incr}}$

*(iv)* $\mathcal{M} \Rightarrow \{t'\mu \gtrdot \ell' \mid t' \gtrdot \ell' \in \mathcal{M}\}$, *if* $t \in \mathcal{V}_{\mathsf{incr}}$

The first two rules are the standard decomposition and clash rules. Moreover, there are two special rules to handle the case where $t$ is a variable. Here, the set of *increasing variables* $\mathcal{V}_{\mathsf{incr}} = \{x \mid \exists n.\, x\mu^n \notin \mathcal{V}\}$ plays a crucial role. It collects all those variables for which $\mu$, if applied often enough, introduces a non-variable term. In other words, $x\mu^n$ will always be a variable for $x \notin \mathcal{V}_{\mathsf{incr}}$.

In our development, instead of using the above relation, we formalized the rules directly as a function *simplify-mp* applying the transformation rules deterministically (thereby avoiding the need for a confluence proof, as was required in [103]). As input it takes two generalized matching problems (represented by lists) where the second problem is assumed to be in solved form. Here, [] and $\cdot$ are the list constructors, and @ denotes list concatenation. The possibility of failure is encoded using Isabelle/HOL's option type, which is either *None*, in case of an error, or *Some* $r$ for the result $r$. In contrast to Definition 13.11 of [103], our algorithm also returns an integer $i$ which provides a lower bound on how often $\mu$ has to be applied to get a solution. The function is given by the following equations (where for brevity do-notation in the option-monad is used):

$$
\begin{array}{lll}
\textit{simplify-mp} & [] & s = \textsf{return}\ (s, 0) \\
\textit{simplify-mp} & ((t, x) \cdot mp) & s = \textit{simplify-mp}\ mp\ ((t, x) \cdot s) \\
\textit{simplify-mp} & ((f(ss), g(ts)) \cdot mp) & s = \textsf{do}\ \{\ \textit{guard}\ (f = g); ps \leftarrow \textit{zip-option}\ ss\ ts; \\
 & & \quad \textit{simplify-mp}\ (ps@mp)\ s\ \}
\end{array}
$$

$$
\begin{array}{lll}
\textit{simplify-mp} & ((x, g(ts)) \cdot mp) & s = \textsf{do}\ \{\ \textit{guard}\ (x \in \mathcal{V}_{\mathsf{incr}}); \\
 & & \quad (mp', i) \leftarrow \textit{simplify-mp} \\
 & & \quad\ (\textit{map-}\mu\ ((x, g(ts)) \cdot mp))\ (\textit{map-}\mu\ s); \\
 & & \quad \textsf{return}\ (mp', i + 1)\ \}
\end{array}
$$

where, $\textit{map-}\mu = \textit{map}\ (\lambda(t, \ell).(t\mu, \ell))$ using the standard map function for lists, *zip-option* combines two lists of equal length into *Some* list of pairs and yields *None* otherwise, and *guard* aborts with *None* if the given predicate is not satisfied.

**Example 13.12.** For $\ell = \mathsf{eq}(x, x)$, only one of the redex problems of Example 13.9 remains (all others are simplified to *None*), namely $\mathsf{eq}(x, y) \gtrdot \mathsf{eq}(x, x)$, for which we obtain the simplified matching problem $\{x \gtrdot x, y \gtrdot x\}$.

In our formalization we show all relevant properties of *simplify-mp*, i.e., termination, preservation of solvability, and that *simplify-mp mp* [], if successful, is in solved form. Moreover, we prove the computed lower bound to be sound.

**Theorem 13.13.** *The function simplify-mp satisfies the following properties:*

- *It is terminating.*
- *It is complete, i.e., if* $(n, \sigma)$ *is a solution for mp then there are* $mp'$ *and* $i$ *such that simplify-mp mp* [] = *Some* $(mp', i)$, $i \leqslant n$, *and* $(n - i, \sigma)$ *is a solution for* $mp'$*;*
- *It is sound, i.e., if* $(n, \sigma)$ *is a solution for* $mp'$ *and simplify-mp mp* [] = *Some* $(mp', i)$ *then* $(n + i, \sigma)$ *is a solution for mp;*
- *If simplify-mp mp* [] = *Some* $(mp', i)$ *then* $mp'$ *is in solved form.*

*Proof.* For termination of *simplify-mp mp s*, where $mp = [(t_1, \ell_1), \ldots, (t_k, \ell_k)]$, we use the lexicographic combination of the following two measures: first, we measure the sum of the sizes of the $\ell_i$; and second, we measure the sum of the distances of the $t_i$ before turning into non-variables. Here, the distance of some term $t_i$ before turning into a non-variable is 0 if $t_i \in \mathcal{V} \setminus \mathcal{V}_{\mathsf{incr}}$ and the least number $d$ such that $t_i\mu^d \notin \mathcal{V}$, otherwise.

For this lexicographic measure, we get a decrease in the first component for the first and the second recursive call, and a decrease in the second component for the third recursive call.

Proving soundness and completeness is done via the following property which is proven by induction on the call structure of *simplify-mp*.

Whenever *simplify-mp mp* $s = r$ then

- if $r =$ *None* then $mp \cup s$ is not solvable,
- if $r =$ *Some* $(mp', i)$, there is no solution $(n, \sigma)$ for $mp \cup s$ where $n < i$, and $(n, \sigma)$ is a solution for $mp'$ iff $(n + i, \sigma)$ is a solution for $mp \cup s$.

Finally, the fact that *simplify-mp mp* [] is in solved form is shown by an easy induction proof on the call structure of *simplify-mp*, where [] is generalized to an arbitrary generalized matching problem that is in solved form. □

Although *simplify-mp* is defined as a recursive function, it cannot directly be used as a certification algorithm, due to the following two problems:

The first problem is that $\mathcal{V}_{\mathsf{incr}}$ is not executable, since it contains an existential statement (remember that we had a similar problem for $\mathcal{W}$ earlier). Again, $\mathcal{V}_{\mathsf{incr}}$ could be computed via a fixpoint computation accompanied by a tedious manual termination proof. Instead, we once more employ reflexive transitive closures to characterize $\mathcal{V}_{\mathsf{incr}}$, which allows us to use the algorithm of [98] to compute it.

**Lemma 13.14.** *Let* $R = \{(x, y) \mid x \neq y, x = y\mu, x \in \mathcal{V}, y \in \mathcal{V}\}$. *Then* $\mathcal{V}_{\mathsf{incr}} = \{y \mid \exists x \in \mathcal{V}, x\mu \notin \mathcal{V}, (x, y) \in R^*\}$.

The second problem is the usage of implicit parameters. Recall that at the end of Sect. 13.4 we just fixed some substitution $\mu$ (which corresponds to what we did in our formalization using Isabelle/HOL's locale mechanism). Obviously, both $\mathcal{V}_{\mathsf{incr}}$ and *simplify-mp*

depend on $\mu$. Hence, we have to pass $\mu$ as argument to both. As a result, the modified version of the last equation of *simplify-mp* looks as follows:

$$\begin{aligned} &\textit{simplify-mp}\ \mu\ ((x, g(ts)) \cdot mp)\ s = \textsf{do}\ \{\textit{guard}\ (x \in \mathcal{V}_{\textsf{incr}}(\mu));\\ &\quad (mp', i) \leftarrow \textit{simplify-mp}\ \mu\ (\textit{map-}\mu\ ((x, g(ts)) \cdot mp))\ (\textit{map-}\mu\ s);\\ &\quad \textsf{return}\ (mp', i+1)\ \} \end{aligned} \tag{7}$$

The problem of equation (7) is its inefficiency: In every recursive call, the set of increasing variables $\mathcal{V}_{\textsf{incr}}(\mu)$ is newly computed. Therefore, the obvious idea is to compute $\mathcal{V}_{\textsf{incr}}(\mu)$ once and for all and pass it as an additional argument $V$.

$$\begin{aligned} &\textit{simplify-mp}\ \mu\ V\ ((x, g(ts)) \cdot mp)\ s = \textsf{do}\ \{\textit{guard}\ (x \in V);\\ &\quad (mp', i) \leftarrow \textit{simplify-mp}\ \mu\ V\ (\textit{map-}\mu\ ((x, g(ts)) \cdot mp))\ (\textit{map-}\mu\ s);\\ &\quad \textsf{return}\ (mp', i+1)\ \} \end{aligned} \tag{8}$$

This version does not have the problem of recomputing $\mathcal{V}_{\textsf{incr}}(\mu)$ and we just have to replace the initial call *simplify-mp* $\mu$ *mp* [] by *simplify-mp* $\mu$ $\mathcal{V}_{\textsf{incr}}(\mu)$ *mp* [].

Although, this looks straightforward and maybe not even worth mentioning, we stress that this solution does not work properly. The problem is that by introducing $V$, we can call *simplify-mp* using some $V \neq \mathcal{V}_{\textsf{incr}}(\mu)$, which can cause nontermination. Take for example $\mu$ as the empty substitution and $V = \{x\}$, then the function call *simplify-mp* $\mu$ $V$ $[(x, g(ts))]$ [] directly leads to exactly the same function call via (8). Hence, termination of *simplify-mp* defined by (8) cannot be proven. Therefore, the function package of Isabelle/HOL [69] weakens equality (8) by the assumption that *simplify-mp* has to be terminating on the arguments $\mu$, $V$, $((x, g(ts)) \cdot mp)$, and $s$.

Of course, we can instantiate (8) by $V = \mathcal{V}_{\textsf{incr}}(\mu)$. Then we can get rid of the additional assumption. But still, the corresponding unconditional equation is not suitable for code generation, since $\mathcal{V}_{\textsf{incr}}$ on the left-hand side is not a constructor.

Our final solution is to use the recent *partial-function* [70] command of Isabelle/HOL which generates unconditional equations even for nonterminating functions, provided that some syntactic restrictions are met (only one defining equation and the function must either return an option type or be tail-recursive).

Since *simplify-mp* already returns an option type, we just had to merge all equations into a single case statement. (If the result is not of option type, we can just wrap the original return type into an option type). Afterwards the *partial-function* command is applicable and we obtain an equation similar to (8) which can be processed by the code generator and efficiently computes *simplify-mp* without recomputing $\mathcal{V}_{\textsf{incr}}(\mu)$. Moreover, since we have already shown termination of the inefficient version of *simplify-mp*, we know that also the efficient version does terminate whenever it is called with $V = \mathcal{V}_{\textsf{incr}}(\mu)$. In our formalization we actually have two versions of *simplify-mp*: an abstract version which is unsuitable for code generation (and also inefficient) and a concrete version. All the above properties are proven on the abstract version neglecting any efficiency problems. Afterwards it is shown that the concrete version computes the same results as the abstract one (which is relatively easy since the call-structure is the same). In this way, we get the best of two worlds: abstraction and ease of reasoning from the abstract version (using sets, existential statements, and the induction rules from the function package), and efficiency from the concrete version (using lists and concrete functions to obtain witnesses).

The above mentioned problem is not restricted to *simplify-mp*. Whenever the termination of a function relies on the correct initialization of some precomputed values, a similar problem arises. Currently, this can be solved by writing a second function via the

*partial-function* command, as shown above. Although the second definition is mainly a copy of the original one, we can currently not recommend to use it as a replacement, since the function package provides much more convenience for standard definitions than when using the *partial-function* command. If the functionality of partial functions is extended, the situation might change (and we would welcome any effort in that direction).

Continuing with deciding matching problems, we are in the situation, that by using *simplify-mp* we can either directly detect that a matching problem is unsolvable or obtain an equivalent generalized matching problem in solved form $\mathcal{M} = \{t_1 \trianglerighteq x_1, \ldots, t_k \trianglerighteq x_k\}$. In principle, $\mathcal{M}$ has the solution $(n, \sigma)$ where $n$ is arbitrary and $\sigma(x_i) = t_i\mu^n$. However, this definition of $\sigma$ is not always well-defined if there are $i$ and $j$ such that $x_i = x_j$ and $i \neq j$. To decide whether it is possible to adapt the proposed solution, we must know whether $t_i\mu^n = t_j\mu^n$ for some $n$, i.e., we must solve the identify problem $t_i \cong t_j$.

The following result of [103, Theorem 14 (iv)] is easily formalized and also poses no challenges for certification. Afterwards it remains to decide identity problems.

**Theorem 13.15.** *Let $\mathcal{M} = \{t_1 \trianglerighteq x_1, \ldots, t_k \trianglerighteq x_k\}$ be a generalized matching problem in solved form. Define $\mathcal{I}_{init} = \{t_i \cong t_j \mid 1 \leqslant i < j \leqslant k, x_i = x_j\}$. Then $\mathcal{M}$ is solvable iff all identity problems in $\mathcal{I}_{init}$ are solvable.*

To prove this theorem, the key observation is that we can always combine several solutions of identity problems: Whenever $n_{ij}$ are solutions to the identity problems $t_i \cong t_j$, respectively, then the maximum $n$ of all $n_{ij}$ is a solution to all identity problems $t_i \cong t_j$. And then also $(n, \sigma)$ is a solution to $\mathcal{M}$ where $\sigma(x_i) = t_i\mu^n$ is guaranteed to be well-defined.

**Example 13.16.** For the remaining matching problem of Example 13.12 we generate one identity problem: $x \cong y$.

**Deciding Identity Problems.** In [103, Section 3.4] a complicated algorithm is presented to decide solvability of an identity problem $s \cong t$. The main idea is to iteratively generate $(s, t)$, $(s\mu, t\mu)$, $(s\mu^2, t\mu^2), \ldots$ until either some $(s\mu^i, t\mu^i)$ with $s\mu^i = t\mu^i$ is generated, or it can be detected that no solution exists. For the latter, some easy conditions for unsolvability are identified, e.g., $s\mu^i = C[f(ss)]$ and $t\mu^i = C[x]$ where $x \notin \mathcal{V}_{\mathsf{incr}}$. However, these conditions do not suffice to detect all unsolvable identity problems. Therefore, in each iteration conflicts (indicating which subterms have to become equal after applying $\mu$ several times, to obtain overall equality), are stored in a set $S$, and two sufficient conditions on pairs of conflicts from $S$ are presented that allow to conclude unsolvability.

For the overall algorithm, soundness is rather easy to establish, completeness is more challenging, and the termination proof is the most difficult part. To be more precise, it is shown that nontermination of the algorithm allows to construct an infinite sequence of terms where no two terms are embedded into each other (which is not possible due to Kruskal's tree theorem). Hence, the formalization would require a formalization of the tree theorem. Moreover, the implicit complexity bound on the number of required iterations is quite high.

The reason for using Kruskal's tree theorem is that in [103] the conflicts in $S$ consist of a variable, a position, and a term which is not bounded in its size. So, there is no a priori bound on $S$. We were able to simplify the decision procedure for $s \cong t$ considerably since we only store conflicts whose constituting terms are in the set of *conflict terms*

$$\mathcal{CT}(s, t) = \{u \mid v \trianglerighteq u, v \in \{s, t\} \cup ran(\mu)\}.$$

To be more precise, all conflicts are of the form $(u, v, m)$ where $(u, v)$ is contained in the finite set $\mathcal{S} = (\mathcal{CT}(s,t) \cap \mathcal{V}) \times \mathcal{CT}(s,t)$. Whenever we see a conflict $(u, v, \_)$ for the second time, the algorithm stops. Thus, we get a decision procedure which needs at most $|\mathcal{S}|$ iterations and whose termination proof is easy. In contrast to [103], our procedure does neither require any preprocessing on $\mu$ nor unification.

The key idea to get an a priory bound on the set of conflicts, is to consider identity problems of a generalized form $s \approxeq t\mu^n$ which can be represented by the triple $(s, t, n)$. Then applying substitutions can be done by increasing $n$, and all terms that are generated during an execution of the algorithm are terms from $\mathcal{CT}(s,t)$.

Before presenting the main algorithm for deciding identity problems $s \approxeq t\mu^n$, we require an auxiliary algorithm *conflicts* $(s, t, n)$ that computes the set of conflicts for an identity problem, i.e., subterms of $s$ and $t\mu^n$ with different roots.

$$
\begin{array}{llll}
\mathit{conflicts} & (s, y, n+1) & = \mathit{conflicts}\ (s, \mu(y), n) \\
\mathit{conflicts} & (x, y, 0) & = \mathit{if}\ x = y\ \mathit{then}\ \emptyset\ \mathit{else}\ \{(x, y, 0)\} \\
\mathit{conflicts} & (f(ss), y, 0) & = \{(y, f(ss), 0)\} \\
\mathit{conflicts} & (x, g(ts), n) & = \{(x, g(ts), n)\} \\
\mathit{conflicts} & (f(ss), g(ts), n) & = \mathit{if}\quad f = g \wedge |ss| = |ts| \\
 & & \quad \mathit{then}\ \bigcup_{(s_i,t_i) \in zip\ ss\ ts} \mathit{conflicts}\ (s_i, t_i, n) \\
 & & \quad \mathit{else}\ \{(f(ss), g(ts), n)\}
\end{array}
$$

We identified and formalized the following properties of *conflicts* and $\mathcal{CT}$.

**Lemma 13.17.**

- $s\sigma = t\mu^n\sigma$ *iff* $\forall (u, v, m) \in \mathit{conflicts}\ (s, t, n).\ u\sigma = v\mu^m\sigma$.
- *if* $(u, v, m) \in \mathit{conflicts}\ (s, t, n)$ *then*
  - $\mathit{root}(u) \neq \mathit{root}(v)$
  - $v \in \mathcal{V}$ *implies* $m = 0 \wedge u \in \mathcal{V}$
  - $\exists k\, p.\ n = m + k \wedge ((s|_p, t\mu^k|_p) = (u, v) \vee ((s|_p, t\mu^k|_p) = (v, u) \wedge m = 0))$
  - $\{u, v\} \subseteq \mathcal{CT}(s,t)$
- $\{u, v\} \subseteq \mathcal{CT}(s,t)$ *implies* $\mathcal{CT}(u, v) \subseteq \mathcal{CT}(s,t)$
- $\mathcal{CT}(u, v) \subseteq \mathcal{CT}(u\mu, v)$ *whenever* $u \in \mathcal{V}$

Using *conflicts* we can now formulate the algorithm *ident-solve* which decides identity problem $s \approxeq t$ if invoked with *ident-solve* $\emptyset$ $(s, t, 0)$.

```
ident-solve S idp =
  let C = conflicts idp in
  if (f(us), _, _) ∈ C ∨ ((u, v, _) ∈ C ∧ (u, v, _) ∈ S) then None else do {
    ns ← map-option (λ(u, v, m). ident-solve ({(u, v, m)} ∪ S) (uμ, v, m + 1)) C;
    return (max {n + 1 | n ∈ ns}) }
```

where *map-option* is a variant of the map function on lists whose overall result is *None* if the supplied function returns *None* for any element of the given list.

**Example 13.18.** We continue Example 13.16 by invoking *ident-solve* $\emptyset$ $(x, y, 0)$. This leads to the conflict $(x, y, 0)$. Afterwards, *ident-solve* $\{(x, y, 0)\}$ $(\mathsf{cons}(z, y), y, 1)$ is invoked which results in the conflict $(y, x, 0)$. Finally, the conflict $(x, y, 0)$ is generated again when calling *ident-solve* $\{(x, y, 0), (y, x, 0)\}$ $(\mathsf{cons}(z, x), x, 1)$ and the result *None* is obtained.

We formalized termination, soundness, and completeness of *ident-solve*.

**Lemma 13.19** (Termination)**.** *ident-solve is terminating.*

*Proof.* Take the measure function $\lambda S\ (s,t,\_).\ |(\mathcal{CT}(s,t)\cap\mathcal{V})\times\mathcal{CT}(s,t)\setminus\{(a,b)\mid(a,b,\_)\in S\}|$. Then the actual termination proof boils down to showing

$$\begin{aligned}L &:= (\mathcal{CT}(s,t)\cap\mathcal{V})\times\mathcal{CT}(s,t)\setminus\{(a,b)\mid(a,b,\_)\in S\}\\ &\supset (\mathcal{CT}(u\mu,v)\cap\mathcal{V})\times\mathcal{CT}(u\mu,v)\setminus\{(a,b)\mid(a,b,\_)\in\{(u,v,m)\}\cup S\} =: R\end{aligned}$$

whenever *ident-solve* $S\ (s,t,n)$ leads to a call *ident-solve* $(\{(u,v,m)\}\cup S)\ (u\mu,v,m+1)$, i.e., whenever $(u,v,m)\in$ *conflicts* $(s,t,n)$, $(u,v,\_)\notin S$, and $u\in\mathcal{V}$. By Lemma 13.17 we obtain $\{u,v\}\subseteq\mathcal{CT}(s,t)$ and $\mathcal{CT}(u\mu,v)\subseteq\mathcal{CT}(u,v)\subseteq\mathcal{CT}(s,t)$. Hence, $L\supseteq R$ and since $(u,v)\in L\setminus R$ we even have $L\supset R$. □

**Lemma 13.20** (Soundness)**.** *If ident-solve* $S\ (s,t,n) =$ *Some* $i$ *then* $s\mu^i = t\mu^n\mu^i$.

*Proof.* We perform induction on the call-structure of *ident-solve*. So, assume *ident-solve* $S\ (s,t,n) =$ *Some* $i$. By definition of *ident-solve*, for all $(u,v,m)\in$ *conflicts* $(s,t,n)$ there is some $j$ such that *ident-solve* $(\{(u,v,m)\}\cup S)\ (u\mu,v,m+1) =$ *Some* $j$ and $i$ is the maximum of all $j+1$. Using the induction hypothesis, we conclude $u\mu^{j+1} = u\mu\mu^j = v\mu^{m+1}\mu^j = v\mu^m\mu^{j+1}$ for all $(u,v,m)\in$ *conflicts* $(s,t,n)$, and since $i\geq j+1$ we also achieve $u\mu^i = v\mu^m\mu^i$. But this is equivalent to $s\mu^i = t\mu^n\mu^i$ by Lemma 13.17 (where $\sigma=\mu^i$). □

**Lemma 13.21** (Completeness)**.** *Whenever the identity problem* $s\approxeq t$ *is solvable then ident-solve* $\emptyset\ (s,t,0)\neq$ *None.*

*Proof.* If $s\approxeq t$ is solvable then there is some $N$ such that $s\mu^N = t\mu^N$. Our actual proof shows the following property $(\star)$ for all $S$, $s'$, $t'$, $n$, $n'$, and $p$ where $(a,b)\overset{\leftrightarrow}{=}(c,d)$ abbreviates $(a,b)=(c,d)\vee(a,b)=(d,c)$.[1]

$$(s\mu^n|_p, t\mu^n|_p)\overset{\leftrightarrow}{=}(s',t'\mu^{n'}) \tag{9}$$
$$\longrightarrow (\forall(u,v,m)\in S.\,(m=0\vee v\notin\mathcal{V})\wedge root(u)\neq root(v)\wedge \tag{10}$$
$$\qquad(\exists q_1\,q_2\,n_1.\,p=q_1q_2\wedge n_1<n\wedge(s\mu^{n_1}|_{q_1}, t\mu^{n_1}|_{q_1})\overset{\leftrightarrow}{=}(u,v\mu^m)))$$
$$\longrightarrow \textit{ident-solve}\ S\ (s',t',n')\neq \textit{None} \tag{11}$$

Once $(\star)$ is established, the lemma immediately follows from $(\star)$ which is instantiated by $S=\emptyset$, $s'=s$, $t'=t$, $n'=n=0$, and $p=\epsilon$ (the empty position).

To prove $(\star)$, we perform induction on the call-structure of *ident-solve*. So, we assume (9) and (10), and have to show (11). By $s\mu^N = t\mu^N$ we conclude $s\mu^n|_p\mu^N = s\mu^N\mu^n|_p = t\mu^N\mu^n|_p = t\mu^n|_p\mu^N$, and thus $s'\mu^N = t'\mu^{n'}\mu^N$ by (9). By Lemma 13.17 this shows $u\mu^N = v\mu^m\mu^N$ for all $(u,v,m)\in$ *conflicts* $(s',t',n') =: \mathcal{C}$. In a similar way we prove $u\mu^N = v\mu^m\mu^N$ for all $(u,v,m)\in S$ using (10).

Next we consider an arbitrary $(u,v,m)\in\mathcal{C}$. By Lemma 13.17 we have $root(u)\neq root(v)$, $m=0\vee v\notin\mathcal{V}$, and there are $q_1$ and $k$ such that $n'=m+k$ and $(s'|_{q_1}, t'\mu^k|_{q_1}) = (u,v)\vee((s'|_{q_1}, t'\mu^k|_{q_1}) = (v,u)\wedge m=0)$. In particular, this implies $(s'|_{q_1}, t'\mu^k|_{q_1}\mu^m)\overset{\leftrightarrow}{=}(u,v\mu^m)$. Moreover, we know $u\mu^N = v\mu^m\mu^N$.

[1] In the formalization, $(\star)$ looks even more complicated, since here we dropped all parts that restrict $p$ and $q_1$ to valid positions.

First, we show that $u \in \mathcal{V}$, and hence the condition $(f(us), \_, \_) \in \mathcal{C}$ is not satisfied. The reason is that $u \notin \mathcal{V}$ also implies $v \notin \mathcal{V}$ by Lemma 13.17 which implies the contradiction $root(u) = root(u\mu^N) = root(v\mu^m\mu^N) = root(v) \neq root(u)$.

Second, *ident-solve* $(\{(u, v, m)\} \cup S)$ $(u\mu, v, m+1) \neq None$. To show this, we just apply the induction hypothesis where it remains to show that (9) and (10) are satisfied (where the values of $S$, $s'$ ,$t'$ ,$n$, $n'$, $p$ are $\{(u, v, m)\} \cup S$, $u\mu$, $v$, $n+1$, $m+1$, and $pq_1$, respectively). To this end, we derive the following equality.

$$\begin{array}{lll} (s\mu^n|_{pq_1}, t\mu^n|_{pq_1}) & = (s\mu^n|_p|_{q_1}, t\mu^n|_p|_{q_1}) & \overset{\leftrightarrow}{=} (s'|_{q_1}, t'\mu^{n'}|_{q_1}) \\ & = (s'|_{q_1}, t'\mu^k|_{q_1}\mu^m) & \overset{\leftrightarrow}{=} (u, v\mu^m). \end{array} \tag{12}$$

Using (12), $root(u) \neq root(v)$, and $m = 0 \vee v \notin \mathcal{V}$, we conclude that (10) is satisfied for the new conflict $(u, v, m)$. Moreover, (10) is trivially satisfied for all (old) conflicts in $S$, by using (10) (for the old inputs $(s', t', n'), \ldots$). Finally, by applying $\mu$ on all terms in (12) we obtain $(s\mu^{n+1}|_{pq_1}, t\mu^{n+1}|_{pq_1}) \overset{\leftrightarrow}{=} (u\mu, v\mu^{m+1})$ which is exactly the required (9).

The last potential reason for *ident-solve* $S$ $(s', t', n')$ to be *None*, is that there is some $m'$ such that $(u, v, m') \in S$. We assume that such an $m'$ exists and eventually show a contradiction (the most difficult part of this proof). By (10) we conclude that $m' = 0 \vee v \notin \mathcal{V}$ and there are $p_1$, $q_3$, and $n_2$ where $p = p_1q_3$, $n_2 < n$, and $(s\mu^{n_2}|_{p_1}, t\mu^{n_2}|_{p_1}) \overset{\leftrightarrow}{=} (u, v\mu^{m'})$. Since $n_2 < n$ there is some $k_1$ with $n = n_2 + k_1$ and $k_1 > 0$. Starting from (12) we derive

$$\begin{array}{lll} & (u, v\mu^m) & \overset{\leftrightarrow}{=} (s\mu^n|_{pq_1}, t\mu^n|_{pq_1}) \\ = & (s\mu^{n_2+k_1}|_{p_1q_3q_1}, t\mu^{n_2+k_1}|_{p_1q_3q_1}) & = (s\mu^{n_2}|_{p_1}\sigma|_q, t\mu^{n_2}|_{p_1}\sigma|_q) \\ \overset{\leftrightarrow}{=} & (u\sigma|_q, v\mu^{m'}\sigma|_q) & \end{array} \tag{13}$$

where $\sigma$ and $q$ are abbreviations for $\mu^{k_1}$ and $q_3q_1$, respectively. Using (13) it is possible to derive a contradiction via a case analysis.

If $m' = m$ then (13) yields both $u\sigma\sigma|_{qq} = u$ and $v\mu^m\sigma\sigma|_{qq} = v\mu^m$. Thus, $u(\sigma\sigma)^i|_{(qq)^i} = u$ and $v\mu^m(\sigma\sigma)^i|_{(qq)^i} = v\mu^m$ for all $i$. For $m' = m$ we can further show $u \neq v\mu^m$ and hence, $u(\sigma\sigma)^i|_{(qq)^i} \neq v\mu^m(\sigma\sigma)^i|_{(qq)^i}$ for all $i$. This leads to the desired contradiction since we know that $u\mu^N = v\mu^m\mu^N$, and hence $u(\sigma\sigma)^N = u\mu^{2k_1N} = u\mu^N\mu^{(2k_1-1)N} = v\mu^m\mu^N\mu^{(2k_1-1)N} = v\mu^m\mu^{2k_1N} = v\mu^m(\sigma\sigma)^N$, which shows that for $i = N$ the previous inequality does not hold.

Otherwise $m \neq m'$. Hence, $m \neq 0 \vee m' \neq 0$ and in combination with $m = 0 \vee v \notin \mathcal{V}$ and $m' = 0 \vee v \notin \mathcal{V}$ we conclude $v \notin \mathcal{V}$. Thus, $u \in \mathcal{V}$ by Lemma 13.17 as $(u, v, m) \in$ *conflicts* $(s', t', n')$. Then by a case analysis on (13) we can show that there are $i$ and $j$ such that $u\mu^i \rhd u\mu^j$. Moreover, from $u\mu^N = v\mu^m\mu^N$ and $u\mu^N = v\mu^{m'}\mu^N$ we obtain $u\mu^{N+m} = u\mu^{N+m'}$. In combination with $m \neq m'$ and $u\mu^i \rhd u\mu^j$ this leads to the desired contradiction. □

Putting all lemmas on *ident-solve* together, we can even give a decision procedure for identity problems which does not require *ident-solve* at all, and shows an explicit bound on a solution.

**Theorem 13.22.** *An identity problem $s \approxeq t$ is solvable iff $s\mu^n = t\mu^n$ where $n = |\mathcal{CT}(s,t) \cap \mathcal{V}| \cdot |\mathcal{CT}(s,t)|$.*

*Proof.* If an identity problem is solvable, then the result of *ident-solve* $\emptyset$ $(s, t, 0) = Some\ i$ for some $i$ by Lemma 13.21. From the termination proof in Lemma 13.19 we know that $i \leqslant |\mathcal{CT}(s,t) \cap \mathcal{V}| \cdot |\mathcal{CT}(s,t)| = n$ (unfortunately, in Isabelle/HOL we could not extract this knowledge from the termination proof and had to formalize this simple result separately). And by Lemma 13.20 we infer that $s\mu^i = t\mu^i$. But then also $s\mu^n = t\mu^n$. □

Note that $|\mathcal{CT}(s,t)| \leqslant |s| + |t| + |\mu|$ where $|\mu|$ is the size of all terms in the range of $\mu$. Hence, the value of $n$ in Theorem 13.22 is quadratic in the size of the input problem. We conjecture that even a linear bound exists, although some proof attempts failed. As an example, we tried to replace the condition $(u, v, \_) \in \mathcal{C} \wedge (u, v, \_) \in S$ by $(u, \_, \_) \in \mathcal{C} \wedge (u, \_, \_) \in S$ in *ident-solve* to get a linear number of iterations. However, then *ident-solve* is not complete anymore.

## 13.6. Conclusions

We have formalized several techniques to certify compositional (innermost) nontermination proofs, where the hardest part was the decision procedure of [103], which decides whether a loop is an innermost loop. In our formalization, we were able to simplify the algorithm and the proofs for identity problems considerably: a complex algorithm can be replaced by a single line due to Theorem 13.22.

With this result we can also show (but have not formalized) that all considered decision problems of this paper are in P.

**Theorem 13.23.** *Deciding whether an identity problem, a matching problem, or a redex problem is solvable is in P. Moreover, deciding whether a loop is an innermost loop is in P.*

*Proof.* We start with identity problems. By Theorem 13.22 we just have to check $s\mu^n = t\mu^n$ for $n = |\mathcal{CT}(s,t) \cap \mathcal{V}| \cdot |\mathcal{CT}(s,t)|$. When using DAG compressed terms we can represent $s\mu^n$ and $t\mu^n$ in polynomial space and in turn use the algorithms of [19,91] to check equality in polynomial time. Note that even if the input $(s, t, \mu)$ is already DAG compressed, the problem is still in P. The reason is that $|\mathcal{CT}(s,t)| \leqslant |s| + |t| + |\mu|$ also holds when sizes of terms are measured according to their DAG representation.

For matching problems $t \trianglerighteq \ell$, we first observe that *simplify-mp* $[(t, \ell)]$ [] requires at most $|\mathcal{V}_{\mathsf{incr}}| \cdot |\ell|$ many iterations, and when using DAG compression, the resulting simplified matching problem can be represented in polynomial space. Hence, the resulting identity problems can all be solved in polynomial time.

Using the result for matching problems, by Theorem 13.8 it follows that redex problems $t \,|\trianglerighteq\, \ell$ are decidable in P: The number of matching problems in $\mathcal{M}_{init}$ as well as the size of each element of $\mathcal{M}_{init}$ is linear in the sizes of $t$, $\ell$, and $\mu$.

Finally, since redex problems can be decided in P, by Lemma 13.5 this also holds for the question, whether a loop is an innermost loop. □

We have also shown how reflexive transitive closures can be used to avoid termination proofs, and how partial functions help to develop efficient algorithms.

We tested our algorithms within our certifier CeTA (version 2.3) in combination with the termination analyzer AProVE [39], which is (as far as we know) currently the only tool, that can prove innermost nontermination of term rewrite systems. Through our experiments, a major soundness bug in AProVE was revealed: one of the two loop-finding methods completely ignored the strategy. After this bug was fixed, all generated nontermination proofs could be certified. Since the overhead for certification is negligible (AProVE required 151 minutes to generate all proofs, whereas CeTA required 4 seconds to certify them), we encourage termination tool users to always certify their proofs. For more details on the experiments, we refer to `http://cl-informatik.uibk.ac.at/software/ceta/experiments/nonterm/`.

Future work consists of integrating further techniques for which completeness is not obvious into our framework. Examples are innermost narrowing [3] and the switch from innermost termination to termination for TRSs and DPPs.

**Acknowledgments**

We thank Lukas Bulwahn for helpful information on Isabelle/HOL's predicate compiler.

# Bibliography


[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. V. Ramírez-Deantes, G. Román, and D. Zanardini. Termination and cost analysis with COSTA and its user interfaces. *ENTCS*, 258(1):109–121, 2009.

[2] T. Aoto, T. Yoshida, and J. Toyama. Proving confluence of term rewriting systems automatically. In *Proc. of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009)*, volume 5595 of *LNCS*, pages 93–102, 2009.

[3] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

[4] M. Avanzini, G. Moser, and A. Schnabl. Automated implicit computational complexity analysis (system description). In *Proc. of the 4th International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *LNAI*, pages 132–138, 2008.

[5] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[6] G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In *Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, volume 3945 of *LNCS*, pages 114–129, 2006.

[7] A. M. Ben-Amram and M. Codish. A SAT-based approach to size change termination with global ranking functions. In *Proc. of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 218–232, 2008.

[8] A. M. Ben-Amram and C. S. Lee. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst.*, 29(1), 2007.

[9] 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.

[10] S. Berghofer, L. Bulwahn, and F. Haftmann. Turning inductive into equational specifications. In *Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 131–146, 2009.

[11] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions.* TCS Texts. Springer, 2004.

[12] F. Blanqui, W. Delobel, S. Coupet-Grimal, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In *Proc. of the 8th International Workshop on Termination (WST 2006)*, pages 69–73, 2006.

[13] F. Blanqui and A. Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011.

[14] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1980.

[15] M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *Proc. of the 24th International Conference on Computer Aided Verification (CAV 2012)*, volume 7358 of *LNCS*, pages 105–122, 2012.

[16] M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination graphs for Java bytecode. In *Verification, Induction, Termination Analysis - Festschrift for Christoph Walther on the Occasion of His 60th Birthday*, volume 6463 of *LNCS*, pages 17–37, 2010.

[17] W. Buchholz. Proof-theoretic analysis of termination proofs. *Ann. Pure Appl. Logic*, 75(1-2):57–65, 1995.

[18] L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In *Proc. of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *LNCS*, pages 38–53, 2007.

[19] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML documents. In *Proc. of the 10th International Symposium on Database Programming Languages (DBPL 2005)*, volume 3774 of *LNCS*, pages 199–216, 2005.

[20] M. Codish, C. Fuhs, J. Giesl, and P. Schneider-Kamp. Lazy abstraction for size-change termination. In *Proc. of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2010)*, volume 6397 of *LNCS*, pages 217–232, 2010.

[21] E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In *Proc. of the 6th International Symposium on Frontiers of Combining Systems (FroCoS 2007)*, volume 4720 of *LNAI*, pages 148–162, 2007.

[22] E. Contejean, C. Marché, B. Monate, and X. Urbain. CiME. `http://cime.lri.fr`.

[23] É. Contejean, A. Paskevich, X. Urbain, P. Courtieu, O. Pons, and J. Forest. A3PAT, an approach for certified automated termination proofs. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2010)*, pages 63–72, 2010.

[24] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*, pages 415–426, 2006.

[25] P. Courtieu, J. Forest, and X. Urbain. Certifying a termination criterion based on graphs, without graphs. In *Proc. of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 183–198, 2008.

[26] N. Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[27] N. Dershowitz. Termination of rewriting. *J. Symb. Comp.*, 3(1-2):69–116, 1987.

[28] N. Dershowitz. Termination dependencies. In *Proc. of the 6th International Workshop on Termination (WST 2003)*, pages 27–30, 2003.

[29] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Comm. of the ACM*, 22(8):465–476, 1979.

[30] F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In *Proc. of the 6th International Joint Conference on Automated Reasoning (IJCAR 2012)*, volume 7364 of *LNAI*, pages 225–240, 2012.

[31] J. Endrullis. Jambox. Available from `http://joerg.endrullis.de`.

[32] 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.

[33] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal termination. In *Proc. of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *LNCS*, pages 110–125, 2008.

[34] A. L. Galdino and M. Ayala-Rincón. A PVS theory for term rewriting systems. *ENTCS*, 247:67–83, 2009.

[35] A. L. Galdino and M. Ayala-Rincón. A formalization of the Knuth-Bendix(-Huet) critical pair theorem. *Journal of Automated Reasoning*, 45(3):301–325, 2010.

[36] A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. *Information and Computation*, 205(4):512–534, 2007.

[37] J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *Appl. Alg. Eng. Comm. Comput.*, 12(1,2):39–72, 2001.

[38] 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*, 33(2):7:1–7:39, 2011.

[39] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNAI*, pages 281–286, 2006.

[40] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2004)*, volume 3452 of *LNAI*, pages 301–331, 2005.

[41] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. of the 5th International Symposium on Frontiers of Combining Systems (FroCoS 2005)*, volume 3717 of *LNAI*, pages 216–231, 2005.

[42] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[43] M. Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction, Essays In Honour of Robin Milner*, pages 169–186. MIT Press, 2000.

[44] M. Gordon. From LCF to HOL: A short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

[45] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *LNCS*. Springer, 1979.

[46] B. Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 24:3–23, 1995.

[47] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proc. of the 36th ACM Symposium on Principles of Programming Languages (POPL 2009)*, pages 127–139, 2009.

[48] F. Haftmann. *Code generation from Isabelle/HOL theories*, Apr. 2009. `http://isabelle.in.tum.de/doc/codegen.pdf`.

[49] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Proc. of the 10th International Symposium on Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117, 2010.

[50] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.

[51] N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.

[52] N. Hirokawa and A. Middeldorp. Decreasing diagrams and relative termination. *Journal of Automated Reasoning*, 47(4):481–501, 2011.

[53] N. Hirokawa, A. Middeldorp, and H. Zankl. Uncurrying for termination. In *Proc. of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2008)*, volume 5330 of *LNAI*, pages 667–681, 2008.

[54] D. Hofbauer and J. Waldmann. Match-bounds for relative termination. In *Proc. of the 11th International Workshop on Termination (WST 2010)*, 2010.

[55] J. Hoffmann, K. Aehlig, and M. Hofmann. Resource aware ML. In *Proc. of the 24th International Conference on Computer Aided Verification (CAV 2012)*, volume 7358 of *LNCS*, pages 781–786, 2012.

[56] H. Hong and D. Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21(1):23–38, 1998.

[57] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proc. of the 14.th Annual IEEE Symposium on Logic in Computer Science (LICS 1999)*, pages 402–411. IEEE Computer Society Press, 1999.

[58] S. Kamin and J. J. Lévy. Two generalizations of the recursive path ordering. Unpublished manuscript, University of Illinois, 1980.

[59] R. Kennaway, J. W. Klop, R. Sleep, and F.-J. de Vries. Comparing curried and uncurried rewriting. *Journal of Symbolic Computation*, 21(1):15–39, 1996.

[60] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Comm. of the ACM*, 53(6):107–115, 2010.

[61] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[62] A. Koprowski. TPA: Termination proved automatically. In *Proc. of the 17th International Conference on Rewriting Techniques and Applications (RTA 2006)*, volume 4098 of *LNCS*, pages 257–266, 2006.

[63] A. Koprowski. Coq formalization of the higher-order recursive path ordering. *Appl. Alg. Eng. Comm. Comput.*, 20(5-6):379–425, 2009.

[64] A. Koprowski and J. Waldmann. Arctic termination ... below zero. In *Proc. of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *LNCS*, pages 202–216, 2008.

[65] K. Korovin and A. Voronkov. Orienting rewrite rules with the Knuth-Bendix order. *Information and Computation*, 183(2):165–186, 2003.

[66] M. Korp and A. Middeldorp. Match-bounds revisited. *Information and Computation*, 207(11):1259–1283, 2009.

[67] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009)*, volume 5595 of *LNCS*, pages 295–304, 2009.

[68] A. Krauss. Certified size-change termination. In *Proc. of the 21st International Conference on Automated Deduction (CADE 2007)*, volume 4603 of *LNCS*, pages 460–475, 2007.

[69] A. Krauss. Partial and nested recursive function definitions in higher-order logic. *Journal of Automated Reasoning*, 44(4):303–336, 2010.

[70] A. Krauss. Recursive definitions of monadic functions. In *Proc. of the Workshop on Partiality and Recursion in Interactive Theorem Proving (PAR 2010)*, volume 43 of *EPTCS*, pages 1–13, 2010.

[71] M. S. Krishnamoorthy and P. Narendran. On recursive path ordering. *Theor. Comput. Sci.*, 40:323–328, 1985.

[72] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):210–225, 1960.

[73] D. Lankford. On proving term rewrite systems are noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

[74] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages (POPL 2001)*, pages 81–92, 2001.

[75] A. Lochbihler. Verifying a compiler for Java threads. In *Proc. of the 19th European Symposium on Programming (ESOP 2010)*, volume 6012 of *LNCS*, pages 427–447, 2010.

[76] S. Lucas. Polynomials over the reals in proofs of termination: From theory to practice. *RAIRO Theor. Inf. Appl.*, 39(3):547–586, 2005.

[77] M. Marchiori. Logic programs as term rewriting systems. In *Proc. of the 4th International Conference on Algebraic and Logic Programming (ALP 1994)*, volume 850 of *LNCS*, pages 223–241, 1994.

[78] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[79] A. Middeldorp. *Modular Properties of Term Rewriting Systems*. PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[80] M. O. Myreen and J. Davis. A verified runtime for a verified theorem prover. In *Proc. of the 2nd International Conference on Interactive Theorem Proving (ITP 2011)*, volume 6898 of *LNCS*, pages 265–280, 2011.

[81] F. Neurauter, H. Zankl, and A. Middeldorp. Revisiting matrix interpretations for polynomial derivational complexity of term rewriting. In *Proc. of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2010)*, volume 6397 of *LNCS*, pages 550–564, 2010.

[82] M. Nguyen, D. D. Schreye, J. Giesl, and P. Schneider-Kamp. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *Theory and Practice of Logic Programming*, 11(1):33–63, 2011.

[83] T. Nipkow. An inductive proof of the wellfoundedness of the multiset order, 1998. Available at `http://www4.in.tum.de/~nipkow/misc/multiset.ps`.

[84] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[85] E. Ohlebusch. A simple proof of sufficient conditions for the termination of the disjoint union of term rewriting systems. *Bulletin of the EATCS*, 50:223–228, 1993.

[86] E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *Appl. Alg. Eng. Comm. Comput.*, 12(1-2):73–116, 2001.

[87] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java bytecode by term rewriting. In *Proc. of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *LIPIcs*, pages 259–276, 2010.

[88] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proc. of the 11th International Conference on Automated Deduction (CADE 1992)*, volume 607 of *LNAI*, pages 748–752, 1992.

[89] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[90] S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[91] M. Schmidt-Schauß. Polynomial equality testing for terms with shared substructures. Frank report 21, Institut für Informatik. FB Informatik und Mathematik. J.W. Goethe-Universität, Frankfurt am Main, 2005.

[92] P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving termination using recursive path orders and SAT solving. In *Proc. of the 6th International Symposium on Frontiers of Combining Systems (FroCoS 2007)*, volume 4720 of *LNAI*, pages 267–282, 2007.

[93] F. Spoto, F. Mesnard, and É. Payet. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[94] C. Sternagel. *Automatic Certification of Termination Proofs*. PhD thesis, Institut für Informatik, Universität Innsbruck, Austria, 2010.

[95] C. Sternagel and A. Middeldorp. Root-Labeling. In *Proc. of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *LNCS*, pages 336–350, 2008.

[96] C. Sternagel and R. Thiemann. Certified subterm criterion and certified usable rules. In *Proc. of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *LIPIcs*, pages 325–340, 2010.

[97] C. Sternagel and R. Thiemann. Signature extensions preserve termination. In *Proc. of the 19th Annual Conference of the EACSL on Computer Science Logic (CSL 2010)*, volume 6247 of *LNCS*, pages 514–528, 2010.

[98] C. Sternagel and R. Thiemann. Executable Transitive Closures of Finite Relations. In *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Transitive-Closure.shtml`, 2011. Formalization.

[99] T. Sternagel and H. Zankl. KBCV - Knuth-Bendix completion visualizer. In *Proc. of the 6th International Joint Conference on Automated Reasoning (IJCAR 2012)*, volume 7364 of *LNAI*, pages 530–536, 2012.

[100] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[101] R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, RWTH Aachen, 2007. Available as technical report AIB-2007-17.

[102] R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Appl. Alg. Eng. Comm. Comput.*, 16(4):229–270, 2005.

[103] R. Thiemann, J. Giesl, and P. Schneider-Kamp. Deciding innermost loops. In *Proc. of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *LNCS*, pages 366–380, 2008.

[104] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 452–468, 2009.

[105] X. Urbain. Modular & incremental automated termination proofs. *Journal of Automated Reasoning*, 32(4):315–355, 2004.

[106] J. Waldmann. Matchbox: A tool for match-bounded string rewriting. In *Proc. of the 15th International Conference on Rewriting Techniques and Applications (RTA 2004)*, volume 3091 of *LNCS*, pages 85–94, 2004.

[107] S. Winkler, H. Sato, A. Middeldorp, and M. Kurihara. Optimizing mkbTT (system description). In *Proc. of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *LIPIcs*, pages 373–384, 2010.

[108] H. Zankl, B. Felgenhauer, and A. Middeldorp. CSI – A confluence tool. In *Proc. of the 23rd International Conference on Automated Deduction (CADE 2011)*, LNAI, 2011.

[109] H. Zankl, C. Sternagel, D. Hofbauer, and A. Middeldorp. Finding and certifying loops. In *Proc. of the 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *LNCS*, pages 755–766, 2009.

[110] H. Zantema. Termination of term rewriting by semantic labelling. *Fund. Inform.*, 24(1-2):89–105, 1995.

[111] H. Zantema. Termination of string rewriting proved automatically. *Journal of Automated Reasoning*, 34(2):105–139, 2005.