Certification of Nontermination Proofs⋆

Christian Sternagel[1] and René Thiemann[2]

[1] Japan Advanced Institute of Science and Technology, Japan
[2] Institute of Computer Science, University of Innsbruck, Austria

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 [1]. 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.

Keywords: nontermination, formalization, interactive theorem proving, term rewriting

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).

⋆ This research is supported by the Austrian Science Fund (FWF): J3202, P22767.

$$x == y \to \mathsf{chk}(\mathsf{eq}(x,y)) \quad (1) \qquad \mathsf{chk}(\mathsf{true}) \to \mathsf{true} \quad (3)$$
$$\mathsf{eq}(x,x) \to \mathsf{true} \quad (2) \qquad \mathsf{chk}(\mathsf{eq}(x,y)) \to \mathsf{false} \quad (4)$$

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 [1] 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 [2,3]. 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 [1] by a single line. Moreover, we report on how we managed to obtain efficient versions of other algorithms from [1] within Isabelle/HOL [4].

The remainder is structured as follows. In Sect. 2 we give preliminaries. Then, in Sect. 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. 4. The main part of this paper is on our formalization of the decision procedure for innermost loops in Sect. 5, before we conclude in Sect. 6.

2 Preliminaries

We assume basic familiarity with term rewriting [5]. Nevertheless, we shortly recapitulate what is used later on. A *term* t (ℓ, 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 $\mathit{root}(x) = x$ and $\mathit{root}(f(t_1,\ldots,t_n)) = f$. The set $\mathit{args}(t)$ of *arguments* of t is defined by the equations $\mathit{args}(x) = \emptyset$ and $\mathit{args}(f(t_1,\ldots,t_n)) = \{t_1,\ldots,t_n\}$. 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* σ (μ) is a mapping from variables to terms whose *domain* $\mathit{dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. The *range* of a substitution is $\mathit{ran}(\sigma) = \{\sigma(x) \mid x \in \mathit{dom}(\sigma)\}$. We represent concrete substitutions using the notation $\{x_1/t_1,\ldots,x_n/t_n\}$. We use σ interchangeably with its homomorphic extension to terms, writing, e.g., $t\sigma$ to denote the application of the substitution σ 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 σ 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 $s \overset{\mathsf{i}}{\to}_{\mathcal{R}} t$. 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

$$t = t_1 \overset{(\mathsf{i})}{\to}_{\mathcal{R}} t_2 \overset{(\mathsf{i})}{\to}_{\mathcal{R}} t_3 \overset{(\mathsf{i})}{\to}_{\mathcal{R}} \cdots$$

A TRS $\mathcal{R}$ is (innermost) nonterminating iff there is a term t that is (innermost) nonterminating w.r.t. $\mathcal{R}$.

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 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. 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 [6,7].

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., [8] and [9] 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) [10]. 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.

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 $t \xrightarrow{\mathsf{i}}_{\mathcal{R}}^{+} C[t\mu]$, since this does not necessarily imply an infinite sequence (5) when restricting to innermost rewriting. Therefore, in [1], 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 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 μ is applied, all steps should be innermost.

Lemma 3. *A loop (6) is an innermost loop iff (5) is an innermost derivation.*

Corollary 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 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 σ.*

Hence, we can easily check, whether a loop is innermost, whenever for two terms t and ℓ, and a substitution μ, we can solve the problem whether there exist n and σ, such that $t\mu^n$ contains a redex $\ell\sigma$. Such problems are called *redex problems* and a large part of [1] is devoted to develop a corresponding decision procedure.

Example 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 μ be an arbitrary but fixed substitution (usually originating from some loop $t \to^+_{\mathcal{R}} C[t\mu]$).

Definition 7 (Redex, Matching, and Identity Problems). *Let s, t, and ℓ be terms. Then a* redex problem *is a pair $t \mathrel{|\!\!\gtrdot} \ell$, 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 $t \mathrel{|\!\!\gtrdot} \ell$ is solvable *iff there is a context C, a substitution σ, 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 σ 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,σ,n), (σ,n), and n, the* solution.

5 Formalization

In [11] 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 [1].

From Redex Problems to Matching Problems. A redex problem $t \mathrel{|\!\!\gtrdot} \ell$ 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., ℓ 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 μ^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 $u \gtrdot \ell$. In total, the redex problem is solvable iff one of the matching problems $u \gtrdot \ell$ 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 [1, Theorem 10].

Theorem 8. *Let $t \mid\!\gtrdot \ell$ be a redex problem. Let*

$$\mathcal{M}_{init}(t, \ell) = \textsf{if } \ell \in \mathcal{V} \textsf{ then } \{t \gtrdot \ell\} \textsf{ else } \{u \gtrdot \ell \mid u \in \mathcal{U}(t)\}$$

be the set of initial matching problems. Then $t \mid\!\gtrdot \ell$ is solvable iff one of the matching problems in $\mathcal{M}_{init}(t, \ell)$ is solvable.

Example 9. Continuing with Example 6, from each redex problem $\mathsf{eq}(x, y) \mid\!\gtrdot \ell$ we obtain the matching problems $\mathsf{eq}(x, y) \gtrdot \ell$, $\mathsf{cons}(z, y) \gtrdot \ell$, $\mathsf{cons}(z, x) \gtrdot \ell$, and $\mathsf{0} \gtrdot \ell$ where ℓ is an arbitrary left-hand side of the TRS.

Theorem 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 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 10 can easily be computed since whenever $(x, y) \in R$ then $x \in \mathit{dom}(\mu)$. 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 [12] (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 $x \in \mathcal{W}(t) \implies \mathcal{V}(x\mu) \subseteq \mathcal{W}(t)$. 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 [12]. 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 [13]. 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 $\{t_1 \gtrdot \ell_1, \dots, t_k \gtrdot \ell_k\}$, in [1], 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 ℓ_i are variables (or $\bot$ is obtained which represents a matching problem without solution).

Definition 11 (Transformation of Matching Problems). *In [1] 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*

1. $\mathcal{M} \Rightarrow \{t_1 \gtrdot \ell_1, \dots, t_k \gtrdot \ell_k\} \cup \mathcal{M}'$*, if* $t = f(t_1, \dots, t_k)$ *and* $\ell = f(\ell_1, \dots, \ell_k)$
2. $\mathcal{M} \Rightarrow \bot$*, if* $t = f(\dots)$*,* $\ell = g(\dots)$*, and* $f \neq g$
3. $\mathcal{M} \Rightarrow \bot$*, if* $t \in \mathcal{V} \setminus \mathcal{V}_{\mathsf{incr}}$
4. $\mathcal{M} \Rightarrow \{t'\mu \gtrdot \ell' \mid t' \gtrdot \ell' \in \mathcal{M}\}$*, 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 μ, 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 [1]). 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 11 of [1], our algorithm also returns an integer i which provides a lower bound on how often μ 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):

$$
\begin{array}{ll}
\textit{simplify-mp}\ [] & s = \textsf{return}\ (s, 0) \\
\textit{simplify-mp}\ ((t, x) \cdot \textit{mp}) & s = \textit{simplify-mp}\ \textit{mp}\ ((t, x) \cdot s) \\
\textit{simplify-mp}\ ((f(\textit{ss}), g(\textit{ts})) \cdot \textit{mp}) & s = \textsf{do}\ \{\ \textit{guard}\ (f = g); \textit{ps} \leftarrow \textit{zip-option}\ \textit{ss}\ \textit{ts}; \\
& \qquad\qquad \textit{simplify-mp}\ (\textit{ps}@\textit{mp})\ s\ \}
\end{array}
$$

```
simplify-mp ((x, g(ts)) · mp) s = do { guard (x ∈ V_incr);
                                         (mp', i) ← simplify-mp
                                          (map-μ ((x, g(ts)) · mp)) (map-μ s);
                                         return (mp', i + 1) }
```

where, $\mathit{map}\text{-}\mu = \mathit{map}\ (\lambda(t,\ell).(t\mu,\ell))$ 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 12. For $\ell = \mathsf{eq}(x,x)$, only one of the redex problems of Example 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. *The function simplify-mp satisfies the following properties:*

– *It is terminating.*
– *It is complete, i.e., if* (n,σ) *is a solution for mp then 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,σ) *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 $\mathit{mp} = [(t_1,\ell_1),\ldots,(t_k,\ell_k)]$, we use the lexicographic combination of the following two measures: first, we measure the sum of the sizes of the ℓ_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 $r = \mathit{None}$ then $\mathit{mp} \cup s$ is not solvable,
– if $r = \mathit{Some}\ (\mathit{mp}', i)$, there is no solution (n,σ) for $\mathit{mp} \cup s$ where $n < i$, and (n,σ) is a solution for mp' iff $(n+i,\sigma)$ is a solution for $\mathit{mp} \cup s$.

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 [12] to compute it.

Lemma 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. 4 we just fixed some substitution μ (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 μ. Hence, we have to pass μ as argument to both. As a result, the modified version of the last equation of *simplify-mp* looks as follows:

$$\begin{aligned} &\mathit{simplify\text{-}mp}\ \mu\ ((x, g(ts)) \cdot \mathit{mp})\ s = \mathsf{do}\ \{\mathit{guard}\ (x \in \mathcal{V}_{\mathsf{incr}}(\mu));\\ &\quad (\mathit{mp}', i) \leftarrow \mathit{simplify\text{-}mp}\ \mu\ (\mathit{map\text{-}\mu}\ ((x, g(ts)) \cdot \mathit{mp}))\ (\mathit{map\text{-}\mu}\ s);\\ &\quad \mathsf{return}\ (\mathit{mp}', i + 1)\ \} \end{aligned} \tag{7}$$

The problem of equation (7) is its inefficiency: In every recursive call, the set of increasing variables $\mathcal{V}_{\mathsf{incr}}(\mu)$ is newly computed. Therefore, the obvious idea is to compute $\mathcal{V}_{\mathsf{incr}}(\mu)$ once and for all and pass it as an additional argument V.

$$\begin{aligned} &\mathit{simplify\text{-}mp}\ \mu\ V\ ((x, g(ts)) \cdot \mathit{mp})\ s = \mathsf{do}\ \{\mathit{guard}\ (x \in V);\\ &\quad (\mathit{mp}', i) \leftarrow \mathit{simplify\text{-}mp}\ \mu\ V\ (\mathit{map\text{-}\mu}\ ((x, g(ts)) \cdot \mathit{mp}))\ (\mathit{map\text{-}\mu}\ s);\\ &\quad \mathsf{return}\ (\mathit{mp}', i + 1)\ \} \end{aligned} \tag{8}$$

This version does not have the problem of recomputing $\mathcal{V}_{\mathsf{incr}}(\mu)$ and we just have to replace the initial call *simplify-mp* μ *mp* [] by *simplify-mp* μ $\mathcal{V}_{\mathsf{incr}}(\mu)$ *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 $V \neq \mathcal{V}_{\mathsf{incr}}(\mu)$, which can cause nontermination. Take for example μ as the empty substitution and $V = \{x\}$, then the function call *simplify-mp* μ 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, Isabelle/HOL's function package [14] weakens equality (8) by the assumption that *simplify-mp* has to be terminating on the arguments μ, V, $((x, g(ts) \cdot \mathit{mp})$, and s.

Of course, we can instantiate (8) by $V = \mathcal{V}_{\mathsf{incr}}(\mu)$. Then we can get rid of the additional assumption. But still, the corresponding unconditional equation is not suitable for code generation, since $\mathcal{V}_{\mathsf{incr}}$ on the left-hand side is not a constructor.

Our final solution is to use the recent *partial-function* [15] 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 $\mathcal{V}_{\mathsf{incr}}(\mu)$. 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 $V = \mathcal{V}_{\mathsf{incr}}(\mu)$. 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 $\mathcal{M} = \{t_1 \gtrdot x_1, \dots, t_k \gtrdot x_k\}$. In principle, $\mathcal{M}$ has the solution (n, σ) where n is arbitrary and $\sigma(x_i) = t_i\mu^n$. However, this definition of σ 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 $t_i \approxeq t_j$.

The following result of [1, Theorem 14 (iv)] is easily formalized and also poses no challenges for certification. Afterwards it remains to decide identity problems.

Theorem 15. *Let $\mathcal{M} = \{t_1 \gtrdot x_1, \dots, t_k \gtrdot x_k\}$ be a generalized matching problem in solved form. Define $\mathcal{I}_{init} = \{t_i \approxeq t_j \mid 1 \leqslant i < j \leqslant k, x_i = x_j\}$. 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 $t_i \approxeq t_j$, respectively, then the maximum n of all n_{ij} is a solution to all identity problems $t_i \approxeq t_j$. And then also (n, σ) is a solution to $\mathcal{M}$ where $\sigma(x_i) = t_i\mu^n$ is guaranteed to be well-defined.

Example 16. For the remaining matching problem of Example 12 we generate one identity problem: $x \approxeq y$.

Deciding Identity Problems. In [1, Section 3.4] a complicated algorithm is presented to decide solvability of an identity problem $s \approxeq t$. 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 μ 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 [1] 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 $s \approxeq t$ considerably since we only store conflicts whose constituting terms are in the set of *conflict terms*

$$\mathcal{CT}(s,t) = \{u \mid v \trianglerighteq u, v \in \{s,t\} \cup \mathit{ran}(\mu)\}.$$

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 [1], our procedure does neither require any preprocessing on μ 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.

$$\begin{array}{ll}
\mathit{conflicts}\ (s, y, n+1) & = \mathit{conflicts}\ (s, \mu(y), n) \\
\mathit{conflicts}\ (x, y, 0) & = \textsf{\textit{if}}\ x = y\ \textsf{\textit{then}}\ \emptyset\ \textsf{\textit{else}}\ \{(x, y, 0)\} \\
\mathit{conflicts}\ (f(ss), y, 0) & = \{(y, f(ss), 0)\} \\
\mathit{conflicts}\ (x, g(ts), n) & = \{(x, g(ts), n)\} \\
\mathit{conflicts}\ (f(ss), g(ts), n) & = \textsf{\textit{if}}\ f = g \wedge |ss| = |ts| \\
& \quad \textsf{\textit{then}} \bigcup_{(s_i,t_i) \in \mathit{zip}\ ss\ ts} \mathit{conflicts}\ (s_i, t_i, n) \\
& \quad \textsf{\textit{else}}\ \{(f(ss), g(ts), n)\}
\end{array}$$

We identified and formalized the following properties of *conflicts* and $\mathcal{CT}$.

Lemma 17. – *$s\sigma = t\mu^n\sigma$ iff $\forall(u,v,m) \in$ conflicts $(s,t,n).\, u\sigma = v\mu^m\sigma$.*

- *if $(u,v,m) \in$ conflicts (s,t,n) then*
  - *$root(u) \neq root(v)$*
  - *$v \in \mathcal{V}$ 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* $\emptyset$ $(s,t,0)$.

```
$ident\text{-}solve\ S\ idp =$
  let $\mathcal{C} = conflicts\ idp$ in
  if $(f(us), \_, \_) \in \mathcal{C} \ \vee ((u,v,\_) \in \mathcal{C} \wedge (u,v,\_) \in S)$ then $None$ else do {
    $ns \leftarrow map\text{-}option\ (\lambda(u,v,m).\ ident\text{-}solve\ (\{(u,v,m)\} \cup S)\ (u\mu, v, m+1))\ \mathcal{C}$;
    return $(max\ \{n+1 \mid n \in ns\})$ }
```

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 18. We continue Example 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 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 recursive 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 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 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* we know that 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 17 (where $\sigma = \mu^i$). □

Lemma 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 $(a, b) = (c, d) \vee (a, b) = (d, c)$.[3]

$$\begin{aligned}
&(s\mu^n|_p, t\mu^n|_p) \overset{\leftrightarrow}{=} (s', t'\mu^{n'}) && (9)\\
&\longrightarrow (\forall (u, v, m) \in S.\, (m = 0 \vee v \notin \mathcal{V}) \wedge \mathit{root}(u) \neq \mathit{root}(v) \wedge && (10)\\
&\qquad\qquad (\exists q_1\, q_2\, n_1.\, p = q_1 q_2 \wedge n_1 < n \wedge (s\mu^{n_1}|_{q_1}, t\mu^{n_1}|_{q_1}) \overset{\leftrightarrow}{=} (u, v\mu^m)))\\
&\longrightarrow \textit{ident-solve}\ S\ (s', t', n') \neq \textit{None} && (11)
\end{aligned}$$

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 17 this shows $u\mu^N = v\mu^m\mu^N$ for all $(u, v, m) \in \mathit{conflicts}\ (s', t', n') =: \mathcal{C}$. 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 17 we have $\mathit{root}(u) \neq \mathit{root}(v)$, $m = 0 \vee v \notin \mathcal{V}$, 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 $u\mu^N = v\mu^m\mu^N$.

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 17 which implies the contradiction $\mathit{root}(u) = \mathit{root}(u\mu^N) = \mathit{root}(v\mu^m\mu^N) = \mathit{root}(v) \neq \mathit{root}(u)$.

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.

$$\begin{aligned}
(s\mu^n|_{pq_1}, t\mu^n|_{pq_1}) &= (s\mu^n|_p|_{q_1}, t\mu^n|_p|_{q_1}) \overset{\leftrightarrow}{=} (s'|_{q_1}, t'\mu^{n'}|_{q_1})\\
&= (s'|_{q_1}, t'\mu^k|_{q_1}\mu^m) \quad \overset{\leftrightarrow}{=} (u, v\mu^m).
\end{aligned} \tag{12}$$

Using (12), $\mathit{root}(u) \neq \mathit{root}(v)$, 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 μ 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 $p = p_1 q_3$,

[3] In the formalization, $(\star)$ looks even more complicated, since here we dropped all parts that restrict p and q_1 to valid positions.

$n_2 < n$, 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

$$\begin{aligned} &(u, v\mu^m) && \overset{\leftrightarrow}{=} (s\mu^n|_{pq_1}, t\mu^n|_{pq_1}) \\ =\ &(s\mu^{n_2+k_1}|_{p_1q_3q_1}, t\mu^{n_2+k_1}|_{p_1q_3q_1}) && = (s\mu^{n_2}|_{p_1}\sigma|_q, t\mu^{n_2}|_{p_1}\sigma|_q) \\ \overset{\leftrightarrow}{=}\ &(u\sigma|_q, v\mu^{m'}\sigma|_q) \end{aligned} \tag{13}$$

where σ and q are abbreviations for μ^{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 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 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 21. From the termination proof in Lemma 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 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 μ. Hence, the value of n in Theorem 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.

6 Conclusions

We have formalized several techniques to certify compositional (innermost) nontermination proofs, where the hardest part was the decision procedure of [1], 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 22.

With this result we can also show (but have not formalized) that all considered decision problems of this paper are in P.

Theorem 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 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 [16,17] to check equality in polynomial time. Note that even if the input (s,t,μ) 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 $t \gtrdot \ell$, 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 8 it follows that redex problems $t \mathrel{|\!\gtrdot} \ell$ 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, ℓ, and μ.

Finally, since redex problems can be decided in P, by Lemma 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 [18], 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 [10] 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.

References

1. Thiemann, R., Giesl, J., Schneider-Kamp, P.: Deciding innermost loops. In: RTA 2008. Volume 5117 of LNCS., Springer (2008) 366–380 doi:10.1007/978-3-540-70590-1_25.
2. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS 2010. Volume 6009 of LNCS., Springer (2010) 103–117 doi:10.1007/978-3-642-12251-4_9.
3. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: TPHOLs 2009. Volume 5674 of LNCS., Springer (2009) 452–468 doi:10.1007/978-3-642-03359-9_31.
4. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002) doi:10.1007/3-540-45949-9.
5. Baader, F., Nipkow, T.: Term Rewriting and *All That*. Paperback edn. Cambridge University Press, New York, USA (August 1999) doi:10.2277/0521779200.
6. Ben Cherifa, A., Lescanne, P.: Termination of rewriting systems by polynomial interpretations and its implementation. Sci. Comput. Program. **9**(2) (1987) 137–159 doi:10.1016/0167-6423(87)90030-X.
7. Lankford, D.S.: On proving term rewriting systems are Noetherian. Memo MTP-3, Louisiana Technical University, Ruston, LA, USA (May 1979)
8. Zantema, H.: Termination of string rewriting proved automatically. J. Autom. Reasoning **34**(2) (2005) 105–139 doi:10.1007/s10817-005-6545-0.
9. Sternagel, C., Thiemann, R.: Signature extensions preserve termination - an alternative proof via dependency pairs. In: CSL 2010. Volume 6247 of LNCS., Springer (2010) 514–528 doi:10.1007/978-3-642-15205-4_39.
10. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theor. Comput. Sci. **236**(1-2) (2000) 133–178 doi:10.1016/S0304-3975(99)00207-8.
11. Zankl, H., Sternagel, C., Hofbauer, D., Middeldorp, A.: Finding and certifying loops. In: SOFSEM 2010. Volume 5901 of LNAI., Springer (2010) 755–766 doi:10.1007/978-3-642-11266-9_63.
12. Sternagel, C., Thiemann, R.: Executable Transitive Closures of Finite Relations. In: The Archive of Formal Proofs. http://afp.sf.net/entries/Transitive-Closure.shtml (March 2011) Formalization.
13. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In: TPHOLs 2009. Volume 5674 of LNCS., Springer (2009) 131–146 doi:10.1007/978-3-642-03359-9_11.
14. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reasoning **44**(4) (2010) 303–336 doi:10.1007/s10817-009-9157-2.
15. Krauss, A.: Recursive definitions of monadic functions. In: PAR 2010. Volume 43 of EPTCS. (2010) 1–13 doi:10.4204/EPTCS.43.1.
16. Busatto, G., Lohrey, M., Maneth, S.: Efficient memory representation of XML documents. In: DBPL 2005. Volume 3774 of LNCS., Springer (2005) 199–216 doi:10.1007/11601524_13.
17. Schmidt-Schauß, M.: 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)
18. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: IJCAR 2006. Volume 4130 of LNAI., Springer (2006) 281–286 doi:10.1007/11814771_24.