Finding and Certifying Loops*

Harald Zankl,[1] Christian Sternagel,[1] Dieter Hofbauer,[2] and Aart Middeldorp[1]

[1] University of Innsbruck, Innsbruck, Austria
[2] Berufsakademie Nordhessen, Bad Wildungen, Germany

Abstract. The first part of this paper presents a new approach for automatically proving nontermination of string rewrite systems. We encode rewrite sequences as propositional formulas such that a loop can be extracted from a satisfying assignment. Alternatively, loops can be found by enumerating forward closures. In the second part we give a formalization of loops in the theorem prover Isabelle/HOL. Afterwards, we use Isabelle's code-generation facilities to certify loops. The integration of our approach in CeTA (a program for automatic certification of termination proofs) makes it the first tool capable of certifying nontermination.

Key words: Rewriting, Nontermination, Certification

Introduction

Proving termination of term rewrite systems is a challenging endeavor since it is undecidable in general. Nonetheless, there are many powerful algorithms that are able to automatically prove termination of a huge class of TRSs as witnessed by the international termination competition.[3] This event (where several termination tools compete on a large set of problems) evolved in 2004. Since then, it stimulated research and focused the efforts towards the automation of termination analysis. Surprisingly—compared to the vast amount of methods devoted to termination—only few techniques concerning nontermination are known and implemented. Nevertheless, checking for nontermination is especially useful for debugging programs, since concrete counterexamples are helpful for tracking down bugs. Most nontrivial approaches in that direction aim to find *looping* reductions and comprise ancestor graphs [28], narrowing [14], match-bounds [11, 25], unfoldings [22], and transport systems [26]. The first automated approach [21] dealing with nonlooping nonterminating systems was presented during the 2008 edition of the termination competition.

The increasing complexity of proofs generated by termination tools makes certification of their output more and more important. Since 2007 a *certified* category is part of the termination competition. The participating tools have to generate proofs that can automatically be certified. Recent approaches for automatic certification of termination proofs are Coccinelle/C*i*ME [6], CoLoR/Rainbow [4],

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

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

and IsaFoR/CeTA [24]. The first two use Coq [3] as theorem prover. Coccinelle and CoLoR are Coq-libraries on rewriting whereas C*i*ME and Rainbow transform proof output of a termination tool into Coq-script using the respective library. Afterwards, Coq is used to certify the result. CeTA uses Isabelle/HOL (Isabelle for short) as theorem prover. IsaFoR (*Isa*belle *F*ormalization *o*f *R*ewriting) is an Isabelle-library on rewriting and CeTA (*Ce*rtified *T*ermination *A*nalysis) is a program that certifies a termination proof directly (without calling a theorem prover). It is generated from IsaFoR using Isabelle's code-generation facilities [16].

In Section 1 we recall term rewriting and in Section 2 we present two powerful methods to find loops for string rewriting. Afterwards, Sections 3 and 4 discuss our Isabelle formalization for IsaFoR and our check-functions for CeTA that are used to certify looping nontermination. An assessment of our contributions can be found in Section 5 before ideas for future work are addressed in Section 6.

Parts of Section 2 were first announced in two separate notes by the authors presented at the 9th and 10th International Workshop on Termination.

1 Preliminaries

We assume familiarity with term rewriting [2] in general and termination [29] in particular. A *signature* $\mathcal{F}$ is a set of function symbols with fixed arities. Let $\mathcal{V}$ denote an infinite set of variables disjoint from $\mathcal{F}$. Then $\mathcal{T}(\mathcal{F},\mathcal{V})$ forms the set of terms over the signature $\mathcal{F}$ using variables from $\mathcal{V}$. For a term $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ the *size* and *number of function symbols* of t is denoted by $|t|$ and $\|t\|$, respectively. If $t = f(\dots)$ then f is called the *root* of t. *Rewrite rules* are pairs of terms (l, r), usually written as $l \to r$, where l is not a variable and all variables of r appear in l. Function symbols that appear as roots of left-hand sides are called *defined*. A *term rewrite system* (TRS) is a finite set of rewrite rules. *Contexts* are terms over the signature $\mathcal{F} \cup \{\Box\}$ with exactly one occurrence of the fresh constant $\Box$ (called *hole*). The expression $C[t]$ denotes the result of replacing the hole in C by the term t. A *substitution* σ is a mapping from variables to terms and $t\sigma$ denotes the result of replacing the variables in t according to σ. Substitutions change only finitely many variables (thus written as $\{x_1/t_1, \dots, x_n/t_n\}$). The *rewrite relation* induced by a TRS $\mathcal{R}$ is a binary relation on terms denoted by $\to_\mathcal{R}$ with $s \to_\mathcal{R} t$ if and only if there exist a rewrite rule $l \to r \in \mathcal{R}$, a context C, and a substitution σ such that $s = C[l\sigma]$ and $t = C[r\sigma]$. The (reflexive and) transitive closure of $\to_\mathcal{R}$ is denoted by $(\to_\mathcal{R}^*)$ $\to_\mathcal{R}^+$. A TRS $\mathcal{R}$ is called *strongly normalizing* (SN) or *terminating* if $\to_\mathcal{R}^+$ is well-founded. A sequence $t_1 \to_\mathcal{R} \cdots \to_\mathcal{R} t_n \to_\mathcal{R} C[t_1\sigma]$ is called a *loop* of length n, written $([t_1, \dots, t_n], C, \sigma)$.

Next we recall the dependency pair framework [1,13,15,17]. The signature $\mathcal{F}$ is extended with *dependency pair symbols* $f^\sharp$ for every defined f, where $f^\sharp$ has the same arity as f, resulting in the signature $\mathcal{F}^\sharp$. In examples we write F for $f^\sharp$. If $l \to r \in \mathcal{R}$ and u is a subterm of r with defined root then the rule $l^\sharp \to u^\sharp$ is a *dependency pair* of $\mathcal{R}$. Here $l^\sharp$ and $u^\sharp$ are the results of replacing the roots of l and u by the corresponding dependency pair symbols. The set of dependency pairs of $\mathcal{R}$ is denoted by $\mathtt{DP}(\mathcal{R})$.

A *DP problem* is a pair of TRSs $(\mathcal{P}, \mathcal{R})$ such that the roots of the rules in $\mathcal{P}$ do neither occur in $\mathcal{R}$ nor in proper subterms of the left- and right-hand sides of rules in $\mathcal{P}$. The problem is said to be *finite* if there is no infinite sequence of the shape $s_1 \to_{\mathcal{P}} t_1 \to^*_{\mathcal{R}} s_2 \to_{\mathcal{P}} t_2 \to^*_{\mathcal{R}} \cdots$. The main theorem underlying the dependency pair approach states that termination of a TRS $\mathcal{R}$ is equivalent to finiteness of the *initial* DP problem $(\mathtt{DP}(\mathcal{R}), \mathcal{R})$.

To prove finiteness of a DP problem, *DP processors* are employed. A DP processor is a function taking a DP problem as input while returning a set of DP problems or "no" as output. For proving termination they need to be *sound*, i.e, if all DP problems returned by a DP processor are finite then so is the original one. To ensure that a DP processor can be used to prove nontermination it must be *complete*, i.e., if one of the DP problems returned by the DP processor is not finite then the original DP problem is not finite. Hence if only complete DP processors are used and one returns "no" then the original TRS is nonterminating.

2 Finding Loops

A *string rewrite system* (SRS) is a TRS with unary function symbols only. Instead of $\mathsf{a}(\mathsf{b}(\mathsf{c}(x)))$ we write abc (i.e., the variable is implicit). For a string s we denote the i-th symbol $(1 \leqslant i \leqslant \|s\|)$ in s by s_i, e.g., $\mathsf{abc}_2 = \mathsf{b}$.

Example 1. Consider the SRS $\mathcal{S} = \{\mathsf{ab} \to \mathsf{bbaa}\}$ which admits the looping reduction $\mathsf{abb} \to_{\mathcal{S}} \mathsf{bbaab} \to_{\mathcal{S}} \mathsf{bbabbaa}$ where the initial string abb is reached again after two rewrite steps wrapped in the context $C = \mathsf{bb}\Box$ and instantiated by the substitution $\sigma = \{x/\mathsf{aa}\}$. Thus $\mathcal{S}$ admits the loop $([\mathsf{abb}, \mathsf{bbaab}], C, \sigma)$ of length 2.

The main benefit of the dependency pair approach (for finding loops) is that leading contexts as in Example 1 are automatically removed by construction of the dependency pairs [14], and as a result a looping reduction in a DP problem $(\mathcal{P}, \mathcal{R})$ takes the form $t \to^+_{\mathcal{P} \cup \mathcal{R}} t\sigma$. Our idea is to encode a looping rewrite sequence within the DP framework in SAT using a matrix of dimension $m \times n$ where $(0, 0)$ denotes the top left entry and $(m-1, n-1)$ the bottom right one. Every row in the matrix corresponds to a string and the intended meaning is that there is a rewrite step from row i to row $i+1$. Nowadays many termination tools interface SAT solvers which makes our contribution little effort to implement.

Example 2. The SRS from Example 1 admits the dependency pairs $\mathsf{Ab} \to \mathsf{Aa}$ and $\mathsf{Ab} \to \mathsf{A}$. In a 3×5 matrix a looping reduction is possible. The entries marked with $\cdot$ indicate that any symbol might appear at these positions.

$$\begin{array}{ccccc} \mathsf{A} & \mathsf{b} & \mathsf{b} & \cdot & \cdot \\ \mathsf{A} & \mathsf{a} & \mathsf{b} & \cdot & \cdot \\ \mathsf{A} & \mathsf{b} & \mathsf{b} & \mathsf{a} & \mathsf{a} \end{array}$$

In the sequel we describe how to represent such matrices for a DP problem $(\mathcal{P}, \mathcal{R})$ in propositional logic where the following variables are used:[4]

[4] The idea of encoding computation as propositional satisfiability goes back to [7]. Encoding cyclic structures in SAT originates from liveness properties in bounded model checking [5].

M_{ij}^a	symbol a occurs at position (i,j) of the matrix
$R_i^{l\to r}$	rule $l \to r$ is applied in row i of the matrix
$\mathbf{p}_i$	the position (= column) in row i where the rule is applied (in Example 2 we have $\mathbf{p}_0 = 0$ and $\mathbf{p}_1 = 1$)
$\mathbf{e}_i$	pointing to the last symbol of the i-th string (in the example $\mathbf{e}_0 = 2$, $\mathbf{e}_1 = 2$, and $\mathbf{e}_2 = 4$)

The variables $\mathbf{p}_i$ and $\mathbf{e}_i$ are not Boolean but represent natural numbers which are implemented as lists of Boolean variables encoding the actual value in binary. To distinguish them from proper propositional variables they are typeset in boldface. Furthermore, operations such as $>, \geqslant, =$, and $+$ are defined as usual for such SAT encodings (see, e.g., [10]).

Exactly one function symbol: To get exactly one function symbol at each matrix position, we ensure at least one symbol per entry and additionally ban multiple symbols at the same entry. Note that dependency pair symbols (those in $\mathcal{F}^\sharp \setminus \mathcal{F}$) can only appear at column 0 of each row. This is encoded as follows (where $X = \mathcal{F}$ if $j > 0$ and $X = \mathcal{F}^\sharp \setminus \mathcal{F}$ otherwise):

$$\alpha_{ij} = \Big(\bigvee_{a\in X} M_{ij}^a\Big) \wedge \bigwedge_{a\in X} \big(M_{ij}^a \to \bigwedge_{b\in X\setminus\{a\}} \neg M_{ij}^b\big)$$

Rule application: If a rule $l \to r$ applies in row i ($R_i^{l\to r}$), the rule must be applied correctly ($\mathsf{app}_i^{l\to r}$) and entries unaffected by the rule application must be copied from row i to row $i+1$ ($\mathsf{cp}_{ij}^{l\to r}$). The position of the rule application is fixed by $\mathbf{p}_i$ and satisfying $\mathsf{cp}_{ij}^{l\to r}$ ensures that only one rule is applied. Hence

$$\beta_i^{l\to r} = R_i^{l\to r} \to \big(\mathsf{app}_i^{l\to r} \wedge \bigwedge_{0\leqslant j<n} \mathsf{cp}_{ij}^{l\to r}\big)$$

where in case of $l \to r \in \mathcal{R}$ we set $\mathsf{app}_i^{l\to r}$ to

$$\bigwedge_{0\leqslant j<\|l\|} M_{i(\mathbf{p}_i+j)}^{l_{j+1}} \wedge \bigwedge_{0\leqslant j<\|r\|} M_{(i+1)(\mathbf{p}_i+j)}^{r_{j+1}} \wedge (\mathbf{e}_{i+1} + \|l\| = \mathbf{e}_i + \|r\|) \wedge (\mathbf{e}_i \geqslant \mathbf{p}_i + \|l\|)$$

and if $l \to r \in \mathcal{P}$ then $\mathbf{p}_i$ specializes to 0. The first (second) conjunct of $\mathsf{app}_i^{l\to r}$ expresses that l (r) matches the string encoded in the matrix at row i ($i+1$) at the abstract position $\mathbf{p}_i$. The last but one conjunct demands that the end pointer in line $i+1$ takes the value of $\mathbf{e}_i - \|l\| + \|r\|$. To ensure that the contracted redex fits the string in line i the last conjunct must be satisfied.

The formula for $\mathsf{cp}_{ij}^{l\to r}$ is defined as $\top$ if $j + \max\{\|l\|, \|r\|\} \geqslant n$ (these entries would be outside of the matrix), as

$$\big((j < \mathbf{p}_i) \wedge \bigwedge_{a\in X} (M_{ij}^a \leftrightarrow M_{(i+1)j}^a)\big) \vee \big((j \geqslant \mathbf{p}_i) \wedge \bigwedge_{a\in\mathcal{F}} (M_{i(j+\|l\|)}^a \leftrightarrow M_{(i+1)(j+\|r\|)}^a)\big)$$

(where $X = \mathcal{F}^\sharp \setminus \mathcal{F}$ if $j = 0$ and $X = \mathcal{F}$ otherwise) if $l \to r \in \mathcal{R}$, and if $l \to r \in \mathcal{P}$ then the encoding specializes to the second disjunct (since we know that $\mathbf{p}_i = 0$).

All entries in the matrix before the position where the rule is applied are copied from row i to $i+1$. The second disjunct copies the entries after $\mathbf{p}_i$ which are unaffected when applying the rule. The positions of these entries change if the applied rule is not length-preserving.

Initial string is reached again: To recognize a loop, the string in some row $i > 0$ has to match the one in row zero. Furthermore the end pointer for this row is not allowed to be smaller than the one of row zero.

$$\gamma = \bigvee_{0<i<m} \Big(\bigwedge_{a \in \mathcal{F}^\sharp \setminus \mathcal{F}} \big(M^a_{00} \leftrightarrow M^a_{i0}\big) \;\wedge\; \bigwedge_{\substack{0<j<n \\ a \in \mathcal{F}}} \big(M^a_{0j} \leftrightarrow M^a_{ij}\big) \;\wedge\; (\mathbf{e}_i \geqslant \mathbf{e}_0) \Big)$$

All together: For a DP problem $(\mathcal{P}, \mathcal{R})$ the formula $\mathsf{loop}^{m,n}_{\mathcal{P},\mathcal{R}}$ is defined as

$$\Big(\bigwedge_{0 \leqslant i < m} \big(\bigwedge_{0 \leqslant j < n} \alpha_{ij} \big) \wedge (\mathbf{e}_i < n) \wedge \beta_i \Big) \wedge \gamma$$

with $\beta_i = \bigvee_{l \to r \in \mathcal{P} \cup \mathcal{R}} R^{l \to r}_i \;\wedge\; \bigwedge_{l \to r \in \mathcal{P} \cup \mathcal{R}} \beta^{l \to r}_i$ which expresses that one rule has to apply in row i and that it is applied properly. The condition $\mathbf{e}_i < n$ ensures that all strings in the loop stay within the allowed matrix dimensions.

Different types of variables—concrete (M^a_{ij}) and abstract ones ($M^a_{i\mathbf{x}}$) where $\mathbf{x}$ is a list of propositional variables representing a natural number in binary—are used. The latter are needed when a rule is applied at the abstract position $\mathbf{p}_i$. By default, abstract variables $M^{\mathsf{a}}_{3[x_1,x_0]}$ and $M^{\mathsf{a}}_{3[y_1,y_0]}$ are different (since the variables differ) and hence may take different values. If the assignments for x_1 and y_1 as well as x_0 and y_0 are the same, we want to enforce that the variables take identical values. In the implementation we test for every such abstract variable whether it matches a concrete one and we identify them if that is the case in order to obtain consistent results: $\varphi_{\mathsf{cons}} = \bigwedge_{M^a_{i\mathbf{x}}} \bigwedge_{0 \leqslant j < n} \big((\mathbf{x} = j) \to (M^a_{ij} \leftrightarrow M^a_{i\mathbf{x}})\big)$. This allows to formulate the main theorem for encoding loops:

Theorem 3. *A DP problem $(\mathcal{P}, \mathcal{R})$ admits a loop of length at most m involving strings of size at most $n+1$ if the formula $\mathsf{loop}^{m+1,n}_{\mathcal{P},\mathcal{R}} \wedge \varphi_{\mathsf{cons}}$ is satisfiable.* □

The previous theorem allows to implement a DP processor for nontermination.

Theorem 4 ([14, Theorem 26]**).** *The DP processor that maps a DP problem $(\mathcal{P}, \mathcal{R})$ to "no" if $(\mathcal{P}, \mathcal{R})$ loops and to $\{(\mathcal{P}, \mathcal{R})\}$ otherwise is sound and complete.* □

For our formalization the following lemma is essential (cf. Section 3).

Lemma 5. *If $\mathcal{P} \subseteq \mathtt{DP}(\mathcal{R})$ then any loop in the DP problem $(\mathcal{P}, \mathcal{R})$ can be transformed into a loop in $\mathcal{R}$.*

Proof. If $\mathcal{P} \subseteq \mathtt{DP}(\mathcal{R})$ then any sequence $t^\sharp_1 \to_{\mathcal{P} \cup \mathcal{R}} t^\sharp_2 \to_{\mathcal{P} \cup \mathcal{R}} t^\sharp_3 \to_{\mathcal{P} \cup \mathcal{R}} \cdots$ can be transformed into a sequence $t_1 \to_{\mathcal{R}} C_1[t_2] \to_{\mathcal{R}} C_2[t_3] \to_{\mathcal{R}} \cdots$ involving only the original system by soundness of the dependency pair transformation [1]. □

The restriction $\mathcal{P} \subseteq \mathtt{DP}(\mathcal{R})$ does no harm since the initial DP problem $(\mathtt{DP}(\mathcal{R}),\mathcal{R})$ obviously satisfies it and all DP processors we employ only remove rules from $\mathcal{P}$.

The next corollary combines Theorem 3 and Lemma 5. Note that strings in the transformed loop might be of size larger than n due to additional contexts.

Corollary 6. *If* $\mathit{loop}^{m+1,n}_{\mathtt{DP}(\mathcal{R}),\mathcal{R}} \wedge \varphi_{\mathit{cons}}$ *is satisfiable then* $\mathcal{R}$ *admits a looping reduction of length at most* m. □

We state one nice property of the encoding from Theorem 3. Even for DP problems where all infinite *minimal* sequences are nonlooping our encoding may find looping nonminimal sequences [27, Example 5.5].

An alternative characterization of loops can be given in terms of *forward closures*. Define the set of *right forward closures* $\mathtt{RFC}(\mathcal{R})$ as the least set of reductions containing $\mathcal{R}$ (as a set of one-step reductions) and being closed under rewriting (if $(t_1,\ldots,t_n) \in \mathtt{RFC}(\mathcal{R})$ and $t_n \to_{\mathcal{R}} t_{n+1}$ then $(t_1,\ldots,t_n,t_{n+1}) \in \mathtt{RFC}(\mathcal{R})$) and under right extension (if $(t_1,\ldots,t_n) \in \mathtt{RFC}(\mathcal{R})$ and $t_n = sl_1$ for some $l_1 l_2 \to r \in \mathcal{R}$ with non-empty l_1 and l_2 then $(t_1 l_2,\ldots,t_n l_2, sr) \in \mathtt{RFC}(\mathcal{R})$). E.g., the loop in Example 1 is a right forward closure. By Theorem 9 from [12], $\mathcal{R}$ admits a loop if and only if there is a loop in $\mathtt{RFC}(\mathcal{R})$. Furthermore, if $\mathcal{R}$ admits a loop of length n then a loop of length at most n exists in $\mathtt{RFC}(\mathcal{R})$.

The set of *left forward closures* $\mathtt{LFC}(\mathcal{R})$ is defined symmetrically via *left extension*. By symmetry, either way we characterize loops of minimal length, but minimal length loops in $\mathtt{RFC}(\mathcal{R})$ and in $\mathtt{LFC}(\mathcal{R})$ can have quite different *widths*, where the *width* of a reduction is the size of the starting string. As a consequence, we will search for loops both in $\mathtt{RFC}(\mathcal{R})$ and in $\mathtt{LFC}(\mathcal{R})$.

3 Formalizing Loops

In the next two sections we assume some familiarity with Isabelle [20]. All lemmas and theorems within these sections have formally been proved in IsaFoR. Next we sketch how we formalized loops (for full rewriting). Figure 1 lists the most important function definitions and types. (Here we deviate from the syntax of IsaFoR, to increase readability, e.g., the application of a substitution would be $t \cdot \sigma$ rather than $t\sigma$.) A binary relation is a set of pairs. A loop is a triple consisting of a list of terms, a context, and a substitution. For a given relation $\mathcal{A}$, an element `a` is strongly normalizing (`SN_elt`) if there is no infinite sequence $\boldsymbol{s}$ such that $\boldsymbol{s}_0 = \mathtt{a}$ while for all i we have $(\boldsymbol{s}_i, \boldsymbol{s}_{i+1}) \in \mathcal{A}$. Strong normalization of a relation $\mathcal{A}$ (`SN`) holds if all elements of the domain are strongly normalizing with respect to $\mathcal{A}$. To guarantee that our definition of `SN` is suitable we proved an easy lemma stating equivalence to the built-in Isabelle notion of well-foundedness (`wf`), i.e., $\mathtt{SN}(\mathcal{A}) = \mathtt{wf}(\mathcal{A}^{-1})$. The rewrite relation $\to_{\mathcal{R}}$ induced by a TRS $\mathcal{R}$ is a binary relation on terms closed under contexts and substitutions. The function `rsteps` checks for a list of terms if between two consecutive terms there is a rewrite step. Then the predicate `is_loop` is defined based on `rsteps`. The function `ith` returns for a loop $([t_1,\ldots,t_n], C, \sigma)$ and any $i > 0$ the i-th term in the sequence:

$$t_1 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_n \to_{\mathcal{R}} C[t_1\sigma] \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} C[t_n\sigma] \to_{\mathcal{R}} C[C[t_1\sigma]\sigma] \to_{\mathcal{R}} \cdots$$

```
types 'a brel     = "('a × 'a)set"
types ('f,'v)loop = "('f,'v)term list × ('f,'v)ctxt × ('f,'v)sub"
definition SN_elt where
 "SN_elt 𝒜 a ≡ ¬(∃s. s₀ = a ∧ (∀i. (sᵢ,sᵢ₊₁) ∈ 𝒜))"
definition SN where "SN(𝒜) ≡ ∀a. SN_elt 𝒜 a"
fun rsteps :: "('f,'v)term list ⇒ ('f,'v)term brel ⇒ bool"
where "rsteps [t] ℛ = True"
    | "rsteps (s#t#ts) ℛ = (s →ℛ t ∧ rsteps (t#ts) ℛ)"
fun is_loop :: "('f,'v)loop ⇒ ('f,'v)term brel ⇒ bool" where
 "is_loop (t#ts,C,σ) ℛ = rsteps (t#ts@[C[tσ]]) ℛ"
fun ith :: "('f,'v)loop ⇒ nat ⇒ ('f,'v)term" where
 "ith(t#ts,C,σ)ᵢ = (if i < length(t#ts)
   then (t#ts)!i
   else C[(ith(t#ts,C,σ)_{i−length(t#ts)})σ])"
```

Fig. 1. Basic definitions

Using `ith` an infinite sequence $\boldsymbol{s}$ can be constructed, contradicting `SN_elt`. In IsaFoR, the main task was to prove the following lemma (which amounts to proving that rewriting is closed under contexts and substitutions).

Lemma 7. *If* `is_loop` ℓ $\mathcal{R}$ *then for all* i *we have* $\texttt{ith}(\ell)_i \to_{\mathcal{R}} \texttt{ith}(\ell)_{i+1}$. □

We obtain the main theorem for the abstract formalization:

Theorem 8. *If* `is_loop` ℓ $\mathcal{R}$ *then* $\to_{\mathcal{R}}$ *is not terminating.*

Proof. From `is_loop` ℓ $\mathcal{R}$ we get an infinite sequence $\boldsymbol{s}$ by defining $\boldsymbol{s}_i = \texttt{ith}(\ell)_i$. By Lemma 7 the sequence $\boldsymbol{s}$ satisfies $\forall i.\ \boldsymbol{s}_i \to_{\mathcal{R}} \boldsymbol{s}_{i+1}$. Hence, for the first term t of the loop ℓ we obtain $\neg\texttt{SN\_elt} \to_{\mathcal{R}} t$ and thus by definition of `SN`, $\neg\texttt{SN}(\to_{\mathcal{R}})$. □

4 Certifying Loops

This section aims at certification, i.e., an automatic check if a suspected loop indeed is a loop. For this task we use the code-generation [16] facilities of Isabelle which allow to generate verified code. We provide an implementation of the predicate `is_loop` from Section 3 by the check-function `check_loop` that tests if a list of terms, a context, and a substitution form a loop. First we state the main theorem for certifying loops:

Theorem 9. *If* `check_loop` ℓ $\mathcal{R}$ *then* $\to_{\texttt{set}(\mathcal{R})}$ *is not terminating.* □

This resembles Theorem 8, but on a *constructive* (executable) level. Here $\mathcal{R}$ is chosen as a list and `set` transforms a list into a set. The reason is that for lists executable code can be generated but for sets not. Before considering `check_loop`

```
types ('f,'v)rule = "('f,'v)term × ('f,'v)term"
types ('f,'v)trsL = "('f,'v)rule list"
fun rewrites where
 "rewrites (s,t) C σ (l,r) R = (s = C[lσ] ∧ t = C[rσ] ∧ (l,r) mem R)"
fun rewrites_to
where "rewrites_to [(s,C,σ,rule)] t R = rewrites (s,t) C σ rule R"
    | "rewrites_to ((s,C,σ,rule)#(t,C',σ',rule')#xs) u R = (
       rewrites (s,t) C σ rule R ∧
       rewrites_to ((t,C',σ',rule')#xs) u R)"
fun check_loop_d
where "check_loop_d [] _ _ _ = False"
    | "check_loop_d xs C σ R = rewrites_to xs C[(fst(hd xs))σ] R"
```

Fig. 2. Checking a loop with all details provided

we focus on a simpler task, namely `check_loop_d` where more details of the loop are supplied. In the lemma below, the list xs contains the full information for every rewrite step $s \to_{\mathcal{R}} t$, i.e., the context C, the substitution σ, and the rewrite rule $l \to r$ such that $s = C[l\sigma] \to_{\mathcal{R}} C[r\sigma] = t$ (cf. Figure 2 for the definition of `check_loop_d` and the functions it relies on). Using `check_loop_d` no further information must be computed and certification of a candidate loop is already possible:

Lemma 10. *If* `check_loop_d` xs C σ $\mathcal{R}$ *then* $\to_{\mathtt{set}(\mathcal{R})}$ *is not terminating.*

Proof. The abstract formalization can be linked to the concrete implementation: If `rewrites_to` xs t $\mathcal{R}$ then `rsteps` (`map fst` xs`@[`t`]`) (`set`$(\mathcal{R})$). By unfolding the definitions of `check_loop_d` and `is_loop`, and using Theorem 8, the proof concludes. □

The main drawback of the function `check_loop_d` is that it requires all information about the rewrite steps. To the best of our knowledge not a single termination prover provides all these details. To make the certification of loops more appealing and user-friendly we turn our focus on the function `check_loop` again. Here for every rewrite step $s \to_{\mathcal{R}} t$ the context C, the substitution σ, and the rewrite rule $l \to r$ such that $s = C[l\sigma]$ and $t = C[r\sigma]$ are computed internally.

A function `get` s t $\mathcal{R}$ computes `Some`$(C, \sigma, (l, r))$ if there is a rewrite step from s to t involving C, σ, and $l \to r \in \mathcal{R}$. Hence `get` has to test for all rules $l \to r \in \mathcal{R}$ if for any context C in s there is a substitution σ satisfying $s = C[l\sigma]$ and $t = C[r\sigma]$.[5] To find this substitution we had to implement matching. With the help of `get` a function `get_list` returns the necessary information for a sequence of rewrite steps and `get_loop` computes all details for a looping reduction. Finally the function `check_loop` just calls `check_loop_d` on the output of `get_loop`.

[5] If $s \to_{\{l\to r\}} t$ and $s = C[l\sigma]$ then $t = C[r\sigma]$ holds for free by our definition of TRS. To handle systems that violate the variable condition we demand both conditions.

5 Experiments

The first part of this section evaluates the power of our approaches to find loops for SRSs. The second part focuses on certifying loops for SRSs and TRSs.

In our tests we considered the 1391 TRSs and 732 SRSs from the Termination Problems Data Base version 5.0 (TPDB). All tests have been performed on a server equipped with eight dual-core AMD Opteron® processors 885 running at a clock rate of 2.6 GHz and on 64 GB of main memory at a time limit of 60 s.

Finding Loops: We integrated the encoding from Section 2 into T$_{\text{T}}$T$_2$ [19] which was configured such that propositional formulas were solved by MiniSat [9] after a satisfiability-preserving transformation to CNF [23]. (Using the SMT solver Yices [8] as back-end produced slightly worse results.)

The implementation of the encoding differs from the presentation in Section 2 for reasons of readability. Our experiments showed that nontermination proving power can be slightly (i.e., a gain of about 10%) extended by addressing the following issues. Mutual exclusion of the M^a_{ij} variables can be expressed more concisely. After fixing an order on the variables, the property that at most one of the variables $x_1, \ldots, x_n$ can be satisfied, is expressed by $x_i \to \neg x_{i+1} \wedge \cdots \wedge \neg x_n$ for all $1 \leqslant i < n$. Due to mutual exclusion of the M^a_{ij} variables, all bi-implications occurring in subformulas of $\mathsf{loop}^{m,n}_{\mathcal{P},\mathcal{R}}$ can safely be replaced by implications. The encoding contains the requirement $\mathbf{e}_{i+1} + \|l\| = \mathbf{e}_i + \|r\|$ where "=" could be weakened to "$\leqslant$". (This corresponds to cutting parts of the substitution.) However, due to the increased search space the more restrictive version performs better. To reduce the search space, fixing $\mathbf{p}_i < \min\{3 + i * \max_{l \to r \in \mathcal{R}}\{\|l\|, \|r\|\}, n\}$ applies rewrite rules close to the root in the first few rows.

Before applying the loop-finder, T$_{\text{T}}$T$_2$ uses termination methods like matrix interpretations [10] and bounds [18] to preprocess DP problems. Sometimes these methods suffice to prove termination and often they simplify DP problems. Heuristics for encoding loops try matrices of different dimensions ranging from 10×10 up to 25×25. Most successful proofs only take a few seconds.

As an alternative, an enumeration of looping forward closures (cf. Section 2) is implemented in KnockedForLoops (KFL), a tool developed by the third author. It is based on a simple brute force, breadth first search strategy. To overcome the problem with excessive memory consumption, we employ bounds on the width of forward closures, i.e., bounds on the number of extension steps. By a concurrent search both in RFC($\mathcal{R}$) and in LFC($\mathcal{R}$) with different width bounds, many long loops with small width can be exhibited. As a simple combinatorial optimization, we disregard reductions with a rewrite step to the left of the previous step in case these steps do not overlap (since those two steps would commute).

We compare our implementations with three powerful nontermination analyzers, namely the 2007 version of Matchbox [25] specialized to nontermination (enumerations of forward closures, reversing, transport systems), nonloop [21], and the 2008 edition of NTI [22]. Earlier versions of Matchbox and NTI participated in the *Standard SRS* category of the 2007 competition and nonloop did so in 2008. These tools were (apart from T$_{\text{T}}$T$_2$) the most powerful ones for

Table 1. Finding and certifying loops

	732 SRSs					1391 TRSs
tool	KFL /CeTA	Matchbox	nonloop	NTI	T_TT_2 /CeTA	T_TT_2 /CeTA
no	158 / 158	149	93	67	97 / 97	203 / 203
time	34853/ 6	35441	37855	39508	23039/ 3	25898/ 3
time (avg.)	2.45 /0.03	3.10	5.73	5.78	11.61/0.02	0.19 /0.01

nontermination in this division. The left part of Table 1[6] presents a comparison of the provers where the row labeled no shows the number of successfully found/certified nontermination proofs, time refers to the accumulated total time used by the tool in seconds and time (avg.) displays the average time needed for successfully finding/certifying a nontermination proof. The columns labeled CeTA are explained in the next subsection. We note that the algorithm underlying NTI performs much better for terms than for strings (cf. [22]) and that the main aim of nonloop is establishing nonlooping nontermination. In our experiments KFL subsumes Matchbox, NTI, and T_TT_2. Only nonloop can disprove nine systems terminating which KFL misses. We anticipate that these systems are nonlooping nonterminating. Most remarkably, KFL finds a loop of length 80 and width 21 for the system Gebhardt/10 from TPDB, the termination status of which has been—at least to the authors' knowledge—open for several years.

Certifying Loops: Our contribution amounts to approximately 500 lines of Isabelle code that were added to IsaFoR (theory `Loop`). This includes the abstract formalization of loops and both approaches for certifying loops (the detailed one using `check_loop_d` and the user-friendly one based on `check_loop`). The key concept for certification presented here, is the code-generation mechanism of Isabelle (currently we export *verified* Haskell code). Thus the whole certifier consists of a bunch of automatically generated sources and a main file that just calls the check function on a given problem and proof. This paper refers to version 1.01 of CeTA the input format of which can be found at its website.[7] When calling CeTA, two arguments have to be supplied, namely the input problem and the proof attempt. The tool then tests if the specified proof attempt corresponds to a loop and terminates with exit code 0 in case of success and exit code 1 if the input could not be proved to be a loop.

Considering Table 1 again (this time only the columns CeTA) separate empirical data for certifying loops for 732 SRSs and 1391 TRSs is given.[8] For the columns labeled XXX/CeTA the tool XXX was used to find loops which CeTA then had to certify. The row labeled no indicates the number of successfully certified

[6] Experiments are available from `http://cl-informatik.uibk.ac.at/ttt2/loops/`.

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

[8] For completeness we mention how T_TT_2 finds loops for TRSs: Apart from the approach proposed in [22] two trivial methods are employed (test for fresh variables on right-hand sides and test if a rule is self-embedding).

systems and time shows the accumulated time in seconds for certifying loops. The certifier was just called for the successfully found loops but still this number demonstrates that the overhead for certification is negligible. This is remarkable, since the certifier has to compute for every two consecutive terms s and t in the loop a context C, a substitution σ, and a rewrite rule $l \to r \in \mathcal{R}$ such that indeed $s = C[l\sigma] \to_{\mathcal{R}} C[r\sigma] = t$. We conjecture that CeTA's efficiency is mainly due to code generation. (For a comparison in run time with other certifiers see [24].) CeTA could certify all 158 (97) SRSs and 203 TRSs nonterminating for which KFL (T$_{\text{T}}$T$_2$) and T$_{\text{T}}$T$_2$ provided a nontermination proof, respectively.

6 Conclusion and Future Work

This paper presents two methods for finding loops in SRSs. Since the encoding from Section 2 takes parameters for the length of looping sequences and the maximal size of strings occurring within the reduction it is especially suitable to find short(est) loops. This eases the task of debugging since the reason for nontermination is concisely represented. Our experiments revealed that detecting loops by enumerating forward closures is powerful. In the second part we formalized strong normalization in the theorem prover Isabelle and sketched how our contribution allows to generate verified code capable of certifying loops. The thereby generated check-function was incorporated into CeTA. Since CeTA is freely available our contribution allows any termination tool to certify its loop output.

Both contributions may be further investigated. One question concerning Section 2 is whether the encoding can be lifted from strings to terms. Concerning the formalization of loops one could try to incorporate the approach from [21] and also formalize nonlooping nontermination. It has to be clarified if from the output provided by nonloop the i-th term in a nonterminating sequence can be extracted easily. This issue will then make the task either easy or undoable.

Acknowledgments. We would like to thank René Thiemann for exploring the code-generation facilities of Isabelle and Johannes Waldmann for helpful comments and providing a version of Matchbox'07 specialized to nontermination.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236(1-2), 133–178 (2000)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. (2004)
4. Blanqui, F., Delobel, W., Coupet-Grimal, S., Hinderer, S., Koprowski, A.: CoLoR, a Coq library on rewriting and termination. In: WST. pp. 69–73 (2006)

5. Clarke, A., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. FMSD 19(1), 7–34 (2001)
6. Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: Certification of automated termination proofs. In: FroCoS. LNAI, vol. 4720, pp. 148–162 (2007)
7. Cook, S.: The complexity of theorem-proving procedures. In: STOC. pp. 151–158 (1971)
8. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: CAV. LNCS, vol. 4144, pp. 81–94 (2006)
9. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. LNCS, vol. 2919, pp. 502–518 (2004)
10. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. JAR 40(2-3), 195–220 (2008)
11. Geser, A., Hofbauer, D., Waldmann, J.: Termination proofs for string rewriting systems via inverse match-bounds. JAR 34(4), 365–385 (2005)
12. Geser, A., Zantema, H.: Non-looping string rewriting. TIA 33(3), 279–302 (1999)
13. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: Combining techniques for automated termination proofs. In: LPAR. LNAI, vol. 3452, pp. 301–331 (2005)
14. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: FroCoS. LNAI, vol. 3717, pp. 216–231 (2005)
15. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. JAR 37(3), 155–203 (2006)
16. Haftmann, F.: Code Generation from Specifications in Higher Order Logic. PhD thesis, Technische Universität München (2009)
17. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. I&C 199(1-2), 172–199 (2005)
18. Korp, M., Middeldorp, A.: Match-bounds revisited. I&C (2009). doi:10.1016/j.ic.2009.02.010
19. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: RTA. LNCS, vol. 5595, pp. 295–304 (2009)
20. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. vol. 2283 of LNCS. (2002)
21. Oppelt, M.: Automatische Erkennung von Ableitungsmustern in nichtterminierenden Wortersetzungssystemen. Master's thesis, HTWK Leipzig (FH) (2008)
22. Payet, É.: Loop detection in term rewriting using the eliminating unfoldings. TCS 403(2-3), 307–327 (2008)
23. Plaisted, D., Greenbaum, S.: A structure-preserving clause form translation. JSC 2(3), 293–304 (1986)
24. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: TPHOLs. LNCS, vol. 5764, pp. 452–468 (2009)
25. Waldmann, J.: Matchbox: A tool for match-bounded string rewriting. In: RTA. LNCS, vol. 3091, pp. 85–94 (2004)
26. Waldmann, J.: Compressed loops (2007). Draft, available from `http://dfa.imn.htwk-leipzig.de/matchbox/methods/loop.pdf`
27. Zankl, H.: Lazy Termination Analysis. PhD thesis, University of Innsbruck (2009)
28. Zantema, H.: Termination of string rewriting proved automatically. JAR 34(2), 105–139 (2005)
29. Zantema, H.: Termination. In: TeReSe (ed.) Term Rewriting Systems. vol. 55 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press 181–259 (2003)