International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 401-416
http://rewriting.loria.fr/rta/

PROVING PRODUCTIVITY IN INFINITE DATA STRUCTURES

HANS ZANTEMA [1,2] AND MATTHIAS RAFFELSIEPER [1]

[1] Department of Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail address: H.Zantema@tue.nl
E-mail address: M.Raffelsieper@tue.nl

[2] Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract. For a general class of infinite data structures including streams, binary trees, and the combination of finite and infinite lists, we investigate the notion of productivity. This generalizes stream productivity. We develop a general technique to prove productivity based on proving context-sensitive termination, by which the power of present termination tools can be exploited. In order to treat cases where the approach does not apply directly, we develop transformations extending the power of the basic approach. We present a tool combining these ingredients that can prove productivity of a wide range of examples fully automatically.

1. Introduction

Some computations potentially go on forever. A standard example is the sieve of Eratosthenes producing the infinitely many prime numbers. The result of such a computation is then an infinite stream of elements. Although the computation itself goes on forever, there is a kind of termination involved that is called *productivity*: every finite initial part will be produced after a finite number of steps. We will consider computations specified by a number of rewrite rules that are interpreted as a lazy functional program. Then productivity can be characterized and investigated as a property of term rewriting, as was investigated before in [6, 11, 4, 19, 12].

Streams can be seen as infinite terms. Even when restricting to data structures representing the result of a computation, it is natural not to restrict to streams. In case the computation possibly ends, then the result is not a stream but a finite list, and when parallelism is considered, naturally infinite trees come in. In this paper we develop techniques for automatically proving productivity of specifications in a wide range of infinite data structures, including streams, the combination with finite lists, and several kinds of infinite trees. Earlier techniques specifically for stream specifications were given in [6, 4, 19]. A key idea of our approach is to prove productivity by proving termination of *context-sensitive rewriting* [17, 9], that is, the restricted kind of rewriting in which rewriting is disallowed inside particular arguments of particular symbols. As strong tools like AProVE [8] and μ-Term

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.401

[13] have been developed to prove termination of context-sensitive rewriting automatically, the power of these tools can now be exploited to prove productivity automatically. As the underlying technique is completely different from the technique of [6, 4], it is expected that both approaches have their own merits. Indeed, there are examples where the technique of [6, 4] fails and our technique succeeds. The comparison the other way around is hard to make as the technique of [6, 4] only applies for proving productivity for a single ground term while our technique applies for proving productivity for all ground terms.

Through this paper we consider two kinds of terms: finite terms and infinite terms. As the elements of the infinite data structures we intend to define are infinite terms, infinite terms are unavoidable here. On the other hand, terms occurring in our specifications are finite. Rewriting has been investigated both for finite and infinite terms. But rewriting finite terms is much easier and better understood than infinitary rewriting, and for many properties, like several variants of termination, there is strong tool support to investigate these properties. We follow the policy to use infinite terms only where necessary, and exploit understanding and tool support for rewriting finite terms as much as possible. In this way we need the concept of infinite terms, but not of infinitary rewriting. Following this policy, elements of infinite data structures over a data set $\mathcal{D}$ are considered as infinite terms in which elements of $\mathcal{D}$ act as constants, and the infinite terms are composed from constructor symbols taken from a set $\mathcal{C}$, and elements of $\mathcal{D}$. In this world of infinite terms we want to avoid that data elements are infinite terms themselves. For instance, in considering streams over natural numbers as infinite terms, we want to be able to consider a stream $0 : 1 : 2 : 3 : 4 : \cdots$, but we do not want the data elements in such a stream to be infinite terms like $\mathsf{s}^\infty(0)$. When specifying elements of infinite data structures over a data set $\mathcal{D}$, the set $\mathcal{D}$ may be described as the set of (finite) ground normal forms of some rewriting system R_d over data signature Σ_d. As an example, for natural numbers with $+$ we can choose $\Sigma_d = \{0, \mathsf{s}, +\}$, and R_d consists of the rules $0 + x \to x$ and $\mathsf{s}(x) + y \to \mathsf{s}(x + y)$. Apart from Σ_d and R_d the specification then is given by a set R_s of rewrite rules over $\mathcal{C} \cup \Sigma_d \cup \Sigma_s$, where Σ_s consists of constants and auxiliary operations to be introduced for the specification. For instance, for defining the above mentioned stream $\mathsf{nat} = 0 : 1 : 2 : 3 : 4 : \cdots$ we introduce an auxiliary function $\mathsf{f} \in \Sigma_s$ on streams that replaces each element by its successor, and specify nat by choosing R_s to consist of the two rules

$$\mathsf{nat} \to 0 : \mathsf{f}(\mathsf{nat}), \quad \mathsf{f}(x : \sigma) \to \mathsf{s}(x) : \mathsf{f}(\sigma).$$

Here we have $\mathcal{C} = \{:\}$, $\Sigma_d = \{0, \mathsf{s}\}$, $R_d = \emptyset$, $\Sigma_s = \{\mathsf{nat}, \mathsf{f}\}$, and x, σ are variable symbols of type data and stream, respectively. In this setting productivity means that for every n the initial term can be rewritten to a (finite) term in which all symbols on depth less than n are constructor symbols. This notion of productivity is consistent with stream productivity as in [6, 4, 19], formalizing the spirit of stream productivity as introduced in [15]. It is also consistent with productivity as defined in [11, 12] for a setting even more general than ours.

Proving productivity may be hard. For the sieve of Eratosthenes proving productivity is beyond the scope of fully automatic techniques as it depends on the fact that there are infinitely many prime numbers. Moreover, we can specify an extra stream by filtering out every element in this stream of prime numbers that is distinct from its predecessor plus 2. This yields a stream specification, easily expressed in the format of this paper, of which productivity is equivalent to the existence of infinitely many prime twins: a well-known open problem in number theory. As expected for such a format suitable for expressing a well-known open problem, productivity is an undecidable property. This has been proved independently by several people in [5, 16].

In contrast to [6, 4], we focus on requiring productivity not only for a single initial term, like nat in the above example, but for all finite ground terms. An easy induction argument shows that productivity holds for all ground terms if and only if every ground term rewrites to a term of which the root is a constructor symbol. As in [19] this characterization is the basis of our productivity investigations, but now for more general infinite data structures than only streams. A main reason for the focus on productivity for all ground terms is this technical convenience. In many cases, however, productivity of a single ground terms is equivalent to productivity for all ground terms over the operations that are relevant for the single ground term. If this does not hold and one is interested in productivity for a single ground term, our approach fails while the approach of [6, 4] may succeed.

The paper is organized as follows. In Section 2 we introduce infinite terms and give examples of several infinite data structures consisting of infinite terms. In Section 3 we introduce our notions of proper specifications and productivity. Being interested in deterministic computations, in proper specifications we require the rewrite systems to be orthogonal. A first basic result (Theorem 3.4) states that a specification is productive if for all rules the root of the right-hand side is a constructor symbol. In Section 4 we relate productivity to context-sensitive rewriting. The main result (Theorem 4.1) states that a specification is productive if context-sensitive termination holds for the rules of the specification, where rewriting is only allowed in the data arguments of the constructor symbols, and in all arguments of the other symbols. For cases where these approaches fail, in Section 5 we investigate transformations that transform a proper specification into another one, such that productivity of the original specification can be concluded from productivity of the transformed specification, the latter typically proved by the basic techniques from the earlier sections. Through these sections we give several examples of specifications of streams and binary trees for which productivity is proved. For many of these, productivity cannot be proved by earlier techniques. In Section 6 we describe an implementation of our techniques, by which productivity of all productive examples presented in this paper can be proved fully automatically. We conclude in Section 7.

2. Infinite Terms

Intuitively, a term (both finite and infinite) is defined by saying which symbol is at which position. Here, a *position* $p \in \mathbf{N}^*$ is a finite sequence of natural numbers. In order to be a proper term, some requirements have to be satisfied as indicated in the following definition. As we will only consider infinite terms over a set $\mathcal{C}$ of constructors and a set $\mathcal{D}$ of data (disjoint from $\mathcal{C}$), our terms will be two-sorted[1]: a sort s for the (infinite) terms to be defined, and a sort d for the data. Every $f \in \mathcal{C}$ is assumed to be of type $d^n \times s^m \to s$ for some $n, m \in \mathbf{N}$. We write $\mathsf{ar}(d, f) = n$ and $\mathsf{ar}(s, f) = m$. We write $\perp$ for undefined.

Definition 2.1. A (possibly infinite) *term* of sort s over $\mathcal{C}, \mathcal{D}$ is defined to be a map $t : \mathbf{N}^* \to \mathcal{C} \cup \mathcal{D} \cup \{\perp\}$ such that

- the root $t(\epsilon)$ of the term t is a constructor symbol, so $t(\epsilon) \in \mathcal{C}$, and

[1]In [11, 12] an arbitrary many-sorted setting is proposed. Our approach easily generalizes to a more general many-sorted setting, but for notational convenience we restrict to the two-sorted setting.

- for all $p \in \mathbf{N}^*$ and all $i \in \mathbf{N}$ we have

$$t(pi) \in \mathcal{D} \iff t(p) \in \mathcal{C} \wedge 1 \leq i \leq \mathsf{ar}(d, t(p)), \text{ and}$$
$$t(pi) \in \mathcal{C} \iff t(p) \in \mathcal{C} \wedge \mathsf{ar}(d, t(p)) < i \leq \mathsf{ar}(d, t(p)) + \mathsf{ar}(s, t(p)).$$

So $t(pi) = \perp$ for all p, i not covered by the above two cases.

We write $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ for the set of all terms over $\mathcal{C}, \mathcal{D}$.

An alternative equivalent definition of $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ can be given based on co-algebra. Another alternative uses metric completion, where infinite terms are limits of finite terms. However, for the results in this paper we do not need these alternatives.

A position $p \in \mathbf{N}^*$ satisfying $t(p) \in \mathcal{C}$ is called a *position of* t of sort s. A position $p \in \mathbf{N}^*$ satisfying $t(p) \in \mathcal{D}$ is called a *position of* t of sort d. The *depth* of a position $p \in \mathbf{N}^*$ is the length of p considered as a string.

The usual notion of finite term coincides with a term in this setting having finitely many positions, that is, $t(p) = \perp$ for all but finitely many p. In case $\mathsf{ar}(s, f) > 0$ for all $f \in \mathcal{C}$ then no finite terms exist. This holds for streams. In case $\mathsf{ar}(d, f) = 0$ for all $f \in \mathcal{C}$ then no position of sort $\mathcal{D}$ exist, and terms do not depend on $\mathcal{D}$.

For $f \in \mathcal{C}$ with $\mathsf{ar}(d, f) = n$, $\mathsf{ar}(s, f) = m$, n elements $u_1, \ldots, u_n \in \mathcal{D}$ and m terms $t_1, \ldots, t_m$ we write $f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ for the term t defined by $t(\epsilon) = f$, $t(i) = u_i$ for every $i = 1, \ldots, n$, $t(ip) = t_{i-n}(p)$ for every $p \in \mathbf{N}^*$ and $i = n+1, \ldots, n+m$, and $t(ip) = \perp$ if $i \notin \{1, \ldots, n+m\}$, or $i \in \{1, \ldots, n\}$ and $p \neq \epsilon$.

Example 2.2 (Streams)**.** Let $\mathcal{D}$ be an arbitrary given non-empty data set, and let $\mathcal{C} = \{:\}$, with $\mathsf{ar}(d, :) = \mathsf{ar}(s, :) = 1$. Then $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ coincides with the usual notion of *streams* over $\mathcal{D}$, being functions from $\mathbf{N}$ to $\mathcal{D}$. More precisely, a function $f : \mathbf{N} \to \mathcal{D}$ gives rise to an infinite term t defined by $t(2^n) = :$ and $t(2^n 1) = f(n)$ for every $n \in \mathbf{N}$, and $t(w) = \perp$ for all other strings $w \in \mathbf{N}^*$. Conversely, every $t : \mathbf{N}^* \to \mathcal{C} \cup \mathcal{D}$ satisfying the requirements of the definition of a term is of this shape. Note that if $\#\mathcal{D} = 1$, then there exists only one such term.

In case $\mathcal{D}$ is finite, an alternative approach is not to consider the binary constructor ':', but unary constructors for every element of $\mathcal{D}$. In this approach $\mathcal{D}$ does not play a role and is irrelevant.

Example 2.3 (Finite and infinite lists)**.** Let $\mathcal{D}$ be an arbitrary given non-empty data set, and let $\mathcal{C} = \{:, \mathsf{nil}\}$, with $\mathsf{ar}(d, :) = \mathsf{ar}(s, :) = 1$ and $\mathsf{ar}(d, \mathsf{nil}) = \mathsf{ar}(s, \mathsf{nil}) = 0$. Then $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ covers both the *streams* over $\mathcal{D}$ as in Example 2.2 and the usual (finite) lists. As in Example 2.2, a function $f : \mathbf{N} \to \mathcal{D}$ gives rise to an infinite term t defined by $t(2^n) = :$ and $t(2^n 1) = f(n)$ for every $n \in \mathbf{N}$, and $t(w) = \perp$ for all other strings $w \in \mathbf{N}^*$. The only way nil can occur is where $t(2^n) = \mathsf{nil}$ for some $n \geq 0$, $t(2^i) = :$ and $t(2^i 1) \in \mathcal{D}$ for every $i < n$, and $t(w) = \perp$ for all other strings $w \in \mathbf{N}^*$, in this way representing a finite list of length n. Conversely, every $t : \mathbf{N}^* \to \mathcal{C} \cup \mathcal{D}$ satisfying the requirements of the definition of a term is of one of these two shapes. In the literature this combination of finite and infinite lists is sometimes called *lazy lists*.

Example 2.4 (Binary trees)**.** For infinite binary trees several variants fit in our format. We will meet the following:

- Infinite binary trees with nodes labeled by $\mathcal{D}$ are obtained by choosing $\mathcal{C} = \{\mathsf{b}\}$ with $\mathsf{ar}(d, \mathsf{b}) = 1$ and $\mathsf{ar}(s, \mathsf{b}) = 2$. In Example 4.4 the nodes are labeled by $\mathcal{D} \times \mathcal{D}$, obtained by choosing $\mathsf{ar}(d, \mathsf{b}) = 2$ instead.

- The combination of finite and infinite binary trees with nodes labeled by $\mathcal{D}$ is obtained by choosing $\mathcal{C} = \{\mathsf{b}, \mathsf{nil}\}$ with $\mathsf{ar}(d, \mathsf{b}) = 1$, $\mathsf{ar}(s, \mathsf{b}) = 2$ and $\mathsf{ar}(d, \mathsf{nil}) = \mathsf{ar}(s, \mathsf{nil}) = 0$. In Example 3.5 the nodes are unlabeled, obtained by choosing $\mathsf{ar}(d, \mathsf{b}) = 0$ instead.

3. Specifications and Productivity

Throughout this paper we use some basics of term rewriting as is introduced e.g. in [1, 17]. In particular, a term rewriting system (TRS) is called *orthogonal* if the left-hand sides of the rules do not overlap, and every variable occurs at most once in every left-hand side of a rule.

We consider *specifications* in order to define elements of $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$. We do this for the special case where $\mathcal{D}$ consists of the ground normal forms of an orthogonal terminating TRS R_d over a signature Σ_d. Here all symbols of Σ_d are considered to be of sort $d^n \to d$ for some $n \geq 0$. For defining elements of $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ we introduce a set Σ_s of defined symbols of sort s, disjoint from $\mathcal{C}$, all being of sort $d^n \times s^m \to s$ for some $n, m \in \mathbf{N}$, just like the elements of $\mathcal{C}$. The real specification is given by a set R_s of rewrite rules of sort s being of a special shape. Although the goal is to define elements of $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$, most times being infinite, all terms in the specification are finite, and rewriting always refers to rewriting finite terms. All terms are well-sorted, that is, for every symbol f occurring in a term the sort of the term on the i-th argument equals the sort expected at that argument.

Definition 3.1. A *proper specification* $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ consists of $\Sigma_d, \Sigma_s, \mathcal{C}, R_d$ as described above and a TRS R_s over $\Sigma_d \cup \mathcal{C} \cup \Sigma_s$ consisting of rules of the shape

$$f(u_1, \ldots, u_n, t_1, \ldots, t_m) \to t,$$

where

- $f \in \Sigma_s$ is of type $d^n \times s^m \to s$,
- for every $i = 1, \ldots, m$ the term t_i is either
  - a variable of sort s, or
  - $t_i = g(d_1, \ldots, d_k, \sigma_1, \ldots, \sigma_l)$ for some $g \in \mathcal{C}$ with $\mathsf{ar}(d, g) = k$ and $\mathsf{ar}(s, g) = l$, where $\sigma_1, \ldots, \sigma_l$ are variables of sort s, and $d_1, \ldots, d_k$ are terms over Σ_d,
- t is a (well-sorted) term of sort s,
- $R_s \cup R_d$ is orthogonal, and
- every term of the shape $f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ for $f \in \Sigma_s$, $u_1, \ldots, u_n \in \mathcal{D}$, and in which every t_i is of the shape $g(u'_1, \ldots, u'_n, t'_1, \ldots, t'_m)$ for $g \in \mathcal{C}$ and $u'_1, \ldots, u'_n \in \mathcal{D}$, matches with the left-hand side of a rule from R_s.

Intuitively, the last bullet requires exhaustiveness of pattern matching, as is a standard requirement in functional programming. Orthogonality is required for forcing unicity of the result of computation. The second bullet requires simplicity of left-hand sides of rules; in case this restriction does not hold it can be obtained by unfolding the rules and introducing extra symbols.

A proper specification is therefore a generalization of proper stream specifications as given in [18, 19]. Fixing $\mathcal{C}, \mathcal{D}$, typically a proper specification will be given by R_d, R_s in which Σ_d, Σ_s and the arities are left implicit since they are implied by the terms occurring in R_d, R_s. If a proper specification is only given by R_s, then R_d is assumed to be empty. This is what we will do several times, starting in Example 3.5.

For a term $t = f(\cdots)$ we write $\mathsf{root}(t) = f$; the symbol f is called the *root* of t.

A specification is called *productive* for a given ground term of sort s if every finite part of the intended resulting infinite term can be computed in finitely many steps. As the intended resulting infinite term consists of constructor symbols and data elements, and all ground terms of sort d rewrite to data elements by assumption, this is equivalent to the following.

Definition 3.2. A proper specification $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ is *productive* for a ground term t of sort s if for every $k \in \mathbf{N}$ there is a reduction $t \to^*_{R_s \cup R_d} t'$ for which every symbol of sort s in t' on depth less than k is in $\mathcal{C}$.

An important consequence of productivity is *well-definedness*: the term admits a unique interpretation as an infinite term. Intuitively, existence follows from taking the limit of the process of computing a constructor on every level, and reduce data terms to normal form. Uniqueness follows form orthogonality. For an investigation of well-definedness of stream specifications we refer to [18].

As in [19], in this paper we are interested in productivity for all finite ground terms of sort s rather than a single one. The following proposition states that for this case reaching a constructor on every arbitrary depth is equivalent to reaching a constructor at the root. As the latter characterization is simpler, this is the basis of all further observations on productivity in this paper. In [11, 12] productivity is also required for infinite terms, being a stronger restriction than ours, see Example 4.2. Again we stress that in this section all terms are finite.

Proposition 3.3. *A specification $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ is productive for all ground terms of sort s if and only if every ground term t of sort s admits a reduction $t \to^*_{R_s \cup R_d} t'$ for which $\mathsf{root}(t') \in \mathcal{C}$.*

Proof. The "only if" direction of the proposition is obvious. For the "if" direction, we prove the following claim by induction on k.

> **Claim.** Let $k \in \mathbf{N}$, and for all ground terms t of sort s we have $t \to^*_{R_s \cup R_d} t'$ with $\mathsf{root}(t') \in \mathcal{C}$. Then $t \to^*_{R_s \cup R_d} t''$ for a term t'' in which every symbol of sort s on depth less than k is in $\mathcal{C}$.

If $k = 1$, then the claim directly holds by choosing $t'' = t'$.

Otherwise, we have $t \to^*_{R_s \cup R_d} t' = f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ with $\mathsf{root}(t') = f \in \mathcal{C}$, with f of type $d^n \times s^m \to s$. Applying the induction hypothesis to $t_1, \ldots, t_m$ yields $t_i \to^*_{R_s \cup R_d} t''_i$ with all symbols of sort s in t''_i are on depth $< k - 1$, for $i = 1, \ldots, m$. Now

$$t \to^*_{R_s \cup R_d} f(u_1, \ldots, u_n, t_1, \ldots, t_m) \to^*_{R_s \cup R_d} f(u_1, \ldots, u_n, t''_1, \ldots, t''_m)$$

proves the claim. ∎

Our first theorem gives a simple syntactic criterion for productivity, which can also be seen as a particular case of the analysis of friendly nesting specifications as given in [4].

Theorem 3.4. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ be a proper specification in which for every $\ell \to r$ in R_s the term r is not a variable and $\mathsf{root}(r) \in \mathcal{C}$. Then $\mathcal{S}$ is productive.*

Proof. According to Proposition 3.3 for every ground term t of sort s it suffices to prove that $t \to^*_{R_s \cup R_d} t'$ for a term t' satisfying $\mathsf{root}(t') \in \mathcal{C}$. We do this by induction on t. Let $t = f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ for $m, n \geq 0$. If $f \in \mathcal{C}$ we are done. So we may assume $f \in \Sigma_s$.

As they are ground terms of sort d, all u_i rewrite to elements of $\mathcal{D}$. By the induction hypothesis, all t_i rewrite to terms with root in $\mathcal{C}$, and in which the arguments of sort d rewrite to elements of $\mathcal{D}$. Now by the last requirement of properness, the resulting term matches with the left-hand side of a rule from R_s. By the assumption, by rewriting according to this rule a term is obtained of which the root is in $\mathcal{C}$. ∎

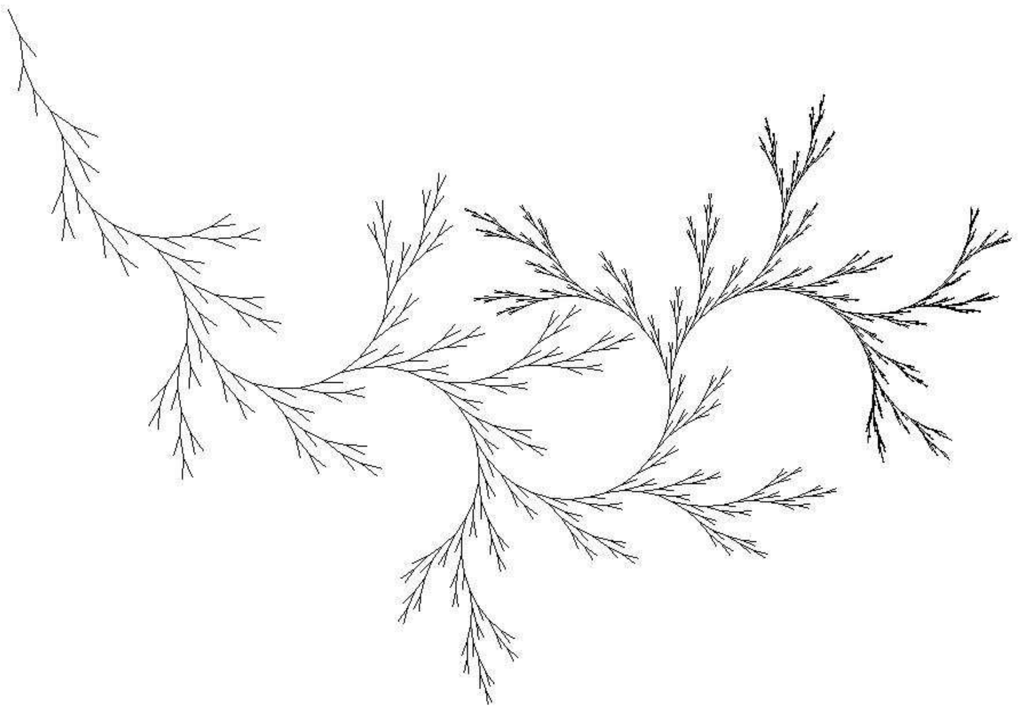
Example 3.5. Choose $\mathcal{C} = \{\mathsf{b}, \mathsf{nil}\}$ with $\mathsf{ar}(s, \mathsf{b}) = 2$ and $\mathsf{ar}(d, \mathsf{b}) = \mathsf{ar}(d, \mathsf{nil}) = \mathsf{ar}(s, \mathsf{nil}) = 0$ representing the combination of finite and infinite unlabeled binary trees. Then

$$\mathsf{c} \to \mathsf{b}(\mathsf{b}(\mathsf{nil}, \mathsf{c}), \mathsf{c})$$

is a proper specification that is productive due to Theorem 3.4; the symbol c represents an infinite tree in which the number of nodes on depth n is exactly the n-th Fibonacci number. In the same setting

$$\begin{array}{rcl} \mathsf{p} & \to & \mathsf{b}(\mathsf{f}(\mathsf{p}), \mathsf{nil}) \\ \mathsf{f}(\mathsf{b}(x, y)) & \to & \mathsf{b}(\mathsf{f}(y), \mathsf{b}(\mathsf{nil}, \mathsf{f}(x))) \\ \mathsf{f}(\mathsf{nil}) & \to & \mathsf{nil} \end{array}$$

is a proper specification that is productive due to Theorem 3.4. The symbol p represents the infinite tree of which the initial part until depth 100 is shown in the following picture, in which the root of the tree is shown on top left:



4. Proving Productivity by Context-Sensitive Termination

As intended for generating infinite terms, the TRS $R_s \cup R_d$ will never be terminating. However, when disallowing rewriting inside arguments of sort s of constructor symbols, it may be terminating. The main result of this section states that if this is the case, then the specification is productive. The variant of rewriting with the restriction that rewriting inside certain arguments of certain symbols is disallowed, is called *context-sensitive rewriting* [17, 9]. In context-sensitive rewriting, for every symbol f the set $\mu(f)$ of arguments of f is specified inside which rewriting is allowed. More precisely, μ-rewriting $\to_{R,\mu}$ with respect to a TRS R is defined inductively by

- if $\ell \to r \in R$ and ρ is a substitution, then $\ell\rho \to_{R,\mu} r\rho$;
- if $i \in \mu(f)$ and $t_i \to_{R,\mu} t'_i$ and $t'_j = t_j$ for all $j \neq i$, then $f(t_1,\dots,t_n) \to_{R,\mu} f(t'_1,\dots,t'_n)$.

In our setting we choose μ by $\mu(c) = \{1,\dots,\mathsf{ar}(d,c)\}$ for all $c \in \mathcal{C}$, and $\mu(f) = \{1,\dots,\mathsf{ar}(f)\}$ for all $f \in \Sigma_d \cup \Sigma_s$, where we write $\mathsf{ar}(f) = \mathsf{ar}(d,f) + \mathsf{ar}(s,f)$ for $f \in \Sigma_s$. In the rest of this paper the only instance of context-sensitive rewriting we consider is with respect to this particular μ, which is left implicit from now on. So in μ-rewriting, rewriting inside s-arguments of constructor symbols is disallowed, and is allowed in all other positions. A TRS is called *μ-terminating* if μ-rewriting is terminating.

Theorem 4.1. *Let $(\Sigma_d,\Sigma_s,\mathcal{C},R_d,R_s)$ be a proper specification for which $R_s \cup R_d$ is μ-terminating for μ as defined above. Then the specification is productive.*

Proof. We define a ground μ-normal form to be a ground term that can not be rewritten by μ-rewriting. We prove the following claim by induction on t:

Claim: If t is a ground μ-normal form of sort s, then the $\mathsf{root}(t) \in \mathcal{C}$.

Assume $\mathsf{root}(t) \notin \mathcal{C}$. Then $t = f(u_1,\dots,u_n,t_1,\dots,t_m)$ for $f \in \Sigma_s$, $u_1,\dots,u_n$ are of sort d, and $t_1,\dots,t_m$ are of sort s. Since $\mu(f) = \{1,\dots,n+m\}$, they are all ground μ-normal forms. So $u_1,\dots,u_n \in \mathcal{D}$. By the induction hypothesis all t_i have their roots in $\mathcal{C}$. Since t_i is a μ-normal form and the arguments of sort d are in $\mu(c)$ for every $c \in \mathcal{C}$, the arguments of t_i of sort d are all in $\mathcal{D}$. Due to the shape of the rules now a rule is applicable on t on the root level, so satisfies the restriction of μ-rewriting, contradicting the assumption that t is a μ-normal form. This concludes the proof of the claim.

According to Proposition 3.3 for productivity we have to prove that every ground term t of sort s rewrites to a term having its root in $\mathcal{C}$. Apply μ-rewriting on t as long as possible. Due to μ-termination this will end in a term on which μ-rewriting is not possible, so a ground μ-normal form. Due to the claim this ground μ-normal form has its root in $\mathcal{C}$. ∎

Example 4.2. Consider the following stream specification

$$\begin{array}{rcl} \mathsf{ones} & \to & 1 : \mathsf{ones} \end{array} \qquad \begin{array}{rcl} \mathsf{f}(0:\sigma) & \to & \mathsf{f}(\sigma) \\ \mathsf{f}(1:\sigma) & \to & 1:\mathsf{f}(\sigma) \end{array}$$

Productivity for all ground terms including $\mathsf{f}(\mathsf{ones})$ follows from Theorem 4.1: entering this rewrite system in the tool AProVE [8] or μ-Term [13] together with the context-sensitivity information that rewriting is disallowed in the second argument of ':' fully automatically yields a proof of context-sensitive termination. Alternatively, by entering this specification in our tool yields exactly the same proof.

In this specification f is the stream function that removes all zeros. So productivity depends on the fact that the stream of all zeros does not occur as the interpretation of a subterm of any ground term in this specification. For instance, by adding the rule $\mathsf{zeros} \to 0 : \mathsf{zeros}$ the specification is not productive any more as $\mathsf{f}(\mathsf{zeros})$ does not rewrite to a term having a constructor as its root.

This also shows the difference between our requirement of productivity of all finite ground terms and the requirement in [11, 12] of productivity of all terms, including infinite terms. There this example is not productive on the infinite term representing the stream of all zeros. Finally we mention that the technique from [4] fails to prove productivity for $\mathsf{f}(\mathsf{ones})$, since the specification is not data obliviously productive.

Example 4.3. We specify the sorted stream of Hamming numbers: all positive natural numbers that are not divisible by other prime numbers than 2, 3 and 5. Here $\mathcal{D} = \{\mathsf{s}^n(0) \mid n \geq 0\}$. For $+$ and $*$ we have the standard rules, we also need comparison cmp for which $\mathsf{cmp}(n, m)$ yields 0 if $n = m$, $\mathsf{s}(0)$ if $n > m$ and $\mathsf{s}(\mathsf{s}(0))$ if $n < m$. So R_d consists of the rules

$$\begin{array}{rcl} x + 0 & \to & x \\ x + \mathsf{s}(y) & \to & \mathsf{s}(x + y) \\ x * 0 & \to & 0 \\ x * \mathsf{s}(y) & \to & (x * y) + x \end{array} \qquad \begin{array}{rcl} \mathsf{cmp}(0,0) & \to & 0 \\ \mathsf{cmp}(\mathsf{s}(x),0) & \to & \mathsf{s}(0) \\ \mathsf{cmp}(0,\mathsf{s}(x)) & \to & \mathsf{s}(\mathsf{s}(0)) \\ \mathsf{cmp}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{cmp}(x,y) \end{array}$$

For R_s we need a function mul to multiply a stream element-wise by a number, a function mer for merging two sorted streams, and an auxiliary function f. Finally we have a constant h for the sorted stream of Hamming numbers. The rules of R_s read:

$$\begin{array}{rcl} \mathsf{mul}(x, y : \sigma) & \to & x * y : \mathsf{mul}(x, \sigma) \\ \mathsf{mer}(x : \sigma, y : \tau) & \to & \mathsf{f}(\mathsf{cmp}(x,y), x : \sigma, y : \tau) \\ & & \end{array} \qquad \begin{array}{rcl} \mathsf{f}(0, x : \sigma, y : \tau) & \to & x : \mathsf{mer}(\sigma, \tau) \\ \mathsf{f}(\mathsf{s}(0), \sigma, y : \tau) & \to & y : \mathsf{mer}(\sigma, \tau) \\ \mathsf{f}(\mathsf{s}(\mathsf{s}(x)), y : \sigma, \tau) & \to & y : \mathsf{mer}(\sigma, \tau) \end{array}$$

$$\mathsf{h} \to \mathsf{s}(0) : \mathsf{mer}(\mathsf{mer}(\mathsf{mul}(\mathsf{s}^2(0), \mathsf{h}), \mathsf{mul}(\mathsf{s}^3(0), \mathsf{h})), \mathsf{mul}(\mathsf{s}^5(0), \mathsf{h}))$$

Now we have a proper stream specification, being the folklore functional program for generating Hamming numbers, up to notational details. Productivity is proved fully automatically by our tool: μ-Term [13] is called together with the context-sensitivity information that rewriting is disallowed in the second argument of ':', yielding a proof of context-sensitive termination. So by Theorem 4.1 productivity can be concluded.

For completeness we mention that the tool of [6, 4] also finds a proof of productivity of h in this example.

Example 4.4. The Calkin–Wilf tree [3] is a binary tree in which every node is labeled by a pair of natural numbers. The root is labeled by $(1, 1)$, and every node labeled by (m, n) has children labeled by $(m, m + n)$ and $(m + n, n)$. It can be proved that for all natural numbers $m, n > 0$ that are relatively prime the pair (m, n) occurs exactly once as a label of a node, and no other pairs occur. So the labels of the nodes represent positive rational numbers, and every positive rational number m/n occurs exactly once as a pair (m, n). There is one constructor b with $\mathsf{ar}(d, \mathsf{b}) = \mathsf{ar}(s, \mathsf{b}) = 2$. From Example 4.3 we take the data set $\mathcal{D}$ consisting of the natural numbers, and also the symbol $+$ and its two rules. Now the Calkin–Wilf tree c is defined by

$$\mathsf{c} \to \mathsf{f}(\mathsf{s}(0), \mathsf{s}(0)), \quad \mathsf{f}(x, y) \to \mathsf{b}(x, y, \mathsf{f}(x, x + y), \mathsf{f}(x + y, y)).$$

Our tool proves productivity of this specification by calling μ-Term [13] that proves context-sensitive termination, hence proving productivity by Theorem 4.1.

Theorem 4.1 can be seen as a strengthening of Theorem 3.4: if all roots of right-hand sides of rules from R_s are in $\mathcal{C}$ then $R_s \cup R_d$ is μ-terminating, as is shown in the following proposition.

Proposition 4.5. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ be a proper specification in which for every $\ell \to r$ in R_s the term r is not a variable and $\mathsf{root}(r) \in \mathcal{C}$. Then $R_s \cup R_d$ is μ-terminating.*

Proof. Assume there exists an infinite μ-reduction. For every term in this reduction count the number of symbols from Σ_s that are on allowed positions. Due to the assumptions by every R_d-step this number remains the same, while by every R_s-step this number decreases by one. So this reduction contains only finitely many R_s-steps. After these finitely many R_s-steps an infinite R_d-reduction remains, contradicting the assumption that R_d is terminating. ∎

The reverse direction of Theorem 4.1 does not hold, as is illustrated in the next example.

Example 4.6. Consider the proper (stream) specification $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$, where $\Sigma_d = \{0, 1\}$, $R_d = \emptyset$, $\mathcal{C} = \{:\}$ with $\mathsf{ar}(d, :) = \mathsf{ar}(s, :) = 1$, and R_s being the below TRS:

$$\begin{array}{rcl} \mathsf{p} & \to & \mathsf{zip}(\mathsf{alt}, \mathsf{p}) \\ \mathsf{alt} & \to & 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : \sigma, \tau) & \to & x : \mathsf{zip}(\tau, \sigma) \end{array}$$

This specification is productive, as we will see later in Example 5.2. However, it admits an infinite context-sensitive reduction $\mathsf{p} \to \mathsf{zip}(\mathsf{alt}, \mathsf{p})$ which is continued by repeatedly reducing the redex p.

The stream p describes the sequence of right and left turns in the well-known *dragon curve*, obtained by repeatingly folding a paper ribbon in the same direction.

5. Transformations for Proving Productivity

To be able to handle examples like the above, we investigate transformations of such specifications for which productivity of the original system can be concluded from productivity of the transfomed one. Whenever productivity of a specification cannot be determined directly, then we apply one of these transformations and try to prove productivity of the transformed specification, instead.

One such transformation is the reduction of right-hand sides, that is, a rule $\ell \to r$ of R_s is replaced by $\ell \to r'$ for a term r' satisfying $r \to^*_{R_s \cup R_d} r'$. Write $R = R_s \cup R_d$, and write R' for the result of this replacement. Then by construction we have $\to_{R'} \subseteq \to^+_R$, and $\to_R \subseteq \to_{R'} \cdot \leftarrow^*_R$, that is, every $\to_R$-step can be followed by zero or more $\to_R$-steps to obtain a $\to_{R'}$-step. We present our theorems in this more general setting such that they are applicable more generally than only for reduction of right-hand sides.

Theorem 5.1. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ and $\mathcal{S}' = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R'_s)$ be proper specifications satisfying $\to_{R'} \subseteq \to^+_R$ for $R = R_s \cup R_d$ and $R' = R'_s \cup R_d$. If $\mathcal{S}'$ is productive, then $\mathcal{S}$ is productive, too.*

Proof. Let $\mathcal{S}'$ be productive, i.e., every ground term t of sort s admits a reduction $t \to^*_{R'} t'$ for which $\mathsf{root}(t') \in \mathcal{C}$. Then by $\to_{R'} \subseteq \to^+_R$ we conclude $t \to^*_R t'$, proving productivity of $\mathcal{S}$. ∎

Example 5.2. We apply this theorem to Example 4.6. Observe that we can rewrite the right-hand side of the rule $\mathsf{p} \to \mathsf{zip}(\mathsf{alt}, \mathsf{p})$ as follows:

$$\mathsf{zip}(\mathsf{alt}, \mathsf{p}) \to \mathsf{zip}(0 : 1 : \mathsf{alt}, \mathsf{p}) \to 0 : \mathsf{zip}(\mathsf{p}, 1 : \mathsf{alt})$$

So we may transform our specification by replacing R_s by the TRS R'_s consisting of the following rules:

$$\begin{array}{rcl} \mathsf{p} & \to & 0 : \mathsf{zip}(\mathsf{p}, 1 : \mathsf{alt}) \\ \mathsf{alt} & \to & 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : \sigma, \tau) & \to & x : \mathsf{zip}(\tau, \sigma) \end{array}$$

Clearly, this is a proper specification that is productive due to Theorem 3.4. Now productivity of the original specification follows from Theorem 5.1 and $\to_{R'_s} \subseteq \to^+_{R_s}$. Our tool finds exactly this proof.

Concluding productivity of the original system from productivity of the transformed system is called *soundness*, the converse is called *completeness*. The following example shows the incompleteness of Theorem 5.1.

Example 5.3. Consider the two proper (stream) specifications $\mathcal{S}$ and $\mathcal{S}'$ defined by

$$R_s\colon \quad \begin{array}{rcl} \mathsf{c} & \to & \mathsf{f}(\mathsf{c}) \\ \mathsf{f}(\sigma) & \to & 0 : \sigma \end{array} \qquad\qquad R'_s\colon \quad \begin{array}{rcl} \mathsf{c} & \to & \mathsf{f}(\mathsf{c}) \\ \mathsf{f}(x : \sigma) & \to & 0 : x : \sigma \end{array}$$

Here $\mathcal{C} = \{:\}$, $R_d = \emptyset$, $\Sigma_d = \{0\}$. Since $\mathsf{c} \to_R \mathsf{f}(\mathsf{c}) \to_R 0 : \mathsf{c}$ and $\mathsf{f}(\cdots) \to_R 0 : \cdots$ we conclude productivity of $\mathcal{S}$, as c and f are the only symbols in Σ_s.

For the TRS R'_s we have that $\to_{R'_s} \subseteq \to^+_{R_s}$, since any step with the rule $\mathsf{f}(x : \sigma) \to 0 : x : \sigma$ of R'_s can also be done with the rule $\mathsf{f}(\sigma) \to 0 : \sigma$ of R_s. However, $\mathcal{S}'$ is not productive, as the only reduction starting in c is $\mathsf{c} \to \mathsf{f}(\mathsf{c}) \to \mathsf{f}(\mathsf{f}(\mathsf{c})) \to \cdots$ in which the root is never in $\mathcal{C}$.

Next we prove that with the extra requirement $\to_R \subseteq \to_{R'} \cdot \leftarrow^*_R$, as holds for reduction of right-hand sides, we have both soundness and completeness.

Theorem 5.4. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ and $\mathcal{S}' = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R'_s)$ be proper specifications satisfying $\to_{R'} \subseteq \to^+_R$ and $\to_R \subseteq \to_{R'} \cdot \leftarrow^*_R$ for $R = R_s \cup R_d$ and $R' = R'_s \cup R_d$.*

Then $\mathcal{S}$ is productive if and only if $\mathcal{S}'$ is productive.

Proof. The "if" direction follows from Theorem 5.1.

For the "only-if" direction first we prove the following claim:

> **Claim:** If $t \to_R t'$ and $t \to^*_R t''$, then there exists a term v satisfying $t' \to^*_R v$ and $t'' \to^*_{R'} v$.

Let $t \to_R t'$ be an application of the rule $\ell \to r$ in R, so $t = C[\ell\rho]$ and $t' = C[r\rho]$ for some C, ρ. According to the Parallel Moves Lemma ([17], Lemma 4.3.3, page 101), we can write $t'' = C''[\ell\rho_1, \ldots, \ell\rho_n]$, and t', t'' have a common R-reduct $C''[r\rho_1, \ldots, r\rho_n]$. Due to $\ell\rho_i \to_R r\rho_i$ and $\to_R \subseteq \to_{R'} \cdot \leftarrow^*_R$ there exist t_i satisfying $\ell\rho_i \to_{R'} t_i$ and $r\rho_i \to^*_R t_i$, for all $i = 1, \ldots, n$. Now choosing $v = C''[t_1, \ldots, t_n]$ proves the claim.

Using this claim, by induction on the number of $\to_R$-steps from t to t' one proves the generalized claim: If $t \to^*_R t'$ and $t \to^*_R t''$, then there exists a term v satisfying $t' \to^*_R v$ and $t'' \to^*_{R'} v$.

Let t be an arbitrary ground term of sort s. Due to productivity of $\mathcal{S}$ there exists t' satisfying $t \to^*_R t'$ and $\mathsf{root}(t') \in \mathcal{C}$. Applying the generalized claim for $t'' = t$ yields a term

v satisfying $t' \to_R^* v$ and $t \to_{R'}^* v$. Since $\mathsf{root}(t') \in \mathcal{C}$ and $t' \to_R^* v$ we obtain $\mathsf{root}(v) \in \mathcal{C}$. Now $t \to_{R'}^* v$ implies productivity of $\mathcal{S}'$. ∎

Example 5.3 generalizes to a general application of Theorem 5.1 other than rewriting right-hand sides as follows. Assume a rule from R_s in a proper transformation contains an s-variable σ in the left-hand side being an argument of the root. Then for every $c \in \mathcal{C}$ this rule may be replaced by an instance of the same rule, obtained by replacing σ by $c(x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_m)$, where $\mathsf{ar}(d, c) = n, \mathsf{ar}(s, c) = m$. If this is done simultaneously for every $c \in \mathcal{C}$, so replacing the original rule by $\#\mathcal{C}$ instances, then the result is again a proper specification. Also the requirements of Theorem 5.1 hold, even $\to_{R'} \subseteq \to_R$. We show this transformation by an example.

Example 5.5. We want to analyze productivity of the following variant of Example 4.6, in which p has been replaced by a stream function, and R_s is the below TRS:

$$\begin{array}{rcl} \mathsf{p}(\sigma) & \to & \mathsf{zip}(\sigma, \mathsf{p}(\sigma)) \\ \mathsf{alt} & \to & 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : \sigma, \tau) & \to & x : \mathsf{zip}(\tau, \sigma) \end{array}$$

Proving productivity by Theorem 3.4 fails. Also proving productivity with the technique of Theorem 4.1 fails, since there exists the infinite context-sensitive reduction

$$\mathsf{p}(\mathsf{alt}) \to \mathsf{zip}(\mathsf{alt}, \underbrace{\mathsf{p}(\mathsf{alt})}) \to \ldots.$$

Furthermore, reducing the right-hand side of $\mathsf{p}(\sigma) \to \mathsf{zip}(\sigma, \mathsf{p}(\sigma))$ can only be done by applying the first rule, not creating a constructor as the root of the right-hand side. What blocks rewriting using the zip rule is the variable σ in the first argument of zip. Therefore, we apply Theorem 5.1 as sketched above, note that $\mathcal{C} = \{:\}$, and replace the rule $\mathsf{p}(\sigma) \to \mathsf{zip}(\sigma, \mathsf{p}(\sigma))$ by the single rule $\mathsf{p}(x : \sigma) \to \mathsf{zip}(x : \sigma, \mathsf{p}(x : \sigma))$ to obtain the TRS R_s'. This now allows us to rewrite the new right-hand side by the zip rule, replacing the previous rule by $\mathsf{p}(x : \sigma) \to x : \mathsf{zip}(\mathsf{p}(x : \sigma), \sigma)$, i.e., we obtain the TRS R_s'' consisting of the following rules:

$$\begin{array}{rcl} \mathsf{p}(x : \sigma) & \to & x : \mathsf{zip}(\mathsf{p}(x : \sigma), \sigma) \\ \mathsf{alt} & \to & 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : \sigma, \tau) & \to & x : \mathsf{zip}(\tau, \sigma) \end{array}$$

Productivity of R_s'' follows from Theorem 3.4. This implies productivity of R_s' due to Theorem 5.1 which in turn implies productivity of our initial specification $\mathcal{S}$, again due to Theorem 5.1. Our tool finds exactly the proof as given here.

Example 5.6. For stream computations it is often natural also to use finite lists. The data structure combining streams and finite lists is obtained by choosing $\mathcal{C} = \{:, \mathsf{nil}\}$, with $\mathsf{ar}(d, :) = \mathsf{ar}(s, :) = 1$ and $\mathsf{ar}(d, \mathsf{nil}) = \mathsf{ar}(s, \mathsf{nil}) = 0$, as mentioned in Example 2.3. An example using this is defining the sorted stream $p = 1 : 2 : 2 : 3 : 3 : 3 : 4 : \cdots$ of natural numbers, in which n occurs exactly n times for every $n \in \mathbf{N}$. This stream can be defined by a specification not involving finite lists, but here we show how to do it in this extended data structure based on standard operations like conc. Apart from conc we use copy, for which $\mathsf{copy}(k, n)$ is the finite list of k copies of n, for $k, n \in \mathbf{N}$, and a function f for generating $\mathsf{p} = \mathsf{f}(0)$. Taking $\mathcal{D}$ to be the set of ground terms over $\{0, \mathsf{s}\}$ and $R_d = \emptyset$, we choose R_s to consist of the following rules:

$$\begin{array}{rclrcl} \mathsf{p} & \to & \mathsf{f}(0) & \mathsf{f}(x) & \to & \mathsf{conc}(\mathsf{copy}(x,x),\mathsf{f}(\mathsf{s}(x))) \\ \mathsf{copy}(\mathsf{s}(x),y) & \to & y:\mathsf{copy}(x,y) & \mathsf{conc}(\mathsf{nil},\sigma) & \to & \sigma \\ \mathsf{copy}(0,x) & \to & \mathsf{nil} & \mathsf{conc}(x:\sigma,\tau) & \to & x:\mathsf{conc}(\sigma,\tau) \end{array}$$

Note that productivity of this system is not trivial: if the rule for f is replaced by $\mathsf{f}(x) \to \mathsf{conc}(\mathsf{copy}(x,x),\mathsf{f}(x))$, then the system is not productive.

Productivity cannot be proved directly by Theorem 3.4 or Theorem 4.1; context-sensitive termination does not even hold for the single f rule. However by replacing the f rule by the two instances

$$\mathsf{f}(0) \to \mathsf{conc}(\mathsf{copy}(0,0),\mathsf{f}(\mathsf{s}(0))) \quad \text{and} \quad \mathsf{f}(\mathsf{s}(x)) \to \mathsf{conc}(\mathsf{copy}(\mathsf{s}(x),\mathsf{s}(x)),\mathsf{f}(\mathsf{s}(\mathsf{s}(x)))),$$

and then applying rewriting right-hand sides by which these two rules are replaced by

$$\mathsf{f}(0) \to \mathsf{f}(\mathsf{s}(0)) \quad \text{and} \quad \mathsf{f}(\mathsf{s}(x)) \to \mathsf{s}(x):\mathsf{conc}(\mathsf{copy}(x,\mathsf{s}(x)),\mathsf{f}(\mathsf{s}(\mathsf{s}(x))))$$

yields a proper specification for which context-sensitive termination is proved by AProVE [8] or μ-Term [13], proving productivity of the original example by Theorem 5.1 and Theorem 4.1. Our tool finds a similar proof as given here: right-hand sides were slightly more rewritten.

Example 5.7. We conclude this section by an example in binary trees, in which the nodes are labeled by natural numbers, so there is one constructor $\mathsf{b} : d \times s^2 \to s$ and $\mathcal{D}$ consists of ground terms over $\{0,\mathsf{s}\}$. The rules are

$$\begin{array}{rclrcl} \mathsf{c} & \to & \mathsf{b}(0,\mathsf{f}(\mathsf{g}(0),\mathsf{left}(\mathsf{c})),\mathsf{g}(0)) & \mathsf{left}(\mathsf{b}(x,xs,ys)) & \to & xs \\ \mathsf{g}(x) & \to & \mathsf{b}(x,\mathsf{g}(\mathsf{s}(x)),\mathsf{g}(\mathsf{s}(x))) & \mathsf{f}(\mathsf{b}(x,xs,ys),zs) & \to & \mathsf{b}(x,ys,\mathsf{f}(zs,xs)) \end{array}$$

To get an impression of the hardness of this example, observe that f and left are similar to zip and tail for streams, respectively, and the recursion in the rule for c has the flavor of $\mathsf{c} \to 0:\mathsf{zip}(\cdots,\mathsf{tail}(\mathsf{c}))$. Our tool proves productivity by Theorem 5.1 and Theorem 4.1, by first rewriting right-hand sides and then proving context-sensitive termination.

6. Implementation

We have implemented a tool to check productivity of proper specifications using the techniques presented in this paper. It is accessible via the web-interface

`http://pclin150.win.tue.nl:8080/productivity`.

The input format requires the following ingredients:

- the variables,
- the operation symbols with their types,
- the rewrite rules.

Details of the format can be seen from the examples that are available. All other information, like which symbols are in $\mathcal{C}$ is extracted by the tool from these ingredients.

As a first step, the tool checks that the input is indeed a proper specification. Checking syntactic requirements, such as no function symbol returning sort d has an argument of sort s, the TRS is 2-sorted and orthogonal, and the left-hand sides have the required shape, are all straightforward. However, to verify the last requirement of a proper specification, namely that the TRS is exhaustive, is a hard job if we allow $\mathcal{D}$ to be the set of ground normal forms of any terminating orthogonal R_d. Instead we restrict to the class of proper specifications in which $\mathcal{D}$ consists of the constructor ground terms of sort d, i.e., the terms in $\mathcal{D}$ do not contain symbols occurring as root symbol in a left-hand side of a rule in R_d. To check whether this is the case, we use anti-matching as described in [14]. It can easily be shown that the normal forms of ground terms w.r.t. R_d are only constructor terms if and only if there is no anti-matching term that has a defined symbol as root and only terms built from constructors and variables as arguments. The idea of the proof is that such a term could be instantiated to a ground term, which is a normal form due to the anti-matching property. Then, checking exhaustiveness of R_s has to only consider constructor terms for both data and structure arguments.

To analyze productivity of a given proper specification, the tool first investigates whether Theorem 3.4 can be applied directly: it checks whether the roots of all right-hand sides are constructors. If this simple criterion does not hold, then it tries to show context-sensitive termination using the existing termination prover μ-Term, by which productivity will follow by Theorem 4.1.

If both of these first attempts fail then the tool tries to transform the given specification. Since rewriting of right-hand sides is both sound and complete, as was shown in Section 5, a productive specification can never be transformed into an unproductive one by this technique. Therefore, this is the first transformation to try. However, large right-hand sides often make it harder for termination tools to prove context-sensitive termination. Therefore, the tool tries to only rewrite positions on right-hand sides that appear to be needed to obtain a constructor prefix tree of a certain, adjustable depth. This is done by traversing the term in an outermost fashion and only trying to rewrite arguments if the possibly matching rules require a constructor for that particular argument. If at least one right-hand side could be rewritten, a new specification with the rewritten right-hand sides is created. Since rewriting of right-hand sides is not guaranteed to terminate, we limit the maximal number of rewriting steps. After rewriting the right-hand sides in this way, the tool again tries to prove productivity of the transformed TRS using our basic techniques.

As shown in Examples 5.5 and 5.6, it can be helpful to replace a variable by all constructors of its sort applied to variables. Therefore, in case productivity could not be shown so far, it is tried to instantiate a variable on a position of a right-hand side that is required by the rules for the defined symbol directly above it. Then the instantiated right-hand sides are rewritten again to obtain new specifications for which productivity is analyzed further.

The described transformations are applied in the order of their presentation a number of times. If a set limit of applications of transformations is reached, the tool finally tries to rewrite to deeper context-prefixes on right-hand sides and does a final check for productivity, using a larger timeout value.

Using these heuristics the tool is able to automatically prove productivity of all productive examples presented in this paper. This especially includes the example of a stream specification given in the following section, which could not be proved to be productive by any other automated technique we are aware of.

7. Conclusions and Related Work

We have presented new techniques to prove productivity of specifications of infinite objects like streams. Until now several techniques were developed for proving productivity of stream specifications, but not for other infinite data structures like infinite trees or the combination of streams and finite lists. In this paper we gave several examples of applying our techniques to these infinite data structures. We implemented a tool by which productivity of all of these examples could be proved fully automatically. For the non-stream examples there are hardly other techniques to compare. For streams there are examples where our technique outperforms all earlier techniques. For instance, the techniques from [6, 4] fail to prove productivity of Example 4.2. For this example the technique from [19] succeeds, but this technique fails as soon binary stream operations come in like zip. To our knowledge our technique is the first that can deal with productivity for $\mathsf{f}(\mathsf{p})$ of the specification consisting of the combination of Example 4.6 (describing the paper folding stream) and the two rules $\mathsf{f}(0 : \sigma) \to \mathsf{f}(\sigma)$, $\mathsf{f}(1 : \sigma) \to 1 : \mathsf{f}(\sigma)$. Our tool first performs rewriting of the right-hand side of the p-rule and then proves context-sensitive termination by μ-Term. Note the subtlety in this example: as soon as a ground term t can be composed of which the interpretation as a stream contains only finitely many ones, then the system will not be productive for $\mathsf{f}(t)$. So as a consequence we conclude that the paper folding stream p contains infinitely many ones, as the specification is productive for $\mathsf{f}(\mathsf{p})$.

Some ideas in this paper are related to earlier observations. In [10] the observation was made that if right-hand sides of stream definitions have ':' as its root, then well-definedness can be concluded, comparable to what we did by Theorem 3.4, and can be concluded from friendly-nestingness in [4]. A similar observation can be made about process algebra, where a recursive specification is called *guarded* if right-hand sides can be rewritten to a choice among terms all having a constructor on top, see e.g. [2], Section 5.5. In that setting every specification has at least one solution, while guardedness also implies there is at most one solution ([2], Theorem 5.5.11). So guardedness implies well-definedness, being of the flavor of combining Theorem 3.4 with rewriting right-hand sides. From both of these observations we obtain well-definedness, which is a slightly weaker notion than productivity. An investigation of well-definedness for stream specifications based on termination was made in [18]. We want to stress that productivity is strictly stronger than well-definedness, which is shown by the stream specification $\mathsf{c} \to \mathsf{f}(\mathsf{c})$, $\mathsf{f}(x : \sigma) \to 0 : \mathsf{c}$, being well-defined but not productive.

As far as we know the relationship of productivity with context-sensitive termination as expressed in Theorem 4.1 is new. Some ingredients of this relationship were given before in [19] where productivity of stream specifications was related to outermost termination and in [7] where outermost termination was related to context-sensitive termination. An alternative way to proceed would have been a further elaboration of combining the approaches from [19] and [7] to prove productivity: if the combination of the specification and a particular overflow rule is outermost terminating, then the specificiation is productive. Here for proving outermost termination the approach from [7] can be used. However, for examples like Example 4.3 this approach fails, even in combination with rewriting right-hand sides, while context-sensitive termination can be proved by standard tools, proving productivity by Theorem 4.1. For the other way around we only found examples where the direct approach is successful, too, in combination with rewriting right-hand sides. Apart from these experiments some intuition why our approach is to be preferred is the following. By the technique from [7] to prove outermost termination by proving context-sensitive termination of a transformed system, the size of the system increases dramatically. If the goal is to prove productivity, compared to the approach of this paper it is quite a detour to first transform the problem to outermost termination and then use such a strongly expanding transformation to relate it to context-sensitive termination, while it can be done directly without such an expansion by Theorem 4.1. Summarizing, we do not expect that the power of our approach can be improved by extending it by trying to prove outermost termination of the specification extended by the overflow rule, neither for streams nor for other infinite data structures.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, UK, 1998.

[2] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, UK, 2009.

[3] N. Calkin and H. Wilf. Recounting the rationals. *American Mathematical Monthly*, 107(4):360–363, 2000.

[4] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-oblivious stream productivity. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08)*, volume 5330 of *Lecture Notes in Computer Science*, pages 79–96. Springer-Verlag, 2008. Web interface tool: `http://infinity.few.vu.nl/productivity/`.

[5] J. Endrullis, C. Grabmayer, and D. Hendriks. Complexity of Fractran and productivity. In *Proceedings of the 22th Conference on Automated Deduction (CADE'09)*, volume 5663 of *Lecture Notes in Computer Science*, pages 371–387. Springer-Verlag, 2009.

[6] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J.W. Klop. Productivity of stream definitions. In *Proceedings of the Conference on Fundamentals of Computation Theory (FCT '07)*, volume 4639 of *Lecture Notes in Computer Science*, pages 274–287. Springer-Verlag, 2007.

[7] J. Endrullis and D. Hendriks. From outermost to context-sensitive rewriting. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 2009.

[8] J. Giesl et al. AProVE. Web interface and download: `http://aprove.informatik.rwth-aachen.de`.

[9] J. Giesl and A. Middeldorp. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*, 14:329–427, 2004.

[10] R. Hinze. Functional pearl: streams and unique fixed points. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 189–200. ACM, 2008.

[11] A. Isihara. Productivity of algorithmic systems. In *SCSS 2008*, volume 08-08 of *RISC-Linz Report*, pages 81–95, 2008.

[12] A. Isihara. *Algorithmic Term Rewriting Systems.* PhD thesis, Free University Amsterdam, 2010.

[13] S. Lucas et al. μ-Term. Web interface and download: `http://zenon.dsic.upv.es/muterm/`.

[14] M. Raffelsieper and H. Zantema. A transformational approach to prove outermost termination automatically. In *Proceedings of the 8th International Workshop in Reduction Strategies in Rewriting and Programming (WRS'08)*, volume 237 of *Electronic Notes in Theoretical Computer Science*, pages 3–21. Elsevier Science Publishers B. V. (North-Holland), 2009.

[15] B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.

[16] J. G. Simonsen. The Π^0_2-completeness of most of the properties of rewriting systems you care about (and productivity). In R. Treinen, editor, *Proceedings of the 20th Conference on Rewriting Techniques and Applications (RTA)*, Lecture Notes in Computer Science. Springer, 2009.

[17] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, UK, 2003.

[18] H. Zantema. Well-definedness of streams by termination. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 164–178. Springer-Verlag, 2009.

[19] H. Zantema and M. Raffelsieper. Stream productivity by outermost termination. In *Proceedings of the 9th International Workshop in Reduction Strategies in Rewriting and Programming (WRS'09)*, volume 15 of *Electronic Proceedings in Theoretical Computer Science*, 2010.