Lazy Productivity via Termination

Jörg Endrullis, Dimitri Hendriks

Vrije Universiteit Amsterdam, Department of Computer Science,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Dedicated to Jan A. Bergstra on the occasion of his 60th birthday.

Abstract

We present a procedure for transforming strongly sequential constructor-based term rewriting systems (TRSs) into context-sensitive TRSs in such a way that productivity of the input system is equivalent to termination of the output system. Thereby automated termination provers become available for proving productivity. A TRS is called productive if all its finite ground terms are constructor normalizing, and all 'inductive constructor paths' through the resulting (possibly non-wellfounded) constructor normal form are finite. To our knowledge, this is the first complete transformation from productivity to termination.

The transformation proceeds in two steps: (i) The strongly sequential TRS is converted into a shallow TRS, where patterns do not have nested constructors. (ii) The shallow TRS is transformed into a context-sensitive TRS, where rewriting below constructors and in arguments not 'consumed from' is disallowed.

Furthermore, we show how lazy evaluation can be encoded by strong sequentiality, thus extending our transformation to, e.g., Haskell programs.

Finally, we present a simple, but fruitful extension of matrix interpretations to make them applicable for proving termination of context-sensitive TRSs.

Keywords: term rewriting, productivity, termination, strong sequentiality, lazy evaluation, context-sensitive rewriting, matrix interpretations

1. Introduction

An important aspect of program correctness is termination. When dealing with programs that construct or process infinite objects, termination cannot be required. But the question whether such a program really computes a unique total *object*, that is, a finite or infinite structure assembled from a given set of

Email addresses: joerg@few.vu.nl (Jörg Endrullis), diem@cs.vu.nl (Dimitri Hendriks)

'building blocks', remains valid. This is the question of productivity. Productivity captures the intuitive notion of unlimited progress, of working programs producing values indefinitely, programs immune to starvation.

In lazy functional programming languages such as Miranda [47], Clean [38] or Haskell [37], the usage of infinite data structures is common practice. For the correctness of programs dealing with such structures one must guarantee that every finite part of the infinite structure can be evaluated in finite time; that is, the program must be productive. This holds both for terminating programs that employ infinite structures (termination is then possible as only finite parts of the lazy evaluated infinite structures are queried), as well as for non-terminating programs that directly construct or process infinite objects.

We study productivity from a rewriting perspective, modelling programs by term rewriting systems and objects by constructor normal forms, which can be infinite. A productive TRS rewrites every finite ground term to a constructor term of a given inductive or coinductive datatype.

For some specifications, productivity is rather obvious. For instance the rule:

$$\mathsf{zeros} \to \mathsf{0} : \mathsf{zeros}$$

produces a constructor prefix $\mathsf{0} : \Box$ upon each unfolding of the constant zeros, thus in the limit resulting in the infinite stream $\mathsf{0}:\mathsf{0}:\mathsf{0}:\ldots$. Productivity of this example is already guaranteed by the syntactic criterium known as 'guardedness' [5, 19]: the recursive call in the right-hand side is guarded by the stream constructor ':'. For other specifications, where, for instance, there is a function application between guard and recursive call, productivity is less obvious. For example, consider the following stream specification $\mathcal{R}_{\mathsf{a}}$, taken from [9]:

$$\begin{aligned} \mathsf{a} &\to \mathsf{0} : \mathsf{f}(\mathsf{a}) \\ \mathsf{f}(\mathsf{0} : \sigma) &\to \mathsf{1} : \mathsf{0} : \mathsf{f}(\sigma) \\ \mathsf{f}(\mathsf{1} : \sigma) &\to \mathsf{f}(\sigma) \end{aligned} \qquad (\mathcal{R}_{\mathsf{a}})$$

Productivity of this TRS requires an inductive argument to show that every finite ground term t over the signature $\{\mathsf{0}, \mathsf{1}, :, \mathsf{a}, \mathsf{f}\}$ evaluates to a constructor normal form with infinitely many $\mathsf{0}$'s: once we have proved the claim for t, then, starting from $\mathsf{f}(t)$, f will always eventually read the symbol $\mathsf{0}$ and produce accordingly, leaving behind again a stream with infinitely many $\mathsf{0}$'s (and $\mathsf{1}$'s).

Contribution and outline.

We follow [55] by transforming productivity into a termination problem with respect to context-sensitive rewriting. Thereby termination tools become available for proving productivity. We present the first *complete* transformation, that is, productivity of the input specification is equivalent to termination of the transformed system. We can handle *all* examples from [55] directly, that is, without additional preprocessing steps as employed by [55].

The basic notions of (context-sensitive) rewriting that we need here are explained in Section 2. In Section 3, we formalize *tree specifications* as sorted,

orthogonal, exhaustive constructor TRSs, and define what it means for a tree specification to be productive. We define a transformation of 'strongly sequential' [21] tree specifications to context-sensitive term rewriting systems such that rewriting in the resulting system truthfully models lazy evaluation (root-needed reduction [36]) of the original specification. We present the transformation in a top-down fashion:

- In Section 4 we sketch how (first-order) functional programs can be rendered as strongly sequential tree specifications;
- In Section 5 we give a transformation from strongly sequential to shallow tree specifications;
- In Section 6 we show how lazy evaluation on shallow tree specifications (where pattern matching is only one constructor symbol deep; see Section 3) can be modelled using context-sensitive rewriting.

The use of lazy evaluation allows us to do away with the restriction of an 'independent data-layer' required in [11, 12, 9, 54, 55], meaning that symbols of inductive sort are disallowed having coinductive arguments. An independent data-layer excludes, e.g., the function $\mathsf{head}(x : y) \to x$ which takes a coinductive argument (a stream) and returns an element of an inductive sort.

Many tree specifications have an independent data-layer though. For such specifications our transformation can be simplified, resulting in smaller transformed systems; see Section 7.

In Section 8 we present a simple extension of matrix interpretations [22, 15] for proving termination of context-sensitive TRSs. The method is trivial to implement in any termination prover that has standard matrix interpretations: we drop the positivity requirement for the upper-left matrix entries of argument positions where rewriting is forbidden by the replacement map.

In Section 9 we discuss related work. We conclude in Section 10.

2. Preliminaries

For a thorough introduction to term rewriting and context-sensitive rewriting, we refer to [43] and [31], respectively. We recall some of the main definitions that we employ here, for the sake of completeness, and fix notations.

2.1. Many-sorted term rewriting, and constructor TRSs.

Definition 2.1 (Sortedness)**.** Let $\mathcal{S} = \{\tau_1, \ldots, \tau_k\}$ be a finite set of *sorts*. An *$\mathcal{S}$-sorted set* A is a family of sets $A = \{A_\tau\}_{\tau \in \mathcal{S}}$. For $\mathcal{S}' \subseteq \mathcal{S}$ we define $A_{\mathcal{S}'} := \bigcup_{\tau \in \mathcal{S}'} A_\tau$. We sometimes write $a \in A$ where we mean $a \in A_{\mathcal{S}}$, i.e., $a \in A_\tau$ for some $\tau \in \mathcal{S}$. An *$\mathcal{S}$-sorted map* is a map $f : A \to B$ between $\mathcal{S}$-sorted sets A and B such that $f(A_\tau) \subseteq B_\tau$ for all $\tau \in \mathcal{S}$.

An *$\mathcal{S}$-sorted signature* Σ is an $\mathcal{S}$-sorted set of symbols, where to every symbol $\mathsf{f} \in \Sigma_\tau$ is associated a fixed *type* $\langle \tau_1, \ldots, \tau_n, \tau \rangle$, where $\tau_1, \ldots, \tau_n \in \mathcal{S}$. For f with

type $\langle \tau_1, \ldots, \tau_n, \tau \rangle$ we write $\mathsf{f} :: \tau_1 \times \cdots \times \tau_n \to \tau$. We use $\sharp\mathsf{f}$ for the *arity of* f defined by $\sharp\mathsf{f} := n$. For constant symbols $\mathsf{a} \in \Sigma_\tau$ (with $\sharp\mathsf{a} = 0$) we simply write $\mathsf{a} :: \tau$. We let Σ range over $\mathcal{S}$-sorted signatures and $\mathcal{X}$ over $\mathcal{S}$-sorted sets of variables.

Definition 2.2 (Terms)**.** The set $Ter^\infty(\Sigma, \mathcal{X})$ of (potentially infinite) *terms over* Σ *and* $\mathcal{X}$, is defined as the $\mathcal{S}$-sorted set $Ter^\infty(\Sigma, \mathcal{X}) = \{ Ter^\infty(\Sigma, \mathcal{X})_\tau \}_{\tau \in \mathcal{S}}$, where $Ter^\infty(\Sigma, \mathcal{X})_\tau$ is coinductively defined[1] as follows:

(i) $\mathcal{X}_\tau \subseteq Ter^\infty(\Sigma, \mathcal{X})_\tau$
(ii) $\mathsf{f}(t_1, \ldots, t_n) \in Ter^\infty(\Sigma, \mathcal{X})_\tau$ for all symbols $\mathsf{f} :: \tau_1 \times \ldots \times \tau_n \to \tau$ and terms $t_1 \in Ter^\infty(\Sigma, \mathcal{X})_{\tau_1}, \ldots, t_n \in Ter^\infty(\Sigma, \mathcal{X})_{\tau_n}$.

A term is called *finite* if it is well-founded. The set of finite terms is denoted by $Ter(\Sigma, \mathcal{X})$. We write $Var(t)$ for the set of variables occurring in a term t.

Definition 2.3 (Positions)**.** The set of *positions* $Pos(t) \subseteq \mathbb{N}^*$ of a term $t \in Ter^\infty(\Sigma, \mathcal{X})$ is defined by: $Pos(x) = \{\epsilon\}$ for $x \in \mathcal{X}$ and $Pos(\mathsf{f}(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{ip \mid 1 \leq i \leq n, p \in Pos(t_i)\}$ for $\mathsf{f} \in \Sigma$ of arity n. We write $root(t)$ to denote the root symbol of t (at position ϵ), and $t|_p$ for the subterm of t rooted at position $p \in Pos(t)$, that is, $t|_\epsilon = t$, and $\mathsf{f}(t_1, \ldots, t_n)|_{pi} = t_i$. Then $root(t|_p)$ is the symbol at position p in t.

Definition 2.4 (Substitution)**.** A *substitution* is an $\mathcal{S}$-sorted map $\sigma : \mathcal{X} \to Ter^\infty(\Sigma, \mathcal{X})$ from variables to terms. For terms $t \in Ter^\infty(\Sigma, \mathcal{X})$ and substitutions σ, $t\sigma$ is defined by guarded recursion[2]: $x\sigma = \sigma(x)$ for $x \in \mathcal{X}$, and $\mathsf{f}(t_1, \ldots, t_n)\sigma = \mathsf{f}(t_1\sigma, \ldots, t_n\sigma)$ for $\mathsf{f} \in \Sigma$ of arity n, and $t_1, \ldots, t_n \in Ter^\infty(\Sigma, \mathcal{X})$.

Definition 2.5 (Contexts)**.** Contexts are defined as terms over the extended signature $\Sigma \cup \{\Box\}$, where $\Box$ is a fresh symbol serving as a *hole*.[3] We use $\mathcal{C}xt_n(\Sigma, \mathcal{X})$ to denote the set of *n-hole contexts over* Σ *and* $\mathcal{X}$: An n-hole context $C \in \mathcal{C}xt_n(\Sigma, \mathcal{X})$ is a term from $Ter^\infty(\Sigma \cup \{\Box\}, \mathcal{X})$ containing exactly n occurrences of $\Box$. Counting holes from left to right in C (i.e., by in-order traversal through the term tree), *filling* the i-th hole of C ($1 \leq i \leq n$) with a term $s \in Ter^\infty(\Sigma, \mathcal{X})$ is denoted by $C[s]_i$. For $C \in \mathcal{C}xt_1(\Sigma, \mathcal{X})$ we write $C[s]$ for $C[s]_1$. We define $\mathcal{C}xt(\Sigma, \mathcal{X}) = \bigcup_{n>0} \mathcal{C}xt_n(\Sigma, \mathcal{X})$, the set of all contexts over Σ and $\mathcal{X}$.

Definition 2.6 (TRS)**.** An *$\mathcal{S}$-sorted term rewriting system (TRS)* $\mathcal{R}$ is a pair $\langle \Sigma, R \rangle$ consisting of an $\mathcal{S}$-sorted signature Σ and an $\mathcal{S}$-sorted set R of *rewrite rules* such that:

(i) for all $\tau \in \mathcal{S}$: $R_\tau \subseteq Ter(\Sigma, \mathcal{X})_\tau \times Ter(\Sigma, \mathcal{X})_\tau$, and
(ii) $\ell \notin \mathcal{X}$ and $Var(r) \subseteq Var(\ell)$ for all $\langle \ell, r \rangle \in R$.

[1] That is, take the greatest fixed point of the underlying set functor.
[2] In the sense of [5, 19].
[3] To be precise, we extend the signature with a hole symbol for each sort $\tau \in \mathcal{S}$.

The rules $\langle \ell, r\rangle \in R$ are written $\ell \to r$, and we call ℓ the *left-hand side* and r *the right-hand side* of the rule.

Definition 2.7 (Metric)**.** On the set of terms $Ter^\infty(\Sigma, \mathcal{X})$ we define a *metric* d by $d(s,t) = 0$ whenever $s = t$, and $d(s,t) = 2^{-k}$ otherwise, where $k \in \mathbb{N}$ is the least length of all positions $p \in \mathbb{N}^*$ such that $root(s|_p) \neq root(t|_p)$.

Definition 2.8 (Rewriting)**.** For a TRS $\mathcal{R}$ we define $\to_\mathcal{R}$, the *rewrite relation* induced by $\mathcal{R}$ as follows: For terms $s, t \in Ter^\infty(\Sigma, \mathcal{X})$ we write $s \to_\mathcal{R} t$ if there exists a rule $\ell \to r \in \mathcal{R}$, a substitution σ, and a context $C \in \mathcal{C}xt_1(\Sigma, \mathcal{X})$ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. We write $s \to_{\mathcal{R},p} r$ to explicitly indicate the rewrite position p, i.e., when $root(C|_p) = \Box$. If $\mathcal{R}$ is clear from the context, we also write $s \to t$. We let $\to^*$ denote the reflexive-transitive closure of $\to$. A term of the form $\ell\sigma$, for some rule $\ell \to r \in \mathcal{R}$, and a substitution σ, is called a *redex*. We define $t{\downarrow_\mathcal{R}}$ to denote the *normal form of t with respect to $\mathcal{R}$* if it exists and is unique.

We say that s *rewrites in the limit to* t, denoted by $s \to^\omega t$ if there is an infinite rewrite sequence $s = s_0 \to_{p_0} s_1 \to_{p_1} \dots$ such that the depth $|p_i|$ of the rewrite steps tends to infinity and the term t is the limit of the sequence $s_0, s_1, \dots$ with respect to the metric d.

We do not consider rewrite sequences of length $> \omega$ since we only work with orthogonal term rewrite systems for which the compression lemma [43] ensures that every rewrite sequence can be compressed to length at most ω.

Definition 2.9 (Constructor TRS)**.** Let $\mathcal{R} = \langle \Sigma, R\rangle$ be a TRS. We define $\Sigma^\mathcal{D} := \{root(l) \mid l \to r \in R\}$, the set of *defined symbols*, and $\Sigma^\mathcal{C} = \Sigma \setminus \Sigma^\mathcal{D}$, the set of *constructor symbols*.

The system $\mathcal{R}$ is called a *constructor TRS* if for every rewrite rule $\rho \in R$, the left-hand side is of the form $\mathsf{f}(t_1, \dots, t_{\sharp\mathsf{f}})$ where all $t_1, \dots, t_{\sharp\mathsf{f}} \in Ter(\Sigma^\mathcal{C}, \mathcal{X})$ are constructor terms. Such a rule ρ is called a *defining rule for* f.

We call $\mathcal{R}$ *exhaustive for* $\mathsf{f} \in \Sigma$ if every term $\mathsf{f}(t_1, \dots, t_{\sharp\mathsf{f}})$ with (possibly infinite) closed constructor terms $t_i \in Ter^\infty(\Sigma^\mathcal{C}, \varnothing)$ $(1 \le i \le \sharp\mathsf{f})$ is a redex; $\mathcal{R}$ is *exhaustive* if it is exhaustive for all $\mathsf{f} \in \Sigma$.

Exhaustivity together with finitary strong normalization guarantees that finite ground terms rewrite to constructor normal forms, a property known as sufficient completeness [27]. However, we are typically interested in non-terminating specifications of infinite data structures. Note that exhaustivity together with infinitary strong normalization does not imply that every finite ground term rewrites to a constructor normal form.

2.2. Context-sensitive term rewriting.

Definition 2.10 (Context-sensitive TRS)**.** A *replacement map for* Σ is a family $\xi = \{\xi_\mathsf{f}\}_{\mathsf{f}\in\Sigma}$ of sets $\xi_\mathsf{f} \subseteq \{1, \dots, \sharp\mathsf{f}\}$. A *context-sensitive term rewriting system* is a pair $\langle \mathcal{R}, \xi\rangle$ consisting of a TRS $\mathcal{R} = \langle \Sigma, R\rangle$ and a replacement map ξ for Σ.

Two context-sensitive TRSs over a signature Σ are said to be *compatible* if their replacement maps coincide.

Definition 2.11 (Context-sensitive rewriting)**.** The set of ξ-*replacing positions* $Pos^\xi(t)$ of a term $t \in Ter^\infty(\Sigma, \mathcal{X})$ is defined by:

(i) $Pos^\xi(x) = \{\epsilon\}$ for $x \in \mathcal{X}$, and
(ii) $Pos^\xi(\mathsf{f}(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{ip \mid i \in \xi_\mathsf{f},\, p \in Pos^\xi(t_i)\}$.

Rewriting then is allowed only at ξ-replacing positions: s ξ-*rewrites to* t, denoted by $s \to_{\mathcal{R},\xi} t$, whenever $s \to_{\mathcal{R},p} t$ with $p \in Pos^\xi(s)$.

As an example, consider the system $\mathcal{R}$ consisting of the single rule:

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

and let ξ be given by $\xi_\mathsf{c} = \{2\}$. Clearly $\mathcal{R}$ is non-terminating, but the context-sensitive TRS $\langle \mathcal{R}, \xi \rangle$ *is* terminating, because the replacement map ξ allows rewriting only in the second argument of the symbol c.

Lemma 2.12. *Let $\langle \mathcal{R}, \xi \rangle$ be a context-sensitive TRS with $\mathcal{R} = \langle \Sigma, R \rangle$ such that there exists a finite ground term $t \in Ter(\Sigma, \varnothing)_\tau$ for every sort $\tau \in \mathcal{S}$. Then termination on all finite ground terms $Ter(\Sigma, \varnothing)$ coincides with termination on all finite terms $Ter(\Sigma, \mathcal{X})$.*

Proof. The direction '$\Leftarrow$' is trivial. For '$\Rightarrow$', note that every rewrite sequence over open terms can be mapped to a rewrite sequence (of equal length) of ground terms by applying a ground substitution to every term in the sequence. □

Of course, this applies to ordinary TRSs as well, by taking the replacement map ξ defined by $\xi_\mathsf{f} = \{1, \ldots, \sharp\mathsf{f}\}$ for all symbols $\mathsf{f} \in \Sigma$.

2.3. Termination and relative termination.

Definition 2.13. A binary relation $\succ \subseteq A \times A$ over a set A is called *well-founded* if no infinite decreasing sequence $a_1 \succ a_2 \succ a_3 \succ \ldots$ exists.

A (context-sensitive) TRS $\mathcal{R}$ is called *terminating* or *strongly normalizing*, denoted by $\mathrm{SN}(\mathcal{R})$, if $\to_\mathcal{R}$ is well-founded.

Definition 2.14. Let $\to_1, \to_2 \subseteq A \times A$ be binary relations. Then we say $\to_1$ *is terminating relative to* $\to_2$ if $\mathrm{SN}(\to_1/\to_2)$, where $\to_1/\to_2$ is defined by:

$$\to_1/\to_2 := \to_2^* \cdot \to_1 \cdot \to_2^*$$

Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be (context-sensitive) TRSs over Σ. Then $\mathcal{R}_1$ is *terminating relative to* $\mathcal{R}_2$, denoted by $\mathrm{SN}(\mathcal{R}_1/\mathcal{R}_2)$, if $\to_{\mathcal{R}_1}$ is terminating relative to $\to_{\mathcal{R}_2}$.

If $\mathcal{R}_1$ is terminating relative to $\mathcal{R}_2$ this means there is no term t that admits an infinite rewrite sequence $t = t_1 \to_{\mathcal{R}_1 \cup \mathcal{R}_2} t_2 \to_{\mathcal{R}_1 \cup \mathcal{R}_2} \ldots$ containing an infinite number of $\to_{\mathcal{R}_1}$ steps. Clearly we also have that $\mathrm{SN}(\mathcal{R})$ if and only if $\mathrm{SN}(\mathcal{R}/\varnothing)$.

2.4. Persistence.

The notion of persistence has been introduced by Zantema in [49]. A property P is called *persistent* if for every many-sorted TRSs $\mathcal{R}$ it holds: $\mathcal{R}$ has P if and only if $\Theta(\mathcal{R})$ has P. Here $\Theta(\mathcal{R})$ denotes the TRS obtained from $\mathcal{R}$ by dropping sorts (mapping all sorts to a single sort).

The following lemma generalizes [26, Theorem 4.4] to context-sensitive rewriting. A related result on the modularity of termination for the disjoint union of left-linear, confluent TRSs has appeared in [46].

Lemma 2.15. *Termination is persistent for orthogonal context-sensitive TRSs.*

Proof sketch. Let $\langle \mathcal{R}, \xi \rangle$ be a terminating, many-sorted, orthogonal context-sensitive TRS. We show that termination is preserved when dropping sorts.

We follow the line of argument of [46]. We partition non-well-sorted terms into (non-overlapping) well-sorted (multi-hole) contexts such that at every hole occurrence (that is, the transition between the contexts) there is a sort-conflict. The *rank* of an unsorted term is the nesting depth of these partitions.

We make some immediate observations: The pattern of a rule always lies entirely within one of these partitions, due to well-sortedness. Moreover, as rewriting respects sorts, it follows that rewriting does not increase the rank of a term, and the only way to resolve a sort-conflict is if a partition collapses to one of its holes. We now take a closer look at collapsing partitions.

Consider a collapsing partition, that is, a non-trivial context $C[\Box_1, \ldots, \Box_n]$ which admits a rewrite sequence to one of its holes $\Box_i$. Then no rule of $\mathcal{R}$ overlaps with $C[\Box_1, \ldots, \Box_n]$ in such a way that a part of the rule pattern is partially above the context, or partially matched by $\Box_i$. For otherwise, there are overlapped symbols on the path from the root of $C[\Box_1, \ldots, \Box_n]$ to $\Box_i$, which by orthogonality cannot be rewritten by any rewrite sequence within the context, contradicting that the context rewrites to $\Box_i$. This observation is important as it guarantees that the partition does not interact with the term above itself or within the hole $\Box_i$ even if the collapse of other partitions resolves the sort-conflict at these positions. As a consequence, we can remove (collapse) collapsing partitions without affecting the reduction above it (here we also use left-linearity).

Assume there exists a non-well-sorted term t that is non-terminating. Let t have minimal rank among all these terms. Then by minimality it follows that t admits a rewrite sequence containing infinitely many rewrite steps in the topmost partition. The above observation on collapsing partitions allows us to remove (collapse) collapsing, non-topmost partitions without affecting the existence of a reduction with infinitely many steps in the topmost context. Thus let t' be obtained from t by collapsing all non-topmost, collapsing partitions. Then by orthogonal projection, the infinite rewrite sequence in t gives rise to an infinite rewrite sequence in t', also containing infinitely many steps in its topmost partition. However, as t' contains no collapsing partitions, there cannot be any interaction between the partitions. Then, in particular, the topmost partition itself is a well-sorted term that admits an infinite reduction, contradicting the assumption that $\mathcal{R}$ is terminating. □

2.5. Call-by-need and root-neededness.

For a detailed study of call-by-need reductions and root-neededness, we refer the reader to [36]. We briefly recall the main definitions.

Definition 2.16. Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a TRS. A term $t \in \mathit{Ter}^\infty(\Sigma, \mathcal{X})$ is *root-stable* if it cannot be rewritten to a redex. A reduction strategy is *root-normalizing* if it leads to a root-stable form for every term that has a root-stable reduct.

Definition 2.17 ([36, Definition 4.1])**.** Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a TRS. A redex ρ in a term $t \in \mathit{Ter}(\Sigma, \mathcal{X})$ is *root-needed* if in every rewrite sequence from t to a root-stable term a descendant of ρ is contracted.

Contracting only root-needed redexes is root-normalizing:

Theorem 2.18 ([36, Corollary 5.7])**.** *Root-needed reduction is root-normalizing for orthogonal term rewriting systems.* □

2.6. Σ-algebras and models.

Let Σ be an unsorted signature, that is, an $\mathcal{S}$-sorted signature for a singleton set $\mathcal{S}$. We give the definition of Σ-algebra:

Definition 2.19. A Σ-*algebra* $\mathcal{A} = \langle A, [\![\cdot]\!] \rangle$ consists of a non-empty set A, called the *domain of* $\mathcal{A}$, and for each n-ary symbol $\mathsf{f} \in \Sigma$ a function $[\![\mathsf{f}]\!] : A^n \to A$, called the *interpretation of* f. Given an *assignment* $\alpha : \mathcal{X} \to A$ of the variables to A, the *interpretation of a term* $t \in \mathit{Ter}(\Sigma, \mathcal{X})$ *with respect to* α is denoted by $[\![t, \alpha]\!]$ and inductively defined by:

$$[\![x, \alpha]\!] = \alpha(x) \qquad [\![\mathsf{f}(t_1, \ldots, t_n), \alpha]\!] = [\![\mathsf{f}]\!]([\![t_1, \alpha]\!], \ldots, [\![t_n, \alpha]\!])$$

For ground terms $t \in \mathit{Ter}(\Sigma, \varnothing)$ we write $[\![t]\!]$ for short.

We introduce *models with respect to a relation*:

Definition 2.20. Let $\mathcal{R}$ be a context-sensitive TRS over Σ, $\mathcal{A} = \langle A, [\![\cdot]\!] \rangle$ be a Σ-algebra, and $\succ \subseteq A \times A$ a binary relation. Then $\langle \mathcal{A}, \succ \rangle$ is a *model for* $\mathcal{R}$ if $[\![\ell, \alpha]\!] \succ [\![r, \alpha]\!]$ for all rules $\ell \to r \in \mathcal{R}$ and assignments $\alpha : \mathit{Var}(\ell) \to A$. If the Σ-algebra $\mathcal{A}$ is clear from the context, we say that $\succ$ *is a model for* $\mathcal{R}$.

Note that $\mathcal{A}$ is a model in the sense of [50] if and only if $\langle \mathcal{A}, = \rangle$ is a model according to the above definition.

3. Tree Specifications and Productivity

3.1. Tree Specifications.

The set of sorts $\mathcal{S}$ is partitioned into $\mathcal{S} = \mu \cup \nu$, where μ is the set of *data sorts*, intended to model inductive data types such as booleans, natural numbers, finite lists, and so on; ν is the set of *codata sorts*, intended for coinductive datatypes

such as streams and infinite trees. A term t of sort τ is called a *data term* if $\tau \in \mu$, and a *codata term* if $\tau \in \nu$. We use $x, y, z, \ldots$ to range over data variables, and use greek letters for codata variables.

Let $\mathsf{f} :: \tau_1 \times \ldots \times \tau_n \to \tau$. We write $\sharp_\mu \mathsf{f}$ and $\sharp_\nu \mathsf{f}$ to denote the *data* and *codata arity* of f, respectively, defined by $\sharp_\varphi \mathsf{f} = |\{i \mid 1 \leq i \leq n\,, \tau_i \in \varphi\}|$ for $\varphi = \mu, \nu$; so $\sharp \mathsf{f} = \sharp_\mu \mathsf{f} + \sharp_\nu \mathsf{f}$. We assume all data arguments are in front, i.e., $\mathsf{g} :: \tau_1 \times \ldots \times \tau_m \times \gamma_1 \times \ldots \times \gamma_k \to \tau$ for all $\mathsf{g} \in \Sigma$ with $\sharp_\mu \mathsf{g} = m$ and $\sharp_\nu \mathsf{g} = k$ (hence $\tau_i \in \mu$ and $\gamma_j \in \nu$, for all $1 \leq i \leq m$ and $1 \leq j \leq k$).

Definition 3.1. Let μ and ν be disjoint sets of sorts. A *tree specification* is a $(\mu \cup \nu)$-sorted, orthogonal, exhaustive constructor term rewriting system.

Example 3.2. The Fibonacci word is the infinite sequence $0100101001001\ldots$, which can be defined as the limit of iterating the (non-uniform) morphism $h : \{0,1\}^* \to \{0,1\}^*$, defined by $h(0) = 01$ and $h(1) = 0$ starting on the word 0.

We give a tree specification $\mathcal{R}_{\mathsf{fib}} = \langle \Sigma, R \rangle$ computing the Fibonacci word. We use a sort B for bits $\mathsf{0}, \mathsf{1}$, and a sort S for streams of bits; so $\mu = \{\mathsf{B}\}$ and $\nu = \{\mathsf{S}\}$. Let $\Sigma = \{\mathsf{0}, \mathsf{1}, :, \mathsf{fib}, \mathsf{h}, \mathsf{tail}\}$ with types:

$$\mathsf{0}, \mathsf{1} :: \mathsf{B} \qquad : :: \mathsf{B} \times \mathsf{S} \to \mathsf{S} \qquad \mathsf{h}, \mathsf{tail} :: \mathsf{S} \to \mathsf{S} \qquad \mathsf{fib} :: \mathsf{S}$$

and let R consist of the rewrite rules:

$$\begin{aligned} \mathsf{fib} &\to \mathsf{h}(\mathsf{0} : \mathsf{tail}(\mathsf{fib})) & \mathsf{h}(\mathsf{0} : \sigma) &\to \mathsf{0} : \mathsf{1} : \mathsf{h}(\sigma) \\ \mathsf{tail}(x : \sigma) &\to \sigma & \mathsf{h}(\mathsf{1} : \sigma) &\to \mathsf{0} : \mathsf{h}(\sigma) \end{aligned}$$

The partitions of Σ are $\Sigma^{\mathcal{C}}_\mu = \{\mathsf{0}, \mathsf{1}\}$, $\Sigma^{\mathcal{D}}_\mu = \varnothing$, $\Sigma^{\mathcal{C}}_\nu = \{:\}$, and $\Sigma^{\mathcal{D}}_\nu = \{\mathsf{fib}, \mathsf{h}, \mathsf{tail}\}$. By the transformations introduced in the sequel, we can show that this tree specification is productive indeed. See Examples 5.2 and 6.7. This ensures that, e.g., the term fib rewrites in ω many steps to an infinite constructor normal form (the Fibonacci word):

$$\mathsf{fib} \to^\omega \mathsf{0} : \mathsf{1} : \mathsf{0} : \mathsf{0} : \mathsf{1} : \mathsf{0} : \mathsf{1} : \mathsf{0} : \mathsf{0} : \mathsf{1} : \mathsf{0} : \mathsf{0} : \mathsf{1} : \ldots$$

This stream specification is not data-obliviously productive [9] for the stream constant fib. The reason is that if we set $\mathsf{0} = \mathsf{1} = \bullet$, then we have to take the identity function as lower bound of the production modulus of h, that is, to be on the safe side, for evaluating h-terms we may use only the rule $\mathsf{h}(\bullet : \sigma) \to \bullet : \mathsf{h}(\sigma)$. But then we get:

$$\mathsf{tail}(\underline{\mathsf{fib}}) \to \mathsf{tail}(\underline{\mathsf{h}}(\bullet : \mathsf{tail}(\mathsf{fib}))) \to \underline{\mathsf{tail}}(\bullet : \mathsf{h}(\mathsf{tail}(\mathsf{fib}))) \to \mathsf{h}(\mathsf{tail}(\mathsf{fib}))$$

and so fib is not data-obliviously productive. We note that this can be remedied by rewriting the right-hand side of the fib-rule:

$$\mathsf{fib} \to \mathsf{0} : \mathsf{1} : \mathsf{h}(\mathsf{tail}(\mathsf{fib}))$$

Remark 3.3. In tree specifications data symbols can have codata as arguments. For example, the commonly used 'observation' function for streams:

$$\mathsf{head}(x : \sigma) \to x$$

takes codata to data. Data symbols with non-zero codata arity are excluded from the specification formats used in [12, 9, 54, 55]. The approach we take lifts this restriction as we transform lazy evaluation on data as well as codata into a termination context-sensitive termination problem.

The work [12, 9] aimed at a decision algorithm for (data-oblivious) productivity for certain formats of stream specifications. There 'stream dependent' data symbols like head are disallowed, for their presence immediately makes productivity a Π^0_2-hard problem, as shown in [10]. The extra complication arising from allowing data symbols with non-zero codata arity is that they may cause reference to a part of the infinite structure that becomes available only in future unfoldings, as in the following example.

Example 3.4. Let $k \geq 0$. Consider the stream specification from Sijtsma [41]:

$$\begin{aligned} \mathsf{T}_k &\to \mathsf{0} : \mathsf{nth}(\underline{k}, \mathsf{T}_k) : \mathsf{T}_k \\ \mathsf{nth}(\mathsf{0}, x : \sigma) &\to x \\ \mathsf{nth}(\mathsf{s}(n), x : \sigma) &\to \mathsf{nth}(n, \sigma) \end{aligned}$$

where $\underline{k}$ denotes the numeral $\mathsf{s}^k(\mathsf{0})$. Two sorts are involved, N for numerals, and S for streams. We set $\mu = \{\mathsf{N}\}$, $\nu = \{\mathsf{S}\}$, and $\Sigma = \{:, \mathsf{T}_k, \mathsf{nth}, \mathsf{0}, \mathsf{s}\}$ with types:

$$: :: \mathsf{N} \times \mathsf{S} \to \mathsf{S} \qquad \mathsf{T}_k :: \mathsf{S} \qquad \mathsf{nth} :: \mathsf{N} \times \mathsf{S} \to \mathsf{N} \qquad \mathsf{0} :: \mathsf{N} \qquad \mathsf{s} :: \mathsf{N} \to \mathsf{N}$$

Note that the data symbol nth takes a codata argument. For $k = 2$, we get that $\mathsf{nth}(\mathsf{s}(\mathsf{s}(t)), \mathsf{T}_2) \to^* \mathsf{nth}(t, \mathsf{T}_2)$ for all numerals t, and $\mathsf{nth}(\underline{1}, \mathsf{T}_2) \to^* \mathsf{nth}(\underline{2}, \mathsf{T}_2)$, and hence $\mathsf{T}_2 \to^\omega \mathsf{0} : \mathsf{0} : \mathsf{0} : \ldots$, producing the infinite stream of zeros. But if we take $k = 3$, the evaluation of each term $\mathsf{nth}(\underline{n}, \mathsf{T}_3)$ for odd n eventually ends up in the loop:

$$\mathsf{nth}(\underline{3}, \mathsf{T}_3) \to^* \mathsf{nth}(\underline{1}, \mathsf{T}_3) \to^* \mathsf{nth}(\underline{3}, \mathsf{T}_3) \to^* \ldots$$

Hence we get $\mathsf{T}_3 \to^\omega \mathsf{0} : \bot : \mathsf{0} : \bot : \ldots$ (where $\bot$ stands for 'undefined') and T_3 is not productive. In general, T_k is productive if and only if k is even.

Productivity of specifications like these, where the evaluation of stream elements needs to be delayed to wait for 'future information' (lazy evaluation), is adequately analyzed using the concept of 'set productivity' in [41]. Hitherto, all methods studied the proper subclass of 'segment productivity' only, where well-definedness of one element requires well-definedness of all previous ones. Our method applies to set productivity, or, *lazy productivity* as we call it.

3.2. Properties of Tree Specifications.

For the restricted formats of [12, 9, 54, 55], productivity was coinciding (and usually defined as) constructor normalization:

Definition 3.5. A tree specification $\langle \Sigma, R \rangle$ is *constructor normalizing* if all finite ground terms $t \in \mathit{Ter}(\Sigma, \varnothing)$ rewrite ($t \to^\omega s$) to a (possibly infinite) constructor normal form $s \in \mathit{Ter}^\infty(\Sigma^{\mathcal{C}}, \varnothing)$.

By orthogonality of tree specifications all normal forms are unique [28, 29]. For constructor normal forms, the uniqueness follows also from [32].

Note that we consider only *finite* ground terms $t \in \mathit{Ter}(\Sigma, \varnothing)$. Considering *all* (finite and infinite) terms is not appropriate for productivity, for otherwise even trivially productive specifications like:

$$\mathsf{zeroid} \to \mathsf{id}(0 : \mathsf{zeroid}) \qquad\qquad \mathsf{id}(x : \sigma) \to x : \mathsf{id}(\sigma)$$

would not be constructor normalizing due to infinite terms like $\mathsf{id}(\mathsf{id}(\mathsf{id}(\ldots)))$.

The global requirement that all finite ground terms be constructor normalizing does not in general imply productivity of functions defined by a specification, in the sense of mapping totally defined objects (constructor normal forms) to totally defined objects. It only tells you that the function returns a constructor normal form when applied to any finite ground term that can be formed in the specification. For instance the rules for f on page 2 do not define a total function on all infinite terms. For instance, when applied to the stream $1 : 1 : 1 : \ldots$, nothing is ever produced.

The following lemma is immediate:

Lemma 3.6 ([55, Proposition 3.3])**.** *A tree specification $\langle \Sigma, R \rangle$ is constructor normalizing if and only if every finite ground term $t \in \mathit{Ter}(\Sigma, \varnothing)$ rewrites in finitely many steps to a term s with a constructor at its root ($\mathit{root}(s) \in \Sigma^{\mathcal{C}}$).*

As we allow for a very general specification format, constructor normalization is no longer equivalent to productivity. Roughly speaking, we need to ensure that data terms are finite. Actually, the setting is slightly more involved as we allow for data constructors to have (infinite) codata arguments, so as to form, e.g., a tuple of streams. Therefore we require that, descending through any constructor normal form of a term $t \in \mathit{Ter}(\Sigma, \varnothing)$ one always eventually encounters a codata constructor. First we define a notion of 'path' in a term.

Definition 3.7. Let $t \in \mathit{Ter}^\infty(\Sigma, \mathcal{X})$, and $\Gamma \subseteq \Sigma$. A Γ-*path in* t is a (finite or infinite) sequence $\langle p_0, \mathsf{c}_0 \rangle, \langle p_1, \mathsf{c}_1 \rangle, \ldots$ such that $\mathsf{c}_i = \mathit{root}(t|_{p_i}) \in \Gamma$ and $p_{i+1} = p_i j$ with $1 \leq j \leq \sharp \mathsf{c}_i$.

Definition 3.8. A tree specification $\langle \Sigma, R \rangle$ is *data-finite* if for all finite ground terms $t \in \mathit{Ter}(\Sigma, \varnothing)$ and constructor normal forms s of t (that is, $t \to^\omega s$): every $\Sigma^{\mathcal{C}}_\mu$-path in s (containing data constructors only) is finite.

In other words, every infinite path in a constructor normal form of a data-finite tree specification contains infinitely many codata constructors.

Example 3.9. The well-known '$\mathsf{ams0}$-TRS' for addition and multiplication of numerals:

$$\mathsf{a}(0, y) \to y \qquad\qquad \mathsf{m}(0, y) \to 0$$
$$\mathsf{a}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{a}(x, y)) \qquad\qquad \mathsf{m}(\mathsf{s}(x), y) \to \mathsf{a}(y, \mathsf{m}(x, y))$$

is productive: all finite ground terms rewrite to a (finite) constructor normal form. If we extend the ams0-TRS (an example due Jan Willem Klop) with a constant ∞ and a rule for computing the infinite term $\mathsf{s}(\mathsf{s}(\mathsf{s}(\ldots)))$:

$$\infty \to \mathsf{s}(\infty)$$

then the so obtained TRS, call it $\mathcal{R}$, is no longer productive: If we take $\mathsf{0}$ and s to be constructors of an inductive sort $\mathsf{N} \in \mu$, then clearly $\mathcal{R}$ is not data-finite. On the other hand, if $\mathsf{0}$ and s are constructors of a coinductive sort $\overline{\mathsf{N}} \in \nu$ (representing the set of *extended natural numbers* $\mathbb{N} \cup \{\infty\}$), then constructor normalization of $\mathcal{R}$ fails, due to the existence of root-active terms $\mathsf{m}(\infty, \mathsf{0})$:

$$\mathsf{m}(\infty, \mathsf{0}) \to \mathsf{m}(\mathsf{s}(\infty), \mathsf{0}) \to \mathsf{a}(\mathsf{0}, \mathsf{m}(\infty, \mathsf{0})) \to \mathsf{m}(\infty, \mathsf{0}) \to \ldots$$

Note that in the coinductive interpretation of $\mathsf{0}$ and s in $\mathcal{R}$, the rules for m can be made productive by a second case analysis in the s-case, that is, by replacing the rule for $\mathsf{m}(\mathsf{s}(x), y)$ by the following two rules:

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

This also shows that the partitioning into data and codata sorts (as done by the programmer) is essential to determine productivity. For example, the single rule TRS $\infty \to \mathsf{s}(\infty)$ is productive if the symbol s is a codata constructor, but not if it is a data constructor.

Remark 3.10. Data-finiteness is similar to what is called 'properness' in [25]. However, properness is a stronger condition where on every infinite path in a constructor normal form eventually only coinductive symbols are encountered.

We arrive at our definition of productivity:

Definition 3.11. A tree specification $\mathcal{R}$ is *productive* if $\mathcal{R}$ is constructor normalizing and data-finite.

3.3. A Global Assumption.

We make the following assumption for all signatures of tree specifications, and justify it in the subsequent remark.

Assumption 3.12. There exists a finite ground term $t \in \mathit{Ter}(\Sigma, \varnothing)_\tau$ of sort τ, for every sort $\tau \in \mu \cup \nu$.

Remark 3.13. The motivation for Assumption 3.12 is based on the fact that we consider *global* productivity, that is, productivity of *all* finite ground terms. Let $\tau \in \mu \cup \nu$ be a sort. If there exist no finite ground terms of sort τ, then rules containing symbols (or variables) of this sort are not applicable in any reduction starting from a finite ground term. Hence, productivity is independent of whatever the rules look like. For example, consider the specification:

$$\mathsf{none}(x) \to \mathsf{none}(x) \tag{1}$$

Now, for all finite ground terms this TRS is productive, as there are none.

The assumption of the existence of finite ground terms of every sort is lacking in [39]. The system (1) extended with the overflow rule $x : \sigma \to \mathsf{overflow}$ is not balanced outermost terminating, but vacuously productive.

Note that the set of sorts $\mathcal{S}' \subseteq \mathcal{S}$ for which there exist finite ground terms is computable; $\mathcal{S}'$ is the smallest set A such that: $\tau \in A$ if for some $\mathsf{f} \in \Sigma$ with $\mathsf{f} :: \tau_1 \times \ldots \times \tau_{\sharp\mathsf{f}} \to \tau$ we have $\tau_1, \ldots, \tau_{\sharp\mathsf{f}} \in A$. Having computed the set $\mathcal{S}'$ of 'non-empty sorts', we can discard all rules from the specification $\mathcal{R}$ that contain a symbol or variable of a sort from $\mathcal{S} \setminus \mathcal{S}'$, without affecting productivity: the thus obtained specification $\mathcal{R}'$ is productive if and only if $\mathcal{R}$ is.

3.4. Shallow Tree Specifications.

We now introduce 'shallow' tree specifications, where pattern matching is only one constructor symbol deep. Shallow tree specifications form the target formalism of the productivity preserving transformation defined in Section 5, which turns strongly sequential tree specification into shallow ones. And shallow tree specifications are the input systems for the transformation to context-sensitive TRSs $\mathbf{T}_{\mathfrak{C}}(\mathcal{R})$ defined in Section 6.

Definition 3.14. A tree specification $\mathcal{R} = \langle \Sigma, R \rangle$ is *shallow* if for every n-ary symbol $\mathsf{f} \in \Sigma^{\mathcal{D}}$ we have a set of *argument indices* $I_{\mathsf{f}} \subseteq \{1, \ldots, n\}$ such that for each of its defining rules:

$$\mathsf{f}(p_1, \ldots, p_n) \to r$$

every pattern p_i satisfies the following conditions:

(i) if $i \in I_{\mathsf{f}}$, then p_i is of the form $\mathsf{c}(x_1, \ldots, x_m)$ for some m-ary constructor $\mathsf{c} \in \Sigma^{\mathcal{C}}$ and variables $x_1, \ldots, x_m$;
(ii) if $i \notin I_{\mathsf{f}}$, then p_i is a variable.

Remark 3.15. In shallow tree specifications patterns contain constructors only at depth one. Moreover, all defining rules for a symbol f are forced to 'consume' from the same arguments, namely from those indicated by I_{f}. The latter requirement can be relaxed to also allow variables at positions $i \in I_{\mathsf{f}}$, i.e., replacing clause (i) by (i)' :

(i)' if $i \in I_{\mathsf{f}}$, then p_i is a variable, or it is of the form $\mathsf{c}(x_1, \ldots, x_m)$ for some m-ary constructor $\mathsf{c} \in \Sigma^{\mathcal{C}}$ and variables $x_1, \ldots, x_m$;

For this extended format, the transformation of Definition 6.1 would still be sound, but no longer complete for proving productivity (Theorem 6.6). The point is that then evaluation of possibly non-needed subterms is allowed. This can lead to non-termination of the transformed system, see Example 9.2.

The stream specifications for T_k of Example 3.4 are shallow. The specification $\mathcal{R}_{\mathsf{fib}}$ from Example 3.2 is not shallow, but can be made shallow, see Example 5.2; in fact $\mathcal{R}_{\mathsf{fib}}$ is an example of a 'semi-shallow' specification, defined in Section 7.

Example 3.16. We give an example of a shallow tree specification $\mathcal{R} = \langle \Sigma, R \rangle$ with data sorts $\mu = \{\mathsf{B}, \mathsf{B}^*\}$ and codata sorts $\nu = \{\mathsf{S}, \mathsf{T}\}$. The constructor symbols (in $\Sigma^{\mathcal{C}}$) are typed as follows:

$$0, 1 :: \mathsf{B} \quad \mathsf{e} :: \mathsf{B}^* \quad \cdot :: \mathsf{B}^* \times \mathsf{B} \to \mathsf{B}^* \quad : :: \mathsf{B}^* \times \mathsf{S} \to \mathsf{S} \quad \mathsf{node} :: \mathsf{B}^* \times \mathsf{T} \times \mathsf{T} \to \mathsf{T}$$

and the defined symbols (in $\Sigma^{\mathcal{D}}$) have types:

$$\mathsf{t} :: \mathsf{T} \qquad \mathsf{f} :: \mathsf{B} \times \mathsf{T} \to \mathsf{T} \qquad \mathsf{lo} :: \mathsf{T} \to \mathsf{S} \qquad \mathsf{zip} :: \mathsf{S} \times \mathsf{S} \to \mathsf{S}$$

We let R consist of the following rules:

$$\begin{aligned} \mathsf{t} &\to \mathsf{node}(\mathsf{e}, \mathsf{f}(0, \mathsf{t}), \mathsf{f}(1, \mathsf{t})) \\ \mathsf{f}(x, \mathsf{node}(w, \alpha, \beta)) &\to \mathsf{node}(w \cdot x, \mathsf{f}(x, \alpha), \mathsf{f}(x, \beta)) \\ \mathsf{lo}(\mathsf{node}(x, \alpha, \beta)) &\to x : \mathsf{zip}(\mathsf{lo}(\alpha), \mathsf{lo}(\beta)) \\ \mathsf{zip}(x : \sigma, \tau) &\to x : \mathsf{zip}(\tau, \sigma) \end{aligned}$$

The constant t defines an infinite binary tree labeled with all words over $\{0, 1\}^*$. The constructor symbol e represents the empty word, and $w \cdot x$ represents appending a letter x to a word w; we let the word constructor '$\cdot$' bind stronger than the stream constructor ':'. The rule for lo defines a function which takes a labeled binary tree and returns a stream of labels of nodes visited in level-order. This specification has a particular form, from which constructor normalization can be derived immediately: all rules produce a constructor; see Proposition 3.17 below.

The term $\mathsf{lo}(\mathsf{t})$ rewrites to the lexicographical enumeration of all binary words:

$$\mathsf{lo}(\mathsf{t}) \to^{\omega} \mathsf{e} : \mathsf{e} \cdot 0 : \mathsf{e} \cdot 1 : \mathsf{e} \cdot 0 \cdot 0 : \mathsf{e} \cdot 0 \cdot 1 : \ldots$$

For a specific class of stream specifications, called 'friendly' in [9], constructor normalization follows immediately. This is generalized to trees in [55], see Theorem 3.4 ibid.

Proposition 3.17. *A shallow tree specification $\mathcal{R} = \langle \Sigma, R \rangle$ is constructor normalizing if for all rules $\ell \to r \in R$ we have that $root(r) \in \Sigma^{\mathcal{C}}$.*

Proof. By Lemma 3.6 it suffices to show that every ground term $t \in Ter(\Sigma, \varnothing)$ rewrites to a term s with $root(s) \in \Sigma^{\mathcal{C}}$. We proceed by induction on t. Let $t = \mathsf{f}(t_1, \ldots, t_n)$. By the induction hypothesis we have that $t_i \to^* s_i$ with $root(s_i) \in \Sigma^{\mathcal{C}}$ for every $1 \leq i \leq n$. So we get $t \to^* \mathsf{f}(s_1, \ldots, s_n)$ and by left-linearity, exhaustivity, and shallowness of $\mathcal{R}$ it follows that $\mathsf{f}(s_1, \ldots, s_n)$ is a redex with respect to a rule $\ell \to r$. Since $root(r) \in \Sigma^{\mathcal{C}}$ we are done. □

The difference between Proposition 3.17 and [55, Theorem 3.4] is that we require shallowness for both data and codata, whereas [55] allows arbitrary matching on data but requires an independent data-layer. A typical example which is excluded by either restriction is:

$$\mathsf{a} \to \mathsf{h}(\mathsf{a}) : \mathsf{a} \qquad \mathsf{h}(0 : \sigma) \to 0 \qquad \mathsf{h}(1 : \sigma) \to 1$$

Note that this specification is not constructor normalizing.

4. From Lazy Evaluation to Strongly Sequential Tree Specifications

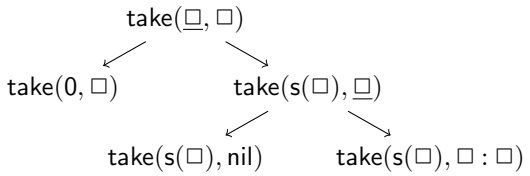
Lazy evaluation is a class of rewrite strategies employed in functional programming languages, such as Haskell [37] and Clean [38], where the evaluation of a subterm is delayed until it is needed, and only to the extent required by the calling function. Unfortunately, there is no general definition of lazy evaluation; every functional programming language has its own variation. Even worse, every Haskell compiler has its own variation of lazy evaluation; there is no operational semantics defined for Haskell (in contrast to Clean [38] which is based on graph rewriting). However, lazy evaluation strategies all have in common that there is a deterministic order for the evaluation of argument positions. These systems can be rendered as strongly sequential term rewriting systems [23, 30].

For constructor TRSs it is known that strong sequentiality coincides with inductive sequentiality, see [21]. A TRS $\mathcal{R} = \langle \Sigma, R \rangle$ is *inductively sequential* if every symbol has a 'definitional tree' [3], that is, a tree that defines the order in which positions in a term are evaluated. We prefer the name 'evaluation tree'.

Before we give the definition of evaluation trees, let us consider an example of a strongly sequential tree specification:

$$\begin{aligned} \mathsf{take}(\mathsf{0}, x) &\to \mathsf{nil} \\ \mathsf{take}(\mathsf{s}(n), \mathsf{nil}) &\to \mathsf{nil} \\ \mathsf{take}(\mathsf{s}(n), x : y) &\to x : \mathsf{take}(n, y) \end{aligned}$$

For 'take' the evaluation tree looks as follows:



The evaluation position is indicated by underlining. The leaves of the tree are patterns of the left-hand sides of the rules for take.

Next we give a formal definition of evaluation trees. A *tree* is a pair $\langle V, E \rangle$ consisting of a set V of *nodes* and a set $E \subseteq V \times V$ of *edges*, satisfying: (i) there is a unique node $r \in V$, the *root* of the tree, that all nodes are reachable from: rE^*t for all $t \in V$; (ii) apart from the root, every node has a unique *parent* node: $\exists!t.\, tEs$ for all nodes $s \neq r$; (iii) E is acyclic, i.e., for no node s we have sE^+s. A tree is *finite* if its set of nodes is finite. We use V_{int} to denote the set of *internal* nodes of a tree, $V_{int} := \{p \mid \exists q.\, pEq\}$, and $V_{ext} := V \setminus V_{int}$ for the *external nodes*.

Definition 4.1. An *evaluation tree* is a tuple $\langle V, E, \lambda \rangle$ where $\langle V, E \rangle$ is a finite tree with $V \subseteq \mathcal{C}xt(\Sigma, \varnothing)$ and $\lambda : V_{int} \to \mathbb{N}$ is a *labeling* function, satisfying:

– if $p \in \mathcal{C}xt_n(\Sigma, \varnothing)$ is an n-hole context, then $\lambda(p) \in \{1, \ldots, n\}$;

– if pEq, then $q = p[\mathsf{c}(\Box, \ldots, \Box)]_{\lambda(p)}$, for some constructor symbol $\mathsf{c} \in \Sigma^{\mathcal{C}}$.

Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a tree specification and let $\mathsf{f} \in \Sigma^{\mathcal{D}}$ be a defined symbol. An *evaluation tree for* f is an evaluation tree with root $\mathsf{f}(\Box, \ldots, \Box)$ such that the set of leaves coincides with the set of lhs patterns of the f-rules $V_{ext} = \{\ell\sigma_{\Box} \mid \ell \to r \in R, root(\ell) = \mathsf{f}\}$, where the substitution $\sigma_{\Box}$ replaces all variables by $\Box$.

The labels $\lambda(p)$ in the evaluation tree for take above are the underlinings of holes, indicating the position where to evaluate next. Since pattern matching in tree specifications is exhaustive (by definition), the number of child nodes of an internal node $p \in V_{int}$ is equal to the number of constructor symbols of the sort of the selected hole. Moreover, each child node is 'one constructor more specific' than its parent.

Theorem 4.2 ([21, Theorem 4.14])**.** *A constructor TRS $\langle \Sigma, R \rangle$ is strongly sequential if and only if there exists an evaluation tree for every $\mathsf{f} \in \Sigma^{\mathcal{D}}$.*

Lemma 4.3. *Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a strongly sequential tree specification. Then, for every $\mathsf{f} \in \Sigma^{\mathcal{D}}$ there exists an argument index $1 \leq i \leq \sharp\mathsf{f}$ such that: for all f-defining rules $\mathsf{f}(t_1, \ldots, t_{\sharp f}) \to r \in R$ we have $root(t_i) \in \Sigma^{\mathcal{C}}$.*

Proof. The argument index is given by $\lambda(p)$ of the root node p of the evaluation tree for the symbol f obtained from the constructive proof of Theorem 4.2. □

We discuss some examples of Haskell programs and show how to encode lazy evaluation with 'rule priorities' by evaluation trees. We thereby informally present a transformation from (first-order) Haskell programs to orthogonal, strongly sequential term rewriting systems.

As a start, we consider the rewrite system:

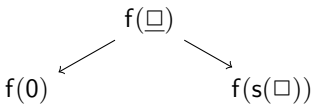
$$\begin{aligned} \infty &\to \mathsf{s}(\infty) \\ \mathsf{f}(\mathsf{0}) &\to \mathsf{0} \\ \mathsf{f}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{0}) \end{aligned}$$

We evaluate the term $\mathsf{f}(\infty)$ using an Haskell-like strategy:

$$\mathsf{f}(\underline{\infty}) \to \underline{\mathsf{f}}(\mathsf{s}(\infty)) \to \mathsf{s}(\mathsf{0})$$

where the evaluation position is indicated by underlining. In the first step, the symbol ∞ is evaluated since the function f needs its argument to be evaluated to a term with a constructor at the root. In the second step, the evaluation of ∞ is delayed since f is applicable without further evaluating the subterm.

The evaluation tree of f can be depicted as follows:



Let us consider a slightly more complicated example:

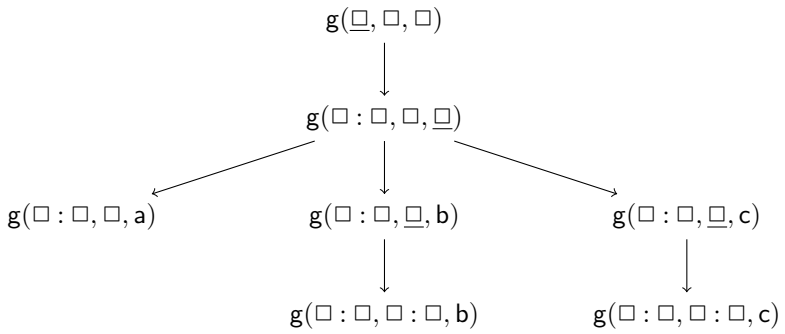
$$\begin{aligned} \mathsf{g}(x:\sigma, y, \mathsf{a}) &\to x \\ \mathsf{g}(x, y:\tau, \mathsf{b}) &\to y \\ \mathsf{g}(\sigma, \tau, x) &\to \mathsf{a} \end{aligned}$$

where the data constructors are a, b, and c.

The rules for g are non-orthogonal (overlapping). In functional programming, these ambiguities are resolved by a priority order on the rules. In this example, the priority order is top to bottom (where top has the highest priority).

The strategy of Haskell to determine the evaluation position is as follows. A rewrite rule $\ell \to r$ is called *feasible* for a term t if $root(t) = root(\ell)$ and the leftmost outermost mismatch between ℓ and t concerns a defined symbol in t, that is, a not yet evaluated position in t. The position of this mismatch for the highest-priority feasible rule is then chosen as evaluation position.

This strategy results in the following evaluation tree of g:



In this example, Haskell does not choose an optimal evaluation strategy. It starts evaluating the first argument since the rule of highest priority for g requires input from that argument. However, it is the third argument which decides on which rule to be applied, and hence the optimal choice would be to start evaluating the third argument. After evaluating the third argument to c, Haskell still considers the second rule for g as feasible, and thus evaluates the second argument. Clearly the strategy evaluates unneeded arguments. The leaves in the evaluation tree correspond to the following slightly adapted rewrite rules:

$$\begin{aligned} \mathsf{g}(x:\sigma, \tau, \mathsf{a}) &\to x \\ \mathsf{g}(x:\sigma, y:\tau, \mathsf{b}) &\to y \\ \mathsf{g}(x:\sigma, y:\tau, \mathsf{c}) &\to \mathsf{a} \end{aligned}$$

These rules together with the above evaluation tree are equivalent to the original rewrite system with the rule priority order combined with the evaluation strategy of Haskell.

5. From Strongly Sequential to Shallow Tree Specifications

We show that all strongly sequential tree specifications can be turned into equivalent shallow tree specifications (Definition 3.14), that is, having the same constructor normal forms and the same behavior with respect to productivity. In Section 6 we describe a complete method for analyzing productivity of shallow tree specifications.

Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a strongly sequential tree specification, and $\mathsf{f} \in \Sigma^{\mathcal{D}}$. We let $\Psi(\mathsf{f})$ stand for the least argument index $1 \leq i \leq \sharp\mathsf{f}$ (bound to exist by Lemma 4.3) such that for all defining rules $\mathsf{f}(t_1, \ldots, t_{\sharp\mathsf{f}}) \to r \in R$ we have that $\mathit{root}(t_i) \in \Sigma^{\mathcal{C}}$.

Definition 5.1. Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a strongly sequential tree specification. We define a transformation from $\mathcal{R}$ to a shallow tree specification $\Xi(\mathcal{R})$, by iteration:

Let $\mathcal{R}_0 := \mathcal{R}$. For $k = 0, 1, \ldots$, check whether $\mathcal{R}_k = \langle \Sigma_k, R_k \rangle$ is shallow. If it is, then set $\Xi(\mathcal{R}) := \mathcal{R}_k$, and the iteration terminates. Otherwise, let $\rho \in R_k$ be a non-shallow rule defining a symbol $\mathsf{f} := \mathit{root}(\mathit{lhs}(\rho))$ of type $\tau_1 \times \ldots \times \tau_n \to \tau$, $i := \Psi(\mathsf{f})$, and define:

- Ξ_k to consist of the following rules, one for each constructor $\mathsf{c} \in \Sigma^{\mathcal{C}}$:

$$\mathsf{f}(x_1, \ldots, x_{i-1}, \mathsf{c}(y_1, \ldots, y_m), x_{i+1}, \ldots, x_n) \\ \to \mathsf{f}_{\mathsf{c}}(x_1, \ldots, x_{i-1}, y_1, \ldots, y_m, x_{i+1}, \ldots, x_n)$$

  where $\mathsf{c} :: \gamma_1 \times \cdots \times \gamma_m \to \tau_i$, and f_{c} is a fresh symbol $\notin \Sigma_k$ (of the induced type, i.e., $\mathsf{f}_{\mathsf{c}} :: \tau_1 \times \cdots \times \tau_{i-1} \times \gamma_1 \times \cdots \times \gamma_m \times \tau_{i+1} \times \cdots \times \tau_n \to \tau$);

- $\mathcal{R}_{k+1} := \langle \Sigma_{k+1}, R_{k+1} \rangle$ where:

$$\Sigma_{k+1} := \Sigma_k \cup \{\mathsf{f}_{\mathsf{c}} \mid \mathsf{c} \in \Sigma^{\mathcal{C}},\ \mathsf{c} :: \gamma_1 \times \cdots \times \gamma_m \to \tau_i\} \\ R_{k+1} := \Xi_k \cup \{\ell{\downarrow_{\Xi_k}} \to r \mid \ell \to r \in R\}$$

  Repeat with $\mathcal{R}_{k+1}$.

To justify the notation $\Xi(\mathcal{R})$, which suggests a deterministic transformation, we note that although the selection order of non-shallow rules may vary, the outcome is unique (up to the names of the freshly chosen symbols).

We give two examples of this Ξ-transformation.

Example 5.2. Reconsider the strongly sequential tree specification $\mathcal{R}_{\mathsf{fib}}$ from Example 3.2, where the defining rules for the stream function h are not shallow due to the presence of the constructor symbols $\mathsf{0}$, $\mathsf{1}$ as argument of the stream constructor ':' in the left-hand sides. Using the algorithm from Definition 5.1 we obtain the following shallow tree specification $\Xi(\mathcal{R}_{\mathsf{fib}})$:

$$\mathsf{fib} \to \mathsf{h}(\mathsf{0} : \mathsf{tail}(\mathsf{fib})) \qquad \mathsf{h}(x : \sigma) \to \mathsf{h}_{:}(x, \sigma)$$
$$\mathsf{tail}(x : \sigma) \to \sigma \qquad \mathsf{h}_{:}(\mathsf{0}, \sigma) \to \mathsf{0} : \mathsf{1} : \mathsf{h}(\sigma)$$
$$\mathsf{h}_{:}(\mathsf{1}, \sigma) \to \mathsf{0} : \mathsf{h}(\sigma)$$

where $\mathsf{h}_{:}$ is a freshly introduced symbol of type $\mathsf{B} \times \mathsf{S} \to \mathsf{S}$.

Example 5.3. Consider the non-shallow strongly sequential TRS:

$$\mathsf{f}(\mathsf{a},\sigma) \to r_1 \qquad\qquad \mathsf{f}(\mathsf{b},x:y:\sigma) \to r_2$$

with some arbitrary right-hand sides r_1 and r_2. We follow Definition 5.1 and transform this TRS into a shallow TRS. We have $\Psi(\mathsf{f}) = 1$. Hence we introduce fresh symbols f_a and f_b and replace the above rules by:

$$\begin{aligned} \mathsf{f}(\mathsf{a},\sigma) &\to \mathsf{f}_\mathsf{a}(\sigma) & \mathsf{f}(\mathsf{b},\sigma) &\to \mathsf{f}_\mathsf{b}(\sigma) \\ \mathsf{f}_\mathsf{a}(\sigma) &\to r_1 & \mathsf{f}_\mathsf{b}(x:y:\sigma) &\to r_2 \end{aligned}$$

The rule for f_b is not shallow yet. Hence we proceed. Introduce a fresh symbol $\mathsf{f}_{\mathsf{b}:}$, and replace the f_b-rule by the following two rules:

$$\mathsf{f}_\mathsf{b}(x:\sigma) \to \mathsf{f}_{\mathsf{b}:}(x,\sigma) \qquad\qquad \mathsf{f}_{\mathsf{b}:}(x,y:\sigma) \to r_2$$

The iteration ends. We have obtained a shallow TRS, consisting of five rules.

Definition 5.4. Let $\mathcal{R}_1 = \langle \Sigma_1, R_1\rangle$ and $\mathcal{R}_2 = \langle \Sigma_2, R_2\rangle$ be tree specifications with $\Sigma_1 \subseteq \Sigma_2$. We say that $\mathcal{R}_2$ *simulates* $\mathcal{R}_1$ if the following conditions hold:

(i) $\to_{\mathcal{R}_1} \subseteq \to_{\mathcal{R}_2}$, and
(ii) $\mathcal{R}_1$ is productive if and only if $\mathcal{R}_2$ is productive.

We prove that the transformation to shallow tree specifications from Definition 5.1 preserves productivity as well as non-productivity. More precisely:

Theorem 5.5. *Let $\mathcal{R} = \langle \Sigma, R\rangle$ be a strongly sequential tree specification. Then $\Xi(\mathcal{R})$ simulates $\mathcal{R}$.*

Proof. We use the notations from Definition 5.1. By transitivity, it suffices to prove that for every transformation step k it holds: $\mathcal{R}_{k+1}$ simulates $\mathcal{R}_k$. Assume that $\mathcal{R}_k$ is not shallow, and let f, i and Ξ_k be as in the definition.

First we show $\to_{\mathcal{R}_k} \subseteq \to_{\mathcal{R}_{k+1}}$. Let $\ell \to r \in \mathcal{R}_k$, we prove that $\ell \to^*_{\mathcal{R}_{k+1}} r$. If $\mathit{root}(\ell) \neq \mathsf{f}$, then $\ell \to r \in \mathcal{R}_{k+1}$. Thus, assume $\mathit{root}(\ell) = \mathsf{f}$. By the choice of $i = \Psi(\mathsf{f})$ there is a rule ρ in Ξ_k matching ℓ, and by orthogonality of Ξ_k this rule is unique. Thus $\ell \xrightarrow{\rho} \ell'$, and ℓ' is a normal form with respect to Ξ_k as f has been replaced by a fresh symbol f_c. Hence, $\{\rho, \ell' \to r\} \subseteq \mathcal{R}_{k+1}$ and $\ell \to_{\mathcal{R}_{k+1}} \ell' \to_{\mathcal{R}_{k+1}} r$, which proves the claim.

Secondly, we prove that $\mathcal{R}_k$ is productive if and only if $\mathcal{R}_{k+1}$ is. To this end we give semantics $[\![\cdot]\!] : \mathit{Ter}(\Sigma_{k+1}, \varnothing) \to \mathit{Ter}(\Sigma_k, \varnothing)$ to terms, mapping terms over the extended signature Σ_{k+1} to terms over Σ_k. Note that every fresh symbol f_c in a term forms a redex occurrence with respect to inverse rules Ξ_k^{-1}. We define $[\![t]\!] = t{\downarrow_{\Xi_k^{-1}}}$ for every $t \in \mathit{Ter}(\Sigma_{k+1}, \varnothing)$. Then by definition, the rules Ξ_k preserve the semantics.

For the direction '$\Rightarrow$', let $\mathcal{R}_k$ be productive, and $t \in \mathit{Ter}(\Sigma_{k+1}, \varnothing)$ a ground term over the extended signature. By assumption $[\![t]\!]$ is productive with respect to $\mathcal{R}_k$. Consequently, $[\![t]\!]$ is productive with respect to $\mathcal{R}_{k+1}$ as $\to_{\mathcal{R}_k} \subseteq \to_{\mathcal{R}_{k+1}}$. It follows that t is productive with respect to $\mathcal{R}_{k+1}$ since $[\![t]\!] \to^*_{\mathcal{R}_{k+1}} t$.

For the direction '$\Leftarrow$', assume $\mathcal{R}_{k+1}$ is productive and let $t \in Ter(\Sigma_k, \varnothing)$ be a ground term. We show that whenever $t \to^*_{\mathcal{R}_{k+1}} t'$ then $t \to^*_{\mathcal{R}_k} [\![t']\!]$; this implies productivity of t with respect to $\mathcal{R}_k$ since the constructor prefix of t' coincides with that of $[\![t']\!]$. We use induction on the length of the reduction $t \to^*_{\mathcal{R}_{k+1}} t'$. For the base case, note that $t = [\![t']\!]$. For the induction step, we consider a rewrite sequence $t \to^*_{\mathcal{R}_{k+1}} t' \to_{\mathcal{R}_{k+1}} t''$. Then $t \to^*_{\mathcal{R}_k} [\![t']\!]$ by induction hypothesis. Let $\rho \in \mathcal{R}_{k+1}$ be the rule corresponding to the step $\rho' : t' \to_{\mathcal{R}_{k+1}} t''$. If $\rho \in \Xi_k$ then it follows $[\![t']\!] = [\![t'']\!]$, and $t \to^*_{\mathcal{R}_k} [\![t'']\!]$. If $\rho \in \mathcal{R}_k$ then contracting the (unique) residual of ρ' in $[\![t']\!]$ after $t' \to^*_{\Xi_k^{-1}} [\![t']\!]$ yields a step $[\![t']\!] \to_{\mathcal{R}_k} [\![t'']\!]$, and hence $t \to^*_{\mathcal{R}_k} [\![t'']\!]$. Finally, if $\rho \notin \mathcal{R}_k$ and $\rho \notin \Xi_k$ then ρ is of the form $\ell' \to r$ and there exists a rule $\ell \to r \in \mathcal{R}_k$ such that $\ell \to_{\Xi_k} \ell'$. Then application of $\ell \to r \in \mathcal{R}_k$ in $[\![t']\!]$ at the residual of the position of ρ' in t' after $t' \to^*_{\Xi_k^{-1}} [\![t']\!]$ results in a step $[\![t']\!] \to_{\mathcal{R}_k} [\![t'']\!]$. Hence $t \to^*_{\mathcal{R}_k} [\![t'']\!]$. □

6. From Productivity of Shallow Tree Specifications to Context-Sensitive Termination

In this section we define a transformation from shallow tree specifications to context-sensitive TRSs in such a way that productivity of the original specification is equivalent to termination of the transformed system. In fact, we give two transformations: one for constructor normalization, and one augmented with rules that consume all data-constructors in order to capture data-finiteness.

Definition 6.1. Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a shallow tree specification. Let ϕ be a fresh sort ($\phi \notin \mu \cup \nu$), and for every data sort $\tau \in \mu$, let $\mathfrak{C}_\tau$ be a symbol not in Σ of type $\mathfrak{C}_\tau :: \tau \to \phi$. We define two many-sorted context-sensitive TRSs:

1. $\mathbf{T}(\mathcal{R}) := \langle \mathcal{R}, \xi \rangle$ with the replacement map ξ defined by:

$$\xi_{\mathsf{f}} = I_{\mathsf{f}} \quad (\mathsf{f} \in \Sigma^{\mathcal{D}}) \qquad\qquad \xi_{\mathsf{c}} = \varnothing \quad (\mathsf{c} \in \Sigma^{\mathcal{C}})$$

2. $\mathbf{T}_{\mathfrak{C}}(\mathcal{R}) := \langle \mathcal{R}', \xi' \rangle$ where $\mathcal{R}' = \langle \Sigma', R' \rangle$ with $\Sigma' = \Sigma \cup \{\mathfrak{C}_\tau \mid \tau \in \mu\}$ and
   (i) R' is the extension of R with the rules:

$$\mathfrak{C}_\tau(\mathsf{c}(x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_m)) \to \mathfrak{C}_{\tau_i}(x_i)$$

   for all data constructors $\mathsf{c} \in \Sigma^{\mathcal{C}}_\mu$ of type $\tau_1 \times \ldots \times \tau_n \times \gamma_1 \times \ldots \times \gamma_m \to \tau$, where $n = \sharp_\mu \mathsf{c}$ and $m = \sharp_\nu \mathsf{c}$, and all indices $1 \leq i \leq n$;
   (ii) ξ' is the extension of ξ to the signature Σ', for every $\tau \in \mu$ defined by:

$$\xi'_{\mathfrak{C}_\tau} = \{1\}\,.$$

The replacement maps ξ and ξ' are *canonical* in the sense of [33]. A canonical replacement map is the most restrictive replacement map guaranteeing that every non-variable position of a subterm of a left-hand side is replacing. For shallow tree specifications, it is not only ensured that every non-variable position

in a left-hand side is replacing, but moreover that the non-variable positions are exactly the replacing positions. This is crucial to obtain completeness for proving productivity, see Theorem 6.6.

The replacement map ξ enforces the contraction of root-needed redexes only. Hence, the context-sensitive reduction with respect to $\mathbf{T}(\mathcal{R})$ will lead to (and end in) a root-stable form with constructor-root ($root(t) \in \Sigma^{\mathcal{C}}$) whenever a term of this form is reachable with respect to $\mathcal{R}$.

Lemma 6.2. *Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a shallow tree specification. A ground term $t \in Ter(\Sigma, \varnothing)$ is in normal form with respect to $\mathbf{T}(\mathcal{R})$ if and only if $root(t) \in \Sigma^{\mathcal{C}}$.*

Proof. Since $\xi_{\mathsf{c}} = \varnothing$ for every constructor symbol $\mathsf{c} \in \Sigma^{\mathcal{C}}$, it suffices to show that every term $t = \mathsf{f}(t_1, \ldots, t_{\sharp\mathsf{f}})$ with $\mathsf{f} \in \Sigma^{\mathcal{D}}$ contains a redex with respect to $\mathbf{T}(\mathcal{R})$. We proceed by induction on the term structure of t. If for some $i \in I_{\mathsf{f}}$ the term t_i does not have a constructor at the root, then t_i contains a redex by the induction hypothesis, and thereby t contains a redex since $i \in \xi_{\mathsf{f}}$. Otherwise, t is a redex by exhaustivity of $\mathcal{R}$. □

Proposition 6.3. *Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a shallow tree specification. Then $\mathbf{T}(\mathcal{R})$ implements lazy evaluation with respect to $\mathcal{R}$, that is, for all $t \in Ter(\Sigma, \varnothing)$ such that $t \to_{\mathcal{R}}^{*} s$ for some term s with $root(s) \in \Sigma^{\mathcal{C}}$ it holds: every $\mathbf{T}(\mathcal{R})$ reduction starting from t reduces only root-needed redexes.*

Proof. By induction on the size of t. If $root(t) \in \Sigma^{\mathcal{C}}$, then t is root-stable, and a $\mathbf{T}(\mathcal{R})$-normal form. Thus, let $t = \mathsf{f}(t_1, \ldots, t_{\sharp\mathsf{f}})$ with $\mathsf{f} \in \Sigma^{\mathcal{D}}$. By confluence of orthogonal TRSs, every $\mathcal{R}$-reduct of t rewrites to a term s with $root(s) \in \Sigma^{\mathcal{C}}$. As a consequence, every rewrite sequence from t to a root-stable term contains a rewrite step at the root. The first of these root-steps must be the application of a defining rule of f. By the shape of the rules, such a rule is applicable at the root of a term $\mathsf{f}(s_1, \ldots, s_{\sharp\mathsf{f}})$ (if and) only if $root(s_i) \in \Sigma^{\mathcal{C}}$ for all $i \in I_{\mathsf{f}}$. Hence, in order for t to reach a root-stable term, first every t_i with $i \in I_{\mathsf{f}}$ has to rewrite to a root-stable term s_i with $root(s_i) \in \Sigma^{\mathcal{C}}$. As a consequence, every root-needed redex of t_i at a position p corresponds to a root-needed redex of t at position ip. By definition $\xi_{\mathsf{f}} = I_{\mathsf{f}}$, and hence rewriting is restricted to terms t_i with $i \in I_{\mathsf{f}}$. Moreover, by the induction hypothesis, $\mathbf{T}(\mathcal{R})$ allows only the contraction of root-needed redexes for t_i. Finally, if $root(t_i) \in \Sigma^{\mathcal{C}}$ for all $i \in I_{\mathsf{f}}$, then t has a root-needed redex at the root, and by context-sensitive rewriting this is the only redex that is permitted to be contracted. □

Corollary 6.4. *Let $\mathcal{R} = \langle \Sigma, R \rangle$ be a shallow tree specification. Then $\mathbf{T}(\mathcal{R})$ is terminating for $t \in Ter(\Sigma, \varnothing)$ if and only if $t \to_{\mathcal{R}}^{*} s$ with $root(s) \in \Sigma^{\mathcal{C}}$.*

Proof. Direct consequence of Lemma 6.2, Proposition 6.3 and Theorem 2.18. □

We arrive at our main results:

Theorem 6.5. *A shallow tree specification $\mathcal{R}$ is constructor normalizing if and only if $\mathbf{T}(\mathcal{R})$ is terminating.*

Proof. By Lemma 3.6 we have that $\mathcal{R}$ is constructor normalizing if and only if every finite ground term t rewrites to a term with a constructor at the root, which in turn holds if and only if $\mathbf{T}(\mathcal{R})$ is terminating for all ground terms by Corollary 6.4. The latter is equivalent to termination on all terms by Lemma 2.12. □

Theorem 6.6. *A shallow tree specification $\mathcal{R}$ is productive if and only if $\mathbf{T}_{\mathfrak{C}}(\mathcal{R})$ is terminating.*

Proof. For soundness of $\mathbf{T}_{\mathfrak{C}}$ ($\Leftarrow$), assume that $\mathbf{T}_{\mathfrak{C}}(\mathcal{R})$ is terminating.

First, we prove constructor normalization of $\mathcal{R}$. Since $\mathbf{T}(\mathcal{R}) \subseteq \mathbf{T}_{\mathfrak{C}}(\mathcal{R})$, we conclude termination of $\mathbf{T}(\mathcal{R})$. Then by Corollary 6.4, for every $t \in \mathit{Ter}(\Sigma, \varnothing)$ we have: $t \to^* s$ such that $\mathit{root}(s) \in \Sigma^{\mathcal{C}}$. Hence $\mathcal{R}$ is constructor normalizing by Lemma 3.6.

Second, we show that $\mathcal{R}$ is data-finite. Let a term t be called *p-bad* if the (unique [28]) normal form of t contains an infinite path visiting only data constructors starting from position p. We show that there are no p-bad terms. Let $t \in \mathit{Ter}(\Sigma, \varnothing)$ be p-bad with constructor normal form s. By compression [43, Theorem 12.7.1] there exists a strongly convergent reduction $\sigma : t \to^{\leq\omega} s$. This rewrite sequence σ contains, because it is strongly convergent, only finitely many rewrite steps above depth $|p|$. Hence, there exists a finite term t' such that $t \to^* t' \to^{\leq\omega} s$, and t' coincides with s up to depth $|p|$. Then t' consists up to depth $|p|$ of constructor symbols only, and consequently $t'|_p \to^{\leq\omega} s|_p$ and hence $t'|_p$ is ϵ-bad. Thus for data-finiteness it suffices to prove that there exist no ϵ-bad terms.

Let $t \in \mathit{Ter}(\Sigma, \varnothing)$ be ϵ-bad with constructor normal form s. By compression, it follows that there exists a rewrite sequence of the form:

$$t \to^* \mathsf{c}(t_1, \ldots, t_n) \to^{\leq\omega} s$$

for some data constructor $\mathsf{c} \in \Sigma^{\mathcal{C}}$ and finite terms $t_1, \ldots, t_n$ where $n = \sharp\mathsf{c}$. Then there exists a $1 \leq j \leq n$ such that t_j is ϵ-bad again. From Corollary 6.4 and Lemma 6.2 it follows that in $\mathbf{T}_{\mathfrak{C}}(\mathcal{R})$ we have a rewrite sequence of the form:

$$t \to^* \mathsf{c}(t'_1, \ldots, t'_n)$$

for some terms $t'_1, \ldots, t'_n$. Let τ and τ_j be the sorts of t and t'_j, respectively. Then we have in $\mathbf{T}_{\mathfrak{C}}(\mathcal{R})$ a rewrite sequence of the form:

$$\mathfrak{C}_\tau(t) \to^* \mathfrak{C}_\tau(\mathsf{c}(t'_1, \ldots, t'_n)) \to \mathfrak{C}_{\tau_j}(t'_j) \tag{2}$$

The term $\mathsf{c}(t'_1, \ldots, t'_n)$ has a constructor normal form ($\mathcal{R}$ is constructor normalizing), and because it is unique [28], it must be s. Hence t'_j is ϵ-bad again. Repeating the above construction yields an infinite composition of (2), which contradicts termination of $\mathbf{T}_{\mathfrak{C}}(\mathcal{R})$.

For completeness of $\mathbf{T}_{\mathfrak{C}}$ ($\Rightarrow$), assume that $\mathcal{R}$ is productive. Termination of $\mathbf{T}_{\mathfrak{C}}$ for terms $t \in \mathit{Ter}(\Sigma, \varnothing)_\tau$ with $\tau \neq \phi$ follows from Theorem 6.5. Therefore let us consider a term $t \in \mathit{Ter}(\Sigma, \varnothing)_\phi$. Then t is of the form $t = \mathfrak{C}_\tau(t')$ where $t' \in$

$Ter(\Sigma, \mathcal{X})_\tau$ and $\tau \neq \phi$ by well-sortedness. Then t' is terminating by Lemma 6.4. Hence, if t is non-terminating, then the $\mathfrak{C}$-rules must be applied infinitely often at the root. However, this is only possible if t' would 'produce' infinitely many data constructors; contradicting data-finiteness and thereby productivity. □

Example 6.7. The transformation $\mathbf{T}_{\mathfrak{C}}$ of Definition 6.1 applied to the shallow tree specification $\Xi(\mathcal{R}_{\mathsf{fib}})$ of Example 5.2 results in the context-sensitive TRS consisting of the rules of $\Xi(\mathcal{R}_{\mathsf{fib}})$ (there is no rule for $\mathfrak{C}_{\mathsf{B}}$ as the data constructors are constants) and the replacement map ξ defined by:

$$\xi_{:} = \varnothing \qquad\qquad \xi_{\mathsf{tail}} = \xi_{\mathsf{h}} = \xi_{\mathsf{h}_{:}} = \{1\}$$

AProVE [17] fails to prove termination (within 120 sec.); Jambox [7] succeeds by first rewriting the right-hand side of the fib-rule to $\mathsf{0} : \mathsf{1} : \mathsf{h}(\mathsf{tail}(\mathsf{fib}))$.

We give two examples that can not be shown productive by any previous method (see Section 9) due to the use of inductive symbols having coinductive arguments.

Example 6.8. We consider a subsystem $\mathcal{R}_{\mathsf{Ord}}$ of [25] defining tree ordinals:

$$\begin{aligned} x + \mathsf{O} &\to x & x \cdot \mathsf{O} &\to \mathsf{O} \\ x + \mathsf{S}(y) &\to \mathsf{S}(x + y) & x \cdot \mathsf{S}(y) &\to x \cdot y + x \\ x + \mathsf{L}(\sigma) &\to \mathsf{L}(x +_{\mathsf{L}} \sigma) & x \cdot \mathsf{L}(\sigma) &\to \mathsf{L}(x \cdot_{\mathsf{L}} \sigma) \\ x +_{\mathsf{L}} (y : \sigma) &\to (x + y) : (x +_{\mathsf{L}} \sigma) & x \cdot_{\mathsf{L}} (y : \sigma) &\to (x \cdot y) : (x \cdot_{\mathsf{L}} \sigma) \\ \mathsf{nats}(x) &\to x : \mathsf{nats}(\mathsf{S}(x)) & \omega &\to \mathsf{L}(\mathsf{nats}(\mathsf{O})) \end{aligned}$$

We use Ord, a data sort for ordinals, and Str, a codata sort for streams of ordinals. The constructor symbols are:

$$\mathsf{O} :: \mathsf{Ord} \qquad \mathsf{S} :: \mathsf{Ord} \to \mathsf{Ord} \qquad \mathsf{L} :: \mathsf{Str} \to \mathsf{Ord} \qquad : :: \mathsf{Ord} \times \mathsf{Str} \to \mathsf{Str}$$

The transformed system $\mathbf{T}_{\mathfrak{C}}(\mathcal{R}_{\mathsf{Ord}})$ extends the above set of rules with:

$$\mathfrak{C}_{\mathsf{Ord}}(\mathsf{S}(x)) \to \mathfrak{C}_{\mathsf{Ord}}(x)$$

(Note that there is no such rule for the L-constructor as it has no data arguments.) Furthermore, the replacement map ξ is defined by $\xi_{:} = \xi_{\mathsf{S}} = \xi_{\mathsf{L}} = \varnothing$, $\xi_{+} = \xi_{\cdot} = \xi_{+_{\mathsf{L}}} = \xi_{\cdot_{\mathsf{L}}} = \{2\}$, $\xi_{\mathsf{nats}} = \varnothing$, and $\xi_{\mathfrak{C}_{\mathsf{Ord}}} = \{1\}$.

Termination of $\mathbf{T}_{\mathfrak{C}}(\mathcal{R}_{\mathsf{Ord}})$ is proved instantly by AProVE. Hence, by Theorem 6.6, we conclude that $\mathcal{R}_{\mathsf{Ord}}$ is productive, ensuring that, e.g., the term $\omega \cdot \omega$ produces a data-finite contructor normal form.

Example 6.9. We apply our transformation to Example 3.4. The replacement map is defined by $\xi_{\mathsf{nth}} = \{1, 2\}$, $\xi_{\mathfrak{C}_{\mathsf{N}}} = \{1\}$, and $\xi_{:} = \xi_{\mathsf{s}} = \varnothing$, and the set of rules is extended by the rule $\mathfrak{C}_{\mathsf{N}}(\mathsf{s}(n)) \to \mathfrak{C}_{\mathsf{N}}(n)$. AProVE proves termination for $k = 0$ and $k = 2$ within 2 minutes.

Unfortunately, automated termination tools are not directly capable of handling sorted TRSs. From the perspective of unsorted term rewriting, termination on the set of well-sorted terms is a *local termination* problem [8], in contrast to *global termination* on all terms. The work [8] describes a transformation from local to global termination, which, in our special case, would boil down to the elimination of collapsing rules (the right-hand side a variable) by instantiation of variables with terms covering all ground instances.

However, since $\mathbf{T}(\mathcal{R})$ is orthogonal we obtain global termination for free. More precisely, Lemma 2.15 allows us to simply forget about the sorts:

Proposition 6.10. $\mathbf{T}(\mathcal{R})$ *is terminating* $\iff \Theta(\mathbf{T}(\mathcal{R}))$ *is terminating.* □

The system $\mathbf{T}_{\mathfrak{C}}(\mathcal{R})$ is no longer orthogonal due to the $\mathfrak{C}$-rules. Nevertheless:

Proposition 6.11. $\mathbf{T}_{\mathfrak{C}}(\mathcal{R})$ *is terminating* $\iff \Theta(\mathbf{T}_{\mathfrak{C}}(\mathcal{R}))$ *is terminating.*

Proof sketch. The proof is an extension of the proof of Lemma 2.15 with the following observations. Every non-root occurrence of a symbol $\mathfrak{C}_\tau$ yields a sort-conflict, and hence each of these occurrences is the root-symbol of one of the partitions. By the shape of the rewrite rules for the $\mathfrak{C}$-symbols, partitions (contexts) with a root-symbol $\mathfrak{C}_\tau$ cannot collapse. The remaining partitions do not contain any $\mathfrak{C}$-symbols, and as $\mathfrak{C}$-symbols cannot be created, rewriting within these partitions is orthogonal (the $\mathfrak{C}$-rules are never applicable). Moreover, the only overlaps are between the $\mathfrak{C}$-rules, and hence, as in 2.15, we can conclude that collapsing partitions cannot interact with their environment.

As a consequence, when partitioning a non-well-sorted term, all non-topmost partitions are either with respect to orthogonal rewriting, or can never collapse. Thus we can again remove (collapse) all non-topmost, collapsing partitions, and conclude as in the proof of 2.15. □

Propositions 6.10 and 6.11 allow us to forget about the sortedness discipline and to feed the context-sensitive TRSs obtained by Theorems 6.5 and 6.6 directly to a termination prover such as AProVE and mu-Term [35].

7. Semi-Shallow Tree Specifications

In this section we consider productivity for the case that data symbols cannot have codata arguments, and that data terms are supposed to be terminating with respect to eager (non-lazy) evaluation.

Definition 7.1. A tree specification $\mathcal{R} = \langle \Sigma, R \rangle$ has an *independent data-layer* if data symbols do not have codata arguments, i.e., $\sharp_\nu \mathsf{c} = 0$ for every $\mathsf{c} \in \Sigma_\mu$.

This enables us to relax the restrictions on the syntactic format of shallow tree specifications: only the pattern matching against codata constructors needs to be shallow (at most one deep).

Definition 7.2. A tree specification $\mathcal{R} = \langle \Sigma, R \rangle$ is *semi-shallow* if for every symbol $\mathsf{f} \in \Sigma^{\mathcal{D}}$ with data arity $\sharp_\mu \mathsf{f} = m$ and codata arity $\sharp_\nu \mathsf{f} = n$ we have a set of indices $J_\mathsf{f} \subseteq \{1, \ldots, n\}$ such that for every defining rule of the form:

$$\mathsf{f}(p_1, \ldots, p_m, q_1, \ldots, q_n) \to r$$

the codata patterns q_i satisfy the following conditions:

(i) if $i \in J_\mathsf{f}$, then $q_i = \mathsf{c}(t_1, \ldots, t_k, x_1, \ldots, x_l)$ for some codata constructor $\mathsf{c} \in \Sigma^{\mathcal{C}}_\nu$ with $\sharp_\mu \mathsf{c} = k$ and $\sharp_\nu \mathsf{c} = l$, data terms $t_1, \ldots, t_k \in \mathit{Ter}(\Sigma, \mathcal{X})_\mu$, and codata variables $x_1, \ldots, x_l$;
(ii) if $i \notin J_\mathsf{f}$, then q_i is a variable.

This format is closely related to the format used in [55]; our format is slightly more restrictive in that we fix for every function symbol $\mathsf{f} \in \Sigma^{\mathcal{D}}$ the codata arguments (the set J_f) from which there is consumption. This is only a minor restriction, as was shown in Sections 4 and 5.

Example 7.3. The specification of the Fibonacci word given in Example 3.2 is semi-shallow, but not shallow.

In Section 5 we have shown that every strongly sequential specification can be made shallow, and hence semi-shallow. Actually, for the semi-shallow format, we need strong sequentiality only for the codata matching: we no longer have restrictions on data patterns. For example, the following non-sequential stream function (compare with [43, Example 9.2.35]) is semi-shallow:

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

because there are no codata at all.

Another advantage of the semi-shallow format is, in comparison to the shallow format, that it allows for more efficient transformations from non-semi-shallow to semi-shallow specifications. Here, efficiency is compared with respect to the number of rules of the transformed system. We illustrate this transformation on an example, but leave the formal details to the reader:

Example 7.4. To illustrate the transformation to the semi-shallow format, we consider the following specification $\mathcal{R}$:

$$\begin{aligned} \mathsf{h}(\mathsf{0} : \sigma, \tau) &\to \tau \\ \mathsf{h}(\mathsf{s}(x) : \sigma, y_1 : y_2 : \tau) &\to x : \tau \end{aligned}$$

Observe that the system is not semi-shallow. We transform the specification into the following semi-shallow one:

$$\begin{aligned}
\mathsf{h}(\mathsf{0}:\sigma,\tau) &\to \tau\\
\mathsf{h}(\mathsf{s}(x):\sigma,\tau) &\to \mathsf{h}_{\mathsf{h}(\mathsf{s}(\Box):\Box,\Box)}(x,\sigma,\tau)\\
\mathsf{h}_{\mathsf{h}(\mathsf{s}(\Box):\Box,\Box)}(x,\sigma,y:\tau) &\to \mathsf{h}_{\mathsf{h}(\mathsf{s}(\Box):\Box,\Box:\Box)}(x,\sigma,y,\tau)\\
\mathsf{h}_{\mathsf{h}(\mathsf{s}(\Box):\Box,\Box:\Box)}(x,\sigma,y_1,y_2:\tau) &\to x:\tau
\end{aligned}$$

where we have introduced fresh symbols h_C with C being a prefix of the original left-hand side indicating the constructor symbols that have already been evaluated.

This is to be compared to the result $\Xi(\mathcal{R})$ of the transformation to the shallow format (Definition 5.1), which consists of the following six rules:

$$\begin{aligned}
\mathsf{h}(x:\sigma,\tau) &\to \mathsf{h}_{\mathsf{h}(\Box:\Box,\Box)}(x,\sigma,\tau)\\
\mathsf{h}_{\mathsf{h}(\Box:\Box,\Box)}(\mathsf{0},\sigma,\tau) &\to \mathsf{h}_{\mathsf{h}(\mathsf{0}:\Box,\Box)}(\sigma,\tau)\\
\mathsf{h}_{\mathsf{h}(\Box:\Box,\Box)}(\mathsf{s}(x),\sigma,\tau) &\to \mathsf{h}_{\mathsf{h}(\mathsf{s}(\Box):\Box,\Box)}(x,\sigma,\tau)\\
\mathsf{h}_{\mathsf{h}(\mathsf{0}:\Box,\Box)}(\sigma,\tau) &\to \tau\\
\mathsf{h}_{\mathsf{h}(\mathsf{s}(\Box):\Box,\Box)}(x,\sigma,y:\tau) &\to \mathsf{h}_{\mathsf{h}(\mathsf{s}(\Box):\Box,\Box:\Box)}(x,\sigma,y,\tau)\\
\mathsf{h}_{\mathsf{h}(\mathsf{s}(\Box):\Box,\Box:\Box)}(x,\sigma,y_1,y_2:\tau) &\to x:\tau
\end{aligned}$$

Having eager termination of data terms, we obtain data-finiteness for free. As a consequence, we can simplify the transformation of productivity to context-sensitive termination (Definition 6.1): the $\mathfrak{C}$-rules are no longer required.

Definition 7.5. Let $\mathcal{R} = \langle \Sigma, R\rangle$ be a semi-shallow tree specification. We define the context-sensitive TRS $\mathbf{T}_\mu(\mathcal{R}) = \langle \mathcal{R}, \xi\rangle$ with the replacement map ξ:

$$\begin{aligned}
\xi_{\mathsf{f}} &= \{1,\ldots,\sharp_\mu\mathsf{f}\} \cup \{\sharp_\mu\mathsf{f}+i \mid i \in J_{\mathsf{f}}\} && \text{for } \mathsf{f}\in\Sigma^{\mathcal{D}}\\
\xi_{\mathsf{c}} &= \{1,\ldots,\sharp_\mu\mathsf{c}\} && \text{for } \mathsf{c}\in\Sigma^{\mathcal{C}}
\end{aligned}$$

The transformation $\mathbf{T}_\mu(\mathcal{R})$ to context-sensitive rewriting consists only of defining a replacement map ξ, that is, restricting the argument positions where rewriting is allowed, and leaves the set of rules R unaltered. In particular, the transformation allows rewriting in all data-arguments, but rewriting in codata-arguments is restricted to defined symbols f, and only to those arguments where evaluation is needed in order to apply an f-rule (indicated by J_{f}).

The essential difference with the transformation from [55] is that there rewriting has been allowed for all codata-arguments of defined symbols. The restriction of the evaluation to only those codata-arguments where the defining rules actually consume from is necessary to obtain a complete transformation; the necessity of this restriction is demonstrated by Example 9.2.

Theorem 7.6. *A semi-shallow tree specification $\mathcal{R}$ with independent data-layer is productive in combination with termination on the data terms if and only if $\mathbf{T}_\mu(\mathcal{R})$ is terminating.*

Proof. For the direction '$\Rightarrow$', let $\mathcal{R}$ be productive and terminating on data terms. Termination of $\mathbf{T}_\mu(\mathcal{R})$ follows analogously to termination of $\mathbf{T}(\mathcal{R})$ in the proof of Theorem 6.5. However, in contrast to $\mathbf{T}(\mathcal{R})$, $\mathbf{T}_\mu(\mathcal{R})$ allows rewriting in all data arguments. Nevertheless, this does not harm termination of $\mathbf{T}_\mu(\mathcal{R})$ since by assumption $\mathcal{R}$ is terminating on all data terms.

For the direction '$\Leftarrow$', let $\mathbf{T}_\mu(\mathcal{R})$ be terminating. This immediately implies termination of $\mathcal{R}$ on data terms since $\mathcal{R}$ has an independent data-layer, and the replacement map ξ of $\mathbf{T}_\mu(\mathcal{R})$ allows rewriting in all arguments of sort data. The proof of constructor normalization of $\mathcal{R}$ is analogous to the proof of Theorem 6.5 by showing that every ground term t with $root(t) \notin \mathcal{R}^{\mathcal{C}}$ contains a $\mathbf{T}_\mu(\mathcal{R})$ redex. The difference is that matching against data arguments is unrestricted due to semi-shallowness of $\mathcal{R}$. This does not disturb constructor normalization of $\mathcal{R}$, as every data term is strongly normalizing with respect to $\mathcal{R}$. The data-finiteness of $\mathcal{R}$ follows immediately from termination of $\mathcal{R}$ on data terms. □

As in Section 6, Lemma 2.15 allows us to forget about the sorts:

Proposition 7.7. *$\mathbf{T}_\mu(\mathcal{R})$ is terminating $\iff$ $\Theta(\mathbf{T}_\mu(\mathcal{R}))$ is terminating.* □

Example 7.8. For the specification $\mathcal{R}_{\mathsf{fib}}$ introduced in Example 3.2 the transformation of Definition 7.5 defines the replacement map by $\xi_{:} = \xi_{\mathsf{h}} = \xi_{\mathsf{tail}} = \{1\}$, and leaves the set of rules unmodified. Termination of the resulting (unsorted) context-sensitive TRS is proved automatically by AProVE, and hence $\mathcal{R}_{\mathsf{fib}}$ is productive by Theorem 7.6.

8. Matrix Interpretations for Proving Context-Sensitive Termination

In the previous sections we have shown how productivity can be transformed to context-sensitive termination. Here we propose a generalization of matrix interpretations [22, 15] for proving termination of context-sensitive rewriting: we drop the positivity requirement for the upper-left matrix entries for argument positions where rewriting is disallowed. Despite the simplicity of this idea, the method turns out to be efficient for proving termination of context-sensitive rewriting, even without the use of advanced techniques like context-sensitive dependency pairs [20, 1].

In this section we treat only unsorted (context-sensitive) TRSs.

Definition 8.1. A *ξ-monotone Σ-algebra* $\langle A, [\![\cdot]\!], \succ\rangle$ is a Σ-algebra $\langle A, [\![\cdot]\!]\rangle$ and a binary relation $\succ$ on A such that for every $\mathsf{f} \in \Sigma$ of arity n the function $[\![\mathsf{f}]\!]$ is ξ-monotone with respect to $\succ$, that is:

$$a_i \succ b_i \implies [\![\mathsf{f}]\!](a_1, \ldots, a_n) \succ [\![\mathsf{f}]\!](b_1, \ldots, b_n)$$

for every $i \in \xi_{\mathsf{f}}$.

A monotone Σ-algebra $\langle A, [\![\cdot]\!], \succ\rangle$ is called *well-founded* if $\succ$ is well-founded.

In [51] it has been shown that context-sensitive termination can be characterized by interpretations in ξ-monotone Σ-algebras:

Theorem 8.2. *Let $\langle \mathcal{R}, \xi \rangle$ be a context-sensitive TRS over Σ. Then $\mathrm{SN}(\mathcal{R})$ holds if and only if there exists a well-founded ξ-monotone Σ-algebra $\mathcal{A} = \langle A, [\![\cdot]\!], \succ \rangle$ such that $\succ$ is a model for $\mathcal{R}$.* □

Following [15] we generalize this theorem to relative termination as follows.

Definition 8.3. An *extended well-founded ξ-monotone Σ-algebra* $\langle A, [\![\cdot]\!], \succ, \sqsupseteq \rangle$ consists of ξ-monotone Σ-algebras $\langle A, [\![\cdot]\!], \succ \rangle$ and $\langle A, [\![\cdot]\!], \sqsupseteq \rangle$ such that $\mathrm{SN}(\succ / \sqsupseteq)$.

Theorem 8.4. *Let $\mathcal{R}_1$, $\mathcal{R}_2$ be compatible context-sensitive TRSs over Σ. Then $\mathrm{SN}(\mathcal{R}_1/\mathcal{R}_2)$ holds if and only if there is an extended well-founded ξ-monotone Σ-algebra $\mathcal{A} = \langle A, [\![\cdot]\!], \succ, \sqsupseteq \rangle$ such that $\succ$ is a model for $\mathcal{R}_1$, and $\sqsupseteq$ is a model for $\mathcal{R}_2$.*

Proof. Straightforward extension of the proof of [15] to context-sensitive rewriting. □

We can employ the theorem for stepwise removal of rules, as follows:

Corollary 8.5. *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be compatible context-sensitive TRSs over Σ. Assume that $\mathcal{A} = \langle A, [\![\cdot]\!], \succ, \sqsupseteq \rangle$ is an extended well-founded monotone Σ-algebra such that*

(i) *$\succ$ is a model for $\mathcal{T} \subseteq \mathcal{R}_1 \cup \mathcal{R}_2$,*
(ii) *$\sqsupseteq$ is a model for $(\mathcal{R}_1 \cup \mathcal{R}_2) \setminus \mathcal{T}$.*

Then $\mathrm{SN}((\mathcal{R}_1 \setminus \mathcal{T})/(\mathcal{R}_2 \setminus \mathcal{T}))$ implies $\mathrm{SN}(\mathcal{R}_1/\mathcal{R}_2)$.

As an instance of this general framework, we extend matrix interpretations [15] to context-sensitive rewriting. The idea is simple: for argument positions where rewriting is forbidden, we no longer require that the upper-left element of the matrix is positive.

Definition 8.6. Let $\langle \mathcal{R}, \xi \rangle$ be a context-sensitive TRS over the signature Σ. A *context-sensitive matrix interpretation* is a tuple $\langle A, [\![\cdot]\!], \succ, \sqsupseteq \rangle$ where:

– The carrier $A = \mathbb{N}^d$ is a vector space of *dimension* $d \in \mathbb{N}$ on which binary relations $\succ$ and $\sqsupseteq$ are defined as follows:

$$(a_1, \ldots, a_d)^{\mathsf{T}} \succ (b_1, \ldots, b_d)^{\mathsf{T}} \iff a_1 > b_1 \wedge a_2 \geq b_2 \wedge \ldots \wedge a_n \geq b_n$$
$$(a_1, \ldots, a_d)^{\mathsf{T}} \sqsupseteq (b_1, \ldots, b_d)^{\mathsf{T}} \iff a_1 \geq b_1 \wedge a_2 \geq b_2 \wedge \ldots \wedge a_n \geq b_n$$

– For every symbol $\mathsf{f} \in \Sigma$ of arity n, $[\![\mathsf{f}]\!]$ is an affine interpretation:

$$[\![\mathsf{f}]\!](\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}) = M_1^{\mathsf{f}} \boldsymbol{v_1} + \ldots + M_n^{\mathsf{f}} \boldsymbol{v_n} + \boldsymbol{v_{\mathsf{f}}}$$

where $M_1^{\mathsf{f}}, \ldots, M_n^{\mathsf{f}} \in \mathbb{N}^{d \times d}$ are matrices, and $\boldsymbol{v_{\mathsf{f}}} \in \mathbb{N}^d$ is a vector such that the upper-left element of M_i^{f} is positive for every $i \in \xi_{\mathsf{f}}$. Note that $\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}$ range over vectors in $\mathbb{N}^d$.

Theorem 8.7. *A context-sensitive matrix interpretation $\langle A, [\![\cdot]\!], \succ, \sqsupseteq\rangle$ is an extended well-founded ξ-monotone Σ-algebra.*

Proof. Monotonicity with respect to $\sqsupseteq$ is immediate. The ξ-monotonicity with respect to $\succ$ follows from the requirement that the upper-left entries of matrices M_i^{f} are positive for every $i \in \xi_{\mathsf{f}}$. Finally, $\mathrm{SN}(\succ/\sqsupseteq)$ is trivial. □

Let $\langle A, [\![\cdot]\!], \succ, \sqsupseteq\rangle$ be a matrix interpretation. For the applicability of Theorem 8.4 and Corollary 8.5 we need to determine whether $\succ$ or $\sqsupseteq$ are a model for some rule $\ell \to r \in \mathcal{R}_1 \cup \mathcal{R}_2$. As the interpretation of every symbol is an affine transformation, the interpretations $[\![\ell, \alpha]\!]$ and $[\![r, \alpha]\!]$ are affine transformations again. Let $Var(\ell) = \{x_1, \ldots, x_k\}$ for some $k \in \mathbb{N}$. Then we can compute matrices $L_1, \ldots, L_k$, $R_1, \ldots, R_k$, and vectors $\boldsymbol{\ell}$, $\boldsymbol{r}$ such that:

$$[\![\ell, \alpha]\!] = L_1\boldsymbol{\alpha}(\boldsymbol{x_1}) + \ldots + L_k\boldsymbol{\alpha}(\boldsymbol{x_k}) + \boldsymbol{\ell}$$
$$[\![r, \alpha]\!] = R_1\boldsymbol{\alpha}(\boldsymbol{x_1}) + \ldots + R_k\boldsymbol{\alpha}(\boldsymbol{x_k}) + \boldsymbol{r}$$

For matrices $M, N \in \mathbb{N}^{d\times d}$ we write $M \sqsupseteq N$ if $M_{i,j} \geq N_{i,j}$ for all $1 \leq i, j \leq d$.

The following is Lemma 1 from [15], rendered in our terminology:

Proposition 8.8 ([15, Lemma 1])**.** *Let $\langle A, [\![\cdot]\!], \succ, \sqsupseteq\rangle$ be a matrix interpretation, $\ell \to r$ a rewrite rule, and $L_1, \ldots, L_k$, $R_1, \ldots, R_k$, $\boldsymbol{\ell}$, $\boldsymbol{r}$ as described above. Then:*

(i) *$\succ$ is a model for $\ell \to r$ if and only if*

$$L_i \sqsupseteq R_i \text{ for every } 1 \leq i \leq k \text{ and } \boldsymbol{\ell} \succ \boldsymbol{r}$$

(ii) *$\sqsupseteq$ is a model for $\ell \to r$ if and only if*

$$L_i \sqsupseteq R_i \text{ for every } 1 \leq i \leq k \text{ and } \boldsymbol{\ell} \sqsupseteq \boldsymbol{r}$$

Thus in order to remove rules, we have to find a matrix interpretation such that for all rules $\ell \to r$, the interpretation $[\![\ell]\!]$ of ℓ is component-wise $\geq$ than the interpretation $[\![r]\!]$ of r. Then we remove the strictly decreasing rules, that is, rules for which the first component of the vector is decreasing $\boldsymbol{\ell} \succ \boldsymbol{r}$.

Example 8.9. We consider the context-sensitive TRS:

$$\mathsf{f}(\mathsf{f}(x)) \to \mathsf{f}(\mathsf{g}(\mathsf{f}(\mathsf{f}(x))))$$

where the replacement map is given by: $\xi_{\mathsf{f}} = \varnothing$ and $\xi_{\mathsf{g}} = \{1\}$. Termination of this system is obvious. However, proving termination using a direct decreasing interpretation (without transforming the system) is not entirely trivial. First, observe that the left-hand side is a subterm of the right-hand side. Hence the context $\mathsf{f}(\mathsf{g}(\Box))$ in the right-hand side has to make the interpretation smaller. Rewriting within g is allowed, thus its interpretation $[\![\mathsf{g}]\!]$ must be strictly monotonic. As a consequence, we have to make the interpretation $[\![\mathsf{f}]\!]$ of f weakly monotonic. However, multiplying the argument with 0 is not possible as then the left-hand side and the right-hand side will have equal interpretations.

A solution using context-sensitive matrix interpretations then is as follows:

$$[\![\mathsf{f}]\!](\boldsymbol{x}) = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot \boldsymbol{x} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$
$$[\![\mathsf{g}]\!](\boldsymbol{x}) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \boldsymbol{x} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Note that $[\![\mathsf{f}]\!]$ is indeed weakly monotonic as the upper-left matrix element is 0. The interpretation of the rewrite rule is as follows:

$$[\![\mathsf{f}(\mathsf{f}(\boldsymbol{x}))]\!] = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \cdot \boldsymbol{x} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} > \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \cdot \boldsymbol{x} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} = [\![\mathsf{f}(\mathsf{g}(\mathsf{f}(\mathsf{f}(\boldsymbol{x}))))]\!]$$

This interpretation is component-wise $\geq$ and there is a strict decrease in the first component of the vector. Hence we conclude termination of the system $\mathcal{R}$ by Theorem 8.4.

Despite the triviality of the idea, context-sensitive matrix interpretations are very effective in practice. Apart from rewriting right-hand sides, this technique is the only context-sensitive termination method implemented in Jambox. To prove termination of a context-sensitive TRS, Jambox first applies the context-sensitive matrix method to remove a few rules, and then forgets about the replacement map and continues by proving standard termination for the remaining rules. In particular, Jambox does not make use of the advanced method for context-sensitive termination like context-sensitive dependency pairs [20, 1]. Nevertheless, Jambox scored first in proving outermost termination in the termination competition 2008 [44] (based on a transformation from outermost to context-sensitive rewriting [13, 14]), and performed respectably in the competition for context-sensitive termination 2009: Jambox proved 28 system terminating; the winners AProVE and mu-Term both scored 34.

9. Related Work

9.1. Data-Oblivious Productivity.

Since Dijkstra [6] coined the name 'productivity' there have been several papers [48, 41, 24, 42, 4, 11, 12] on defining sufficient criteria for proving productivity. In [9] the methods of [11, 12] are extended, resulting in a decision algorithm for data-oblivious productivity of certain formats of stream specifications. For the pure stream format defined in [11, 12] and extended in [9], data-oblivious productivity coincides with productivity. The term 'data-oblivious' refers to a purely quantitative analysis, where the concrete values of data elements are ignored—productivity for the color-blind, as it were.

All aforementioned methods for analyzing productivity are data-oblivious, The method of [9] is the only one that is data-obliviously *optimal* for a large class of stream specifications. This means that in order to improve on the algorithm one has to proceed in a data-aware fashion.

To see the limitations of a data-oblivious analysis, note that the term a in the TRS $\mathcal{R}_{\mathsf{a}}$ above is not data-obliviously productive; if one cannot distinguish $\mathsf{0}$ from $\mathsf{1}$, it is not sure whether a produces ever more elements.

9.2. Data-Aware Productivity via Outermost Termination.

The first 'data-aware' method (i.e., one that does take into account the identity of data) for proving productivity of stream specifications was presented in [54]. The main theorem of [54] states that a stream specification $\mathcal{R}$ is productive if and only if $\mathcal{R}$ extended with the following rule is terminating with respect to 'balanced outermost' rewriting:

$$x : \sigma \to \mathsf{overflow}$$

where $\mathsf{overflow}$ is a fresh symbol. A *balanced outermost* rewrite sequence is an outermost-fair rewrite sequence which contracts only outermost redexes. A rewrite sequence $t_0 \to t_1 \to \dots$ is called *outermost-fair* [43] if there is no t_n containing an outermost redex which remains an outermost redex infinitely long, that is, which is never contracted.

The idea is that the overflow-rule in combination with the strategy of outermost rewriting prohibits rewriting below the stream constructor symbol ':' and thereby introduces a form of lazy evaluation. Unfortunately, the completeness part of the main theorem of [54] turns out to be wrong:

Example 9.1. The following tree specification $\mathcal{R}$ forms a counterexample to Theorem 4 of [54]: $\mathcal{R} = \langle \Sigma, R \rangle$ is $(\mu \cup \nu)$-sorted with $\mu = \{\mathsf{N}\}$ and $\nu = \{\mathsf{S}\}$. The symbols $\mathsf{0}, : \in \Sigma^{\mathcal{C}}$ and $\mathsf{b}, \mathsf{zeros}, \mathsf{f} \in \Sigma^{\mathcal{D}}$ are typed as follows:

$$\mathsf{0} :: \mathsf{N} \qquad : :: \mathsf{N} \times \mathsf{S} \to \mathsf{S} \qquad \mathsf{b}, \mathsf{zeros} :: \mathsf{S} \qquad \mathsf{f} :: \mathsf{S} \times \mathsf{S} \times \mathsf{S} \to \mathsf{S}$$

We let the set R consist of the following rules:

$$\begin{aligned} \mathsf{b} &\to \mathsf{f}(\mathsf{zeros}, \mathsf{zeros}, \mathsf{b}) \\ \mathsf{zeros} &\to \mathsf{0} : \mathsf{zeros} \\ \mathsf{f}(x : \sigma, y : \tau, \gamma) &\to x : y : \mathsf{f}(\sigma, \tau, \gamma) \end{aligned}$$

This system extended with the rule $x : \sigma \to \mathsf{overflow}$ admits the following balanced outermost rewrite sequence:

$$\begin{aligned} \underline{\mathsf{b}} &\to \mathsf{f}(\underline{\mathsf{zeros}}, \mathsf{zeros}, \mathsf{b}) \\ &\to \mathsf{f}(\mathsf{0} : \underline{\mathsf{zeros}}, \mathsf{zeros}, \mathsf{b}) \\ &\to \mathsf{f}(\mathsf{overflow}, \underline{\mathsf{zeros}}, \mathsf{b}) \\ &\to \mathsf{f}(\mathsf{overflow}, \mathsf{0} : \underline{\mathsf{zeros}}, \mathsf{b}) \\ &\to \mathsf{f}(\mathsf{overflow}, \mathsf{overflow}, \underline{\mathsf{b}}) \\ &\to \dots \end{aligned}$$

Hence the extended system is not balanced outermost terminating although $\mathcal{R}$ is productive, establishing a counterexample to completeness of Theorem 4 of [54].

The problem is not that the stream function symbol f simultaneously consumes from two arguments; by introducing one auxiliary symbol and rule, we can construct a similar example where each function consumes exactly from one

argument. The problem is that rewriting in all coinductive arguments, including those not consumed from, is allowed.

The transformation from Definition 6.1 solves this problem by using context-sensitive rewriting to disallow rewriting in exactly those arguments ($i \notin I_{\mathsf{f}}$). The replacement map ξ is defined by $\xi_{\mathsf{f}} = \{1, 2\}$, and $\xi_{:} = \varnothing$. Without this restriction the system is not terminating. Note that the transformed system $\mathbf{T}_{\mathfrak{C}}(\mathcal{R}) = \langle \mathcal{R}, \xi \rangle$ contains no $\mathfrak{C}$-rules; all data symbols are constants ($\mathsf{0}$) and hence data-finiteness is guaranteed.

In [39], [45] and [13, 14] transformations from outermost termination to standard, innermost and context-sensitive termination are presented, respectively. The combination of [54] with these approaches results in a method for proving data-aware productivity automatically. In [55], Zantema and Raffelsieper extract the essence of the combination of [54] and [13] and propose a simplified direct transformation.

9.3. Data-Aware Productivity via Context-Sensitive Termination.

The work [55] introduces an elegant, direct transformation from productivity to context-sensitive termination. However, in contrast to our transformation, it is not complete:

Example 9.2. The following specification is given in [55] to illustrate the limitations of their transformation to context-sensitive rewriting:

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

The context-sensitive TRS resulting from the transformation in [55] allows rewriting in all coinductive arguments of defined symbols (hence also in the second argument of zip), the first rule admits an infinite rewrite sequence by unfolding the constant p repeatedly.

In [55] this specification is used to show the need of preprocessing specifications by rewriting right-hand sides. Our transformation does not need this extra preprocessing, and proves productivity directly by forbidding rewriting in the second argument of the symbol zip. Nevertheless, rewriting right-hand sides may simplify termination proofs, also in combination with the methods proposed here.

9.4. Comparison of Input Formats.

In [52, 53], Zantema defines 'proper' stream specifications. These are similar to shallow tree specifications, but differ in two aspects:

(i) In a 'proper' specification, the defining rules of a function symbol are not required to consume from the same arguments. Giving up on completeness of the transformation in Definition 6.1, the format of shallow tree specifications can be extended in this direction; see further Remark 3.15.

(ii) Unlike the shallow format, the 'proper' stream format requires the data-layer to be independent (see Definition 7.1); functions from codata to data, like $\mathsf{head}(x:\sigma) \to x$, are excluded. Employing this restriction, the 'proper' format allows for arbitrary pattern matching on data (not just one deep).

The format of semi-shallow tree specifications, investigated in Section 7, is even closer to 'proper' specifications: the formats differ only in aspect (i).

We emphasize that restriction (i) of shallow tree specifications hardly influences the applicability of the method. Specifications of infinite data structures are typically strongly sequential, and hence can be transformed into the shallow format.

In [52, 53] it is claimed without proof that every pure specification [9] can be transformed ('unfolded') into a 'proper' one. The following (non-strongly sequential) specification forms a counterexample to this claim:

$$\begin{array}{llllll}
\mathsf{h}(& \mathsf{a}:\sigma, & \mathsf{b}:\tau, & \gamma) & \to & \gamma \\
\mathsf{h}(& \sigma, & \mathsf{a}:\tau, & \mathsf{b}:\gamma) & \to & \sigma \\
\mathsf{h}(& \mathsf{b}:\sigma, & \tau, & \mathsf{a}:\gamma) & \to & \tau \\
\mathsf{h}(& \mathsf{a}:\sigma, & \mathsf{a}:\tau, & \mathsf{a}:\gamma) & \to & \gamma \\
\mathsf{h}(& \mathsf{b}:\sigma, & \mathsf{b}:\tau, & \mathsf{b}:\gamma) & \to & \tau
\end{array}$$

This function specification is a valid stream specification in the format of [9] (it is orthogonal, exhaustive and a constructor rewrite system). The first step of the transformation of [52, 53] (of 'unfolding' a specification into a 'proper' one) results in the following intermediate system:

$$\begin{array}{llllll}
\mathsf{h}(& x:\sigma, & \tau, & \gamma &) & \to & \mathsf{g}(x,\sigma,\tau,\gamma) \\
\mathsf{g}(& \mathsf{a}, & \sigma, & \mathsf{b}:\tau, & \gamma) & \to & \gamma \\
\mathsf{h}(& \sigma, & \mathsf{a}:\tau, & \mathsf{b}:\gamma &) & \to & \sigma \\
\mathsf{g}(& \mathsf{b}, & \sigma, & \tau, & \mathsf{a}:\gamma) & \to & \tau \\
\mathsf{g}(& \mathsf{a}, & \sigma, & \mathsf{a}:\tau, & \mathsf{a}:\gamma) & \to & \gamma \\
\mathsf{g}(& \mathsf{b}, & \sigma, & \mathsf{b}:\tau, & \mathsf{b}:\gamma) & \to & \tau
\end{array}$$

The problem is that non-orthogonality is introduced (the rules for h overlap), and so the transformation does not yield a 'proper' stream specification (the non-orthogonality remains in the further 'unfolding' steps).

Moreover, there is no hope of repairing this shortcoming. The above system cannot be transformed into the 'proper' stream format. Let us briefly elaborate on this fact. For every argument of h, there is one defining rule that does not consume from this argument. As a consequence, it is a priori not clear which argument needs to be evaluated to a term with a constructor at the root. Since in a 'proper' stream specification pattern matching on streams is allowed only one symbol deep, we can only match against the stream constructor ':'. For that reason it is impossible to distinguish the three different cases: either one fixes an evaluation order on the arguments, or one loses orthogonality.

Hence, contrary to the claim in [52, 53], their 'unfolding' transformation does not work for all specifications. Only strongly sequential specifications can be 'unfolded' to 'proper' specifications. However, even for strongly sequential specifications, the 'unfolding' transformation needs to be adapted: in order to avoid overlap, the order in which the arguments are unfolded is important. For example, consider the left-hand sides $\mathsf{f}(x{:}y{:}\sigma, \mathsf{a}{:}\tau)$ and $\mathsf{f}(\sigma, \mathsf{b}{:}\tau)$. Then, 'unfolding' the first argument would introduce an overlap again. Our transformation of turning non-shallow into shallow specifications given in Definition 5.1, deals with this problem by unfolding only those arguments that every defining rule consumes from.

We remark that the 'proper' format defined in [55] is slightly more general, allowing pattern matching on data even below a stream constructor. Nevertheless, the general problem remains, and a similar counterexample can be given. In particular, it is not hard to construct (non-strongly sequential) specifications which can automatically be proven productive by the method of [9], but are out of the scope of [52, 53, 55] as well as of the current paper. For example, consider the following, rather artificial, stream specification:

$$\mathsf{W} \to a : b : \mathsf{g}(\mathsf{W}, \mathsf{W}, \mathsf{W})$$

$$\begin{array}{llllll}
\mathsf{g}(& x : \mathsf{a} : \sigma, & \mathsf{b} : \tau, & \gamma) & \to & \mathsf{a} : \mathsf{g}(b : \sigma, \tau, \gamma) \\
\mathsf{g}(& \sigma, & \mathsf{a} : \tau, & \mathsf{b} : \gamma) & \to & \mathsf{a} : \mathsf{g}(\tau, \gamma, \sigma) \\
\mathsf{g}(& x : \mathsf{b} : \sigma, & \tau, & \mathsf{a} : \gamma) & \to & \mathsf{b} : \mathsf{b} : \mathsf{a} : \mathsf{g}(\tau, \tau, \tau) \\
\mathsf{g}(& x : \mathsf{a} : \sigma, & \mathsf{a} : \tau, & \mathsf{a} : \gamma) & \to & \mathsf{a} : \mathsf{b} : \mathsf{g}(\sigma, \tau, \gamma) \\
\mathsf{g}(& x : \mathsf{b} : \sigma, & \mathsf{b} : \tau, & \mathsf{b} : \gamma) & \to & x : x : x : \mathsf{g}(\sigma, \tau, \gamma)
\end{array}$$

This specification cannot be transformed into the format of [52, 53, 55]. However, productivity of W is proved automatically by the tool of [9].

9.5. Lazy Evaluation.

Lazy evaluation has been introduced in [16] using graph rewriting. In [34] Lucas presents a transformation from lazy evaluation to context-sensitive rewriting based on an extension of the rewrite system with so-called *activation rules*. In [40] Schernhammer and Gramlich improve this transformation to make it complete with respect to termination.

There are several differences with the transformation proposed by us. First, our transformation proceeds in several simple steps: (i) from lazy rewriting of $\mathcal{R}$ to a strongly sequential $\mathcal{R}'$ (Section 4), (ii) to a shallow specification $\Xi(\mathcal{R}')$ (Section 5), and finally (iii) to a context-sensitive rewrite system $\mathbf{T}(\Xi(\mathcal{R}'))$ (Section 6). We think our transformation is easier to understand and implement.

Second, we sketch a transformation that works for every *deterministic* lazy evaluation strategy. The works [34, 40] restrict themselves to a particular *non-deterministic* lazy evaluation strategy, that is, the strategy allows freedom (non-determinism) in the order of evaluation of arguments. For the termination behavior, non-determinism can only have a negative impact as it allows for

'choosing the worst case'. For example, consider the specification:

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

The term non is non-terminating with respect to the strategy from [34, 40] as the activation relation allows for activating the first as well as the second argument of f (activating the second argument leads to non-termination). If we replace the rule $\mathsf{non} \to \mathsf{f}(\mathsf{g},\mathsf{non})$ by:

$$\mathsf{non} \to \mathsf{f}(\mathsf{g},\mathsf{f}(\mathsf{non},\mathsf{g}))\,,$$

then we even obtain a system where *every* deterministic lazy evaluation strategy is terminating. On the other hand, the non-deterministic strategy from [34, 40] leads to non-termination. For this reason deterministic strategies are clearly preferable.

The term 'lazy evaluation' usually refers to evaluation strategies such as those used in functional programming languages such as Haskell and Clean. These languages employ deterministic evaluation strategies, and hence, apart from their higher-order nature, are in the scope of the transformation described in Section 4. In Haskell and Clean, the above specification would be terminating.

9.6. Context-Sensitive Matrix Interpretations

In Section 8 we present an extension of matrix interpretations [22, 15] to context-sensitive termination. A similar extension has been proposed in [2] where matrices over real coefficients are considered. We argue that its simplicity is what makes our proposal interesting. As an historical note we want to mention that Jambox was the first tool to use context-sensitive matrix interpretations, namely in the Termination Competition of 2008 [44] in the category of *Outermost Termination*, using the transformation from outermost to context-sensitive rewriting from our paper [14]. The context-sensitive termination method employed by Jambox in this competition is described in Section 8.

10. Conclusion

We defined tree specifications as sorted, exhaustive, orthogonal constructor-based TRSs and have presented a transformation from strongly sequential [23] tree specifications $\mathcal{R}$ to context-sensitive TRSs $\mathbf{T}_{\mathfrak{C}}(\Xi(\mathcal{R}))$ such that $\mathcal{R}$ is productive if and only if $\mathbf{T}_{\mathfrak{C}}(\Xi(\mathcal{R}))$ is terminating (Theorems 5.5 and 6.6). Here, the transformation Ξ is an intermediate step from strongly sequential to shallow tree specifications, where pattern matching is only one constructor symbol deep.

This is the first complete transformation from productivity to termination. Moreover we have extended the input format with respect to existing formats [55]. Strongly sequential tree specifications allow data functions to carry codata arguments.

We argued that first-order Haskell programs can be viewed as strongly sequential rewrite systems. Future work should investigate as to how far our method can be generalized to productivity and termination analysis of higher-order programs. In particular, we envisage improving the existing Haskell termination analysis [18] using the method proposed here to more adequately model lazy evaluation.

Acknowledgements. We thank Aart Middeldorp, Hans Zantema and the anonymous referees for valuable suggestions for improving draft versions of this paper and fruitful discussions on its topic.

References

[1] B. Alarcón, F. Emmes, C. Fuhs, J. Giesl, R. Gutiérrez, S. Lucas, P. Schneider-Kamp, and R. Thiemann. Improving Context-Sensitive Dependency Pairs. In *Proc. 15th Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2008)*, volume 5330 of *LNCS*, pages 636–651. Springer, 2008.

[2] B. Alarcón, S. Lucas, and R. Navarro-Marset. Using Matrix Interpretations over the Reals in Proofs of Termination. In *Proc. 9th Conf. on Programming and Computer Languages (PROLE 2009)*, 2009.

[3] S. Antoy. Definitional Trees. In *Proc. 3rd Conf. on Algebraic and Logic Programming (ALP 1992)*, volume 632 of *LNCS*, pages 143–157. Springer, 1992.

[4] W. Buchholz. A Term Calculus for (Co-)Recursive Definitions on Stream-like Data Structures. *Annals of Pure and Applied Logic*, 136(1–2):75–90, 2005.

[5] Th. Coquand. Infinite Objects in Type Theory. In *Postproc. Conf. on Types for Proofs and Programs (TYPES 1993)*, volume 806 of *LNCS*, pages 62–78. Springer, 1993.

[6] E. W. Dijkstra. On the Productivity of Recursive Definitions. EWD749, http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD749.PDF, 1980.

[7] J. Endrullis. Jambox, 2009. Available at http://joerg.endrullis.de.

[8] J. Endrullis, R. C. de Vrijer, and J. Waldmann. Local Termination: Theory and Practice. *Logical Methods in Computer Science*, 6(3), 2010.

[9] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-Oblivious Stream Productivity. In *Proc. 15th Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2008)*, number 5330 in LNCS, pages 79–96. Springer, 2008.

[10] J. Endrullis, C. Grabmayer, and D. Hendriks. Complexity of Fractran and Productivity. In *Proc. 22nd Conf. on Automated Deduction (CADE 2009)*, volume 5663 of *LNCS*, pages 371–387. Springer, 2009.

[11] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of Stream Definitions. In *Proc. 16th Symp. on Fundamentals of Computation Theory (FCT 2007)*, number 4639 in LNCS, pages 274–287. Springer, 2007.

[12] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of Stream Definitions. *Theoretical Computer Science*, 411:765–782, 2010. Extended version of [11].

[13] J. Endrullis and D. Hendriks. From Outermost to Context-Sensitive Rewriting. In *Proc. 20th Conf. on Rewriting Techniques and Applications (RTA 2009)*, volume 5595 of *LNCS*, pages 305–319. Springer, 2009.

[14] J. Endrullis and D. Hendriks. Transforming Outermost into Context-Sensitive Rewriting. *Logical Methods in Computer Science*, 6(2), 2010.

[15] J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

[16] W. Fokkink, J. Kamperman, and P. Walters. Lazy Rewriting on Eager Machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86, 2000.

[17] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proc. 3rd Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNAI*, pages 281–286. Springer, 2006.

[18] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *Proc. 17th Conf. on Rewriting Techniques and Applications (RTA 2006)*, volume 4098 of *LNCS*, pages 297–312. Springer, 2006.

[19] E. Giménez. Codifying Guarded Definitions with Recursive Schemes. In *Postproc. Conf. on Types for Proofs and Programs (TYPES 1993)*, volume 806 of *LNCS*, pages 39–59. Springer, 1993.

[20] R. Gutiérrez and S. Lucas. Proving Termination in the Context-Sensitive Dependency Pair Framework. In *Proc. 8th Workshop on Rewriting Logic and its Applications (WRLA 2010)*, volume 6381 of *LNCS*, pages 18–34. Springer, 2010.

[21] M. Hanus, S. Lucas, and A. Middeldorp. Strongly Sequential and Inductively Sequential Term Rewriting Systems. *Information Processing Letters*, 67(1):1–8, 1998.

[22] D. Hofbauer and J. Waldmann. Proving Termination with Matrix Interpretations. In *Proc. 17th Conf. Rewriting Techniques and Applications (RTA 2006)*, volume 4098, pages 328–342. Springer, 2006.

[23] G.P. Huet and J.-J. Lévy. Computations in Orthogonal Rewriting Systems, I and II. In *Computational Logic — Essays in Honor of Alan Robinson*, pages 396–443. MIT Press, 1991.

[24] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. 23rd Symp. on Principles of Programming Languages (POPL 1996)*, pages 410–423, 1996.

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

[26] M. Iwami. Persistence of Termination for Non-Overlapping Term Rewriting Systems. In *Proc. of Int. Forum on Information and Computer Technology*, pages 198–202, 2003.

[27] D. Kapur, P. Narendran, D. J. Rosenkrantz, and H. Zhang. Sufficient-Completeness, Ground-Reducibility and their Complexity. *Acta Informatica*, 28(4):311–350, 1991.

[28] R. Kennaway, J. W. Klop, R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation*, 119(1):18–38, 1995.

[29] J. W. Klop and R. C. de Vrijer. Infinitary Normalization. In *We Will Show Them: Essays in Honour of Dov Gabbay (2)*, pages 169–192. College Publications, 2005. ftp://ftp.cwi.nl/pub/CWIreports/SEN/SEN-R0516.pdf.

[30] J. W. Klop and A. Middeldorp. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, 12(2):161–196, 1991.

[31] S. Lucas. Context-Sensitive Computations in Functional and Functional Logic Programs. *Journal of Functional and Logic Programming*, 1998(1), 1998.

[32] S. Lucas. Transfinite Rewriting Semantics for Term Rewriting Systems. In *Proc. 12th Conf. on Rewriting Techniques and Applications (RTA 2001)*, volume 2051 of *LNCS*, pages 216–230. Springer, 2001.

[33] S. Lucas. Context-Sensitive Rewriting Strategies. *Information and Computation*, 178(1):294–343, 2002.

[34] S. Lucas. Lazy Rewriting and Context-Sensitive Rewriting. In *Proc. of 10th Workshop of Functional and (Constraint) Logic Programming (WFLP 2001)*, number 64 in ENTCS. Elsevier, 2002.

[35] S. Lucas. mu-Term: A Tool for Proving Termination of Context-Sensitive Rewriting. In *Proc. 15th Conf. on Rewriting Techniques and Applications (RTA 2004)*, volume 3091 of *LNCS*, pages 200–209. Springer, 2004.

[36] A. Middeldorp. Call by Need Computations to Root-Stable Form. In *Proc. 24th Symp. on Principles of Programming Languages (POPL 1997)*, pages 94–105. ACM, 1997.

[37] S. Peyton-Jones. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[38] M. J. Plasmeijer and M. van Eekelen. The Concurrent Clean Language Report (version 2.0). Technical report, University of Nijmegen, 2001.

[39] M. Raffelsieper and H. Zantema. A Transformational Approach to Prove Outermost Termination Automatically. In *Proc. 8th Workshop on Reduction Strategies in Rewriting and Programming (WRS 2008)*, volume 237 of *ENTCS*, pages 3–21. Elsevier, 2009.

[40] F. Schernhammer and B. Gramlich. Termination of Lazy Rewriting Revisited. In *Proc. 7th Workshop on Reduction Strategies in Rewriting and Programming (WRS 2007)*, volume 204 of *ENTCS*, pages 35–51. Elsevier, 2008.

[41] B. A. Sijtsma. On the Productivity of Recursive List Definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.

[42] A. Telford and D. Turner. Ensuring Streams Flow. In *Proc. 5th Conf. on Algebraic Methodology and Software Technology (AMAST 1997)*, volume 1349 of *LNCS*, pages 509–523. Springer, 1997.

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

[44] Termination Portal. http://www.termination-portal.org/, 2008. Termination Competition and Termination Problems Data Base (TPDB).

[45] R. Thiemann. From Outermost Termination to Innermost Termination. In *Proc. 35th Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM 2009)*, volume 5404 of *LNCS*, pages 533–545. Springer, 2009.

[46] Y. Toyama, J. W. Klop, and H. P. Barendregt. Termination for Direct Sums of Left-Linear Complete Term Rewriting Systems. *J. ACM*, 42:1275–1304, 1995.

[47] D. A. Turner. An Overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, 1986.

[48] W. W. Wadge. An Extensional Treatment of Dataflow Deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

[49] H. Zantema. Termination of Term Rewriting: Interpretation and Type Elimination. *Journal of Symbolic Computation*, 17(1):23–50, 1994.

[50] H. Zantema. Termination of Term Rewriting by Semantic Labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

[51] H. Zantema. Termination of Context-Sensitive Rewriting. In *Proc. 8th Conf. on Rewriting Techniques and Applications (RTA 1997)*, volume 1232 of *LNCS*, pages 172–186. Springer, 1997.

[52] H. Zantema. Well-Definedness of Streams by Termination. In *Proc. 20th Conf. on Rewriting Techniques and Applications (RTA 2009)*, volume 5595 of *LNCS*, pages 164–178. Springer, 2009.

[53] H. Zantema. Well-Definedness of Streams by Transformation and Termination. *Logical Methods in Computer Science*, 6(3), 2010.

[54] H. Zantema and M. Raffelsieper. Stream Productivity by Outermost Termination. In *Proc. 9th Workshop on Reduction Strategies in Rewriting and Programming (WRS 2009)*, volume 15 of *EPTCS*, pages 83–95, 2009.

[55] H. Zantema and M. Raffelsieper. Proving Productivity in Infinite Data Structures. In *Proc. 21st Conf. on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *LIPIcs*, pages 401–416. Schloss Dagstuhl, 2010.