Theoretical Computer Science 411 (2010) 765–782

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Productivity of stream definitions☆

Jörg Endrullis[a], Clemens Grabmayer[b], Dimitri Hendriks[a,*], Ariya Isihara[a], Jan Willem Klop[a]

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

[b] Universiteit Utrecht, Department of Philosophy, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

ARTICLE INFO

Keywords:
Recursive stream definitions
Productivity
Functional programming
Dataflow networks

ABSTRACT

We give an algorithm for deciding productivity of a large and natural class of recursive stream definitions. A stream definition is called 'productive' if it can be evaluated continually in such a way that a uniquely determined stream in constructor normal form is obtained as the limit. Whereas productivity is undecidable for stream definitions in general, we show that it can be decided for 'pure' stream definitions. For every pure stream definition the process of its evaluation can be modelled by the dataflow of abstract stream elements, called 'pebbles', in a finite 'pebbleflow net(work)'. And the production of a pebbleflow net associated with a pure stream definition, that is, the amount of pebbles the net is able to produce at its output port, can be calculated by reducing nets to trivial nets.

1. Introduction

In functional programming, term rewriting and λ-calculus, there is a wide arsenal of methods for proving termination such as recursive path orders, dependency pairs (for term rewriting systems, [24]) and the method of computability (for λ-calculus, [22]). All of these methods pertain to finite data only. In the last two decades interest has grown towards infinite data, as witnessed by the application of type theory to infinite objects [8], and the emergence of coalgebraic techniques for infinite data types like streams [20]. While termination cannot be expected when infinite data are processed, infinitary notions of termination become relevant. For example, in frameworks for the manipulation of infinite objects such as infinitary rewriting [15] and infinitary λ-calculus [16], basic notions are the properties WN^{∞} and SN^{∞} of infinitary weak and strong normalization [17], and UN^{∞} of uniqueness of (infinitary) normal forms.

In the functional programming literature the notion of 'productivity' has arisen, initially in the pioneering work of Sijtsma [21], as a natural strengthening of what in our setting are the properties WN^{∞} and UN^{∞}. A stream definition is called productive if not only can the definition be evaluated continually to build up a unique infinite normal form, but the resulting infinite expression is also meaningful in the sense that it is a constructor normal form which allows us to consecutively read off individual elements of the stream. Since productivity of stream definitions is undecidable in general, the challenge is to find ever larger classes of stream definitions significant to programming practice for which productivity is decidable, or for which at least a powerful method for proving productivity exists.

Contribution and overview. We show that productivity is decidable for a rich class of recursive stream specifications that hitherto could not be handled automatically. (Since a stream definition, in the sense most commonly used, *defines* a stream only in the case that it is productive, here and henceforth we use the more accurate term 'stream specification'.) We start

☆ This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.502.

* Corresponding author.
E-mail addresses: joerg@few.vu.nl (J. Endrullis), clemens@phil.uu.nl (C. Grabmayer), diem@cs.vu.nl (D. Hendriks), ariya@few.vu.nl (A. Isihara), jwk@cs.vu.nl (J.W. Klop).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.10.014

with a brief introduction to infinitary rewriting, and define some preliminary notions in Section 2. In Section 3 we define 'pure stream constant specifications' (SCSs) as orthogonal term rewriting systems, which are based on 'weakly guarded stream function specifications' (SFSs). In Section 4 we develop a 'pebbleflow calculus' as a tool for computing the 'degree of definedness' of SCSs. The idea is that a stream element is modelled by an abstract 'pebble', a stream specification by a finite 'pebbleflow net', and the process of evaluating a stream specification by the dataflow of pebbles in the associated net. In Section 5, we give a translation of SCSs into 'rational' pebbleflow nets, and prove that this translation is production preserving. Finally in Section 6, we show that the production of a 'rational' pebbleflow net, that is, the amount of pebbles such a net is able to produce at its output port, can be calculated by an algorithm that reduces nets to trivial nets. We obtain that productivity is decidable for pure SCSs. We believe our approach is natural because it is based on building a pebbleflow net corresponding to an SCS as a model that is able to reflect the local consumption/production steps during the evaluation of the stream specification in a quantitatively precise manner.

This paper is a revised and extended version of the paper [11] presented at FCT 2007. We follow [21,7] in describing the quantitative input/output behaviour of a stream function f by a 'modulus of production' $\nu_{\mathsf{f}} : (\overline{\mathbb{N}})^r \to \overline{\mathbb{N}}$ with the property that the first $\nu_{\mathsf{f}}(n_1, \ldots, n_r)$ elements of $\mathsf{f}(t_1, \ldots, t_r)$ can be computed whenever the first n_i elements of t_i are defined. In fact, our approach is distinguished by the use of *optimal* moduli. Moreover, our decision algorithm exploits moduli that are 'rational' functions $\nu : (\overline{\mathbb{N}})^r \to \overline{\mathbb{N}}$ which, for $r = 1$, have eventually periodic difference functions $\Delta_\nu(n) := \nu(n+1) - \nu(n)$, that is $\exists n, p \in \mathbb{N}.\ \forall m \geq n.\ \Delta_\nu(m) = \Delta_\nu(m+p)$. This class of moduli is effectively closed under composition, and allows us to calculate fixed points of unary functions. Rational production moduli generalise those employed by [25,13,8,23], and enable us to precisely capture the consumption/production behaviour of a large class of stream functions.

Related work. In order to facilitate a comparison of our contribution with previous approaches, we describe the various notions of production moduli that have been proposed.

It is well-known that networks are devices for computing least fixed points of systems of equations [14]. The notion of 'productivity' (sometimes also referred to as 'liveness') was first mentioned by Dijkstra [9]. Since then several papers [25,21, 8,13,23,7] have been devoted to criteria ensuring productivity. The common essence of these approaches is a quantitative analysis.

In [25] Wadge uses dataflow networks to model fixed points of equations. He devises a so-called 'cyclic sum test', using production moduli of the form $\nu(n_1, \ldots, n_r) = \min(n_1 + a_1, \ldots, n_r + a_r)$ with $a_i \in \mathbb{Z}$, i.e. the output 'leads' or 'lags' the input by a fixed value a_i.

Sijtsma [21] points out that this class of production moduli is too restrictive to capture the behaviour of commonly used stream functions like even or zip. For instance, consider:

$$\mathsf{M} = 0 : \mathsf{zip}(\mathsf{inv}(\mathsf{even}(\mathsf{M})), \mathsf{tail}(\mathsf{M}))\ ,$$

a definition of the Thue–Morse sequence (see also Fig. 1), which we use as a running example, cannot be dealt with by the cyclic sum test and other methods mentioned below. Therefore Sijtsma develops an approach allowing arbitrary production moduli $\nu_f : \mathbb{N}^r \to \mathbb{N}$, having the only drawback of not being automatable in full generality.

In order to formalise coinductive types in type theory, Coquand [8] defines a syntactic criterion called 'guardedness' for ensuring productivity. Giménez [12] implements a modified version of this criterion in the Coq proof assistant. This notion of guarded recursion avoids the introduction of non-normalisable terms, but is too restrictive for programming practice, because it disallows function applications to recursive calls, like even(M) in the definition of M above.

Telford and Turner [23] extend the notion of guardedness with a method in the flavour of Wadge. They use a sophisticated counting scheme to compute the 'guardedness level' of a stream function, an element in $\mathbb{Z} \cup \{-\omega, \omega\}$. With this, a stream specification is recognised to be productive if the result of computing its guardedness level (by plain addition in the case of unary functions) from the guardedness levels of the stream functions occurring is positive. However, their approach does not overcome Sijtsma's criticism: their production moduli are essentially the same as Wadge's. Determining a guardedness level x, hence a modulus of the form $n \mapsto n + x$, for the stream function even leaves $x = -\omega$ as the only possibility. As a consequence, their algorithm does not recognise the specification of M to be productive.

Hughes, Pareto and Sabry [13] introduce a type system using production moduli with the property that $\nu_f(a \cdot x + b) = c \cdot x + d$ for some $a, b, c, d \in \mathbb{N}$. For instance, the type assigned there to the stream function tail is $\mathrm{St}^{i+1} \to \mathrm{St}^i$. Hence their system rejects the stream specification for M because the subterm tail(M) cannot be typed. Moreover, their class of moduli is not closed under composition, leading to the need for approximations and a loss of power.

Buchholz [7] presents a formal type system for proving productivity, whose basic ingredients are, closely connected to [21], unrestricted production moduli $\nu_f : \mathbb{N}^r \to \mathbb{N}$. In order to obtain an automatable method, Buchholz also devises a syntactic criterion to ensure productivity. This criterion easily handles all the examples of [23], but fails to deal with functions that have a negative effect like even, and hence with the specification of M above.

2. Infinitary rewriting

The theoretical foundation and background of our work is that of infinitary rewriting, ensuring us of unique normalisation results when dealing with infinite objects such as streams, that are computed in an infinite timescale. For general reference and a more complete introduction we mention [24,17,15]. Here we just give a succinct introduction to the notions of

$\mathsf{M} \to 0 : \mathsf{zip}(\mathsf{inv}(\mathsf{even}(\mathsf{M})), \mathsf{tail}(\mathsf{M}))$		SCS layer
$\mathsf{zip}(x : \sigma, \tau)$	$\to x : \mathsf{zip}(\tau, \sigma)$	
$\mathsf{inv}(x : \sigma)$	$\to \mathsf{i}(x) : \mathsf{inv}(\sigma)$	
$\mathsf{even}(x : \sigma)$	$\to x : \mathsf{odd}(\sigma)$	SFS layer
$\mathsf{odd}(x : \sigma)$	$\to \mathsf{even}(\sigma)$	
$\mathsf{tail}(x : \sigma)$	$\to \sigma$	
$\mathsf{i}(0) \to 1$	$\mathsf{i}(1) \to 0$	data layer

Fig. 1. Example of a productive SCS.

infinitary rewriting. We will do this in the following somewhat informal glossary, starting with the notion of infinitary rewriting itself. Thereafter, these preliminary notions will be given in more technical detail.

2.1. Glossary

(I) *Infinitary rewriting* is a natural extension of ordinary, finitary rewriting, by allowing terms to have infinite branches. Canonical examples are infinite (term) trees, infinite streams of data, or infinite λ-terms. For infinite λ-terms, see [5] for untyped infinite λ-terms, and [1] for simply typed infinite λ-terms. In mathematical and physical theories the use of such infinite objects is commonplace in the form of infinite expansions and power series. In λ-calculus a particular class of infinite λ-terms is well known as Böhm trees [4].

(II) *Infinite terms*. Formally, one can introduce infinite terms in several ways: most concretely as partial mappings from the set of positions $\mathbb{N}^*$ to the alphabet symbols of some signature Σ, or by means of coinductive notions, or as the completion of the metric space of finite terms with the usual metric based on the familiar notion of distance that yields distance $2^{-(n+1)}$ for terms that are identical up to and including level n from the root, but then have a difference (see Definition 2.2). In this complete metric space of finite and infinite terms we have the notion of Cauchy convergence.

(III) *Reduction (or rewriting) sequences*. In ordinary, finitary, rewriting theory rewriting sequences (we also use 'reduction' for 'rewriting') are just finite or infinite. This view is much more refined in infinitary rewriting, by allowing rewrite sequences of any countable ordinal length. The passage over limit ordinals is given by a strengthened notion of Cauchy convergence, called 'strong convergence'.

Note that Cauchy convergence of reduction sequences is not yet sufficient to make a reduction sequence 'connected', as it should be; without more it could jump at a limit stage to a totally unrelated term. So, we evidently have to impose the requirement of continuity that at a limit stage λ the reduction sequence proceeds with the limit of the prefix up to λ. Thus, e.g., in the reduction sequence $t_0 \to r_1 \to \ldots t_\omega \to t_{\omega+1} \to \ldots$ the term t_ω equals $\lim_{i\to\omega} t_i$ (see [17] for several examples of transfinite reductions sequences).

(IV) *Strong convergence* is Cauchy convergence with the extra requirement that the 'activity', that is the depth of the successive redex contractions ('firings') in a rewrite sequence, has to go deeper and deeper when approaching a limit ordinal. For the rationale of this requirement and a more detailed introduction we refer to [15,24]. Here we just mention the essential benefit of this requirement: it provides us with a natural notion of 'descendant' or 'residual', also in limit passages. In classical λ-calculus and term rewriting, the ubiquitous notion of descendant or residual has proved to be indispensable for a fruitful development of the theory.

(V) *Compressing transfinite rewriting sequences*. The full-fledged framework of infinitary rewriting comprises transfinitely long rewrite sequences. However, when one wishes to avoid the transfinite realm, there is the sub-framework where one restricts to the first infinite ordinal ω. What we definitely do retain (see Definition 2.3) is the notion of strong convergence: also in a rewrite sequence of length ω, the redex depth of the sequence must tend to ∞: the evaluation is not allowed to stagnate at some finite level, but must proceed and deliver more and more levels of the constructors that we are interested in. Technically, the property that yields this modest framework of reductions of length not exceeding ω, is the Compression Property, stating that every transfinitely long rewrite sequence can be compressed, by a dove-tailing strategy of redex selection, to a rewrite sequence with the same begin and end point, but of length $\leq \omega$. The property holds when we deal with a class of well-behaved systems of rewrite rules, known as 'orthogonal rewrite systems'.

(VI) *Orthogonal term rewriting systems (TRSs) with constructors*. We will employ orthogonal TRSs, formally introduced in Definition 2.1. These systems have been widely studied, and admit an extensive and elegant theory. Orthogonality means that the reduction rules are left-linear (no variable occurs twice in the left-hand side), and there are no critical pairs. The signatures will contain, next to 'defined function symbols', also 'constructor symbols'; they are not meant to be rewritten, but are generating the finite data or infinite 'codata'. We adhere to a sorting discipline; the straightforward details are stated in Section 2.2.

(VII) *The basic finitary notions* CR, UN, SN, WN. We briefly recall the basic properties pertaining to finite rewriting: CR is the Church–Rosser or confluence property, stating that every pair of coinitial reductions can be prolonged to a common reduct; UN is the immediate corollary to CR that ensures the uniqueness of normal forms (terms without redexes): two

finite reductions ending in normal forms, end in the same normal form. As is well-known, for orthogonal TRSs the properties CR and hence UN always hold, and this is even so for weakly orthogonal TRSs, where trivial critical pairs are allowed. The property SN, Strong Normalisation, states that there are no infinite reductions, or rephrased, that every reduction must terminate eventually, if prolonged 'long enough'. Weak Normalization (WN) just states that there exists a reduction to normal form.

(VIII) *The basic infinitary notions* CR^∞, UN^∞, SN^∞, WN^∞. In the realm of infinitary reductions the finitary properties introduced above generalise to analogous notions indicated by the superscript ∞. CR^∞ states that infinitary reductions starting from the same (finite or infinite) term can be prolonged to a common reduct, using infinitary reductions. The property UN^∞ states that two such infinitary reductions cannot end in two different possibly infinite normal forms. The generalisation to WN^∞ is obvious too: there exists an infinitary reduction to normal form. The definition of the stronger property SN^∞ is more subtle: roughly, it means that we are bound to find a normal form when we reduce, transfinitely, long enough. This entails that it is guaranteed that the reduction cannot 'stagnate' at some finite level. For a more precise discussion of this notion we refer to [17] or [24]. Here we are restricting ourselves to the versions of the infinitary generalisations of WN^∞ and SN^∞ where we deal only with reduction sequences not exceeding length ω. For the general background we mention that even for orthogonal TRSs the property CR^∞ fails, due to the possible presence of 'collapsing' reduction rules (they have as right-hand side a single variable). However, UN^∞ does hold for orthogonal TRSs. Remarkably, for weakly orthogonal TRSs UN^∞ fails. In [17] it is proved that the global versions of SN^∞ and WN^∞, where 'global' means that they hold for all terms, are in fact equivalent, even without the condition of non-erasing rules (as the analogous equivalence for finite reductions requires).

(IX) *Productivity*. In the setting of orthogonal TRSs, productivity is a strengthening of WN^∞ where we require that the normal form, whose existence is assured by WN^∞, must be not just any (possibly infinite) normal form, but one that is 'intended', namely, built solely from constructors. Productivity is the main property that we will be concerned with in this paper.

2.2. Preliminaries

Let $\mathbb{N}_+ := \mathbb{N} \setminus \{0\}$. We consider a finite or infinite term as a function on a prefix closed subset of $\mathbb{N}_+^*$ taking values in a first-order signature, adhering to a sortedness discipline. Let U be a finite set of *sorts*. A *U-sorted set* A is a family of sets $\{A_u\}_{u\in U}$. We sometimes write $x \in A$ as a shorthand for $\exists u \in U.\ x \in A_u$. A *$U$-sorted signature* Σ is a U-sorted set of symbols f, each equipped with an arity $\langle u_1 \cdots u_n, u\rangle \in U^* \times U$, for which we will use the suggestive type notation $u_1 \times \cdots \times u_n \to u$, where $f \in \Sigma_u$. Let $\mathcal{X}$ be a U-sorted set of variables. Then, a *term over Σ and $\mathcal{X}$ of sort $u \in U$* is a partial map $t : \mathbb{N}_+^* \rightharpoonup \Sigma \cup \mathcal{X}$ such that:

(i) its *root* $t(\epsilon)$ is defined and has sort u;
(ii) its domain is prefix closed: $p \in \mathrm{dom}(t)$ whenever $pi \in \mathrm{dom}(t)$, for all $p \in \mathbb{N}_+^*$, $i \in \mathbb{N}_+$;
(iii) symbol arities define the number of immediate subterms and their respective sorts: for all $p \in \mathbb{N}_+^*$, if $t(p) \in \mathcal{X}$ then $pi \notin \mathrm{dom}(t)$, for all $i \in \mathbb{N}_+$, and if $t(p) \in \Sigma$ with arity $u_1 \times \cdots \times u_n \to u$, then $t(pi) \in (\Sigma \cup \mathcal{X})_{u_i}$ if $1 \le i \le n$, and $pi \notin \mathrm{dom}(t)$ otherwise.

The set of terms over Σ and $\mathcal{X}$ of sort $u \in U$ is denoted by $\mathit{Ter}_\infty(\Sigma, \mathcal{X})_u$. Usually we keep the variables implicit, assuming a countably infinite set $\mathcal{X}$, and write $\mathit{Ter}_\infty(\Sigma)_u$. The set $\mathit{Ter}_\infty(\Sigma)$ of all terms is defined by $\mathit{Ter}_\infty(\Sigma) := \bigcup_{u\in U} \mathit{Ter}_\infty(\Sigma)_u$. The set of *positions* $\mathcal{P}os(t)$ *of a term* $t \in \mathit{Ter}_\infty(\Sigma)$ is the domain of t. A term t is called *finite* if the set $\mathcal{P}os(t)$ is finite. We write $\mathit{Ter}(\Sigma)$ for the set of finite terms. We use the symbol $\equiv$ to indicate syntactical equality of terms. For positions $p \in \mathcal{P}os(t)$ we use $t|_p$ to denote the *subterm of t at position p*, defined by $t|_p(q) := t(pq)$ for all $q \in \mathbb{N}_+^*$.

For $f \in \Sigma$ with arity $u_1 \times \cdots \times u_n \to u$ and terms $t_i \in \mathit{Ter}_\infty(\Sigma)_{u_i}$ we write $f(t_1, \ldots, t_n)$ to denote the term $t \in \mathit{Ter}_\infty(\Sigma)_u$ that is defined by $t(\epsilon) = f$, and $t(ip) = t_i(p)$ for all $1 \le i \le n$ and $p \in \mathbb{N}_+^*$. For *constants* $c \in \Sigma$ we simply write c instead of $c()$. We use $x, y, z, \ldots$ to range over variables.

Definition 2.1. A *U-sorted term rewriting system (TRS)* is a pair $\langle \Sigma, R\rangle$ consisting of a finite, U-sorted signature Σ and a finite, U-sorted set R of *rules* with finite left- and right-hand sides that satisfy well-sortedness: $R_u \subseteq \mathit{Ter}(\Sigma, \mathcal{X})_u \times \mathit{Ter}(\Sigma, \mathcal{X})_u$ for all $u \in U$, as well as the standard TRS requirements: for all rules $\ell \to r \in R$, ℓ is not a variable, and all variables in r also occur in ℓ.

Let $\mathcal{T} = \langle \Sigma, R\rangle$ be a U-sorted TRS. We define $\mathcal{D}(\Sigma) := \{\mathit{root}(l) \mid l \to r \in R\}$, the set of *defined symbols*, and $\mathcal{C}(\Sigma) := \Sigma \setminus \mathcal{D}(\Sigma)$, the set of *constructor symbols*. $\mathcal{T}$ is called a *constructor TRS* if, for every rewrite rule $\rho \in R$, the left-hand side is of the form $F(t_1, \ldots, t_n)$ with $F \in \mathcal{D}(\Sigma)$ and $t_i \in \mathit{Ter}(\mathcal{C}(\Sigma))$; then ρ is a *defining rule for F*.

A *substitution* is a U-sorted map $\sigma : \mathcal{X} \to \mathit{Ter}_\infty(\Sigma, \mathcal{X})$, that is, $\forall u \in U, x \in \mathcal{X}_u.\ \sigma(x) \in \mathit{Ter}_\infty(\Sigma, \mathcal{X})_u$. For terms $t \in \mathit{Ter}_\infty(\Sigma, \mathcal{X})$ and substitutions σ we define $t\sigma$ as the result of replacing each $x \in \mathcal{X}$ in t by $\sigma(x)$. Formally, $t\sigma$ is defined, for all $p \in \mathbb{N}_+^*$, by: $t\sigma(p) = \sigma(t(p_0))(p_1)$ if there exist $p_0, p_1 \in \mathbb{N}_+^*$ such that $p = p_0p_1$ and $t(p_0) \in \mathcal{X}$, and $t\sigma(p) = t(p)$, otherwise. Let $[]$ be a fresh symbol, $[] \notin \Sigma \cup \mathcal{X}$. A *context* C is a term from $\mathit{Ter}_\infty(\Sigma, \mathcal{X} \cup \{[]\})$ containing precisely one occurrence of $[]$. By $C[s]$ we denote the term $C\sigma$ where $\sigma([]) = s$ and $\sigma(x) = x$ for all $x \in \mathcal{X}$.

Definition 2.2. On the set of terms $\mathit{Ter}_\infty(\Sigma)$ we define a *metric* d by $d(s, t) = 0$ whenever $s \equiv t$, 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 $s(p) \neq t(p)$.

A TRS $\mathcal{T}$ induces a rewrite relation on the set of terms as follows.

Definition 2.3. Let $\mathcal{T}$ be a TRS over Σ. For terms $s, t \in Ter_\infty(\Sigma)$ and $p \in \mathbb{N}_+^*$ we write $s \to_{\mathcal{T},p} t$ if there exist a rule $\ell \to r \in R$, a substitution σ and a context C with $C(p) = [\,]$ such that $s \equiv C[\ell\sigma]$ and $t \equiv C[r\sigma]$. We write $s \to_R t$ if there exists a position p such that $s \to_{\mathcal{T},p} t$.

A reduction sequence $t_0 \to_{\mathcal{T},p_0} t_1 \to_{\mathcal{T},p_0} \dots$ of length ω is called *strongly convergent* if $\lim_{i\to\infty} |p_i| = \infty$, that is, the lengths of the positions of the redexes contracted in the rewrite sequence tend to infinity.

Definition 2.4. Let $\mathcal{T}$ be a TRS and $t_0 \in Ter_\infty(\Sigma)$. Then $\mathcal{T}$ is called *ω-convergent for t_0*, denoted by $\mathrm{SN}^\omega_\mathcal{T}(t_0)$, if every reduction $t_0 \to_{\mathcal{T},p_0} t_1 \to_{\mathcal{T},p_0} \dots$ of length ω is strongly convergent. $\mathcal{T}$ is called *weakly ω-normalising for t_0*, denoted by $\mathrm{WN}^\omega_\mathcal{T}(t_0)$, if there exists a reduction $t_0 \to_{\mathcal{T},p_0} t_1 \to_{\mathcal{T},p_0} \dots$ which is either finite and ends in a normal form, or is strongly convergent, and the limit term $t_\omega := \lim_{i\to\infty} t_i$ is a normal form.

For the general definitions of SN^∞ and WN^∞ based on transfinite rewrite sequences we refer to [17]. For orthogonal TRSs (i.e. left-linear, and non-overlapping redexes, see [24]) infinitary strong normalisation SN^∞ and infinitary weak normalisation WN^∞ coincide with the properties SN^ω and WN^ω, respectively:

Lemma 2.5. *Let $\mathcal{T}$ be an orthogonal TRS and $t_0 \in Ter_\infty(\Sigma)$. Then we have*

- *$\mathcal{T}$ is* infinitary strongly normalising for t_0, *denoted by* $\mathrm{SN}^\infty_\mathcal{T}(t_0)$, *if and only if* $\mathrm{SN}^\omega_\mathcal{T}(t_0)$ *holds, and*
- *$\mathcal{T}$ is* infinitary weakly normalising for t_0, *denoted by* $\mathrm{WN}^\infty_\mathcal{T}(t_0)$, *if and only if* $\mathrm{WN}^\omega_\mathcal{T}(t_0)$ *holds.*

We write $\mathrm{SN}^\infty_\mathcal{T}$ shortly for $\mathrm{SN}^\infty_\mathcal{T}(Ter_\infty(\Sigma))$, that is, infinitary normalisation on all terms. Furthermore, the subscript $\mathcal{T}$ may be suppressed if it is clear from the context.

Note that, for non-left-linear TRSs the lemma does not hold. For instance, consider the TRS $f(x, x) \to f(a, b)$, $a \to s(a)$ and $b \to s(b)$. Then every reduction of length ω starting from $f(a, b)$ is strongly convergent, that is, the depths of the redexes contracted in the terms of the reduction tend to infinity. Nevertheless, $f(a, b)$ is neither strongly nor weakly infinitary normalising: we have $f(a, b) \to^\omega f(s^\omega, s^\omega)$, but the limit term $f(s^\omega, s^\omega)$ is not a normal form and even in transfinitely many steps it does not rewrite to one.

Since outermost-fair rewriting is an infinitary normalising strategy for orthogonal TRSs, it is also possible to characterise WN^∞ as follows.

Lemma 2.6. *Let $\mathcal{T}$ be an orthogonal TRS and $t \in Ter_\infty(\Sigma)$. Then $\mathrm{WN}^\infty_\mathcal{T}(t)$ holds if and only if every outermost-fair rewrite sequence $t_0 \to_{\mathcal{T},p_0} t_1 \to_{\mathcal{T},p_0} \dots$ of length ω is strongly convergent.*

For every strongly convergent, outermost-fair rewrite sequence $t_0 \to_{\mathcal{T},p_0} t_1 \to_{\mathcal{T},p_0} \dots$ in an orthogonal TRS, the limit term $\lim_{i\to\infty} t_i$ is a normal form. Therefore, the important direction of the above lemma is the 'only-if'-part. That is, in case $\mathrm{WN}^\infty_\mathcal{T}(t)$ holds, then every outermost-fair rewrite sequence converges towards a normal form. This normal form is unique, since orthogonal TRSs have the property UN^∞ (infinitary unique normal forms), that is, whenever $t \twoheadrightarrow n_1$ and $t \twoheadrightarrow n_2$ for normal forms n_1 and n_2, then $n_1 \equiv n_2$.

Productivity is a strengthening of infinitary weak normalisation, where we require that the unique normal form is a constructor normal form.

Definition 2.7 (*Productivity*). Let $\mathcal{T}$ be an orthogonal TRS, and let $t \in Ter_\infty(\Sigma)$. Then $\mathcal{T}$ is called *productive for t* if $\mathrm{WN}^\infty_\mathcal{T}(t)$ holds and the unique normal form of t is a constructor normal form.

Note that, we define productivity as a strengthening of WN^∞ and not of SN^∞. Definition 2.7 captures the intuitive notion of well-definedness of specifications of infinite structures in lazy functional programming languages like Haskell, Miranda or Clean. For example, consider the following Haskell program:

```
alt = tail(alt')
alt' = 0:1:alt'
```

Here, `alt` is perfectly well-defined, rewriting to an infinite list in the limit. However, if we only unfold `alt'` without reducing `tail`, then we obtain `tail(0:1:0:1...)` after ω many steps in the limit. Thus, although the stream constant `alt` in this system is SN^∞ we have to use an outermost-fair strategy to obtain a constructor normal form of `alt` within ω many steps. Another example is the Haskell program:

```
zeros = f(c)
c = c
f(x) = 0:f(x)
```

The system is not SN^∞, since the term `f(c)` rewrites to itself. Nevertheless, `zeros` is productive and Haskell evaluates `zeros` to a list of zeros, infinite in the limit. The reason is the one mentioned above: every lazy functional programming language essentially uses some form of outermost-fair (or outermost-needed) rewriting strategy, also called 'lazy evaluation', see e.g. [19].

3. Recursive stream specifications

We introduce the concepts of 'stream constant specification' (SCS) and 'stream function specification' (SFS). An SCS consists of three layers: the *SCS layer* where stream constants are specified using stream and data function symbols that are defined by the rules of the underlying SFS. An SFS consists of an *SFS layer* and a *data layer*. These notions are illustrated by the SCS given in Fig. 1. This SCS is productive and defines the well-known Thue–Morse sequence; indeed the constant M rewrites to $0:1:1:0:1:0:0:1:\ldots$ in the limit. A subtle point here is the definition of the stream function zip; had we used the rule $\mathsf{zip}^*(x:\sigma, y:\tau) \to x:y:\mathsf{zip}^*(\sigma,\tau)$ instead, then M would not produce a second element, for, in the right-hand side of M, zip^* will never match against a constructor in its second argument. Furthermore, we mention that the rule for M could be simplified to $\mathsf{M} \to 0:\mathsf{zip}(\mathsf{inv}(\mathsf{M}),\mathsf{tail}(\mathsf{M}))$. We have chosen a variant including the stream function even to demonstrate the strength of our approach. As explained in the introduction, previously stream functions like even cannot be dealt with automatically. Note that, our example is not artificial, because the simplification is based on a mathematical insight. Moreover, every computable stream can be specified entirely without using stream functions: $\mathsf{A} \to \mathsf{B}(0)$, $\mathsf{B}(n) \to \mathsf{t}(n):\mathsf{B}(n+1)$, with an appropriate specification of the data function t. Then the actual computation of the stream elements is 'hidden away' into the computation of the data function t. Thus for showing productivity, the burden has shifted from analysing the stream functions to analysing the data functions. In particular, it has to be shown that $\mathsf{t}(n)$ is finitary strongly normalising and rewrites to a constructor normal form for every $n \in \mathbb{N}$.

To formalise the definition of SCSs and SFSs, we use many-sorted term rewriting. Only the rules in the SFS-layer will be subjected to syntactic restrictions, in order to ensure well-definedness of the stream functions specified. No conditions other than well-sortedness will be imposed on how the defining rules for the stream constant symbols in the SCS-layer can make use of the function symbols in the other two layers.

In the sequel we use sorts D and S for *data terms* and *stream terms*, respectively. A *stream TRS* is an $\{S, D\}$-sorted, orthogonal TRS $\langle \Sigma, R\rangle$ such that $':' \in \Sigma_S$, the *stream constructor symbol*, with arity $D \times S \to S$ is the single constructor symbol in Σ_S. The members of Σ_D and Σ_S are referred to as *data symbols* and *stream symbols*, respectively. Without loss of generality we assume that all stream arguments of a stream function, if present, are in front. That is, for all $\mathsf{f} \in \Sigma$, f has arity $S^{\sharp_s(\mathsf{f})} \times D^{\sharp_d(\mathsf{f})} \to S$, and we refer to $\sharp_s(\mathsf{f}) \in \mathbb{N}$ and $\sharp_d(\mathsf{f}) \in \mathbb{N}$ as its *stream arity* and *data arity*, respectively. A stream symbol is called a *stream constant* if it is a constant of sort S.

In this paper we restrict our attention to *constructor stream TRSs*. The sets $\mathit{Ter}(\mathcal{C}(\Sigma))_D$ and $\mathit{Ter}_\infty(\mathcal{C}(\Sigma))_S$ are the sets of data terms and of stream terms in constructor normal form, respectively. Note that stream constructor normal forms are inherently infinite. Moreover, we only consider stream TRSs with *strict data symbols*: for all $\mathsf{m} \in \Sigma_D$ we have $\sharp_s(\mathsf{m}) = 0$.

Definition 3.1. Let $\mathcal{T}$ be a stream TRS. For every $t \in \mathit{Ter}(\Sigma)_S$, we say that $\mathcal{T}$ *is productive for* t if t has a unique infinite normal form $u_1 : u_2 : u_3 : \ldots \in \mathit{Ter}_\infty(\mathcal{C}(\Sigma))_S$, for some $u_1, u_2, u_3, \ldots \in \mathit{Ter}(\mathcal{C}(\Sigma_D))$.

Definition 3.2. Let $\mathcal{T}$ be a stream TRS. The *production function* $\Pi_\mathcal{T} : \mathit{Ter}(\Sigma)_S \to \overline{\mathbb{N}}$ *of* $\mathcal{T}$ is defined for all $t \in \mathit{Ter}(\Sigma)_S$ by $\Pi_\mathcal{T}(t) := \sup\{\, \#_:(s) \mid t \twoheadrightarrow_\mathcal{T} s \,\}$, called *the production of* t, where $\#_:(t) := \sup\{n \in \mathbb{N} \mid t = t_1 : \ldots : t_n : t'\}$, and $\overline{\mathbb{N}} := \mathbb{N} \cup \{\infty\}$ is the set of *extended natural numbers*.

The following proposition characterises productivity of a stream TRS $\mathcal{T}$ for a term t by the unboundedness of the production of t in $\mathcal{T}$. This is an easy consequence of the fact that orthogonal TRSs are finitary confluent, and enjoy the property UN^∞ [17].

Proposition 3.3. *Let $\mathcal{T}$ be a stream TRS, and let $t \in \mathit{Ter}(\Sigma)_S$. Then $\mathcal{T}$ is productive for t if and only if $\Pi_\mathcal{T}(t) = \infty$.*

Definition 3.4. A *stream function specification (SFS)* is a stream TRS $\mathcal{T} = \langle \Sigma, R\rangle$ such that:

(i) $\langle \Sigma_D, R_D\rangle$ is a strongly normalising (finitary SN) TRS in which all ground terms have constructor normal forms.
(ii) For every stream function symbol $\mathsf{f} \in \Sigma_S \setminus \{:\}$ with stream arity $k = \sharp_s(f)$ and data arity $l = \sharp_d(f)$ there is precisely one rule in R_S, denoted by ρ_f, the *defining rule* for f which has the form:

$$\mathsf{f}((\boldsymbol{x_1} : \sigma_1), \ldots, (\boldsymbol{x_k} : \sigma_k), y_1, \ldots, y_l) \to t_1 : \ldots : t_m : u \qquad (\rho_\mathsf{f})$$

where $\boldsymbol{x_i} : \sigma_i$ stands for $x_{i,1} : \ldots : x_{i,n_i} : \sigma_i$, the σ_i are variables of sort S, and u is of one of the forms:

$$u \equiv \mathsf{g}(\sigma_{\phi_\mathsf{f}(1)}, \ldots, \sigma_{\phi_\mathsf{f}(k')}, t'_1, \ldots, t'_{l'}) \qquad (a)$$

$$u \equiv \sigma_i \qquad (b)$$

where $\mathsf{g} \in \Sigma_S$ with $k' = \sharp_s(\mathsf{g})$ and $l' = \sharp_d(\mathsf{g})$, $\phi_\mathsf{f} : \{1, \ldots, k'\} \to \{1, \ldots, k\}$ is an injection used to permute stream arguments, $n_1, \ldots, n_k, m \in \mathbb{N}$, and $1 \le i \le k$.

We use $\mathit{out}(\rho_\mathsf{f}) := m$ to denote the *production* of ρ_f, and $\mathit{in}(\rho_\mathsf{f}, i) := n_i$ to denote the *consumption of* ρ_f *at the* i*-th position*.

The SFS $\mathcal{T}$ is called *weakly guarded* if there are no rules $\ell_1 \to r_1, \ldots, \ell_n \to r_n \in R_{sf}$ such that $\mathit{root}(\ell_1) = \mathit{root}(r_n)$, $\forall i.\ \mathit{root}(r_i) \neq ':'$, and $\forall i < n.\ \mathit{root}(\ell_{i+1}) = \mathit{root}(r_i)$; that is, there do not exist unproductive rewrite sequences of the form $\mathsf{f}(\boldsymbol{t}) \to^+ \mathsf{f}(\boldsymbol{t'})$.

Remark 3.5. (i) This definition covers a large class of stream functions including for instance tail, even, odd, and zip. By the restriction to strict data symbols, we exclude data rules such as $\mathsf{head}(x : \sigma) \to x$, possibly creating 'look-ahead' as in the well-defined example $\mathsf{S} \to 0 : \mathsf{head}(\mathsf{tail}^2(\mathsf{S})) : \mathsf{S}$ from [21].

(ii) For an extension of the format of SFSs we refer to [10], where the conditions on stream functions imposed here are relaxed in four different ways (while productivity stays decidable for stream specifications built upon stream functions of the extended class). First, the requirement of right-linearity of stream variables (a consequence of the permutation function ϕ_{f} for stream arguments being injective) is dropped, allowing rules like $\mathsf{f}(\sigma) \to \mathsf{g}(\sigma, \sigma)$. Second, 'additional supply' to the stream arguments is allowed, in rules like $\mathsf{diff}(x : y : \sigma) \to \mathsf{xor}(x, y) : \mathsf{diff}(y : \sigma)$, where the variable y is 'supplied' to the recursive call of diff. Third, the use of non-productive stream functions is allowed. Finally, even a restricted form of pattern matching is allowed in defining rules for stream functions as long as, for every stream function f, the quantitative ('data-oblivious') consumption/production behaviour of all defining rules for f is the same, see Example 3.9 below. Extending terminology introduced in Definition 3.6 below, stream specifications built upon stream functions of this enlarged class are also called 'pure' in [10].

Definition 3.6. Let $\mathcal{T} = \langle \Sigma, R \rangle$ be a stream TRS with an additional partition $\Sigma_S = \Sigma_{sf} \uplus \Sigma_{sc} \uplus \{:\}$ of the stream signature and a partition $R_S = R_{sf} \uplus R_{sc}$ of the set of stream rules. Then $\mathcal{T}$ is called a *pure stream constant specification (SCS)* if the following conditions hold:

(i) $\mathcal{T}_0 = \langle \Sigma_D \uplus \Sigma_{sf} \uplus \{:\}, R_D \uplus R_{sf} \rangle$ is a weakly guarded SFS. We say: $\mathcal{T}$ *is based on* $\mathcal{T}_0$.
(ii) Σ_{sc} is a set of constant symbols containing a distinguished symbol M_0, called the *root of* $\mathcal{T}$. R_{sc} is the set of *defining rules* $\rho_{\mathsf{M}} \colon \mathsf{M} \to t$ for every $\mathsf{M} \in \Sigma_{sc}$.

Given an SCS, we speak of its *data*, *SFS*, and *SCS layer* to mean R_D, R_{sf}, and R_{sc}, respectively. An SCS $\mathcal{T}$ is called *productive* if $\mathcal{T}$ is productive for its root M_0.

In the sequel we restrict to SCSs in which all stream constants in Σ_{sc} are reachable from the root: $\mathsf{M} \in \Sigma_{sc}$ is *reachable* if there is a term t such that $\mathsf{M}_0 \twoheadrightarrow t$ and M occurs in t. Note that reachability of stream constants is decidable, and that unreachable symbols may be neglected for investigating whether or not an SCS is productive.

Since every SCS is a stream TRS, Proposition 3.3 entails the following characterisation of productivity of stream terms, which will be useful in the correctness proof of our method for deciding productivity of SCSs.

Proposition 3.7. *Let $\mathcal{T}$ be a SCS. Then $\mathcal{T}$ is productive if and only if $\Pi_{\mathcal{T}}(\mathsf{M}_0) = \infty$.*

The signature of the SCS given in Fig. 1 is partitioned such that $\Sigma_D = \{0, 1, \mathsf{i}\}$, $\Sigma_{sf} = \{\mathsf{zip}, \mathsf{inv}, \mathsf{even}, \mathsf{odd}, \mathsf{tail}\}$ and $\Sigma_{sc} = \{\mathsf{M}\}$; the set of rules R is partitioned as indicated.

Example 3.8. Consider the SCS $\langle \Sigma, R \rangle$ with $\Sigma = \{0, 1, \mathsf{even}, \mathsf{odd}, \mathsf{J}, :\}$ and where R has the SCS layer $R_{sc} = \{\mathsf{J} \to 0 : 1 : \mathsf{even}(\mathsf{J})\}$, the SFS layer consisting of the mutual recursive rules for even and odd (see Fig. 1), and the empty data layer $R_D = \varnothing$. The infinite normal form of J is $0 : 1 : 0 : 0 : \mathsf{even}(\mathsf{even}(\ldots))$, which is not a constructor normal form. Hence J is WN^{∞} (in fact SN^{∞}), but not productive.

Example 3.9. For an example that (only just) falls outside the format of SCSs, consider the stream TRS $\mathcal{T} = \langle \Sigma, R \rangle$ with $\Sigma = \{0, 1, \mathsf{tail}, \mathsf{f}, \mathsf{T}, :\}$ and with R consisting of the stream constant layer rules $R_{sc} = \{\mathsf{T} \to 0 : 1 : \mathsf{f}(\mathsf{tail}(\mathsf{T}))\}$, the stream function layer rules $R_{sf} = \{\mathsf{tail}(x : \sigma) \to \sigma\ ,\ \mathsf{f}(0 : \sigma) \to 0 : 1 : \mathsf{f}(\sigma)\ ,\ \mathsf{f}(1 : \sigma) \to 1 : 0 : \mathsf{f}(\sigma)\}$, and an empty set $R_D = \varnothing$ of data layer rules. $\mathcal{T}$ specifies the Thue–Morse stream based on the D0L-system $\{0 \to 01,\ 1 \to 10\}$. Now note that $\mathcal{T}$ is not an SCS as defined in Definition 3.6 because this specification uses pattern matching on data symbols for the stream function symbol f, and, in particular, two defining rules for f rather than just one.

Remark 3.10. (i) Although the specification in Example 3.9 is not an SCS, it is easy to transform it into one. The two defining rules for f satisfy the special property that their consumption/production behaviour is the same. This makes it possible to transform $\mathcal{T}$ into the following closely related SCS $\mathcal{T}'$ that also specifies the Thue–Morse sequence: let $\mathcal{T}' = \langle \Sigma', R' \rangle$ with $\Sigma' = \{0, 1, \mathsf{i}, \mathsf{f}, \mathsf{T}, :\}$ and with R' consisting of the stream constant layer rules $R'_{sc} = \{\mathsf{T} \to 0 : 1 : \mathsf{f}(\mathsf{tail}(\mathsf{T}))\}$, the stream function layer rules $R'_{sf} = \{\mathsf{tail}(x : \sigma) \to \sigma\ ,\ \mathsf{f}(x : \sigma) \to x : \mathsf{i}(x) : \mathsf{f}(\sigma)\}$, and the data layer rules $R'_D = \{\mathsf{i}(0) \to 1,\ \mathsf{i}(1) \to 0\}$.

(ii) All k-automatic streams [2] can be defined as SCSs. In itself this is an immediate consequence of the fact that every computable stream can be defined as an SCS by simulating the effect of a Turing machine by data layer rules. However, there is a much more straightforward translation of k-DFAO's (see [2]) into a stream specification, resembling the one in Example 3.9, which can be transformed into an SCS similar as described in (i). We note that the data layer of the SCS after this transformation consists of rules that can be viewed as a collection of finite substitutions and that form a trivially terminating TRS.

(iii) The stream specification in Example 3.9 corresponds to a 'pure stream specification' as introduced in subsequent work [10], an extension of the present SCSs framework that admits limited pattern matching on data (see also Remark 3.5, ii), and for which productivity is still decidable.

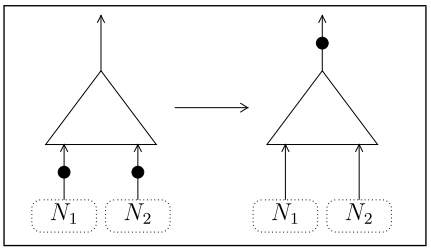


Fig. 2. Rule (P1).

4. Pebbleflow nets

We introduce pebbleflow nets as a means to model the 'data-oblivious' consumption/production behaviour of SCSs. That is, we abstract from the actual stream elements (data) in an SCS in favour of occurrences of the symbol $\bullet$, which we call a 'pebble'. Thus, a stream term $d : s$ is translated to $[d : s] = \bullet([s])$. Pebbleflow nets are inspired by interaction nets [18], and could be implemented in the framework of interaction nets with little effort. We give an operational description of pebbleflow nets and define a production preserving translation of SCSs into 'rational' nets.

Pebbleflow nets are networks built of pebble processing units (fans, boxes, meets, sources) connected by wires. We first introduce a term syntax for nets and the rules governing the flow of pebbles through a net, and then give an operational meaning of the units a net is built of.

Definition 4.1. Let $\mathcal{V}$ be a set of variables. The set $\mathcal{N}$ of *pebbleflow terms* (shortly called *nets*) is generated by:

$$N ::= \mathsf{src}(\underline{k}) \mid x \mid \bullet(N) \mid \mathsf{box}(\sigma, N) \mid \mu x.N \mid \triangle(N, N)$$

where $x \in \mathcal{V}$, σ is a term representation of an I/O sequence in $\pm^\omega \subseteq \{+, -\}^\omega$ (defined in Definition 4.3 below), and where, for $n \in \overline{\mathbb{N}}$, $\underline{n}$ is the *numeral* (a term representation) for n that is defined by $\underline{n} := \mathsf{s}^n(0)$ if $n \in \mathbb{N}$, and $\underline{\infty} := \mathsf{s}^\omega$. A net is called *closed* if it has no free variables.

Pebbleflow terms can be viewed as term specifications of cyclic term graphs (for the latter see [3,24]). They are μ-terms that employ μ-bindings to describe back-pointers in cyclic graph representations. An explicit translation $\mathcal{G}$ of μ-terms into cyclic term graphs is described in [3, Def. 2.7, p. 7]. The image of the translation $\mathcal{G}$ has been characterised as the class of cyclic term graphs without 'horizontal sharing' [6].

Definition 4.2. The *pebbleflow rewrite relation* $\to_{\mathsf{P}}$ is defined by the following rules which may be applied in arbitrary contexts:

$$\triangle(\bullet(N_1), \bullet(N_2)) \to \bullet(\triangle(N_1, N_2)) \tag{P1}$$
$$\mu x.\bullet(N(x)) \to \bullet(\mu x.N(\bullet(x))) \tag{P2}$$
$$\mathsf{box}(+\sigma, N) \to \bullet(\mathsf{box}(\sigma, N)) \tag{P3}$$
$$\mathsf{box}(-\sigma, \bullet(N)) \to \mathsf{box}(\sigma, N) \tag{P4}$$
$$\mathsf{src}(\mathsf{s}(\underline{k})) \to \bullet(\mathsf{src}(\underline{k})) \tag{P5}$$

Wires are unidirectional FIFO communication channels. They are idealised in the sense that there is no upper bound on the number of pebbles they can store; arbitrarily long queues are allowed. Wires have no counterpart on the term level; in this sense they are akin to the edges of a term tree. Wires connect *boxes*, *meets*, *fans*, and *sources*, that we describe next.

A *meet* is waiting for a pebble at each of its input ports and only then produces one pebble at its output port, see Fig. 2. Put differently, the number of pebbles a meet produces equals the minimum of the numbers of pebbles available at each of its input ports. Meets enable explicit branching; they are used to model stream functions of stream arity > 1, as will be explained below. A meet with an arbitrary number $n \geq 1$ of input ports is implemented by using a single wire in case $n = 1$, and if $n = k + 1$ with $k \geq 1$, by connecting the output port of a 'k-ary meet' to one of the input ports of a (binary) meet.

The behaviour of a *fan* is dual to that of a meet: a pebble at its input port is duplicated along its output ports. A fan can be seen as an explicit sharing device, and thus enables the construction of cyclic nets. More specifically, we use fans only to implement feedback when drawing nets; there is no explicit term representation for the fan in our term calculus. In Fig. 3 a pebble is sent over the output wire of the net and at the same time is fed back to the 'recursion wire(s)'. We represent cyclic nets by μ-terms: a fan is represented by a binder μx, and a recursion wire connected to one of its output ports is represented by a variable x. In rule (P2) feedback is accomplished by substituting $\bullet(x)$ for all free occurrences x of N.

A *source* has an output port only, contains a number $k \in \overline{\mathbb{N}}$ of pebbles, and can fire if $k > 0$, see Fig. 6. In Section 6 we show how to reduce closed nets, i.e. nets without free input ports, to sources.

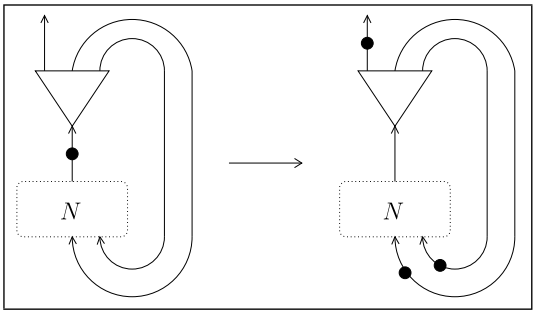


Fig. 3. Rule (P2).

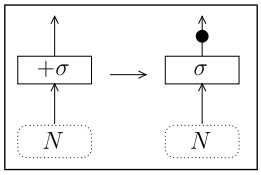


Fig. 4. Rule (P3).

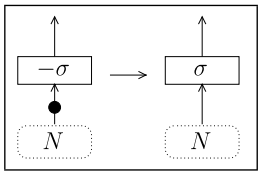


Fig. 5. Rule (P4).

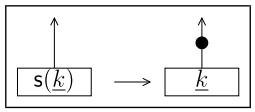


Fig. 6. Rule (P5).

A *box* consumes pebbles at its input port and produces pebbles at its output port, controlled by an infinite sequence $\sigma \in \{+,-\}^\omega$ associated with the box. For example, consider the unary stream function dup, defined as follows, and its corresponding 'I/O sequence':

$$\mathsf{dup}(x:\sigma) = x:x:\mathsf{dup}(\sigma) \qquad -++-++-++\ldots$$

which is to be thought of as: *for* dup *to produce two outputs, it first has to consume one input, and this process repeats indefinitely*. Intuitively, the symbol $-$ represents a requirement for one input pebble, and $+$ represents a ready state for one output pebble. Pebbleflow through boxes is visualised in Figs. 4 and 5.

Definition 4.3. The set $\pm^\omega$ of *I/O sequences* is defined as the set of infinite sequences over the alphabet $\{+,-\}$ that contain an infinite number of $+$'s:

$$\pm^\omega := \left\{ \sigma \in \{+,-\}^\omega \mid \forall n.\, \exists m \geq n.\, \sigma(m) = + \right\}.$$

A sequence $\sigma \in \pm^\omega$ is *rational* if there exist lists $\alpha, \beta \in \{+,-\}^*$ such that $\sigma = \alpha\overline{\beta}$, where β is not the empty list and $\overline{\beta}$ denotes the infinite sequence $\beta\beta\beta\ldots$. The pair $\langle\alpha, \beta\rangle$ is called a *rational representation* of σ. The set of rational I/O sequences is denoted by $\pm^\omega_{rat}$. A net is called *rational* if all its boxes contain rational I/O sequences; by $\mathcal{N}_{rat}$ we denote the set of rational nets.

In the next section we define a translation from SCSs to rational nets. In Section 6 we introduce a rewrite system for reducing nets to trivial nets (pebble sources). That system, the kernel of our decision algorithm, is terminating for rational nets, and enables us to determine the total production of a rational net. We stress that the restriction to rational nets in our algorithm does not entail a restriction to deal only with SCSs that define rational *streams*; actually, the SCS given in Fig. 1 defining the Thue–Morse sequence, an irrational stream, is translated to a rational net.

A stream function f with a stream arity n is modelled by a *gate*: an n-ary component $\triangle_n$ composed with n boxes expressing the contribution of each individual stream argument to the total production of f, see Fig. 8. We define gates as n-ary contexts:

$$\mathsf{gate}(\sigma_1,\ldots,\sigma_n) := \triangle_n(\mathsf{box}(\sigma_1,[\,]_1),\ldots,\mathsf{box}(\sigma_n,[\,]_n))$$

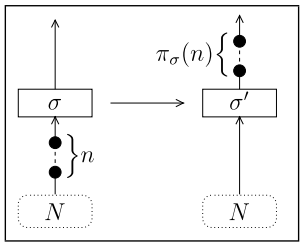


Fig. 7. $\mathsf{box}(\sigma, \bullet^n(N)) \to \bullet^{\pi_\sigma(n)}(\mathsf{box}(\sigma', N))$.

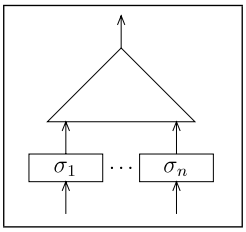


Fig. 8. A gate with n input ports.

and by writing $\mathsf{gate}(\sigma_1, \ldots, \sigma_n)(N_1, \ldots, N_n)$ for context filling we deviate from the standard notation to mean $\triangle_n(\mathsf{box}(\sigma_1, N_1), \ldots, \mathsf{box}(\sigma_n, N_n))$.

Definition 4.4. The *production function* $\pi_\sigma : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ of (a box containing) a sequence $\sigma \in \pm^\omega$ defined, for all $n \in \overline{\mathbb{N}}$, by $\pi_\sigma(n) := \pi(\sigma, n)$, where $\pi(\sigma, n) \in \overline{\mathbb{N}}$ is, for all $\upsilon \in \pm^\omega$ and $n \in \overline{\mathbb{N}}$, corecursively defined by:

$$\pi(+\upsilon, n) = 1 + \pi(\upsilon, n) \qquad \pi(-\upsilon, 0) = 0 \qquad \pi(-\upsilon, n+1) = \pi(\upsilon, n).$$

Intuitively, $\pi_\sigma(n)$ is the number of outputs of a box containing sequence σ when fed with n inputs, see Fig. 7. Notice that π_σ is well-defined because σ contains infinitely many $+$'s by definition.

Lemma 4.5. *The pebbleflow rewrite relation $\to_{\mathsf{P}}$ is confluent.*

Proof. The rules of $\to_{\mathsf{P}}$ can be viewed as a higher-order rewriting system (HRS) that is orthogonal. Applying Theorem 11.6.9 in [24] then establishes the lemma. □

Definition 4.6. The *production function* $\Pi_{\mathsf{P}} : \mathcal{N} \to \overline{\mathbb{N}}$ *of nets* is defined for all $N \in \mathcal{N}$ by $\Pi_{\mathsf{P}}(N) := \sup\{n \in \mathbb{N} \mid N \twoheadrightarrow_{\mathsf{P}} \bullet^n(N')\}$, called *the production of* N. Moreover, for a net N and an *assignment* $\alpha : \mathcal{V} \to \overline{\mathbb{N}}$, let $\Pi_{\mathsf{P}}(N, \alpha) := \Pi_{\mathsf{P}}(N^\alpha)$ where N^α denotes the net obtained by replacing each free variable x of N with $\bullet^{\alpha(x)}(x)$. We will employ the notation $\alpha[x \mapsto n]$ to denote an *update* of α, defined by $\alpha[x \mapsto n](y) = n$ if $y = x$, and $\alpha[x \mapsto n](y) = \alpha(y)$ otherwise.

Note that for closed nets N we have $N^\alpha = N$ and therefore $\Pi_{\mathsf{P}}(N, \alpha) = \Pi_{\mathsf{P}}(N)$, for all assignments α.

We define an alternative net production function $\Pi_{\mathcal{N}}$ (equivalent to Π_{P}) that provides some useful intuition and will allow us to get a handle on proving that production is preserved by the net reduction relation introduced in Section 6.

Definition 4.7. The mapping $\Pi_{\mathcal{N}} : \mathcal{N} \times (\mathcal{V} \to \overline{\mathbb{N}}) \to \overline{\mathbb{N}}$ is defined inductively by:

$$\Pi_{\mathcal{N}}(\mathsf{src}(\underline{k}), \alpha) = k \qquad \Pi_{\mathcal{N}}(\mathsf{box}(\sigma, N), \alpha) = \pi_\sigma(\Pi_{\mathcal{N}}(N, \alpha))$$
$$\Pi_{\mathcal{N}}(\bullet(N), \alpha) = 1 + \Pi_{\mathcal{N}}(N, \alpha) \qquad \Pi_{\mathcal{N}}(\mu x.N, \alpha) = \mathrm{lfp}(\lambda n.\Pi_{\mathcal{N}}(N, \alpha[x \mapsto n]))$$
$$\Pi_{\mathcal{N}}(x, \alpha) = \alpha(x) \qquad \Pi_{\mathcal{N}}(\triangle(N_1, N_2), \alpha) = \min(\Pi_{\mathcal{N}}(N_1, \alpha), \Pi_{\mathcal{N}}(N_2, \alpha))$$

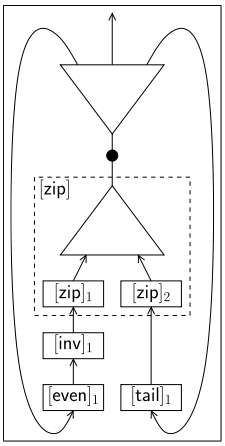
Notice that $\Pi_{\mathcal{N}}$ is monotonic in its second argument. The net production functions Π_{P} and $\Pi_{\mathcal{N}}$ coincide (see the Appendix for a proof):

Lemma 4.8. *For all nets N and assignments α, we have $\Pi_{\mathsf{P}}(N, \alpha) = \Pi_{\mathcal{N}}(N, \alpha)$.*

5. Translating stream specifications into nets

In this section we define a 'production preserving' translation from stream constants M in an SCS to rational nets $[\mathsf{M}]$. In particular, the root M_0 of an SCS $\mathcal{T}$ will be mapped to a net $[\mathsf{M}_0] \in \mathcal{N}_{rat}$ with the property that its production equals the production of M_0 in $\mathcal{T}$.

As a first step, we give a translation of the stream function symbols in an SFS into rational gates (gates with boxes containing rational I/O sequences) that precisely model their quantitative consumption/production behaviour. The idea is to define, for a stream function symbol f, a rational gate by keeping track of the 'production' (the guards encountered) and the 'consumption' of the rules applied, during the finite or eventually periodic dependency sequence on f.



$$
\begin{aligned}
[\mathsf{zip}] &= \mathsf{gate}([\mathsf{zip}]_1, [\mathsf{zip}]_2)\\
[\mathsf{zip}]_1 &= -+[\mathsf{zip}]_2 = -++[\mathsf{zip}]_1 = \overline{-++}\\
[\mathsf{zip}]_2 &= +[\mathsf{zip}]_1 = +-+[\mathsf{zip}]_2 = \overline{+-+}\\
[\mathsf{inv}] &= \mathsf{gate}([\mathsf{inv}]_1)\\
[\mathsf{inv}]_1 &= -+[\mathsf{inv}]_1 = \overline{-+}\\
[\mathsf{even}] &= \mathsf{gate}([\mathsf{even}]_1)\\
[\mathsf{even}]_1 &= -+[\mathsf{odd}]_1 = -+-[\mathsf{even}]_1 = \overline{-+-}\\
[\mathsf{odd}]_1 &= -[\mathsf{even}]_1 = --+[\mathsf{odd}]_1 = \overline{--+}\\
[\mathsf{tail}] &= \mathsf{gate}([\mathsf{tail}]_1)\\
[\mathsf{tail}]_1 &= -\overline{-+}
\end{aligned}
$$

Fig. 9. The net translation $[\mathsf{M}] = \mu M.\bullet([\mathsf{zip}]([\mathsf{inv}]([\mathsf{even}](M)), [\mathsf{tail}](M)))$ of M.

Definition 5.1. Let $\mathcal{T} = \langle \Sigma_D \uplus \Sigma_{sf} \uplus \{:\}, R_D \uplus R_{sf} \rangle$ be an SFS. Then, for each $\mathsf{f} \in \Sigma_{sf}$ with stream arity $k = \sharp_s(\mathsf{f})$ and data arity $l = \sharp_d(\mathsf{f})$ the *translation* of f is a rational gate $[\mathsf{f}] : \mathcal{N}^k \to \mathcal{N}$ defined by:

$$[\mathsf{f}] = \mathsf{gate}([\mathsf{f}]_1, \ldots, [\mathsf{f}]_k)$$

where $[\mathsf{f}]_i \in \pm^{\omega}_{rat}$ is defined as follows. We distinguish the two formats a rule $\rho_\mathsf{f} \in R_{sf}$ can have. Let $\boldsymbol{x_i} : \sigma_i$ stand for $x_{i,1} : \ldots : x_{i,n_i} : \sigma_i$. If ρ_f has the form: $\mathsf{f}(\boldsymbol{x_1} : \sigma_1, \ldots, \boldsymbol{x_k} : \sigma_k, y_1, \ldots, y_l) \to t_1 : \ldots : t_m : u$, where:

(a) $u \equiv \mathsf{g}(\sigma_{\phi_\mathsf{f}(1)}, \ldots, \sigma_{\phi_\mathsf{f}(\sharp_s(\mathsf{g}))}, t'_1, \ldots, t'_{\sharp_d(\mathsf{g})})$, then

$$[\mathsf{f}]_i = \begin{cases} -^{n_i}+^{m}[\mathsf{g}]_j & \text{if } \phi_\mathsf{f}(j) = i \\ -^{n_i}\overline{+} & \text{if } \neg\exists j.\ \phi_\mathsf{f}(j) = i \end{cases}$$

(b) $u \equiv \sigma_j$, then

$$[\mathsf{f}]_i = \begin{cases} -^{n_i}+^{m}\overline{-+} & \text{if } i = j \\ -^{n_i}\overline{+} & \text{if } i \neq j \end{cases}$$

In the second step, we now define the translation of the stream constants in an SCS into rational nets. Here the idea is that the recursive definition of a stream constant M is unfolded step by step; the terms thus arising are translated according to their structure by making use of the translation of the stream function symbols encountered; whenever a stream constant is met that has been unfolded before, the translation stops after establishing a binding to a μ-binder created earlier.

Definition 5.2. Let $\mathcal{T} = \langle \Sigma_D \uplus \Sigma_{sf} \uplus \Sigma_{sc} \uplus \{:\}, R_D \uplus R_{sf} \uplus R_{sc} \rangle$ be an SCS. Then, for each $\mathsf{M} \in \Sigma_{sc}$ with rule $\rho_\mathsf{M} \equiv \mathsf{M} \to \mathit{rhs}_\mathsf{M}$ the translation $[\mathsf{M}] := [\mathsf{M}]_\varnothing$ of M to a closed pebbleflow net is recursively defined by (α a set of stream constant symbols):

$$
\begin{aligned}
[\mathsf{M}]_\alpha &= \begin{cases} \mu M.[\mathit{rhs}_\mathsf{M}]_{\alpha\cup\{\mathsf{M}\}} & \text{if } \mathsf{M} \notin \alpha \\ M & \text{if } \mathsf{M} \in \alpha \end{cases}\\
[t : u]_\alpha &= \bullet([u]_\alpha)\\
[\mathsf{f}(u_1, \ldots, u_{\sharp_s(\mathsf{f})}, t_1, \ldots, t_{\sharp_d(\mathsf{f})})]_\alpha &= [\mathsf{f}]([u_1]_\alpha, \ldots, [u_{\sharp_s(\mathsf{f})}]_\alpha).
\end{aligned}
$$

Example 5.3. Reconsider the SCS given in Fig. 1. The translation of the stream constant M and of the stream functions involved, is illustrated in Fig. 9. (Note that to obtain rational representations of the translated stream functions we use loop checking on top of Definition 5.1.)

The root of the SCS of Example 3.8 is translated by: $\mu J.\bullet(\bullet([\mathsf{even}](J)))$.

By stating that the translation of the root of an SCS into a pebbleflow net is 'production preserving', the theorem below will provide the basis for our decision algorithm for productivity of SCSs, which will be detailed in the next section. A proof of this theorem is given in Section A.2.

Theorem 5.4. *For every SCS* $\mathcal{T}$*:* $\Pi_\mathcal{T}(\mathsf{M}_0) = \Pi_\mathsf{P}([\mathsf{M}_0])$ *holds.*

6. Deciding productivity

We define a rewriting system for pebbleflow nets that, for every net N, allows us to reduce N to a single source while preserving the production of N.

Definition 6.1. We define *the net reduction relation* $\to_{\mathsf{R}}$ by the compatible closure of the following rule schemata:

$$\bullet(N) \to \mathsf{box}(+\overline{-+}, N) \tag{R1}$$
$$\mathsf{box}(\sigma, \mathsf{box}(\tau, N)) \to \mathsf{box}(\sigma \circ \tau, N) \tag{R2}$$
$$\mathsf{box}(\sigma, \triangle(N_1, N_2)) \to \triangle(\mathsf{box}(\sigma, N_1), \mathsf{box}(\sigma, N_2)) \tag{R3}$$
$$\mu x.\triangle(N_1, N_2) \to \triangle(\mu x.N_1, \mu x.N_2) \tag{R4}$$
$$\mu x.N \to N \qquad \text{if } x \notin \mathrm{FV}(N) \tag{R5}$$
$$\mu x.\mathsf{box}(\sigma, x) \to \mathsf{src}(\underline{\mathsf{fix}(\sigma)}) \tag{R6}$$
$$\triangle(\mathsf{src}(\underline{k_1}), \mathsf{src}(\underline{k_2})) \to \mathsf{src}(\underline{\mathsf{min}(k_1, k_2)}) \tag{R7}$$
$$\mathsf{box}(\sigma, \mathsf{src}(\underline{k})) \to \mathsf{src}(\underline{\pi_\sigma(k)}) \tag{R8}$$
$$\mu x.x \to \mathsf{src}(0) \tag{R9}$$

where $k, k_1, k_2 \in \overline{\mathbb{N}}$, σ and τ are term representations of I/O sequences in $\pm^\omega$, and where $\underline{\mathsf{min}(n, m)}$, $\underline{\pi_\sigma(k)}$ (see Definition 4.4), $\sigma \circ \tau$ (see Definition 6.2), and $\underline{\mathsf{fix}(\sigma)}$ (see Definition 6.4) are numerals that represent operation results.

The rewrite system given in Definition 6.1 contains rules with infinite left-hand sides. However, by turning the I/O sequences and the numbers in $\overline{\mathbb{N}}$ into constants, the system can be transformed into a finite higher-order rewriting system (HRS). Then the operations min, π, $\circ$, and fix have to be defined on the constants.

Definition 6.2. The operation *composition* $\circ : \pm^\omega \times \pm^\omega \to \pm^\omega$, $\langle \sigma, \tau \rangle \mapsto \sigma \circ \tau$ *of I/O sequences* is defined corecursively by the following equations:

$$+\sigma \circ \tau = +(\sigma \circ \tau) \qquad -\sigma \circ +\tau = \sigma \circ \tau \qquad -\sigma \circ -\tau = -(-\sigma \circ \tau).$$

Composition of sequences $\sigma \circ \tau \in \pm^\omega$ exhibits analogous properties as composition of functions over natural numbers: it is associative, but not commutative.

Lemma 6.3. *For all* $\sigma, \tau, \upsilon \in \pm^\omega$*:* (i) $\sigma \circ (\tau \circ \upsilon) = (\sigma \circ \tau) \circ \upsilon$*, and* (ii) $\pi_\sigma \circ \pi_\tau = \pi_{\sigma \circ \tau}$.

Because we formalised the I/O behaviour of boxes by sequences and because we are interested in proving and disproving productivity, for the formalisation of the pebbleflow rewrite relation in Definition 4.2 the choice has been made to give output priority over input. This becomes apparent in the definition of composition above: the net $\mathsf{box}(+\overline{-+}, \mathsf{box}(-\overline{-+}, x))$ is able to consume an input pebble at its free input port x as well as to produce an output pebble, whereas the result $\mathsf{box}(+-\overline{-+}, x)$ of the composition can only consume input *after* having fired.

The fixed point of a box is the production of the box when fed its own output.

Definition 6.4. The operations *fixed point* $\mathsf{fix} : \pm^\omega \to \overline{\mathbb{N}}$ and *requirement removal* $\delta : \pm^\omega \to \pm^\omega$ *on I/O sequences* are corecursively defined as follows:

$$\mathsf{fix}(+\sigma) = 1 + \mathsf{fix}(\delta(\sigma)) \qquad \delta(+\sigma) = +\delta(\sigma)$$
$$\mathsf{fix}(-\sigma) = 0 \qquad \delta(-\sigma) = \sigma.$$

Lemma 6.5. *For all* $\sigma \in \pm^\omega$*, we have* $\mathrm{lfp}(\pi_\sigma) = \mathsf{fix}(\sigma)$.

Observe that $\pi_{\sigma\circ\sigma\circ\sigma\circ\ldots} = \pi_\sigma(\pi_\sigma(\pi_\sigma(\ldots))) = \mathsf{fix}(\sigma)$. Therefore, the infinite self-composition of the form $\mathsf{box}(\sigma, \mathsf{box}(\sigma, \mathsf{box}(\sigma, \ldots)))$ is 'production equivalent' to $\mathsf{src}(\underline{\mathsf{fix}(\sigma)})$.

An important property used in the following lemma is that functions of the form $\lambda n \in \overline{\mathbb{N}}.\ \Pi_{\mathsf{P}}(N, \alpha[x \mapsto n])$ are monotonic functions over $\overline{\mathbb{N}}$. Every monotonic function $f : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ in the complete chain $\overline{\mathbb{N}}$ has a least fixed point $\mathrm{lfp}(f)$ which can be computed by $\mathrm{lfp}(f) = \lim_{n\to\infty} f^n(0)$. In what follows we employ, for monotonic $f, g : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$, two basic properties:

$$\forall n, m.\, f(\min(n, m)) = \min(f(n), f(m)) \tag{1}$$
$$\mathrm{lfp}(\lambda n.\, \min(f(n), g(n))) = \min(\mathrm{lfp}(f), \mathrm{lfp}(g)). \tag{2}$$

Lemma 6.6. *Net reduction preserves production:* $\Pi_{\mathsf{P}}(N) = \Pi_{\mathsf{P}}(N')$ *if* $N \to_{\mathsf{R}} N'$.

Proof. By Lemma 4.8 it suffices to prove:

$$C[\ell^\sigma] \to_{\mathsf{R}} C[r^\sigma] \implies \forall \alpha.\ \Pi_{\mathcal{N}}(C[\ell^\sigma], \alpha) = \Pi_{\mathcal{N}}(C[r^\sigma], \alpha)\,,$$

where $\ell \to r$ is a rule of the net reduction TRS, and C a unary context over $\mathcal{N}$. We proceed by induction on C. For the base case, $C = [\,]$, we give the essential proof steps only (no definition unfoldings): For rule (R1), observe that $\pi_{\overline{-+}}$ is the identity function on $\overline{\mathbb{N}}$. For rule (R2), we apply Lemma 6.3 (ii). For rule (R3) the desired equality follows from (1) above. For rule (R4) we conclude by (2) above. For rule (R6) we use Lemma 6.5. For the remaining rules the statement trivially holds. For the induction step, the statement easily follows from the induction hypotheses. □

Lemma 6.7. *The net reduction relation $\to_{\mathsf{R}}$ is terminating and confluent, and every closed net normalises to a unique normal form, a source.*

Proof. To see that $\to_{\mathsf{R}}$ is terminating, let $[\![-]\!] : \mathcal{N} \to \mathbb{N}$ be defined by:

$$[\![x]\!] = 1 \qquad [\![\bullet(N)]\!] = 2 \cdot [\![N]\!] + 1 \qquad [\![\mu x.N]\!] = 2 \cdot [\![N]\!]$$
$$[\![\mathsf{src}(\underline{k})]\!] = 1 \qquad [\![\mathsf{box}(\sigma, N)]\!] = 2 \cdot [\![N]\!] \qquad [\![\triangle(N_1, N_2)]\!] = [\![N_1]\!] + [\![N_2]\!] + 1\,,$$

and observe that $N \to_{\mathsf{R}} M$ implies $[\![N]\!] > [\![M]\!]$.

Some of the rules of $\to_{\mathsf{R}}$ overlap; e.g. rule (R2) with itself. For each of the five critical pairs we can find a common reduct (the critical pair $\langle \sigma \circ (\tau \circ \upsilon), (\sigma \circ \tau) \circ \upsilon \rangle$ due to an (R2)/(R2)-overlap can be joined by Lemma 6.3 (i)), and hence $\to_{\mathsf{R}}$ is locally confluent, by the Critical Pairs Lemma (cf. [24]). By Newman's Lemma, we obtain that $\to_{\mathsf{R}}$ is confluent. Thus normal forms are unique.

To show that every closed net normalises to a source, let N be an arbitrary normal form. Note that the set of free variables of a net is closed under $\to_{\mathsf{R}}$, and hence N is a closed net. Clearly, N does not contain pebbles, otherwise (R1) would be applicable. To see that N contains no subterms of the form $\mu x.M$, suppose it does and consider the innermost such subterm, viz. M contains no μ. If $M \equiv \mathsf{src}(\underline{k})$ or $M \equiv x$, then (R5), resp. (R9) is applicable. If $M \equiv \mathsf{box}(\sigma, M')$, we further distinguish four cases: if $M' \equiv \mathsf{src}(\underline{k})$ or $M' \equiv x$, then (R8) resp. (R6) is applicable; if the root symbol of M' is one of box, $\triangle$, then M constitutes a redex with respect to (R2), (R3), respectively. If $M \equiv \triangle(M_1, M_2)$, we have a redex with respect to (R4). Thus, there are no subterms $\mu x.M$ in N, and therefore, because N is closed, also no variables x. To see that N has no subterms of the form $\mathsf{box}(\sigma, M)$, suppose it does and consider the innermost such subterm. Then, if $M \equiv \mathsf{src}(\underline{k})$ or $M \equiv \triangle(M_1, M_2)$ then (R8) resp. (R3) is applicable; other cases have been excluded above. Finally, N does not contain subterms of the form $\triangle(N_1, N_2)$. For if it does, consider the innermost occurrence and note that, since the other cases have been excluded already, N_1 and N_2 have to be sources, and so we have a redex with respect to (R7). We conclude that $N \equiv \mathsf{src}(\underline{k})$ for some $k \in \overline{\mathbb{N}}$. □

Observe that net reduction employs infinitary rewriting for fixed point computation and composition (Definitions 6.2 and 6.4). To compute normal forms in finite time we make use of finite representations of rational sequences and exchange the numeral s^ω with a constant ∞.

Lemma 6.8. *There is an algorithm that, if $N \in \mathcal{N}_{rat}$ and rational representations of the sequences $\sigma \in \pm^\omega_{rat}$ in N are given, computes the $\to_{\mathsf{R}}$-normal form of N.*

Proof. Note that composition preserves rationality, that is, $\sigma \circ \tau \in \pm^\omega_{rat}$ whenever $\sigma, \tau \in \pm^\omega_{rat}$. Similarly, it is straightforward to show that for sequences $\sigma, \tau \in \pm^\omega_{rat}$ with given rational representations the fixed point $\mathsf{fix}(\sigma)$ and a rational representation of the composition $\sigma \circ \tau$ can be computed in finite time. □

Theorem 6.9. *Productivity is decidable for pure stream constant specifications.*

Proof. The following steps describe a decision algorithm for productivity of a stream constant M in an SCS $\mathcal{T}$: First, the translation $[\mathsf{M}]$ of M into a pebbleflow net is built according to Definition 5.2. It is easy to verify that $[\mathsf{M}]$ is in fact a rational net. Second, by the algorithm stated by Lemma 6.8, $[\mathsf{M}]$ is collapsed to a source $\mathsf{src}(\underline{n})$ with $n \in \overline{\mathbb{N}}$. By Theorem 5.4 it follows that $[\mathsf{M}]$ has the same production as M in $\mathcal{T}$, and by Lemma 6.6 that the production of $[\mathsf{M}]$ is n. Consequently, $\Pi_{\mathcal{T}}(\mathsf{M}) = n$. Hence the answers "$\mathcal{T}$ is productive for M" and "$\mathcal{T}$ is not productive for M" are obtained if $n = \infty$ and if $n \in \mathbb{N}$, respectively. □

We end this section with showing how our algorithm decides productivity of our running examples, the SCSs for J and M given in Example 3.8 and Fig. 1. Besides, we illustrate that productivity is sensitive to the precise definitions of the stream functions used by considering a slightly modified version of the SCS for M.

Example 6.10. For the definition of J from Example 3.8 we obtain:

$$[\mathsf{J}] = \mu J.\bullet(\bullet(\mathsf{box}(\overline{-+-}, J))) \to^2_{\mathsf{R1}} \mu J.\mathsf{box}(+\overline{-+}, \mathsf{box}(+\overline{-+}, \mathsf{box}(\overline{-+-}, J)))$$
$$\to_{\mathsf{R2}} \mu J.\mathsf{box}(++\overline{-+}, \mathsf{box}(\overline{-+-}, J)) \to_{\mathsf{R2}} \mu J.\mathsf{box}(++\overline{-+-}, J) \to_{\mathsf{R6}} \mathsf{src}(\underline{4})\,,$$

proving that J is not productive (only 4 elements can be evaluated).

Example 6.11. By rewriting $[\mathsf{M}]$ from Fig. 9 with parallel outermost rewriting (except that the composition of boxes is preferred to reduce the size of the terms) according to $\to_{\mathsf{R}}$ we get:

$$[\mathsf{M}] = \mu M.\bullet(\triangle(\mathsf{box}(\overline{-++}, \mathsf{box}(\overline{-+}, \mathsf{box}(\overline{-+-}, M))), \mathsf{box}(\overline{+-+}, \mathsf{box}(-\overline{-+}, M))))$$
$$\to^3_{\mathsf{R2}} \mu M.\bullet(\triangle(\mathsf{box}(\overline{-++-}, M), \mathsf{box}(+-\overline{-++}, M)))$$
$$\to_{\mathsf{R1}\cdot\mathsf{R3}} \mu M.\triangle(\mathsf{box}(+\overline{-+}, \mathsf{box}(\overline{-++-}, M)), \mathsf{box}(+\overline{-+}, \mathsf{box}(+-\overline{-++}, M)))$$
$$\to^2_{\mathsf{R2}} \mu M.\triangle(\mathsf{box}(+\overline{-++-}, M), \mathsf{box}(++-\overline{-++}, M))$$
$$\to_{\mathsf{R4}} \triangle(\mu M.\mathsf{box}(+\overline{-++-}, M), \mu M.\mathsf{box}(++-\overline{-++}, M))$$
$$\to_{\mathsf{R6}} \triangle(\mathsf{src}(\mathsf{fix}(+\overline{-++-})), \mathsf{src}(\mathsf{fix}(++-\overline{-++}))) = \triangle(\mathsf{src}(\infty), \mathsf{src}(\infty))$$
$$\to_{\mathsf{R7}} \mathsf{src}(\infty)\,,$$

witnessing productivity of the SCS for M. Note that the 'fine' definitions of zip and even are crucial in this setting. If we replace the definition of zip in the SCS for M by the 'coarser' one: $\mathsf{zip}^*(x:\sigma, y:\tau) \to x:y:\mathsf{zip}^*(\sigma,\tau)$ we obtain an SCS $\mathcal{T}^*$ where:

$$\begin{aligned}
[\mathsf{M}] &= \mu M.\bullet(\triangle(\mathsf{box}(\overline{-++}, \mathsf{box}(\overline{-+}, \mathsf{box}(\overline{-+-}, M))), \mathsf{box}(\overline{-++}, \mathsf{box}(-\overline{-+}, M))))\\
&\to^{3}_{\mathrm{R2}} \mu M.\bullet(\triangle(\mathsf{box}(\overline{-++-}, M), \mathsf{box}(-\overline{-++}, M)))\\
&\to_{\mathrm{R1\cdot R3}} \mu M.\triangle(\mathsf{box}(+\overline{-+}, \mathsf{box}(\overline{-++-}, M)), \mathsf{box}(+\overline{-+}, \mathsf{box}(-\overline{-++}, M)))\\
&\to^{2}_{\mathrm{R2}} \mu M.\triangle(\mathsf{box}(+\overline{-++-}, M), \mathsf{box}(+-\overline{-++}, M))\\
&\to_{\mathrm{R4\cdot R6}} \triangle(\mathsf{src}(\mathsf{fix}(+\overline{-++-})), \mathsf{src}(\mathsf{fix}(+-\overline{-++}))) = \triangle(\mathsf{src}(\infty), \mathsf{src}(\underline{1})) \to_{\mathrm{R7}} \mathsf{src}(\underline{1}).
\end{aligned}$$

Hence M is not productive in $\mathcal{T}^*$ (here it produces only one element).

7. Conclusion and ongoing research

We have shown that productivity is decidable for stream specifications that belong to the format of pure SCSs. The class of pure SCSs contains specifications that cannot be recognised automatically to be productive by the methods of [25,21,8, 13,23,7] (e.g. the SCS in Fig. 1). These previous approaches established criteria for productivity that are not applicable for disproving productivity; furthermore, these methods are either applicable to general stream specifications, but cannot be mechanised fully, or can be automated, but give a 'productive'/'don't know' answer only for a very restricted subclass. Our approach combines the features of being automatable and of obtaining a definite 'productive'/'not productive' decision for a rich class of stream specifications.

Note that we obtain decidability of productivity by restricting only the stream function layer of an SCS (formalised as an orthogonal TRS), while imposing no conditions on how the SCS layer makes use of the stream functions. The restriction to weakly guarded SFSs in pure SCSs is motivated by the wish to formulate a format of stream specifications for which productivity is decidable. More general formats to which our method can be applied are possible. In particular, we refer to [10] where the restrictions imposed on the stream function layer are relaxed in favour of a single remaining condition: stream function symbols do not occur nested on either side of their defining rules (again we do not impose any restrictions on the stream constant layer). In this way we obtain the more general format of 'flat stream specifications' which allows for the use of pattern matching on data for the definitions of stream functions. In [10] we give (i) a computable, 'data-obliviously optimal', sufficient condition for productivity of flat stream specifications, and we show (ii) decidability of productivity for flat stream specifications that are 'pure' (see Remark 3.5), a significant extension of the class of SCSs.

Beyond specific formats of stream specifications, our results can also be used in the following way: Suppose that a stream specification $\mathcal{T}$ has the property that for the stream functions occurring it holds that their quantitative consumption/production behaviour can be faithfully modelled by rational I/O sequences. Then the stream specification is productive if and only if the pebbleflow net built for $\mathcal{T}$ according to Definition 5.2, using the assumed modelling I/O sequences, rewrites to $\mathsf{src}(\infty)$. Hence productivity is still decidable under the assumption that the user is able to come up with modelling I/O sequences for the stream functions. Also lower and upper 'rational' bounds on the production of stream functions can be considered to obtain computable criteria for productivity and its complement. This will allow us to deal with stream functions that depend quantitatively on the value of stream elements and data parameters. Our approach can also be extended to calculate the precise production modulus of stream functions that are contexts built up of weakly guarded stream functions only, by reducing nets with free input ports to gates. All of these extensions of the result presented here are the subject of ongoing research.

The reader is invited to visit http://infinity.few.vu.nl/productivity/ where all additional material is available presently or http://en.wikipedia.org/wiki/ProPro for up-to-date links. Via these links also two software tools can be found: (i) an applet for the animation of pebbleflow nets, and (ii) an implementation of the decision algorithm for productivity of SCSs as part of a more powerful tool ProPro that automates a computable criterion for productivity on the substantially larger class of flat stream specifications introduced in [10]. We have tested the usefulness and feasibility of the implementation of our decision algorithm on various pure SCSs from the literature, and so far have not encountered excessive run-times. However, a precise analysis of the run-time complexity of our algorithm remains to be carried out.

Acknowledgements

For useful discussions we thank Clemens Kupke, Milad Niqui, Vincent van Oostrom, Femke van Raamsdonk, and Jan Rutten.

Appendix. Technical appendix

In this Appendix we provide a proof of Lemma 4.8, and foremost we prove preservation of production for the translation from SCSs to pebbleflow nets, that is, Theorem 5.4.

A.1. A Proof of Lemma 4.8: $\Pi_{\mathcal{N}} = \Pi_{\mathsf{P}}$

The statement of the lemma, $\Pi_{\mathsf{P}}(N,\alpha) = \Pi_{\mathcal{N}}(N,\alpha)$ for all $N \in \mathcal{N}$ and $\alpha : \mathcal{X} \to \overline{\mathbb{N}}$, can be proved by a straightforward induction on the number of μ-bindings of a net N, with a subinduction on the size of N. In the cases $N \equiv \mathsf{box}(\sigma, N')$ and $N \equiv \mu x.M$ Lemmas A.1 and A.2 are applied, respectively. □

Lemma A.1. *For $N \in \mathcal{N}$, $\sigma \in \pm^\omega$, $\alpha : \mathcal{V} \to \overline{\mathbb{N}}$: $\Pi_{\mathsf{P}}(\mathsf{box}(\sigma, N), \alpha) = \pi_\sigma(\Pi_{\mathsf{P}}(N,\alpha))$.*

Proof. We show that the relation $R \subseteq \overline{\mathbb{N}} \times \overline{\mathbb{N}}$ defined as follows is a bisimulation:

$$R := \{\langle \Pi_{\mathsf{P}}(\mathsf{box}(\sigma, N), \alpha), \pi_\sigma(\Pi_{\mathsf{P}}(N,\alpha))\rangle \mid \sigma \in \pm^\omega,\ N \in \mathcal{N},\ \alpha : \mathcal{V} \to \overline{\mathbb{N}}\},$$

that is, we prove that, for all $k_1, k_2 \in \overline{\mathbb{N}}, \sigma \in \pm^\omega, N \in \mathcal{N}$, and $\alpha : \mathcal{V} \to \overline{\mathbb{N}}$, if $k_1 = \Pi_{\mathsf{P}}(\mathsf{box}(\sigma, N), \alpha)$ and $k_2 = \pi_\sigma(\Pi_{\mathsf{P}}(N,\alpha))$, then either $k_1 = k_2 = 0$ or $k_1 = 1 + k_1'$, $k_2 = 1 + k_2'$ and $\langle k_1', k_2'\rangle \in R$. Let $k_1, k_2, \sigma, N, \alpha : \mathcal{V} \to \overline{\mathbb{N}}$, be such that $k_1 = \Pi_{\mathsf{P}}(\mathsf{box}(\sigma, N), \alpha)$ and $k_2 = \pi_\sigma(\Pi_{\mathsf{P}}(N,\alpha))$. By definition of $\pm^\omega$, we have that $\sigma \equiv -^n{+}\tau$ for some $n \in \mathbb{N}$ and $\tau \in \pm^\omega$. We proceed by induction on n. If $n = 0$, then $k_1 = 1 + k_1'$ with $k_1' = \Pi_{\mathsf{P}}(\mathsf{box}(\tau, N), \alpha)$ and $k_2 = 1 + k_2'$ with $k_2' = \pi_\tau(\Pi_{\mathsf{P}}(N,\alpha))$, and $\langle k_1', k_2'\rangle \in R$. If $n = n' + 1$, we distinguish cases: If $\Pi_{\mathsf{P}}(N,\alpha) = 0$, then $k_1 = k_2 = 0$. If $\Pi_{\mathsf{P}}(N,\alpha) = 1 + m$, then $N \twoheadrightarrow_{\mathsf{P}} \bullet(M)$ for some $M \in \mathcal{N}$ with $\Pi_{\mathsf{P}}(M,\alpha) = m$. Thus we get $k_1 = \Pi_{\mathsf{P}}(\mathsf{box}(-^{n'}{+}\tau, M), \alpha)$ and $k_2 = \pi_{-^{n'}+\tau}(\Pi_{\mathsf{P}}(M,\alpha))$, and $\langle k_1, k_2\rangle \in R$ by induction hypothesis. □

Lemma A.2. *For all nets $M \in \mathcal{N}$ and all assignments α, we have that $\Pi_{\mathsf{P}}(\mu x.M, \alpha)$ is the least fixed point of $\lambda n.\Pi_{\mathsf{P}}(M, \alpha[x \mapsto n])$.*

Proof. Let $\alpha : \mathcal{V} \to \overline{\mathbb{N}}$ be an arbitrary assignment and $M_0 := M^{\alpha[x\mapsto 0]}$. Observe that $\Pi_{\mathsf{P}}(\mu x.M, \alpha) = \Pi_{\mathsf{P}}(\mu x.M_0)$ and consider a rewrite sequence of the form

$$\mu x.M_0 \twoheadrightarrow_{\mathsf{P}} \ldots \twoheadrightarrow_{\mathsf{P}} \bullet^{n_i}(\mu x.M_i) \twoheadrightarrow_{\mathsf{P}} \bullet^{n_i}(\mu x.\bullet^{p_i}(M_i')) \twoheadrightarrow_{\mathsf{P}} \bullet^{n_i+p_i}(\mu x.M_{i+1}) \twoheadrightarrow_{\mathsf{P}} \ldots$$

where $p_i = \Pi_{\mathsf{P}}(M_i)$, $n_0 = 0$, $n_{i+1} = n_i + p_i$, and $M_{i+1} := M_i'(\bullet^{p_i}(x))$. Note that $\lim_{m\to\infty} n_m = \Pi_{\mathsf{P}}(\mu x.M_0)$; '$\le$' follows from $\forall m.\ \mu x.M_0 \twoheadrightarrow_{\mathsf{P}} \bullet^{n_m}(\mu x.M_m)$, and '$\ge$' since if $\lim_{m\to\infty} n_m < \infty$ then $\exists m \in \mathbb{N}$ such that $p_m := \Pi_{\mathsf{P}}(M_m) = 0$ and therefore $\Pi_{\mathsf{P}}(\mu x.M_0) = \Pi_{\mathsf{P}}(\bullet^{n_m}(\mu x.M_m)) = n_m$ by confluence.

Let $f_i = \lambda n.\Pi_{\mathsf{P}}(M_i(\bullet^n(x)))$, and $f_i' = \lambda n.\Pi_{\mathsf{P}}(M_i'(\bullet^n(x)))$. We prove

$$\forall k \in \mathbb{N}.\ f_0(n_m + k) = n_m + f_m(k) \qquad (*)$$

by induction over m. The base case $m = 0$ is trivial; we consider the induction step. We have $M_m \twoheadrightarrow_{\mathsf{P}} \bullet^{p_m}(M_m')$ and by substituting $\bullet^k(x)$ for x we get

$$\forall k \in \mathbb{N}.\ f_m(k) = p_m + f_m'(k). \qquad (**)$$

Moreover, since $f_{m+1}(k) = f_m'(p_m + k)$, we get $n_{m+1} + f_{m+1}(k) = n_{m+1} + f_m'(p_m + k) = n_m + p_m + f_m'(p_m + k) \overset{(**)}{=} n_m + f_m(p_m + k) \overset{(*)}{=} f_0(n_m + p_m + k) = f_0(n_{m+1} + k)$.

Let $f := f_0$. We proceed with showing $\forall m.\ f^m(0) = n_m$ by induction over $m \in \mathbb{N}$. For the base case $m = 0$ we have $f^0(0) = 0$ and $n_0 = 0$, and for the induction step we get $f^{m+1}(0) = f(f^m(0)) \overset{\text{IH}}{=} f(n_m) \overset{(*)}{=} n_m + f_m(0) = n_m + p_m = n_{m+1}$.

Hence $\mathrm{lfp}(f) = \lim_{m\to\infty} f^m(0) = \lim_{m\to\infty} n_m = \Pi_{\mathsf{P}}(\mu x.M_0) = \Pi_{\mathsf{P}}(\mu x.M, \alpha)$. □

A.2. A Proof of Theorem 5.4: Translating SCSs to nets preserves production

We recall the statement of Theorem 5.4: $\Pi_{\mathcal{T}}(\mathsf{M}_0) = \Pi_{\mathsf{P}}([\mathsf{M}_0])$ for all SCSs $\mathcal{T}$. For a given SCS $\mathcal{T}$, the proof proceeds by making intermediate steps via the production in a rewrite system $\mu\mathcal{T}$ for μ-terms over $\mathcal{T}$ with rewrite relation $\to_{\mu\mathcal{T}}$, and the production with respect to an alternative pebbleflow rewrite relation $\to_{\mathsf{P}'}$. Using these notions that are defined below together with a translation of stream terms t in $\mathcal{T}$ into μ-terms $[t]^{\mu\mathcal{T}}$ in $\mu\mathcal{T}$, the proof consists of the following three steps: $\Pi_{\mathcal{T}}(\mathsf{M}_0) = \Pi_{\mu\mathcal{T}}([\mathsf{M}_0]^{\mu\mathcal{T}}) = \Pi_{\mathsf{P}'}([\mathsf{M}_0]) = \Pi_{\mathsf{P}}([\mathsf{M}_0])$, which are justified by Lemmas A.3–A.5, respectively (Fig. 10). □

For the lemmas used in this proof, we introduce the following concepts: For an SCS $\mathcal{T}$, the rewrite system $\mu\mathcal{T}$ is defined as follows: its objects are μ-terms over the signature Σ of $\mathcal{T}$, and its set of rewrite steps $\cdot \to_{\mu\mathcal{T}} \cdot$ consists of steps $C[l^\sigma] \to C[r^\sigma]$ applying rules $l \to r$ of $\mathcal{T}$ *outside* of μ-bindings, and of steps that are applications of unfolding $\mu x.t(x) \to_{unf} t(\mu x.t(x))$. We denote by the symbol $\Pi_{\mu\mathcal{T}}$ the production function, and its version relativised to assignments, on μ-terms in $\mu\mathcal{T}$: these functions are defined analogously to the definition of $\Pi_{\mathcal{T}}$ in Definition 3.2 with the difference that $\to_{\mu\mathcal{T}}$ is used instead of $\to_{\mathcal{T}}$.

We also define a translation of stream terms t in an SCS $\mathcal{T}$ into corresponding μ-terms $[t]^{\mu\mathcal{T}}$ in $\mu\mathcal{T}$ that is very similar to the translation into pebbleflow nets in Definition 5.2. For every $t \in Ter(\Sigma_S)$, the μ-term translation $[t]^{\mu\mathcal{T}}$ of t is defined as $[t]^{\mu\mathcal{T}} := [t]_\emptyset^{\mu\mathcal{T}}$, based on the following inductive definition of translations $[t]_\alpha^{\mu\mathcal{T}}$ of terms $t \in Ter(\Sigma_S)$ with respect to

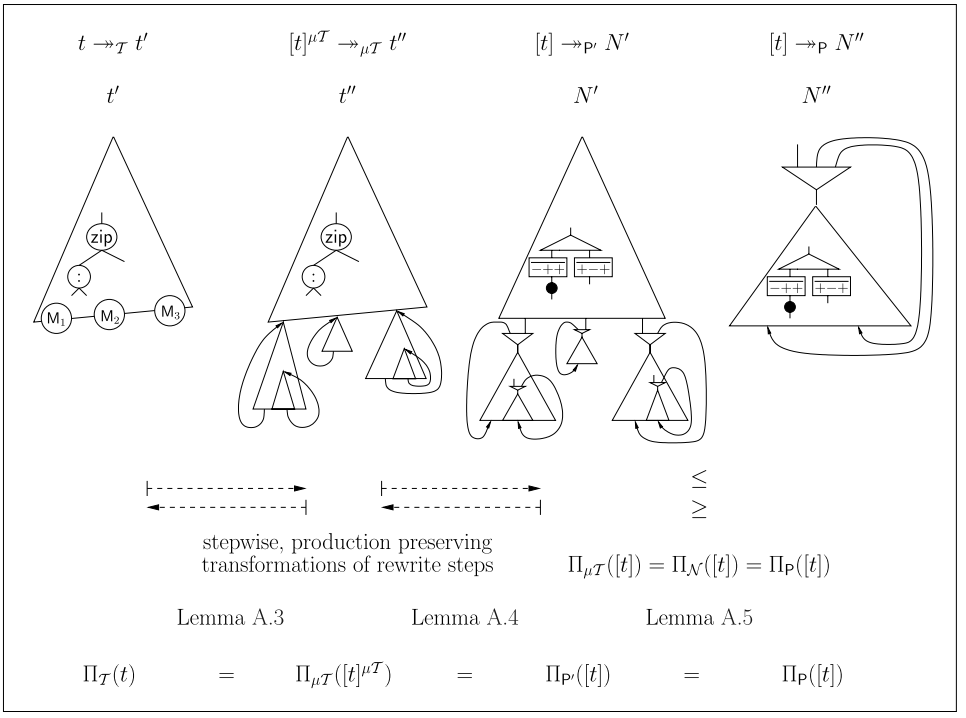


Fig. 10. The three steps in the proof of Theorem 5.4.

finite sets α of stream constant symbols in Σ_{sc} (the clauses below assume $\mathsf{M} \in \Sigma_{sc}$ and $\mathsf{f} \in \Sigma_{sf}$):

$$[\mathsf{M}]^{\mu\mathcal{T}}_{\alpha} = \begin{cases} \mu M.[rhs_{\mathsf{M}}]^{\mu\mathcal{T}}_{\alpha\cup\{\mathsf{M}\}} & \text{if } \mathsf{M} \notin \alpha \\ M & \text{if } \mathsf{M} \in \alpha \end{cases}$$

$$[u : s]^{\mu\mathcal{T}}_{\alpha} = u : [s]^{\mu\mathcal{T}}_{\alpha}$$

$$[\mathsf{f}(s_1, \ldots, s_{\sharp_s(\mathsf{f})}, u_1, \ldots, u_{\sharp_d(\mathsf{f})})]^{\mu\mathcal{T}}_{\alpha} = \mathsf{f}([s_1]^{\mu\mathcal{T}}_{\alpha}, \ldots, [s_{\sharp_s(\mathsf{f})}]^{\mu\mathcal{T}}_{\alpha}, u_1, \ldots, u_{\sharp_d(\mathsf{f})}).$$

Let the rewrite relation $\to_{\mathsf{P}''}$ be defined by the rules P1, P3–P5 in Definition 4.2 (ignoring P2) and of the unfolding rule $\mu x.N(x) \to_{unf} N(\mu x.N(x))$; all of these rules may be applied in arbitrary contexts. Using $\to_{\mathsf{P}''}$, the *alternative pebbleflow relation* $\to_{\mathsf{P}'}$ is defined as the restriction of $\to_{\mathsf{P}''}$ to applications of pebbleflow rules *outside* of μ-bindings. By $\Pi_{\mathsf{P}'}$ we mean the production function, and its version relativised to assignments, that are defined analogously to the production functions Π_{P} in Definition 4.6 with the difference of using $\to_{\mathsf{P}'}$ instead of $\to_{\mathsf{P}}$.

We use the following notation: for a binary relation $\to \subseteq A \times B$ and $A' \subseteq A$ let $\to(A') := \{b \mid \exists a \in A'.\, a \to b\}$; for a function $f : A \to B$ let $f(A') := \{f(a) \mid a \in A'\}$.

Lemma A.3. *For all $t \in Ter(\Sigma_S)$ in an SCS $\mathcal{T}$: $\Pi_{\mathcal{T}}(t) = \Pi_{\mu\mathcal{T}}([t]^{\mu\mathcal{T}})$ holds.*

Proof. For this proof we restrict the unfolding steps in the rewrite relation $\to_{\mu\mathcal{T}}$ to outermost-unfolding, noting that this does not affect the production function. Let $\Sigma_{sc} = \{\mathsf{M}_1, \ldots, \mathsf{M}_m\}$. Let $s \in Ter(\mu\mathcal{T})$, let $\varsigma(s)$ denote the term obtained from s by replacing all subterms M_i and $\mu M_i.s'$ with M_i, respectively; we say that 's has the property $\wp(s)$' if for all subterms $\mu x.s'$ of s: $\exists i.\, x = M_i \wedge \varsigma(s') \equiv rhs_{\mathsf{M}_i}$. Note that (i) $\wp([u]^{\mu\mathcal{T}})$ for every $u \in Ter(\mathcal{T})$, and (ii) $\wp$ is preserved under $\mu\mathcal{T}$ reduction. We show $(*)$ $\forall n \in \mathbb{N}.\forall s \in Ter(\mu\mathcal{T})$ with $\wp(s)$. $\to_{\mathcal{T}}^{\leq n} (\varsigma(s)) = \varsigma(\to_{\mu\mathcal{T}}^{\leq n}(s))$ by induction on the length n of reduction sequences. The case $n = 0$ is trivial. For the induction step we employ $\to^{n+1}(\_) = \to^{n}(\to(\_))$ together with (ii); therefore it suffices to prove $\to_{\mathcal{T}}(\varsigma(s)) = \varsigma(\to_{\mu\mathcal{T}}(s))$. The $\to_{R_S \cup R_D}$ steps carry over directly in both directions. From $\wp(s)$ we infer that $\to_{R_{sc}}$ steps in $\mathcal{T}$ can be translated into $\to_{unf}$ steps in $\mu\mathcal{T}$ and vice versa. Finally (i) and $(*)$ imply $\Pi_{\mathcal{T}}(t) = \Pi_{\mu\mathcal{T}}([t]^{\mu\mathcal{T}})$. □

Lemma A.4. *For all $t \in Ter(\Sigma_S)$ in an SCS $\mathcal{T}$: $\Pi_{\mu\mathcal{T}}([t]^{\mu\mathcal{T}}) = \Pi_{\mathsf{P}'}([t])$.*

Sketch of proof. The statement of the lemma will ultimately be established by a close correspondence between $\mu\mathcal{T}$-steps and $\to_{\mathsf{P}'}$-steps for SCSs in which none of the rules is collapsing, and neither erases nor permutes stream arguments. In order to use this correspondence, we transform an SCS $\mathcal{T}$ in three steps into this special form in such a way that $\mathcal{T}$-production $\Pi_{\mathcal{T}}(t)$ and pebbleflow net translation $[t]$ of terms t are preserved. For the sake of simplicity, we assume that stream function symbols have no data parameters.

First, we *eliminate collapsing rules* by adding a fresh symbol id to Σ_{sf} and the rule $\mathsf{id}(x:\sigma) \to x:\mathsf{id}(\sigma)$ to R_{sf}, and by replacing all collapsing rules $l \to \sigma_j$ in R_{sf} by $l \to \mathsf{id}(\sigma_j)$, respectively.

Second, we *transform R_{sf} to be non-erasing*. As a preprocessing, we replace every stream function symbol $\mathsf{f} \in \Sigma_{sf}$ by a symbol $\mathsf{f}_{\sharp_s(\mathsf{f})}$ that carries its stream arity as a subscript. Let $m_\sharp := \max \sharp_s(\Sigma_{sf})$ the maximum stream arity in Σ_{sf}. For every f_r now in Σ_{sf}, and every $n \in \mathbb{N}$ with $r < n \leq m_\sharp$, add an additional stream function symbol f_n. Then replace every stream function rule $\rho : \mathsf{f}_{\mathsf{r}_1}(\boldsymbol{s}^{r_1}) \to t_1 : \ldots : t_m : \mathsf{g}_{\mathsf{r}_2}(\boldsymbol{s'}^{r_2})$ by the following rules:

$$\mathsf{f}_{\mathsf{r}_1+\mathsf{n}}(\boldsymbol{s}^{r_1}, \tau_1, \ldots, \tau_n) \to t_1 : \ldots : t_m : \mathsf{g}_{\mathsf{r}_1+\mathsf{n}}(\boldsymbol{s'}^{r_2}, \sigma_{i_1}, \ldots, \sigma_{i_{r_1-r_2}}, \tau_1, \ldots, \tau_n)$$

for $n = 0, \ldots, m_\sharp - r_1$ where $\sigma_{i_1}, \ldots, \sigma_{i_{r_1-r_2}}$ are the erased stream variables of ρ and $\tau_1, \ldots, \tau_n$ are stream variables for matching so-called *phantom arguments*. As an example, consider $R_{sf} = \{\mathsf{f}_2(\sigma, x{:}\tau) \to x{:}\mathsf{g}_1(\sigma),\ \mathsf{g}_1(x{:}y{:}\sigma) \to x{+}y{:}\mathsf{g}_1(\sigma)\}$. The first rule is transformed into the non-erasing rule $\mathsf{f}_2(\sigma, x : \tau) \to x : \mathsf{g}_2(\sigma, \tau)$, and the second rule gives rise to $\mathsf{g}_1(x : y : \sigma) \to x + y : \mathsf{g}_1(\sigma)$ and $\mathsf{g}_2(x : y : \sigma, \tau_1) \to x + y : \mathsf{g}_2(\sigma, \tau_1)$.

Third, we *remove permutations of stream arguments*. We annotate function symbols with permutations instead of performing the permutation. For every $\mathsf{f} \in \Sigma_{sf}$ and $\phi : N_{\sharp_s(\mathsf{f})} \to N_{\sharp_s(\mathsf{f})}$ a bijection, where $N_{\sharp_s(\mathsf{f})} = \{1, \ldots, \sharp_s(\mathsf{f})\}$, let f_ϕ be a fresh symbol having the same arity as f. For $n \in \mathbb{N}$ let $\boldsymbol{s}^n$ be shorthand for $s_1, \ldots, s_n$ and for $\phi : N_n \to N_n$ let $\boldsymbol{s}^{n(\phi)}$ denote the permutation $s_{\phi^{-1}(1)}, \ldots, s_{\phi^{-1}(n)}$ w.r.t. ϕ. We replace every stream function rule $\mathsf{f}(\boldsymbol{s}^r) \to t_1 : \ldots : t_m : \mathsf{g}(\boldsymbol{s'}^{r(\phi_\mathsf{f}^{-1})})$ by all rules

$$\mathsf{f}_\phi(\boldsymbol{s}^{r(\phi)}) \to t_1 : \ldots : t_m : \mathsf{g}_{\phi\circ\phi_\mathsf{f}}(\boldsymbol{s'}^{r(\phi)})$$

for $\phi : N_r \to N_r$ a bijection. Note that after the third transformation step all permutation functions ϕ_f for $\mathsf{f} \in \Sigma_{sf}$ are the identity on $N_{\sharp_s(\mathsf{f})}$, respectively.

It is technical but not difficult to prove that $\mathcal{T}$-production $\Pi_\mathcal{T}(t)$ and pebbleflow net translation $[t]$ of terms t are preserved under these three transformations. Therefore in the sequel we can assume without loss of generality that none of the rules of $\mathcal{T}$ is collapsing, and neither erases nor permutes stream arguments: $\forall \mathsf{f} \in \Sigma_{sf}.\ \phi_\mathsf{f} = \mathrm{id}_{N_{\sharp_s(\mathsf{f})}}$.

To gain control about pebbleflow rewriting, we label gates with the function symbols from which they arise. In particular the translation $[\mathsf{f}]^\ell$ is a *labelled gate*:

$$[\mathsf{f}] = \mathsf{gate}^\mathsf{f}([\mathsf{f}]_1, \ldots, [\mathsf{f}]_{\sharp_s(\mathsf{f})})\,.$$

where $\mathsf{gate}^\mathsf{f}(\ldots)$ means that the leftmost box is labelled with f. For closed μ-terms v we define $[v]^\ell$, the translation of v into a *labelled pebbleflow* net (using labelled gates), as follows: $[\mu M.t]^\ell = \mu M.[t]^\ell$, $[t : u]^\ell = \bullet([u]^\ell)$, and $[\mathsf{f}(\boldsymbol{u}^{\sharp_s(\mathsf{f})}, \boldsymbol{t}^{\sharp_d(\mathsf{f})})]^\ell = [\mathsf{f}]^\ell([\boldsymbol{u}^{\sharp_s(\mathsf{f})}]^\ell)$. Moreover, for labelled nets N we use $\not\!\!\downarrow(N)$ to denote the pebbleflow net obtained from N by dropping the labels. Note that $[u] \equiv \not\!\!\downarrow([[u]^{\mu\mathcal{T}}]^\ell)$ for every $u \in \mathit{Ter}(\mathcal{T})$.

For every $\mathsf{f} \in \Sigma_{sf}$ there exists $\mathsf{g} \in \Sigma_{sf}$ such that for every $i \in \mathbb{N}$ with $1 \leq i \leq \sharp_s(\mathsf{f})$ we have $[\mathsf{f}]_i = -^{in(\mathsf{f},i)}{+}^{out(\mathsf{f})}[\mathsf{g}]_i$. On labelled pebbleflow nets we define the rewrite system $\to_{\lfloor\mathsf{P}'\rfloor}$ to consist of unfolding $\mu x.t(x) \to_{unf} t(\mu x.t(x))$ of μ-bindings and rewrite steps, *outside* of μ-bindings, with respect to the rules:

$$[\mathsf{f}]^\ell(\bullet^{in(\mathsf{f},1)}(N_1), \ldots, \bullet^{in(\mathsf{f},\sharp_s(\mathsf{f}))}(N_{\sharp_s(\mathsf{f})})) \to_{\lfloor\mathsf{P}'\rfloor} \bullet^{out(\mathsf{f})}([\mathsf{g}]^\ell(N_1, \ldots, N_{\sharp_s(\mathsf{f})}))$$

for every $\mathsf{f}(\ldots) \to \ldots \mathsf{g}(\ldots)$ in R_{sf}. Note that $\not\!\!\downarrow(\to_{\lfloor\mathsf{P}'\rfloor}) \subseteq \twoheadrightarrow_{\mathsf{P}'}$ and from confluence of $\twoheadrightarrow_{\mathsf{P}'}$ we infer: $\Pi_{\mathsf{P}'}([t]) = \Pi_{\lfloor\mathsf{P}'\rfloor}([[t]^{\mu\mathcal{T}}]^\ell)$.

We proceed with showing $\Pi_{\mu\mathcal{T}}([t]^{\mu\mathcal{T}}) = \Pi_{\mathsf{P}'}([t])$. Employing the above observations it is sufficient to prove that $[\twoheadrightarrow_{\mu\mathcal{T}}([t]^{\mu\mathcal{T}})]^\ell = \twoheadrightarrow_{\lfloor\mathsf{P}'\rfloor}([[t]^{\mu\mathcal{T}}]^\ell)$. The latter is implied by the one-step correspondence: $(*)$ for all closed μ-terms s, $[\to_{\mu\mathcal{T}}(s)]^\ell = \to_{\lfloor\mathsf{P}'\rfloor}([s]^\ell)$, using induction over the length of the reduction sequence. Now we prove $(*)$, therefor let s be an arbitrary closed μ-term. We start with '$\subseteq$': let s' with $\chi : s \to_{\mu\mathcal{T}} s'$. In case χ is an unfolding step, we get $[s]^\ell \to_{\lfloor\mathsf{P}'\rfloor} [s']^\ell$ likewise via an unfolding step. Otherwise χ is of the form:

$$C[\mathsf{f}(\boldsymbol{u_1}^{\boldsymbol{in(\mathsf{f},1)}} : t_1, \ldots, \boldsymbol{u}_{\boldsymbol{\sharp_s(\mathsf{f})}}^{\boldsymbol{in(\mathsf{f},\sharp_s(\mathsf{f}))}} : t_{\sharp_s(\mathsf{f})})] \to C[v_1 : \ldots : v_{out(\mathsf{f})} : \mathsf{g}(t_1, \ldots, t_{\sharp_s(\mathsf{f})})]$$

due to a stream function rule in R_{sf}. Then

$$[s]^\ell \equiv D[[\mathsf{f}]^\ell(\bullet^{in(\mathsf{f},1)}([t_1]^\ell), \ldots, \bullet^{in(\mathsf{f},\sharp_s(\mathsf{f}))}([t_{\sharp_s(\mathsf{f})}]^\ell))]$$
$$[s']^\ell \equiv D[\bullet^{out(\mathsf{f})}([\mathsf{g}]^\ell([t_1]^\ell, \ldots, [t_{\sharp_s(\mathsf{f})}]^\ell))]$$

for some context D and clearly $[s]^\ell \to_{\lfloor\mathsf{P}'\rfloor} [s']^\ell$. The direction '$\supseteq$' is analogous. □

Lemma A.5. *For all $N \in \mathcal{N}$: $\Pi_{\mathsf{P}'}(N) = \Pi_\mathsf{P}(N)$.*

Proof. In view of Lemma 4.8 it suffices to prove that for all nets $N \in \mathcal{N}$: $\Pi_{\mathsf{P}'}(N) = \Pi_\mathcal{N}(N)$ holds, and moreover, $\Pi_{\mathsf{P}'}(N, \alpha) = \Pi_\mathcal{N}(N, \alpha)$ for all assignments α. The proof of this statement proceeds by an inductive proof parallel to that used in the proof of Lemma 4.8, making use of confluence of $\to_{\mathsf{P}'}$ and of statements analogous to that of Lemma A.2, and Lemma A.1. Confluence of $\to_{\mathsf{P}'}$ follows easily from the fact that $\to_{\mathsf{P}''}$, which can be viewed as an orthogonal HRS, is confluent. The statement corresponding to Lemma A.1 can be shown analogously to the proof of that lemma.

It remains to show that $(*)$ $\Pi_{\mathsf{P}'}(\mu x.M, \alpha) = \mathrm{lfp}(\lambda n.\Pi_{\mathsf{P}'}(M, \alpha[x \mapsto n]))$, for all assignments α and $\mu x.M \in \mathcal{N}$. For this, let $\mu x.M(x) \in \mathcal{N}$ and let α be an assignment. Furthermore, let us denote by *FP* the fixed point in $(*)$. Then it follows

that $\{n_i\}_i \to_{i\to\infty} FP$ where the sequence $\{n_i\}_i$ in $\overline{\mathbb{N}}$ is defined as follows: $n_0 := \Pi_{\mathsf{P}'}(M, \alpha[x \mapsto 0])$, and, for all $i \in \mathbb{N}$, $n_{i+1} := \Pi_{\mathsf{P}'}(M, \alpha[x \mapsto n_i])$. Using confluence of $\to_{\mathsf{P}'}$, it is easy to show $\Pi_{\mathsf{P}'}(N_1(N_2(x)), \beta[x \mapsto 0]) = \Pi_{\mathsf{P}'}(N_1(x), \beta[x \mapsto \Pi_{\mathsf{P}'}(N_2(x), \beta[x \mapsto 0])])$ holds for all $N_1(x), N_2(x) \in \mathcal{N}$ and assignments β. Using this statement in a proof by induction, $(**)$ $n_k = \Pi_{\mathsf{P}'}(M^{k+1}(x), \alpha[x \mapsto 0])$ can be shown for all $k \in \mathbb{N}$, where M^{k+1} denotes the net $M(M(\ldots M(x)\ldots))$ with $k+1$ occurrences of M. Now since $\mu x.M(x) \twoheadrightarrow_{\mathsf{P}'} M^k(\mu x.M(x))$ by k unfolding steps, for all $k \in \mathbb{N}$, it follows for all $k \in \mathbb{N}$ that $\Pi_{\mathsf{P}'}(\mu x.M, \alpha) \geq n_k$, and hence that "$\geq$" holds in $(*)$.

For showing "$\leq$" in $(*)$, let $m \in \mathbb{N}$ with $m \leq \Pi_{\mathsf{P}'}(\mu x.M, \alpha)$ arbitrary. Then $\mu x.M \to_{\mathsf{P}'} \bullet^m(M')$ for some $M' \in \mathcal{N}$. If $k+1$ is the number of unfolding steps applied to a subterm $\mu x.M(x)$ in this rewrite sequence, then there also exists a rewrite sequence $\mu x.M(x) \twoheadrightarrow_{unf} M^{k+1}(\mu x.M(x)) \twoheadrightarrow_{\mathsf{P}'} \bullet^m(M')$ for some $M' \in \mathcal{N}$, where in the $\twoheadrightarrow_{\mathsf{P}'}$-steps on $M^{k+1}(\mu x.M(x))$ subterms $\mu x.M(x)$ are not rewritten. It follows that there is also a rewrite sequence $M^{k+1}(x) \twoheadrightarrow_{\mathsf{P}'} \bullet^m(M'')$, for some $M'' \in \mathcal{N}$. Now by $(**)$ it follows that $m \leq n_k \leq FP$. Since m was assumed arbitrarily with $m \leq \Pi_{\mathsf{P}'}(\mu x.M, \alpha)$, now also "$\leq$" in $(*)$ follows. □

References

[1] K. Aehlig, A finite semantics of simply-typed lambda terms for infinite runs of automata, Logical Methods in Computer Science (2007).
[2] J.-P. Allouche, J. Shallit, Automatic Sequences: Theory, Applications, Generalizations, Cambridge University Press, New York, 2003.
[3] Z.M. Ariola, J.W. Klop, Equational Term Graph Rewriting. Technical Report IR-391, Vrije Universiteit Amsterdam, 1995. ftp://ftp.cs.vu.nl/pub/papers/theory/IR-391.ps.Z.
[4] H.P. Barendregt, The Lambda Calculus, Its Syntax and Semantics, North-Holland, 1984.
[5] H.P. Barendregt, J.W. Klop, Applications of infinitary lambda calculus, Information of Computation (2008) (special issue on the occasion of the 60th birthday of Giuseppe Longo.
[6] S. Blom, Term Graph Rewriting — Syntax and Semantics, Ph. D. Thesis, Vrije Universiteit Amsterdam, 2001.
[7] W. Buchholz, A term calculus for (Co-)recursive definitions on streamlike data structures, Annals of Pure and Applied Logic 136 (1–2) (2005) 75–90.
[8] Th. Coquand, in: H. Barendregt, T. Nipkow (Eds.), Infinite Objects in Type Theory, in: TYPES, vol. 806, Springer-Verlag, Berlin, 1994, pp. 62–78.
[9] E.W. Dijkstra, On the Productivity of Recursive Definitions, 1980, EWD749.
[10] J. Endrullis, C. Grabmayer, D. Hendriks, Data-oblivious stream productivity, in: LPAR, in: LNCS, vol. 5330, Springer, 2008, pp. 79–96.
[11] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, J.W. Klop, Productivity of stream definitions, in: Proceedings of FCT 2007, in: LNCS, vol. 4639, Springer, 2007, pp. 274–287.
[12] E. Giménez, Codifying guarded definitions with recursive schemes, in: TYPES, 1994, pp. 39–59.
[13] J. Hughes, L. Pareto, A. Sabry, Proving the correctness of reactive systems using sized types, in: POPL '96, 1996 pp. 410–423.
[14] G. Kahn, The semantics of a simple language for parallel programming, in: Information Processing, 1974, pp. 471–475.
[15] R. Kennaway, J.W. Klop, M.R. Sleep, F.-J. de Vries, Transfinite reductions in orthogonal term rewriting systems, Information and Computation 119 (1) (1995) 18–38.
[16] R. Kennaway, J.W. Klop, M.R. Sleep, F.-J. de Vries, Infinitary lambda calculus, Theoretical Computer Science 175 (1) (1997) 93–125.
[17] J.W. Klop, R. de Vrijer, Infinitary normalization, in: We Will Show Them: Essays in Honour of Dov Gabbay (2), College Publications, 2005, pp. 169–192. ftp://ftp.cwi.nl/pub/CWIreports/SEN/SEN-R0516.pdf.
[18] Y. Lafont, Interaction nets, in: POPL '90, ACM Press, 1990, pp. 95–108.
[19] S. Peyton Jones, The Implementation of Functional Programming Languages, Prentice-Hall, 1987.
[20] J.J.M.M. Rutten, Behavioural differential equations: A coinductive calculus of streams, automata, and power series, Theoretical Computer Science 308 (1–3) (2003) 1–53.
[21] B.A. Sijtsma, On the productivity of recursive list definitions, ACM Transactions on Programming Languages and Systems 11 (4) (1989) 633–649.
[22] W.W. Tait, Intentional interpretations of functionals of finite type I, Journal of Symbolic Logic 32 (2) (1967).
[23] A. Telford, D. Turner, Ensuring the Productivity of Infinite Structures, Technical Report 14-97, The Computing Laboratory, Univ. of Kent at Canterbury, 1997.
[24] Terese, Term Rewriting Systems, in: Cambridge Tracts in Theoretical Computer Science, vol. 55, Cambridge University Press, 2003.
[25] W.W. Wadge, An extensional treatment of dataflow deadlock, Theoretical Computer Science 13 (1981) 3–15.