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

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.

Key words: recursive stream definitions, productivity, functional programming, dataflow networks

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, [16]) and the method of computability (for λ-calculus, [14]). 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 [2], and the emergence of coalgebraic techniques for infinite data types like streams [12]. While termination cannot be expected when infinite data are processed, infinitary notions of

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

termination become relevant. For example, in frameworks for the manipulation of infinite objects such as infinitary rewriting [8] and infinitary λ-calculus [9], basic notions are the properties WN^∞ and SN^∞ of infinitary weak and strong normalisation [10], 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 [13], 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 to read off consecutively 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 case that it is productive, here and henceforth we use the more accurate term 'stream specification'.) In Section 2 we define 'pure stream constant specifications' (SCSs) as orthogonal term rewriting systems, which are based on 'weakly guarded stream function specifications' (SFSs). In Section 3 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. More precisely, we give a translation of SCSs into 'rational' pebbleflow nets, and prove that this translation is production preserving. Finally in Section 4, 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 [5] presented at FCT'07. We follow [13,1] 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 to calculate fixed points of unary functions. Rational production moduli generalise those employed by [17,6,2,15], and enable us to precisely capture the consumption/production behaviour of a large class of stream functions.

Related Work. It is well-known that networks are devices for computing least fixed points of systems of equations [7]. The notion of 'productivity' (sometimes also referred to as 'liveness') was first mentioned by Dijkstra [3]. Since then several papers [17,13,2,6,15,1] have been devoted to criteria ensuring productivity. The common essence of these ap-

proaches is a quantitative analysis. In [17], 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 [13] 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, the definition $\mathsf{M} = 0 : \mathsf{zip}(\mathsf{inv}(\mathsf{even}(\mathsf{M})), \mathsf{tail}(\mathsf{M}))$ 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. Coquand [2] defines a syntactic criterion called 'guardedness' for ensuring productivity. This criterion is too restrictive for programming practice, because it disallows function applications to recursive calls. Telford and Turner [15] 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 recognized 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 recognize the specification of M to be productive. Hughes, Pareto and Sabry [6] 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 they have to assign 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 $\mathsf{tail}(\mathsf{M})$ cannot be typed. Moreover, their class of moduli is not closed under composition, leading to the need of approximations and a loss of power. Buchholz [1] presents a formal type system for proving productivity, whose basic ingredients are, closely connected to [13], 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 [15], but fails to deal with functions that have a negative effect like even, and hence with the specification of M above.

2. 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 we have: $\mathsf{M} \twoheadrightarrow 0 : 1 : 1 : 0 : 1 : 0 : 0 : 1 : \ldots$. Note 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 choosen a variant including the stream function even to demonstrate the strength of our approach. Previously, definitions like even could not be dealt with automatically (see the introduction).

A subtle point here is the definition of the stream function zip; had we used the

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

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. 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. But 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.

Let U be a finite set of *sorts*. A *U-sorted set* A is a family of sets $\{A_u\}_{u\in U}$; for $V \subseteq U$ we define $A_V := \bigcup_{v\in V} A_v$. A *$U$-sorted signature* Σ is a U-sorted set of symbols f, each equipped with an arity $ar(f) = \langle u_1 \cdots u_n, u\rangle \in U^* \times U$ where u is the sort of f. Let X be a U-sorted set of variables. The *U-sorted set of terms* $Ter(\Sigma, X)$ is inductively defined by: for all $u \in U$, $X_u \subseteq Ter(\Sigma, X)_u$, and $f(t_1, \ldots, t_n) \in Ter(\Sigma, X)_u$ whenever $f \in \Sigma$ with arity $ar(f) = \langle u_1 \cdots u_n, u\rangle$ and $t_i \in Ter(\Sigma, X)_{u_i}$. Usually we keep the variables implicit, assuming countably infinite X_u, and write $Ter(\Sigma)$. A *U-sorted term rewriting system (TRS)* is a pair $\langle \Sigma, R\rangle$ consisting of a U-sorted signature Σ and a U-sorted set R of *rules* that satisfy well-sortedness: $R_u \subseteq Ter(\Sigma, X)_u \times Ter(\Sigma, X)_u$ 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. A U-sorted TRS is called *finite* if both its signature and the set of its rules are finite.

Let $\mathcal{T} = \langle \Sigma, R\rangle$ a U-sorted TRS. We define $\mathcal{D}(\Sigma) := \{root(l) \mid l \to r \in R\}$, 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 $t_i \in Ter(\mathcal{C}(\Sigma))$; then ρ is a *defining rule for F*.

In the sequel we use sorts D and S for *data terms* and *stream terms*, respectively. We use ':' with arity $\langle DS, S\rangle$ as the *stream constructor* symbol. A *stream TRS* is an $\{S, D\}$-sorted TRS $\langle \Sigma, R\rangle$ with ':' $\in \Sigma_S$. The members of Σ_D and Σ_S are referred to as *data symbols* and *stream symbols*, respectively. A stream symbol is called a *stream constant* if its arity is $\langle \epsilon, S\rangle$. Without loss of generality we assume that all stream arguments of a stream function, if present, are in front. That is, for all $\mathsf{f} \in \Sigma$, $ar(\mathsf{f}) \in (S^{ar_s(\mathsf{f})} D^{ar_d(\mathsf{f})}) \times \{S, D\}$, and we refer to $ar_s(\mathsf{f}), ar_d(\mathsf{f}) \in \mathbb{N}$ as its *stream arity* and *data arity*, respectively. In this paper we restrict attention to *constructor stream TRSs*. The sets $Ter(\mathcal{C}(\Sigma))_D$ and $Ter_\infty(\mathcal{C}(\Sigma))_S$ 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 $ar_s(\mathsf{m}) = 0$.

Definition 2.1 Let $\mathcal{T}$ a stream TRS. The *production function* $\Pi_\mathcal{T} : Ter(\Sigma)_S \to \overline{\mathbb{N}}$ *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 *coinductive natural numbers*. For every $t \in \mathit{Ter}(\Sigma)_S$, we say that $\mathcal{T}$ *is productive for* t if $\Pi_{\mathcal{T}}(t) = \infty$.

Definition 2.2 A *stream function specification (SFS)* is a finite stream TRS $\mathcal{T} = \langle \Sigma, R \rangle$ such that: (i) $\langle \Sigma_D, R_D \rangle$ is a strongly normalising and orthogonal TRS (i.e. left-linear, non-overlapping redexes, see [16]), in which all ground terms have constructor normal forms; and (ii) for every stream function symbol $\mathsf{f} \in \Sigma_S \setminus \{:\}$ there is precisely one rule in R_S, denoted by ρ_{f}, the *defining rule* for f which has the form:

$$\mathsf{f}((\boldsymbol{x_1} : \sigma_1), \ldots, (\boldsymbol{x_{ar_s(\mathsf{f})}} : \sigma_{ar_s(\mathsf{f})}), y_1, \ldots, y_{ar_d(\mathsf{f})}) \to t_1 : \ldots : t_m : u \qquad (\rho_{\mathsf{f}})$$

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

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

$$u \equiv \sigma_i \qquad (2)$$

where $\mathsf{g} \in \Sigma_S$, $\phi_{\mathsf{f}} : \{1, \ldots, ar_s(\mathsf{g})\} \to \{1, \ldots, ar_s(\mathsf{f})\}$ is an injection used to permute stream arguments, $n_1, \ldots, n_{ar_s(\mathsf{f})}, m \in \mathbb{N}$, and $1 \leq i \leq ar_s(\mathsf{f})$. 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 $\mathsf{f}(\boldsymbol{t}) \to^+ \mathsf{f}(\boldsymbol{t'})$.

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 [13].

Definition 2.3 Let $\mathcal{T} = \langle \Sigma, R \rangle$ be a finite 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* ρ_{M}: $\mathsf{M} \to t$ 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 productivity.

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

Example 2.4 Consider the SCS $\langle \Sigma, R \rangle$ with $\Sigma = \{\mathsf{0}, \mathsf{1}, \mathsf{even}, \mathsf{odd}, \mathsf{J}, :\}$ and where R has the SCS layer $R_{sc} = \{\mathsf{J} \to \mathsf{0} : \mathsf{1} : \mathsf{even}(\mathsf{J})\}$, 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.

The following proposition justifies the definition of productivity of an SCS in Def. 2.3. It follows from the easily verifiable fact that every SCS is an orthogonal TRS, which entails that the rewrite relation generated is confluent and enjoys the property UN^{∞}.

Proposition 2.5 *Let $\mathcal{T}$ be a pure SCS. Then $\mathcal{T}$ is productive if and only if M_0 has an infinite constructor term $u_1 : u_2 : u_3 : \ldots$ as its unique infinitary normal form.*

3. Modelling with 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 'pebble'. Thus, a stream term $d : s$ is translated to $[d : s] = \bullet([s])$. Pebbleflow nets are inspired by interaction nets [11], 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 pure 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 3.1 Let $\mathcal{X}$ be a set. The set $\mathcal{N}$ of *pebbleflow terms* 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 $x \in \mathcal{X}$, $\sigma \in {\pm}^{\omega} \subseteq \{+, -\}^{\omega}$ is defined in Def. 3.3 below, and where, for $n \in \overline{\mathbb{N}}$, $\underline{n}$ is the *numeral* for n that is defined by $\underline{n} := \mathsf{s}^n(\mathsf{0})$ if $n \in \mathbb{N}$, and $\underline{\infty} := \mathsf{s}^{\omega}$.

Definition 3.2 The *pebbleflow rewrite relation* $\to_{\mathsf{P}}$ is defined as the compatible closure of the union of the following rules:

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

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 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

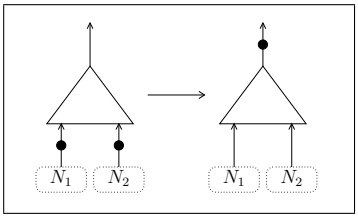


Fig. 2. Rule (P1).

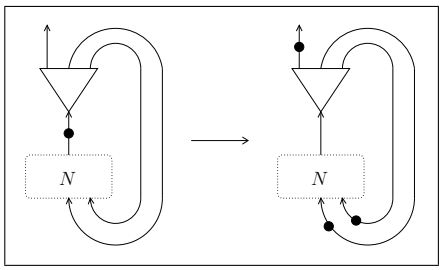


Fig. 3. Rule (P2).

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)'. Turning a cyclic net into a term (tree) means to introduce a notion of binding; certain nodes need to be labelled by a name (μx) so that a wire pointing to that node is replaced by a name (x) *referring* to the labelled node. 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 the next section we show how to reduce 'closed' nets, i.e. nets without free input ports, to sources.

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':

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

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 an input pebble, and $+$ represents a ready state for an output pebble. Pebbleflow through boxes is visualised in Figs. 4 and 5.

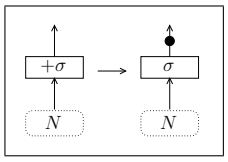


Fig. 4. Rule (P3).

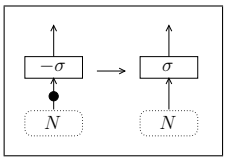


Fig. 5. Rule (P4).

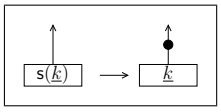


Fig. 6. Rule (P5).

Definition 3.3 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:

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

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.

The importance of rational nets will become evident in Sec. 4, where 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.

Stream functions f with a stream arity n are 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.

By $\mathsf{gate}(\sigma_1, \ldots, \sigma_n)$ we denote the n-ary context $\triangle_n(\mathsf{box}(\sigma_1, []_1), \ldots, \mathsf{box}(\sigma_n, []_n))$.

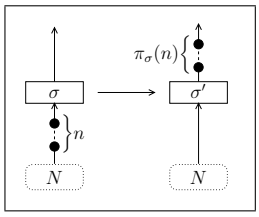


Fig. 7. $\mathsf{box}(\sigma, \bullet^n(N)) \to \bullet^{\pi_\sigma(n)}(\mathsf{box}(\sigma', N))$

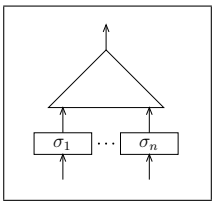


Fig. 8. A gate with n input ports.

Definition 3.4 The *production function* $\pi_\sigma : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ of (a box containing) a sequence $\sigma \in \pm^\omega$ is corecursively defined, for all $n \in \overline{\mathbb{N}}$, by $\underline{\pi_\sigma(n)} := \pi(\sigma, \underline{n})$:

$$\pi(+\sigma, n) = \mathsf{s}(\pi(\sigma, n)) \qquad \pi(-\sigma, 0) = 0 \qquad \pi(-\sigma, \mathsf{s}(n)) = \pi(\sigma, n)$$

Intuitively, $\pi_\sigma(n)$ is the number of outputs of a box containing sequence σ when fed n inputs, see Fig. 7. Note that production functions are well-defined due to our requirement on I/O sequences.

Lemma 3.5 *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 Thm. 11.6.9 in [16] then establishes the lemma. □

Definition 3.6 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* $\alpha \in \mathcal{X} \to \overline{\mathbb{N}}$, 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 (where $\mathrm{FV}(N) = \varnothing$), $N^\alpha = N$ and therefore $\Pi_{\mathsf{P}}(N, \alpha) = \Pi_{\mathsf{P}}(N)$, for all assignments α.

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 $\mu(f)$ which can be computed by $\mu(f) = \lim_{n\to\infty} f^n(0)$. In what follows we employ, for monotonic $f, g : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$, two basic properties: (C_1) $\forall n, m.\ f(\min(n, m)) = \min(f(n), f(m))$, and (C_2) $\mu(\lambda n.\, \min(f(n), g(n))) = \min(\mu(f), \mu(g))$.

Lemma 3.7 *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 $\alpha : \mathcal{X} \to \overline{\mathbb{N}}$ 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

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

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 $\lim_{m\to\infty} n_m = \Pi_{\mathsf{P}}(\mu x.M_0)$; '$\leq$' follows from $\forall m.\, \mu x.M_0 \twoheadrightarrow \bullet^{n_m}(\mu x.M_m)$, and '$\geq$' 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 $M_m \twoheadrightarrow \bullet^{p_m}(M_m')$ 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 $\mu(f) = \lim_{m\to\infty} f^m(0) = \lim_{m\to\infty} n_m = \Pi_{\mathsf{P}}(\mu x.M_0) = \Pi_{\mathsf{P}}(\mu x.M, \alpha)$. □

Lemma 3.8 *For $N \in \mathcal{N}$, $\sigma \in \pm^\omega$, $\alpha : \mathcal{X} \to \overline{\mathbb{N}}$: $\Pi_{\mathsf{P}}(\mathsf{box}(\sigma, N), \alpha) = \pi_\sigma(\Pi_{\mathsf{P}}(N, \alpha))$.*

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

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

that is, we prove that, for all $k_1, k_2 \in \overline{\mathbb{N}}$, $\sigma \in \pm^\omega$, $N \in \mathcal{N}$, and $\alpha : \mathcal{X} \to \overline{\mathbb{N}}$, 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 $k_1, k_2, \sigma, N, \alpha : \mathcal{X} \to \overline{\mathbb{N}}$, 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. □

We define an alternative net production function $\Pi_{\mathcal{N}}$ (equivalent to Π_{P}) that provides some useful intuition and will allow to get a handle on proving that production is preserved by the net reduction relation introduced in Section 4.

Definition 3.9 The mapping $\Pi_{\mathcal{N}} : \mathcal{N} \times (\mathcal{X} \to \overline{\mathbb{N}}) \to \overline{\mathbb{N}}$ is defined inductively by:

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

Next, we show that the net production functions Π_{P} and $\Pi_{\mathcal{N}}$ coincide. Note that $\Pi_{\mathcal{N}}$ is monotonic in its second argument.

Lemma 3.10 *For all nets N and assignments α, it holds $\Pi_{\mathsf{P}}(N, \alpha) = \Pi_{\mathcal{N}}(N, \alpha)$.*

Proof. The statement of the lemma 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 $N \equiv \mu x.M$ Lem. 3.8 and Lem. 3.7 are applied, respectively. $\square$

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.

Definition 3.11 Let $\mathcal{T} = \langle \Sigma_D \uplus \Sigma_{sf} \uplus \{:\}, R_D \uplus R_{sf} \rangle$ be an SFS. Then, for each $\mathsf{f} \in \Sigma_{sf}$ the *translation* of f is a rational gate $[\mathsf{f}] : \mathcal{N}^{ar_s(\mathsf{f})} \to \mathcal{N}$ defined by:

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

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: $\mathsf{f}(\boldsymbol{x_1} : \sigma_1, \ldots, \boldsymbol{x_{ar_s(\mathsf{f})}} : \sigma_{ar_s(\mathsf{f})}, y_1, \ldots, y_{ar_d(\mathsf{f})}) \to t_1 : \ldots : t_m : u$, where:

(a) $u \equiv \mathsf{g}(\sigma_{\phi_{\mathsf{f}}(1)}, \ldots, \sigma_{\phi_{\mathsf{f}}(ar_s(\mathsf{f})')}, t'_1, \ldots, t'_{ar_d(\mathsf{f})'})$, 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 define a 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 3.12 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 $\rho_{\mathsf{M}} \equiv \mathsf{M} \to rhs_{\mathsf{M}}$ 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):

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

$$[t : u]_\alpha = \bullet([u]_\alpha)$$

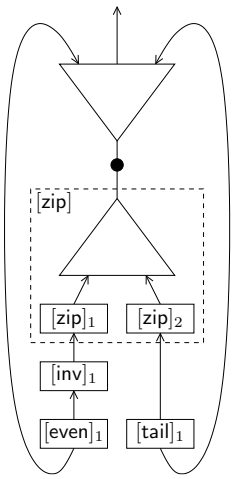
$$[\mathsf{f}(u_1, \ldots, u_{ar_s(\mathsf{f})}, t_1, \ldots, t_{ar_d(\mathsf{f})})]_\alpha = [\mathsf{f}]([u_1]_\alpha, \ldots, [u_{ar_s(\mathsf{f})}]_\alpha)$$

Example 3.13 Reconsider the SCS given in Figure 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 Def. 3.11.)

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.

Theorem 3.14 *For every pure SCS $\mathcal{T}$ it holds:* $\Pi_{\mathcal{T}}(\mathsf{M}_0) = \Pi_{\mathsf{P}}([\mathsf{M}_0])$.

Proof. For a given pure 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



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

$$[\mathsf{M}] = \mu M.\bullet(\triangle(\mathsf{box}(\overline{-++}, \mathsf{box}(\overline{-+}, \mathsf{box}(\overline{-+-}, M))), \mathsf{box}(\overline{+-+}, \mathsf{box}(-\overline{-+}, M))))$$

Fig. 9. The net translation $[\mathsf{M}]$ of the SCS for M.

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 Lem. 3.15, Lem. 3.16, and Lem. 3.17, respectively. □

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 Def. 2.1 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 Def. 3.12. For every $t \in \mathit{Ter}(\Sigma_S)$, the μ-term translation $[t]^{\mu\mathcal{T}}$ of t is defined as $[t]^{\mu\mathcal{T}} := [t]^{\mu\mathcal{T}}_{\emptyset}$, based on the following inductive definition of translations $[t]^{\mu\mathcal{T}}_{\alpha}$ of terms $t \in \mathit{Ter}(\Sigma_S)$ with respect to finite sets α of stream constant symbols in Σ_{sc} (the clauses below assume $\mathsf{M} \in \Sigma_{sc}$ and $\mathsf{f} \in \Sigma_{sf}$):

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

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

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

Let the rewrite relation $\to_{\mathsf{P}''}$ be defined as the compatible closure of the union of the pebbleflow rules P1, P3–P5 in Def. 3.2 (ignoring P2, the rolling rule for leading pebbles), and of the unfolding rule $\mu x.N(x) \to_{unf} N(\mu x.N(x))$. 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 Def. 3.6 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 3.15 *For all $t \in Ter(\Sigma_S)$ in an SCS $\mathcal{T}$ it holds: $\Pi_{\mathcal{T}}(t) = \Pi_{\mu\mathcal{T}}([t]^{\mu\mathcal{T}})$.*

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 $\Pi_{\mathcal{T}}(t) = \Pi_{\mu\mathcal{T}}([t]^{\mu\mathcal{T}})$. $\square$

Lemma 3.16 *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 are 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 $\mathsf{f}_{ar_s(\mathsf{f})}$ that carries its arity as a subscript. Let $m_{ar} := \max ar_s(\Sigma_{sf})$ the maximum stream arity in Σ_{sf}. For every f_r now in Σ_{sf}, and every $n \in \mathbb{N}$ with $r < n \leq m_{ar}$, 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 $n = 0, \ldots, m_{ar} - r_1$ 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 $\phi : N_{ar_s(\mathsf{f})} \to N_{ar_s(\mathsf{f})}$ a bijection, where $N_{ar_s(\mathsf{f})} = \{1, \ldots, ar_s(\mathsf{f})\}$, 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 $N_{ar_\mathsf{s}(\mathsf{f})}$, 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 w.l.o.g. that none of the rules of $\mathcal{T}$ is collapsing, and neither erases nor permutes stream arguments: $\forall \mathsf{f} \in \Sigma_{sf}.\ \phi_\mathsf{f} = \mathrm{id}_{N_{ar_\mathsf{s}(\mathsf{f})}}$.

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*:

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

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 $[\mathsf{f}(\boldsymbol{u}^{ar_\mathsf{s}(\mathsf{f})}, \boldsymbol{t}^{ar_\mathsf{d}(\mathsf{f})})]^\ell = [\mathsf{f}]^\ell([\boldsymbol{u}^{ar_\mathsf{s}(\mathsf{f})}]^\ell)$. 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 $u \in Ter(\mathcal{T})$.

For every $\mathsf{f} \in \Sigma_{sf}$ there exists $\mathsf{g} \in \Sigma_{sf}$ such that for every $i \in \mathbb{N}$ with $1 \le i \le ar_\mathsf{s}(\mathsf{f})$ 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:

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

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:

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

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

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

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

Lemma 3.17 *For all $N \in \mathcal{N}$: $\Pi_{\mathsf{P}'}(N) = \Pi_\mathsf{P}(N)$.*

Proof. In view of Lem. 3.10 it suffices to prove that for all nets $N \in \mathcal{N}$ it holds: $\Pi_{\mathsf{P}'}(N) = \Pi_\mathcal{N}(N)$, 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 Lem. 3.10, making use of confluence of $\to_{\mathsf{P}'}$ and of statements analogous to that of Lem. 3.7, and Lem. 3.8. 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 Lem. 3.8 can be shown analogously to the proof of that lemma.

It remains to show that $(*)$ $\Pi_{\mathsf{P}'}(\mu x.M, \alpha) = \mu(\lambda n.\Pi_{\mathsf{P}'}(M, \alpha[x \mapsto n]))$, 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 l 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. □

4. Deciding Productivity

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

Definition 4.1 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 $\sigma, \tau \in \pm^\omega$, $k, k_1, k_2 \in \overline{\mathbb{N}}$, and $\underline{\mathsf{min}(n, m)}$, $\underline{\pi_\sigma(k)}$, $\underline{\sigma \circ \tau}$ (see Def. 4.2) and $\underline{\mathsf{fix}(\sigma)}$ (see Def. 4.4) are term representations of operation results.

Definition 4.2 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 4.3 *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 Def. 3.2 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 4.4 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:

$$\underline{\mathsf{fix}(+\sigma)} = \mathsf{s}(\underline{\mathsf{fix}(\delta(\sigma))}) \qquad \delta(+\sigma) = +\delta(\sigma)$$
$$\underline{\mathsf{fix}(-\sigma)} = \mathsf{0} \qquad \delta(-\sigma) = \sigma$$

Lemma 4.5 *For all $\sigma \in \pm^\omega$, we have $\mu(\pi_\sigma) = \mathsf{fix}(\sigma)$.*
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 $\mathsf{box}(\sigma, \mathsf{box}(\sigma, \mathsf{box}(\sigma, \ldots)))$ is 'production equivalent' to $\mathsf{src}(\underline{\mathsf{fix}(\sigma)})$.

Lemma 4.6 *Net reduction preserves production: $\Pi_{\mathsf{P}}(N) = \Pi_{\mathsf{P}}(N')$ if $N \to_{\mathsf{R}} N'$.*
Proof. By Lem. 3.10 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 Lem. 4.3 (ii). For rule (R3) the desired equality follows from C_1 on page 8. For rule (R4) we conclude by C_2 ibid. For rule (R6) we use Lemma 4.5. For the remaining rules the statement trivially holds. For the induction step, the statement easily follows from the induction hypotheses. □

Lemma 4.7 *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 Lem. 4.3 (i)), and hence $\to_{\mathsf{R}}$ is locally confluent, by the Critical Pairs Lemma (cf. [16]). (For a complete proof, see [4].) 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 $\mathsf{box}, \triangle$, then M constitutes a redex w.r.t. (R2), (R3), respectively. If $M \equiv \triangle(M_1, M_2)$, we have a redex w.r.t. (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 w.r.t. (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 (Def. 4.2 and 4.4). 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 $\boldsymbol{\infty}$. The reader may confer [4] for further details.

Lemma 4.8 *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 4.9 *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 Def. 3.12. It is easy to verify that $[\mathsf{M}]$ is in fact a rational net. Second, by the algorithm stated by Lem. 4.8, $[\mathsf{M}]$ is collapsed to a source $\mathsf{src}(\underline{n})$ with $n \in \overline{\mathbb{N}}$. By Thm. 3.14 it follows that $[\mathsf{M}]$ has the same production as M in $\mathcal{T}$, and by Lem. 4.6 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 $n = \boldsymbol{\infty}$ 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 Ex. 2.4 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 4.10 For the definition of J from Ex. 2.4 we obtain:

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

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

Example 4.11 By rewriting $[\mathsf{M}]$ from Fig. 9 with parallel outermost rewriting (except that composition of boxes is preferred to reduce the size of the terms) according to $\to_{\mathsf{R}}$ we get:

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

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:

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

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

5. 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 [17,13,2,6,15,1] (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 an effectively recognisable format of stream specifications for which productivity is decidable. More general recognisable formats to which our method can be applied are possible. If the requirement of a recognisable format is dropped, our approach allows to show decidability of productivity for stream specifications that are based on SFSs which can (quantitatively) faithfully be described by 'rational' I/O sequences. 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 (see also [4]).

The reader may want to visit `http://infinity.few.vu.nl/productivity/` for additional material. There, an implementation of the decision algorithm for productivity of SCSs as

well as an animation tool for pebbleflow nets can be found. 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.

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

References

[1] W. Buchholz. A Term Calculus for (Co-)Recursive Definitions on Streamlike Data Structures. *Annals of Pure and Applied Logic*, 136(1-2):75–90, 2005.
[2] Th. Coquand. Infinite Objects in Type Theory. In H. Barendregt and T. Nipkow, editors, *TYPES*, volume 806, pages 62–78. Springer-Verlag, Berlin, 1994.
[3] E.W. Dijkstra. On the Productivity of Recursive Definitions, 1980. EWD749.
[4] J. Endrullis, C. Grabmayer, and D. Hendriks. Productivity of Stream Definitions. Technical report, Vrije Universiteit Amsterdam, 2007. Available via `http://infinity.few.vu.nl/productivity/`.
[5] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J.W. Klop. Productivity of Stream Definitions. In *Proceedings of FCT 2007*, number 4639 in LNCS, pages 274–287. Springer, 2007.
[6] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL '96*, pages 410–423, 1996.
[7] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Information Processing*, pages 471–475, 1974.
[8] R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Inf. and Comput.*, 119(1):18–38, 1995.
[9] R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Infinitary Lambda Calculus. *TCS*, 175(1):93–125, 1997.
[10] J.W. Klop and R. de Vrijer. Infinitary Normalization. In *We Will Show Them: Essays in Honour of Dov Gabbay (2)*, pages 169–192. College Publications, 2005. Cf. `http://web.mac.com/janwillemklop/iWeb/Site/Bibliography.html`.
[11] Y. Lafont. Interaction Nets. In *POPL '90*, pages 95–108. ACM Press, 1990.
[12] J.J.M.M. Rutten. Behavioural Differential Equations: a Coinductive Calculus of Streams, Automata, and Power Series. *TCS*, 308(1-3):1–53, 2003.
[13] B.A. Sijtsma. On the Productivity of Recursive List Definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.
[14] W.W. Tait. Intentional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2), 1967.
[15] A. Telford and D. Turner. Ensuring the Productivity of Infinite Structures. Technical Report 14-97, The Computing Laboratory, Univ. of Kent at Canterbury, 1997.
[16] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
[17] W.W. Wadge. An Extensional Treatment of Dataflow Deadlock. *TCS*, 13:3–15, 1981.