Productivity of Stream Definitions*

[1] Vrije Universiteit Amsterdam, Department of Computer Science,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{ariya,joerg}@few.vu.nl, {diem,klop}@cs.vu.nl
[2] Radboud Universiteit Nijmegen, Department of Computer Science,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
[3] Universiteit Utrecht, Department of Philosophy,
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
clemens@phil.uu.nl

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 continuously in such a way that a uniquely determined stream 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, [15]) and the method of computability (for λ-calculus, [13]). 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 [11]. While termination cannot be expected when infinite data are processed, infinitary notions of termination become relevant. For example, in formal frameworks for the manipulation of infinite objects such as infinitary rewriting [7] and infinitary λ-calculus [8], basic notions are the properties WN^∞ of infinitary weak normalisation and SN^∞ of infinitary strong normalisation [9].

In the functional programming literature the notion of 'productivity' has arisen, initially in the pioneering work of Sijtsma [12], as a natural strengthening

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

E. Csuhaj-Varjú and Z. Ésik (Eds.): FCT 2007, LNCS 4639, pp. 274–287, 2007.

of the property WN^{∞}. A recursive stream definition is called productive if not only can the definition be evaluated continuously to build up an 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 increasingly 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 definitions that hitherto could not be handled automatically. 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 definition by a finite 'pebbleflow net', and the process of evaluating a definition 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 definition in a quantitatively precise manner.

We follow [12] in describing the quantitative input/output behaviour of a stream function f by a non-decreasing 'production function' $\beta_f : (\overline{\mathbb{N}})^r \to \overline{\mathbb{N}}$ such that the first $\beta_f(n_1, \ldots, n_r)$ elements of $f(t_1, \ldots, t_r)$ can be computed whenever the first n_i elements of t_i are defined. More specifically, we employ 'rational' production functions $\beta : (\overline{\mathbb{N}})^r \to \overline{\mathbb{N}}$ that, for $r = 1$, have eventually periodic difference functions $\Delta_\beta(n) := \beta(n+1) - \beta(n)$, that is $\exists n, p \in \mathbb{N}.\, \forall m \geq n.\, \Delta_\beta(m) = \Delta_\beta(m+p)$. This class is effectively closed under composition, and allows to calculate fixed points of unary functions. Rational production functions generalise those employed by [16], [5], [2], and [14], 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 [6]. The notion of 'productivity' (sometimes also referred to as 'liveness') was first mentioned by Dijkstra [3]. Since then several papers [16,12,2,5,14,1] have been devoted to criteria ensuring productivity. The common essence of these approaches is a quantitative analysis. In [16], Wadge uses dataflow networks to model fixed points of equations. He devises a so-called *cyclic sum test*, using production functions of the form $\beta_f(n_1, \ldots, n_r) = \min(n_1 + a_{f,1}, \ldots, n_r + a_{f,r})$ with $a_{f,i} \in \mathbb{Z}$, i.e. the output *leads* or *lags* the input by a fixed value $a_{f,i}$. Sijtsma [12] points out that this class of production functions is too restrictive to capture the behaviour of commonly

used stream operations like $\mathsf{even}, \mathsf{dup}, \mathsf{zip}$ and so forth. Therefore he develops an approach allowing arbitrary production functions $\beta_f : \mathbb{N}^r \to \mathbb{N}$, 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 [14] extend the notion of guardedness with a method in the flavour of Wadge. However, their approach does not overcome Sijtsma's criticism. Hughes, Pareto and Sabry [5] introduce a type system using production functions with the property that $\beta_f(a \cdot x + b) = c \cdot x + d$ for some $a, b, c, d \in \mathbb{N}$. This class of functions is not closed under composition, leading to the need of approximations and a loss of power. Moreover their typing system rejects definitions like $\mathsf{M} = a : b : \mathsf{tail}(\mathsf{M})$, where ':' is the infix stream constructor, because tail is applied to the recursive call. Buchholz [1] presents a formal type system for proving productivity, whose basic ingredients are, closely connected to [12], unrestricted production functions $\beta_f : \mathbb{N}^r \to \mathbb{N}$. In order to obtain an automatable method, Buchholz also devises a syntactic criterion to ensure productivity. This criterion easily handles all the examples of [14], but fails to deal with functions that have a negative effect 'worse than tail'.

2 Recursive Stream Specifications

In this section the concepts of 'stream constant specification' (SCS) and 'stream function specification' (SFS) are introduced. We use a two-layered set-up, which is illustrated by the well-known definition $\mathsf{M} = 0{:}1{:}\mathsf{zip}(\mathsf{tail}(\mathsf{M}), \mathsf{inv}(\mathsf{tail}(\mathsf{M})))$ of the Thue–Morse sequence. This corecursive definition employs separate definitions of the stream functions zip and tail, contained in Ex. 1 below, and of the definition $\mathsf{inv}(x{:}\sigma) = (1{-}x){:}\mathsf{inv}(\sigma)$ of the stream function inv. Stream constants are written using uppercase letters, stream and data functions are written lowercase.

In order to distinguish between *data terms* and *streams* we use the framework of many-sorted term rewriting. Let S be a finite set of sorts. An S-sorted set A is a family of sets $(A_s)_{s \in S}$. An S-sorted signature Σ is a set of function symbols, each having a fixed arity $\mathsf{ar}(f) \in S^* \times S$. Let X be an S-sorted set of variables. The S-sorted set of terms $\mathit{Ter}(\Sigma, X)$ is inductively defined by: $X_s \subseteq \mathit{Ter}(\Sigma, X)_s$ for all $s \in S$ and $f(t_1, \ldots, t_n) \in \mathit{Ter}(\Sigma, X)_s$ whenever $f \in \Sigma$ with arity $\langle s_1 \cdots s_n, s \rangle$ and $t_i \in \mathit{Ter}(\Sigma, X)_{s_i}$. An S-sorted term rewriting system (TRS) over an S-sorted signature Σ is an S-sorted set R where $R_s \subseteq \mathit{Ter}(\Sigma, X)_s \times \mathit{Ter}(\Sigma, X)_s$ for all $s \in S$, satisfying the standard TRS requirements for rules. An S-sorted TRS is called *finite* if both its signature and the set of all of its rules are finite.

In the sequel let $S = \{d, s\}$ where d is the sort of *data terms* and s is the sort of *streams*. We say that a $\{d, s\}$-sorted TRS $\langle \Sigma, R \rangle$ is a *stream TRS* if there exists a partition of the signature $\Sigma = \Sigma_d \uplus \Sigma_{sf} \uplus \Sigma_{sc} \uplus \{:\}$ such that the arity of the symbols from Σ_d is in $\langle d^*, d \rangle$, for Σ_{sf} in $\langle \{s, d\}^*, s \rangle$, for Σ_{sc} in $\langle \epsilon, s \rangle$ and ':' has arity $\langle ds, s \rangle$. Accordingly, the symbols in Σ_d are referred to as the *data symbols*, ':' as the *stream constructor symbol*, the symbols in Σ_{sf} as the *stream function symbols* and the symbols in Σ_{sc} as the *stream constant symbols*.

Without loss of generality we assume that for all $\mathsf{f} \in \Sigma_{sf}$ the stream arguments are in front. That is, f has arity $\langle s^{r_s} d^{r_d}, s\rangle$ for some $r_s, r_d \in \mathbb{N}$; we say that f has arity $\langle r_s, r_d\rangle$ for short.

Definition 1. Let $\mathcal{T} = \langle \Sigma, R\rangle$ be a finite stream TRS with $\Sigma = \Sigma_d \uplus \Sigma_{sf} \uplus \{:\}$ and a partition $R = R_d \uplus R_{sf}$ of its set of rules. Then $\mathcal{T}$ (together with these partitions) is called a *weakly guarded stream function specification (SFS)* if:

(i) $\mathcal{T}$ is orthogonal (i.e. left-linear, non-overlapping redex patterns, see [15]).
(ii) $\langle \Sigma_d, R_d\rangle$ is a strongly normalising TRS.
(iii) For every stream function symbol $\mathsf{f} \in \Sigma_{sf}$ there is precisely one rule in R_{sf}, denoted by ρ_f, the *defining rule* for f. Furthermore, for all $\mathsf{f} \in \Sigma_{sf}$ with arity $\langle r_s, r_d\rangle$, the rule $\rho_\mathsf{f} \in R_{sf}$ has the form:

$$\mathsf{f}((\boldsymbol{x_1} : \sigma_1), \ldots, (\boldsymbol{x_{r_s}} : \sigma_{r_s}), y_1, \ldots, y_{r_d}) \to u$$

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:

$$u \equiv t_1 : \ldots : t_{m_\mathsf{f}} : \mathsf{g}(\sigma_{\pi_\mathsf{f}(1)}, \ldots, \sigma_{\pi_\mathsf{f}(r'_s)}, t'_1, \ldots, t'_{r'_d}), \tag{1}$$

$$u \equiv t_1 : \ldots : t_{m_\mathsf{f}} : \sigma_i \tag{2}$$

Here, the terms $t_1, \ldots, t_{m_\mathsf{f}} \in \mathit{Ter}(\Sigma_d)$ are called *guards* of f. Furthermore, $\mathsf{g} \in \Sigma_{sf}$ with arity $\langle r'_s, r'_d\rangle$, $\pi_\mathsf{f} : \{1, \ldots, r'_s\} \to \{1, \ldots, r_s\}$ is an injection used to permute stream arguments, $n_1, \ldots, n_{r_s}, m_\mathsf{f} \in \mathbb{N}$, and $1 \le i \le r_s$. In case (1) we write $\mathsf{f} \rightsquigarrow \mathsf{g}$, and say f 'depends on' g.
(iv) Every stream function symbol $\mathsf{f} \in \Sigma_{sf}$ is *weakly guarded* in $\mathcal{T}$, i.e. on every dependency cycle $\mathsf{f} \rightsquigarrow \mathsf{g} \rightsquigarrow \cdots \rightsquigarrow \mathsf{f}$ there is at least one guard.

It is easy to show that every function symbol $\mathsf{f} \in \Sigma_{sf}$ in an SFS defines a unique function that maps stream arguments and data arguments to a stream, which can be computed, for given infinite stream terms $u_1, \ldots, u_{r_s}$ in constructor normal form (that is, being of the form $s_0{:}s_1{:}s_2{:}\ldots$) and data terms $t_1, \ldots, t_{r_d}$, by infinitary rewriting as the infinite normal form of the term $\mathsf{f}(u_1, \ldots, u_{r_s}, t_1, \ldots, t_{r_d})$. Note that the definition covers a large class of stream functions including tail, even, odd, zip, add. However, the function head defined by $\mathsf{head}(x : \sigma) = x$, possibly creating 'look-ahead' as in the well-defined example $\mathsf{S} = 0 : \mathsf{head}(\mathsf{tail}^2(\mathsf{S})) : \mathsf{S}$ from [12], is not included.

Now we are ready to define the concept of 'stream constant specification'.

Definition 2. Let $\mathcal{T} = \langle \Sigma, R\rangle$ be a finite stream TRS with a partition $\Sigma = \Sigma_d \uplus \Sigma_{sf} \uplus \Sigma_{sc} \uplus \{:\}$ of its signature and a partition $R = R_d \uplus R_{sf} \uplus R_{sc}$ of its set of rules. Then $\mathcal{T}$ (together with these partitions) is called a *pure stream constant specification (SCS)* if the following conditions hold:

(i) $\langle \Sigma_d \uplus \Sigma_{sf} \uplus \{:\}, R_d \uplus R_{sf}\rangle$ is an SFS.
(ii) $\Sigma_{sc} = \{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$ is a non-empty set of constant symbols, and $R_{sc} = \{\mathsf{M}_i \to \mathit{rhs}_{\mathsf{M}_i} \mid 1 \le i \le n, \mathit{rhs}_{\mathsf{M}_i} \in \mathit{Ter}(\Sigma)_s\}$. The rule $\rho_{\mathsf{M}_i} := \mathsf{M}_i \to \mathit{rhs}_{\mathsf{M}_i}$ is called *the defining rule for* M_i *in* $\mathcal{T}$.

Note that an SCS $\mathcal{T}$ is orthogonal as a consequence of (i) and (ii).

An SCS is called *productive* if every $\mathsf{M} \in \Sigma_{sc}$ has a stream of data terms as infinite normal form (an infinite *constructor normal form*). Note that orthogonality implies that infinite normal forms are unique.

Example 1. Let $\mathcal{T}_\mathsf{D} = \langle \Sigma_d \uplus \Sigma_{sf} \uplus \Sigma_{sc} \uplus \{:\}, R_d \uplus R_{sf} \uplus R_{sc} \rangle$ be the SCS with $\Sigma_d = \{\mathsf{s}, \mathsf{0}, \mathsf{a}\}$, $\Sigma_{sf} = \{\mathsf{tail}, \mathsf{even}, \mathsf{odd}, \mathsf{zip}, \mathsf{add}\}$, $\Sigma_{sc} = \{\mathsf{D}\}$, and R_{sc} consists of

$$\mathsf{D} \to 0 : 1 : 1 : \mathsf{zip}(\mathsf{add}(\mathsf{tail}(\mathsf{D}), \mathsf{tail}(\mathsf{tail}(\mathsf{D}))), \mathsf{even}(\mathsf{tail}(\mathsf{D}))),$$

R_{sf} consists of the rules

$$\mathsf{tail}(x : \sigma) \to \sigma \qquad \mathsf{even}(x : \sigma) \to x : \mathsf{odd}(\sigma) \qquad \mathsf{odd}(x : \sigma) \to \mathsf{even}(\sigma)$$
$$\mathsf{zip}(x : \sigma, \tau) \to x : \mathsf{zip}(\tau, \sigma) \qquad \mathsf{add}(x : \sigma, y : \tau) \to \mathsf{a}(x, y) : \mathsf{add}(\sigma, \tau)$$

and $R_d = \{\mathsf{a}(x, \mathsf{s}(y)) \to \mathsf{s}(\mathsf{a}(x, y)),\ \mathsf{a}(x, \mathsf{0}) \to x\}$. Note that D has the infinite constructor normal form $0:1:1:2:1:3:2:3:3:4:3:5:4:5:5:6:5:7:6:7:7:\ldots$, and hence is productive in $\mathcal{T}_\mathsf{D}$.

Example 2. Consider the rule $\mathsf{J} \to 0 : 1 : \mathsf{even}(\mathsf{J})$ together with Σ, R_d, R_{sf} as in Ex. 1. 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.

3 Modelling with Nets

We introduce nets as a means to model SCSs and to visualise the *flow* of stream elements. As our focus is on productivity of SCSs, we are interested in the *production* of such a net, that is, the number of stream elements produced by a net. Therefore, stream elements are abstracted from in favour of occurrences of the symbol $\bullet$, which we call *pebble*. The nets we study are called *pebbleflow nets*; they are inspired by interaction nets [10], and could be implemented in the framework of interaction nets with little effort.

First we give an operational description of pebbleflow nets, explaining what the components of nets are and the way how the components process pebbles. To ease manipulation of and reasoning about nets, we employ term representations. Term constructs corresponding to net components, as well as the rules governing the flow of pebbles through a net, are given on the fly. Their formal definitions are given in Subsec. 3.2. Finally, in Subsec. 3.3, we define a production preserving translation of pure stream specifications into rational nets.

We denote the set of *coinductive natural numbers* by $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$ and the *numerals* representing the elements of $\overline{\mathbb{N}}$ by $\underline{n} = \mathsf{s}^n(\mathsf{0})$ for $n \in \mathbb{N}$, and $\underline{\infty} = \mathsf{s}^\omega$.

3.1 Nets

Wires. The directed edges of a net, along which pebbles travel, are called *wires*. Wires 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

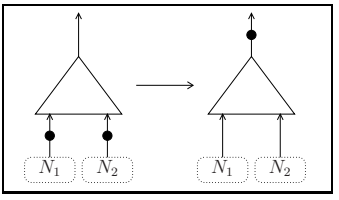

Fig. 1. $\triangle(\bullet(N_1), \bullet(N_2)) \to \bullet(\triangle(N_1, N_2))$

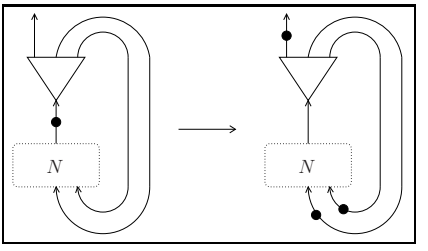

Fig. 2. $\mu x.\bullet(N(x)) \to \bullet(\mu x.N(\bullet(x)))$

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.

Meets. 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. 1. 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 in the part "Boxes and gates" 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.

Fans. 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. 2 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.

Sources. A *source* has an output port only, contains a number $k \in \overline{\mathbb{N}}$ of pebbles, and can fire if $k > 0$. In Sec. 4 we show how to reduce 'closed' nets to sources.

Boxes and Gates. 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. This consumption/production behaviour of the box is then also be expressed by the 'production function' $\beta_\sigma : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ of the box, see Fig. 5. 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. 3 and 4.

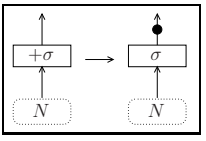

Fig. 3. $\mathsf{box}(+\sigma, N) \to \bullet(\mathsf{box}(\sigma, N))$

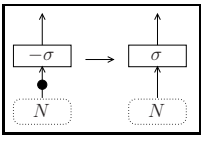

Fig. 4. $\mathsf{box}(-\sigma, \bullet(N)) \to \mathsf{box}(\sigma, N)$

Definition 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:

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

Further, we define the set $\pm^\omega_{rat} \subseteq \pm^\omega$ of *rational I/O sequences*. A sequence $\sigma \in \pm^\omega$ is called *rational* if there exist lists $\alpha, \gamma \in \{+,-\}^*$ such that $\sigma = \alpha\overline{\gamma}$, where γ is not the empty list and $\overline{\gamma}$ denotes the infinite sequence $\gamma\gamma\gamma\ldots$. The pair $\langle\alpha, \gamma\rangle$ is called a *rational representation* of σ.

To model stream functions of arbitrary arity, we introduce *gates*. Gates are compounded of meets and boxes, as depicted in Fig. 6. The precise construction of a gate corresponding to a given stream function is described in Subsec. 3.3.

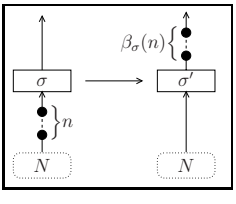

Fig. 5. $\mathsf{box}(\sigma, \bullet^n(N)) \to \bullet^{\beta_\sigma(n)}(\mathsf{box}(\sigma', N))$

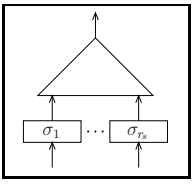

Fig. 6. A gate for modelling r_s-ary stream functions

Definition 4. The *production function* $\beta_\sigma : \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ of (a box containing) a sequence $\sigma \in \pm^\omega$ is corecursively defined, for all $n \in \overline{\mathbb{N}}$, by $\underline{\beta_\sigma(n)} := \beta(\sigma, \underline{n})$:

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

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

3.2 A Rewrite System for Pebbleflow

We define terms representing nets, and a rewrite system to model pebbleflow.

Definition 5. Let $\mathcal{V}$ be a set of variables. The set $\mathcal{N}$ of *terms for pebbleflow 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 $k \in \overline{\mathbb{N}}$, $x \in \mathcal{V}$, and $\sigma \in \pm^\omega$. Furthermore, the set $\mathcal{N}_{rat}$ of *terms for rational pebbleflow nets* is defined by the same inductive clauses, but now with the restriction $\sigma \in \pm^\omega_{rat}$.

The importance of identifying the subset 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 will be terminating for rational nets, and will enable us to determine the total production of a rational net.

The rules that govern pebbleflow are listed in Def. 6.

Definition 6. The *pebbleflow rewrite relation* $\to_{\mathsf{P}}$ is defined as the compatible closure of the union of the following rules:

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

The first four rewrite rules in the definition above are visualised in Figures 1, 2, 3, and 4, respectively. In rule (P2) the feedback of pebbles along the recursion wire(s) of the net N is accomplished by substituting $\bullet(x)$ for all free occurrences x of N. Observe that $\to_{\mathsf{P}}$ constitutes an orthogonal CRS [15], hence:

Theorem 1. *The relation* $\to_{\mathsf{P}}$ *is confluent.*

3.3 Translating Pure Stream Specifications

First 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' (sequence of guards encountered) and the 'consumption' of the rules applied, during the finite or eventually periodic dependency sequence on f.

Definition 7. 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 arity $\langle r_s, r_d \rangle$ the *translation* of f is a rational gate $[\mathsf{f}] : \mathcal{N}^{r_s} \to \mathcal{N}$ as defined by:

$$[\mathsf{f}](N_1, \ldots, N_{r_s}) = \triangle_{r_s}(\mathsf{box}([\mathsf{f}]_1, N_1), \ldots, \mathsf{box}([\mathsf{f}]_{r_s}, N_{r_s}))$$

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: $\mathsf{f}(\boldsymbol{x_1} : \sigma_1, \ldots, \boldsymbol{x_{r_s}} : \sigma_{r_s}, y_1, \ldots, y_{r_d}) \to t_1 : \ldots : t_{m_\mathsf{f}} : u$, where:

(a) $u \equiv \mathsf{g}(\sigma_{\pi_\mathsf{f}(1)}, \ldots, \sigma_{\pi_\mathsf{f}(r'_s)}, t'_1, \ldots, t'_{r'_d})$, then

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

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

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

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 8. 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 pebbleflow net is recursively defined by (α a set of stream constant symbols):

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

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

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

Example 3. Reconsider the SCS defined in Example 1. The translation of the stream function symbols $\mathsf{tail}, \mathsf{zip} \in \Sigma_{sf}$ is carried out as follows:

$$\begin{aligned} [\mathsf{tail}](N) &= \triangle_1(\mathsf{box}([\mathsf{tail}]_1, N)) \\ &= \mathsf{box}([\mathsf{tail}]_1, N) \\ [\mathsf{tail}]_1 &= -\overline{-+} \end{aligned} \qquad \begin{aligned} [\mathsf{zip}](N_1, N_2) &= \triangle_2(\mathsf{box}([\mathsf{zip}]_1, N_1), \mathsf{box}([\mathsf{zip}]_2, N_2)) \\ [\mathsf{zip}]_1 &= -+[\mathsf{zip}]_2 = -++[\mathsf{zip}]_1 = \overline{-++} \\ [\mathsf{zip}]_2 &= +[\mathsf{zip}]_1 = +-+[\mathsf{zip}]_2 = \overline{+-+} \end{aligned}$$

(Note that to obtain rational representations of the translated stream functions we use loop checking on top of Def. 7.) Then, the stream constant D is translated to the following pebbleflow net, depicted in Fig. 7:

$$[\mathsf{D}] = \mu D.\bullet(\bullet(\bullet([\mathsf{zip}]([\mathsf{add}]([\mathsf{tail}](D), [\mathsf{tail}]([\mathsf{tail}](D))), [\mathsf{even}]([\mathsf{tail}](D)))))) \,.$$

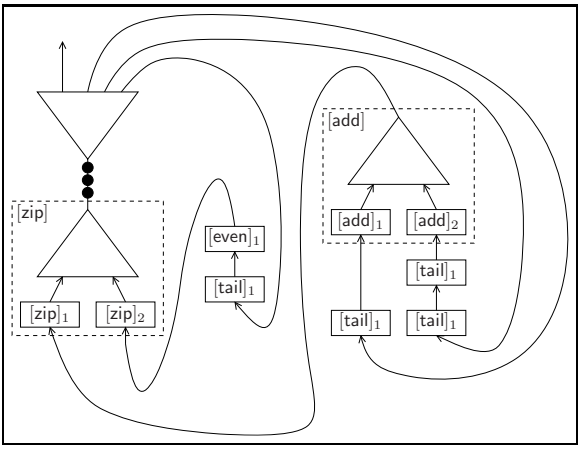

Fig. 7. The pebbleflow net $[\mathsf{D}]$ corresponding to the stream D

The theorem below is the basis of our decision algorithm. It states that the translation is 'production preserving', based on the following terminology: The *production* $\pi(N)$ of a pebbleflow net N is the supremum of the number of pebbles the net can 'produce': $\pi(N) := \mathsf{sup}\{n \in \mathbb{N} \mid N \twoheadrightarrow_{\mathsf{P}} \bullet^n(N')\}$, where $\twoheadrightarrow_{\mathsf{P}}$ denotes the reflexive–transitive closure of $\rightarrow_{\mathsf{P}}$. Likewise for an SCS $\mathcal{T} = \langle \Sigma, R \rangle$ the *production* $\pi_{\mathcal{T}}(t)$ of a term $t \in \mathit{Ter}(\Sigma)$ is the supremum of the number of data elements t can 'produce': $\pi_{\mathcal{T}}(t) := \mathsf{sup}\{n \in \mathbb{N} \mid t \twoheadrightarrow s_1 : \ldots : s_n : t'\}$.

Theorem 2. *Let $\mathcal{T}$ be a pure SCS. Then, $\pi([\mathsf{M}]) = \pi_{\mathcal{T}}(\mathsf{M})$ for all $\mathsf{M} \in \Sigma_{sc}$.*

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 9. We define *the net reduction relation $\rightarrow_{\mathsf{R}}$ on closed pebbleflow nets* by the compatible closure of the following rule schemata:

$$\bullet(N) \rightarrow \mathsf{box}((+\overline{-+}), N) \tag{R1}$$

$$\mathsf{box}(\sigma, \mathsf{box}(\tau, N)) \rightarrow \mathsf{box}((\sigma \cdot \tau), N) \tag{R2}$$

$$\mathsf{box}(\sigma, \triangle(N_1, N_2)) \rightarrow \triangle(\mathsf{box}(\sigma, N_1), \mathsf{box}(\sigma, N_2)) \tag{R3}$$

$$\mu x.\triangle(N_1, N_2) \rightarrow \triangle(\mu x.N_1, \mu x.N_2) \tag{R4}$$

$$\mu x.N \rightarrow N \qquad \text{if } x \notin \mathsf{FV}(N) \tag{R5}$$

$$\mu x.\mathsf{box}(\sigma, x) \rightarrow \mathsf{src}(\underline{\mathsf{fix}(\sigma)}) \tag{R6}$$

$$\triangle(\mathsf{src}(\underline{k_1}), \mathsf{src}(\underline{k_2})) \rightarrow \mathsf{src}(\underline{\mathsf{min}(k_1, k_2)}) \tag{R7}$$

$$\mathsf{box}(\sigma, \mathsf{src}(\underline{k})) \rightarrow \mathsf{src}(\underline{\beta_\sigma(k)}) \tag{R8}$$

$$\mu x.x \rightarrow \mathsf{src}(0) \tag{R9}$$

where $\sigma, \tau \in \pm^\omega$, $k, k_1, k_2 \in \overline{\mathbb{N}}$, and $\underline{\mathsf{min}(n, m)}$, $\underline{\beta_\sigma(k)}$, $\sigma \cdot \tau$ (see Def. 10) and $\underline{\mathsf{fix}(\sigma)}$ (see Def. 11) are term representations of operation results.

Definition 10. The operation *composition* $\cdot : \pm^\omega \times \pm^\omega \rightarrow \pm^\omega$, $\langle \sigma, \tau \rangle \mapsto \sigma \cdot \tau$ *of I/O sequences* is defined corecursively by the following equations:

$$(+\sigma) \cdot \tau = +(\sigma \cdot \tau) \qquad (-\sigma) \cdot (+\tau) = \sigma \cdot \tau \qquad (-\sigma) \cdot (-\tau) = -((-\sigma) \cdot \tau)$$

Composition of sequences $\sigma \cdot \tau \in \pm^\omega$ exhibits analogous properties as composition of functions over natural numbers: it is associative, but not commutative. Furthermore, for all $\sigma, \tau \in \pm^\omega, n \in \overline{\mathbb{N}}$ we have $\beta_{\sigma \cdot \tau}(n) = \beta_\sigma(\beta_\tau(n))$. Because we formalised the I/O behaviour of boxes by sequences and because we are interested in (dis)proving productivity, for the formalisation of the pebbleflow rewrite relation in Def. 6 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 11. 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:

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

For all $\sigma \in \pm^\omega$, we have $\beta_\sigma(\mathsf{fix}(\sigma)) = \mathsf{fix}(\sigma)$. Moreover, $\mathsf{fix}(\sigma)$ is the *least* fixed point of β_σ. Observe that $\beta_{\sigma\cdot\sigma\cdot\sigma\cdot\ldots} = \beta_\sigma(\beta_\sigma(\beta_\sigma(\ldots))) = \mathsf{fix}(\sigma)$. Therefore, the infinite self-composition $\mathsf{box}(\sigma, \mathsf{box}(\sigma, \mathsf{box}(\sigma, \ldots)))$ is 'production equivalent' to $\mathsf{src}(\mathsf{fix}(\sigma))$.

Lemma 1. *The net reduction relation* $\to_\mathsf{R}$ *is production preserving, that is,* $N \to_\mathsf{R} N'$ *implies* $\pi(N) = \pi(N')$*, for all nets* $N, N' \in \mathcal{N}$*. Furthermore,* $\to_\mathsf{R}$ *is terminating and every closed net normalises to a unique normal form, a source.*

Observe that net reduction employs infinitary rewriting for fixed point computation and composition (Def. 10 and 11). 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 2. *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 (Hint). Note that composition preserves rationality, that is, $\sigma \cdot \tau \in \pm^\omega_{rat}$ 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 $\sigma \cdot \tau$ can be computed in finite time. □

Theorem 3. *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. 8. It is easy to verify that $[\mathsf{M}]$ is in fact a rational net. Second, by the algorithm stated by Lem. 2, $[\mathsf{M}]$ is collapsed to a source $\mathsf{src}(n)$ with $n \in \overline{\mathbb{N}}$. By Thm. 2 it follows that $[\mathsf{M}]$ has the same production as M in $\mathcal{T}$, and by Lem. 1 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. □

5 Examples

We give three examples to show how our algorithm decides productivity of SCSs. First we recognise our running example (Ex. 1) to be productive. Next, we give a simple example of an SCS that is not productive. Finally, we illustrate that productivity is sensitive to the precise definitions of the stream functions used.

Example 4. We revisit Ex. 3 where we calculated the pebbleflow net $[\mathsf{D}]$ for D and show the last five steps of the reduction to $\to_{\mathsf{R}}$-normal form.

$$\begin{aligned}
[\mathsf{D}] &\to_{\mathsf{R}} \triangle(\triangle(\mu D.\mathsf{box}(+++-\overline{-++}, D), \mu D.\mathsf{box}(+++-\overline{-++}, \mathsf{box}(-\overline{-+}, D))), \mathsf{src}(\infty))\\
&\to_{\mathsf{R6}} \triangle(\triangle(\mathsf{src}(\infty), \mu D.\mathsf{box}(+++-\overline{-++}, \mathsf{box}(-\overline{-+}, D))), \mathsf{src}(\infty))\\
&\to_{\mathsf{R2}} \triangle(\triangle(\mathsf{src}(\infty), \mu D.\mathsf{box}(+++--\overline{-++}, D)), \mathsf{src}(\infty))\\
&\to_{\mathsf{R6}} \triangle(\triangle(\mathsf{src}(\infty), \mathsf{src}(\infty)), \mathsf{src}(\infty)) \to_{\mathsf{R7}} \triangle(\mathsf{src}(\infty), \mathsf{src}(\infty)) \to_{\mathsf{R7}} \mathsf{src}(\infty)\,.
\end{aligned}$$

Hence D is productive in the SCS of Ex. 1.

Example 5. For the definition of J from Ex. 2 we get:

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

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

Example 6. 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 where $\Sigma_d = \{0\}$, $\Sigma_{sf} = \{\mathsf{zip}, \mathsf{tail}, \mathsf{even}, \mathsf{odd}\}$, $\Sigma_{sc} = \{\mathsf{C}\}$, $R_d = \varnothing$, R_{sc} consists of:

$$\mathsf{C} \to 0 : \mathsf{zip}(\mathsf{C}, \mathsf{even}(\mathsf{tail}(\mathsf{C})))\,,$$

and R_{sf} consists of the rules:

$$\begin{aligned}
\mathsf{tail}(x:\sigma) &\to \sigma & \mathsf{zip}(x:\sigma, \tau) &\to x : \mathsf{zip}(\tau, \sigma)\\
\mathsf{even}(x:\sigma) &\to x : \mathsf{odd}(\sigma) & \mathsf{odd}(x:\sigma) &\to \mathsf{even}(\sigma)\,.
\end{aligned}$$

Then, we obtain the following translations:

$$\begin{aligned}
[\mathsf{zip}](N_1, N_2) &= \triangle_2(\mathsf{box}(\overline{-++}, N_1), \mathsf{box}(\overline{+-+}, N_2))\\
[\mathsf{even}](N) &= \mathsf{box}(\overline{-+-}, N)\\
[\mathsf{tail}](N) &= \mathsf{box}(-\overline{-+}, N)\\
[\mathsf{C}] &= \mu C.\bullet(\triangle(\mathsf{box}(\overline{-++}, C), \mathsf{box}(\overline{+-+}, \mathsf{box}(\overline{-+-}, \mathsf{box}(-\overline{-+}, C)))))\,.
\end{aligned}$$

Now by rewriting $[\mathsf{C}]$ 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:

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

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

$$[\mathsf{zip}^*](N_1, N_2) = \triangle_2(\mathsf{box}(\overline{-++}, N_1), \mathsf{box}(\overline{-++}, N_2))$$

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

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

Similarly, if we change the definition of even to $\mathsf{even}(x : y : \sigma) \to x : \mathsf{even}(\sigma)$, giving rise to the translation $[\mathsf{even}](N) = \mathsf{box}(\overline{--+}, N)$, then only the first two elements of C can be evaluated.

6 Conclusion and Ongoing Research

We have shown that productivity is decidable for stream definitions that belong to the format of SCSs. The class of SCSs contains definitions that cannot be recognised automatically to be productive by the methods of [16,12,2,5,14,1] (e.g. the stream constant definition in Ex. 1). These previous approaches established criteria for productivity that are not applicable for disproving productivity; furthermore, these methods are either applicable to general stream definitions, 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 definitions.

Note that we obtain decidability of productivity by restricting only the stream function definition part of a stream definition (formalised as an orthogonal TRS), while imposing no conditions on how the stream constant definition part makes use of the stream functions. The restriction to weakly guarded stream function definitions in SCSs is motivated by the wish to formulate an effectively recognisable format of stream definitions 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 definitions that are based on stream function specifications which can (quantitatively) faithfully be described by 'rational' I/O sequences. Finally, 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. 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 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 want to thank Clemens Kupke, Milad Niqui, Vincent van Oostrom, Femke van Raamsdonk, and Jan Rutten. Also, we would like to thank the anonymous referees for their encouraging comments.

References

1. Buchholz, W.: A term calculus for (co-)recursive definitions on streamlike data structures. Annals of Pure and Applied Logic 136(1-2), 75–90 (2005)
2. Coquand, Th.: Infinite Objects in Type Theory. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 62–78. Springer, Heidelberg (1994)
3. Dijkstra, E.W.: On the productivity of recursive definitions, EWD749 (1980)
4. Endrullis, J., Grabmayer, C., Hendriks, D.: Productivity of stream definitions. Technical report, Vrije Universiteit Amsterdam (2007), available via `http://infinity.few.vu.nl/productivity/`
5. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: POPL '96, pp. 410–423 (1996)
6. Kahn, G.: The semantics of a simple language for parallel programming. Information Processing, 471–475 (1974)
7. Kennaway, R., Klop, J.W., Sleep, M.R., de Vries, F.-J.: Transfinite reductions in orthogonal term rewriting systems. Inf. and Comput. 119(1), 18–38 (1995)
8. Kennaway, R., Klop, J.W., Sleep, M.R., de Vries, F.-J.: Infinitary lambda calculus. TCS 175(1), 93–125 (1997)
9. Klop, J.W., de Vrijer, R.: Infinitary normalization. In: We Will Show Them: Essays in Honour of Dov Gabbay (2). College Publications, pp. 169–192 (2005), Item 95 at `http://web.mac.com/janwillemklop/iWeb/Site/Bibliography.html`
10. Lafont, Y.: Interaction nets. In: POPL '90, pp. 95–108. ACM Press, New York (1990)
11. Rutten, J.J.M.M.: Behavioural differential equations: a coinductive calculus of streams, automata, and power series. TCS 308(1-3), 1–53 (2003)
12. Sijtsma, B.A.: On the productivity of recursive list definitions. ACM Transactions on Programming Languages and Systems 11(4), 633–649 (1989)
13. Tait, W.W.: Intentional interpretations of functionals of finite type I. Journal of Symbolic Logic 32(2) (1967)
14. Telford, A., Turner, D.: Ensuring the Productivity of infinite structures. Technical Report 14-97, The Computing Laboratory, Univ. of Kent at Canterbury (1997)
15. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
16. Wadge, W.W.: An extensional treatment of dataflow deadlock. TCS 13, 3–15 (1981)