


# Data-Oblivious Stream Productivity

View metadata, citation and similar papers at [core.ac.uk](http://core.ac.uk)

brought to you by  CORE

provided by DSpace at VU

De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

joerg@few.vu.nl, diem@cs.vu.nl

<sup>2</sup> Universiteit Utrecht, Department of Philosophy

Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

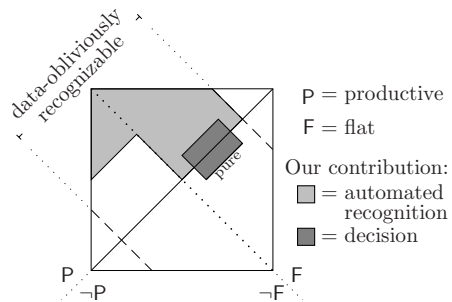
clemens@phil.uu.nl

**Abstract.** We are concerned with demonstrating productivity of specifications of infinite streams of data, based on orthogonal rewrite rules. In general, this property is undecidable, but for restricted formats computable sufficient conditions can be obtained. The usual analysis, also adopted here, disregards the identity of data, thus leading to approaches that we call data-oblivious. We present a method that is provably optimal among all such data-oblivious approaches. This means that in order to improve on our algorithm one has to proceed in a data-aware fashion.<sup>1</sup>

## 1 Introduction

For programming with infinite structures, productivity is what termination is for programming with finite structures. Productivity captures the intuitive notion of unlimited progress, of ‘working’ programs producing defined values indefinitely. In functional languages, usage of infinite structures is common practice. For the correctness of programs dealing with such structures one must guarantee that every finite part of the infinite structure can be evaluated, that is, the specification of the infinite structure must be productive.

We investigate this notion for stream specifications, formalized as orthogonal term rewriting systems. Common to all previous approaches for recognizing productivity is a quantitative analysis that abstracts away from the concrete values of stream elements. We formalize this by a notion of ‘data-oblivious’ rewriting, and introduce the concept of data-oblivious productivity. Data-oblivious (non-)productivity implies



**Fig. 1.** Map of stream specifications

<sup>1</sup> This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.502.

(non-)productivity, but neither of the converse implications holds. Fig. 1 shows a Venn diagram of stream specifications, highlighting the subset of ‘data-obliviously recognizable’ specifications where (non-)productivity can be recognized by a data-oblivious analysis.

We identify two syntactical classes of stream specifications: ‘flat’ and ‘pure’ specifications, see the description below. For the first we devise a decision algorithm for data-oblivious (d-o) productivity. This gives rise to a computable, d-o optimal, criterion for productivity: every flat stream specification that can be established to be productive by whatever d-o argument is recognized as productive by this criterion (see Fig. 1). For the subclass of pure specifications, we establish that d-o productivity coincides with productivity, and thereby obtain a decision algorithm for productivity of this class. Additionally, we extend our criterion beyond the class of flat stream specifications, allowing for ‘friendly nesting’ in the specification of stream functions; here d-o optimality is not preserved.

In defining the different formats of stream specifications, we distinguish between rules for stream constants, and rules for stream functions. Only the latter are subjected to syntactic restrictions. In flat stream specifications the defining rules for the stream functions do not have nesting of stream function symbols; however, in defining rules for stream constants nesting of stream function symbols *is* allowed. This format makes use of exhaustive pattern matching on data to define stream functions, allowing for multiple defining rules for an individual stream function symbol. Since the quantitative consumption/production behaviour of a symbol  $f$  might differ among its defining rules, in a d-o analysis one has to settle for the use of lower bounds when trying to recognize productivity. If for all stream function symbols  $f$  in a flat specification  $\mathcal{T}$  the defining rules for  $f$  coincide, disregarding the identity of data-elements, then  $\mathcal{T}$  is called pure.

Our decision algorithm for d-o productivity determines the tight d-o lower bound on the production behaviour of every stream function, and uses these bounds to calculate the d-o production of stream constants. We briefly explain both aspects. Consider the stream specification  $A \rightarrow 0 : f(A)$  together with the rules  $f(0 : \sigma) \rightarrow 1 : 0 : 1 : f(\sigma)$ , and  $f(1 : \sigma) \rightarrow 0 : f(\sigma)$ , defining the stream  $0 : 1 : 0 : 1 : \dots$  of alternating bits. The tight d-o lower bound for  $f$  is the function  $id: n \mapsto n$ . Further note that  $suc: n \mapsto n + 1$  captures the quantitative behaviour of the function prepending a data element to a stream term. Therefore the d-o production of  $A$  can be computed as  $\text{lfp}(suc \circ id) = \infty$ , where  $\text{lfp}(f)$  is the least fixed point of  $f : \overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}$  and  $\overline{\mathbb{N}} := \mathbb{N} \cup \{\infty\}$ ; hence  $A$  is productive. As a comparison, only a ‘data-aware’ approach is able to establish productivity of  $B \rightarrow 0 : g(B)$  with  $g(0 : \sigma) \rightarrow 1 : 0 : g(\sigma)$ , and  $g(1 : \sigma) \rightarrow g(\sigma)$ . The d-o lower bound of  $g$  is  $n \mapsto 0$ , due to the latter rule. This makes it impossible for any conceivable d-o approach to recognize productivity of  $B$ .

We obtain the following results:

- (i) For the class of flat stream specifications we give a computable, d-o optimal, sufficient condition for productivity.
- (ii) We show decidability of productivity for the class of pure stream specifications, an extension of the format in [3].

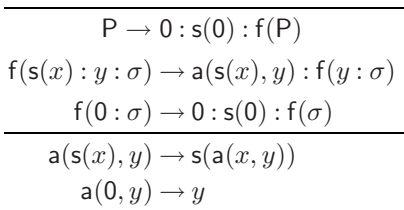
- (iii) Disregarding d-o optimality, we extend (i) to the bigger class of friendly nesting stream specifications.
- (iv) A tool automating (i), (ii) and (iii), which can be downloaded from, and used via a web interface at: <http://infinity.few.vu.nl/productivity>.

*Related work.* Previous approaches [6,4,7,1] employed d-o reasoning (without using this name for it) to find sufficient criteria ensuring productivity, but did not aim at optimality. The d-o production behaviour of a stream function  $f$  is bounded from below by a ‘modulus of production’  $\nu_f : \mathbb{N}^k \rightarrow \mathbb{N}$  with the property that the first  $\nu_f(n_1, \dots, n_k)$  elements of  $f(t_1, \dots, t_k)$  can be computed whenever the first  $n_i$  elements of  $t_i$  are defined. Sijtsma develops an approach allowing arbitrary production moduli  $\nu : \mathbb{N}^k \rightarrow \mathbb{N}$ , which, while providing an adequate mathematical description, are less amenable to automation. Telford and Turner [7] employ production moduli of the form  $\nu(n) = n + a$  with  $a \in \mathbb{Z}$ . Hughes, Pareto and Sabry [4] use  $\nu(n) = \max\{c \cdot x + d \mid x \in \mathbb{N}, n \geq a \cdot x + b\} \cup \{0\}$  with  $a, b, c, d \in \mathbb{N}$ . Both classes of production moduli are strictly contained in the class of ‘periodically increasing’ functions which we employ in our analysis. We show that the set of d-o lower bounds of flat stream function specifications is exactly the set of periodically increasing functions. Buchholz [1] presents a type system for productivity, using unrestricted production moduli. For a restricted subclass he gives an automatable method for ensuring productivity, but this excludes the use of stream functions with a negative effect like `odd` defined by  $\text{odd}(x:y:\sigma) \rightarrow y:\text{odd}(\sigma)$  with a (periodically increasing) modulus  $\nu_{\text{odd}}(n) = \lfloor \frac{n}{2} \rfloor$ .

*Overview.* In Sec. 2 we define the notion of stream specification, and the syntactic format of flat and pure specifications. In Sec. 3 we formalize the notion of d-o rewriting. In Sec. 4 we introduce a production calculus as a means to compute the production of the data-abstracted stream specifications, based on the set of periodically increasing functions. A translation of stream specifications into production terms is defined in Sec. 5. Our main results, mentioned above, are collected in Sec. 6. We conclude and discuss some future research topics in Sec. 7.

## 2 Stream Specifications

We introduce the notion of stream specification. An example is given in Fig. 2, a productive specification of Pascal’s triangle where the rows are separated by



**Fig. 2.** A flat stream specification

zeros. Indeed, evaluating this specification, we get:  $P \rightsquigarrow 0 : 1 : 0 : 1 : 1 : 0 : 1 : 2 : 1 : 0 : 1 : 3 : 3 : 1 : \dots$

We define stream specifications to consist of a *stream layer* (top) where stream constants and functions are specified, and a *data layer* (bottom) such that the stream layer may use symbols of the data layer, but not vice-versa. Thus, the data layer is a term

rewriting system on its own. In order to abstract from the termination problem when investigating productivity, we require the data layer to be strongly normalizing. Let us explain the reason for this hierarchical setup. Stream dependent data symbols (whose defining rules do contain stream symbols), like  $\text{head}(x : \sigma) \rightarrow x$ , might cause the output of undefined data terms. Let  $\sigma(n) := \text{head}(\text{tail}^n(\sigma))$ , and consider the following bitstream specifications:

$$S \rightarrow 0 : S(2) : S \qquad T \rightarrow 0 : T(3) : T ,$$

taken from [6]. Here we have that  $S(n) \rightarrow^* S(n-2)$  for all  $n \geq 2$ , and  $S(1) \rightarrow^* S(2)$ , and hence  $S \twoheadrightarrow 0 : 0 : 0 : \dots$ , producing the infinite stream of zeros. On the other hand, the evaluation of each data term  $T(2n+1)$  eventually ends up in the loop  $T(3) \rightarrow^* T(1) \rightarrow^* T(3) \rightarrow^* \dots$ . Hence we have that  $T \twoheadrightarrow 0 : ? : 0 : ? : \dots$  (where  $?$  stands for ‘undefined’) and  $T$  is not productive.

Such examples, where the evaluation of stream elements needs to be delayed to wait for ‘future information’, can only be productive using a lazy evaluation strategy like in the programming language Haskell. Productivity of specifications like these is adequately analyzed using the concept of ‘set productivity’ in [6]. A natural first step is to study its proper subclass called ‘segment productivity’, where well-definedness of one element requires well-definedness of all previous ones. The restriction to this subclass is achieved by disallowing stream dependent data functions. While conceptually more general, in practice stream dependent data functions usually can be replaced by pattern matching;  $\text{add}(\sigma, \tau) \rightarrow (\text{head}(\sigma) + \text{head}(\tau)) : \text{add}(\text{tail}(\sigma), \text{tail}(\tau))$ , for example, can be replaced by the better readable  $\text{add}(x : \sigma, y : \tau) \rightarrow (x + y) : \text{add}(\sigma, \tau)$ .

Stream specifications are formalized as many-sorted, orthogonal, constructor term rewriting systems [8]. We distinguish between *stream terms* and *data terms*. For the sake of simplicity we consider only one sort  $S$  for stream terms and one sort  $D$  for data terms. Without any complication, our results extend to stream specifications with multiple sorts for data terms and for stream terms.

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  $\Sigma$  is a  $U$ -sorted set of function symbols  $f$ , each equipped with an arity  $\text{ar}(f) = \langle u_1 \dots u_n, u \rangle \in U^* \times U$  where  $u$  is the sort of  $f$ ; we write  $u_1 \times \dots \times u_n \rightarrow u$  for  $\langle u_1 \dots u_n, u \rangle$ . Let  $X$  be a  $U$ -sorted set of variables. The  $U$ -sorted set of terms  $\text{Term}(\Sigma, X)$  is inductively defined by: for all  $u \in U$ ,  $X_u \subseteq \text{Term}(\Sigma, X)_u$ , and  $f(t_1, \dots, t_n) \in \text{Term}(\Sigma, X)_u$  if  $f \in \Sigma$ ,  $\text{ar}(f) = u_1 \times \dots \times u_n \rightarrow u$ , and  $t_i \in \text{Term}(\Sigma, X)_{u_i}$ .  $\text{Term}_\infty(\Sigma, X)$  denotes the set of (possibly) infinite terms over  $\Sigma$  and  $X$  (see [8]). Usually we keep the set of variables implicit and write  $\text{Term}(\Sigma)$  and  $\text{Term}_\infty(\Sigma)$ . A  $U$ -sorted term rewriting system (TRS) is a pair  $\langle \Sigma, R \rangle$  consisting of a  $U$ -sorted signature  $\Sigma$  and a  $U$ -sorted set  $R$  of rules that satisfy well-sortedness, for all  $u \in U$ :  $R_u \subseteq \text{Term}(\Sigma, X)_u \times \text{Term}(\Sigma, X)_u$ , as well as the standard TRS requirements.

Let  $\mathcal{T} = \langle \Sigma, R \rangle$  be a  $U$ -sorted TRS. For a term  $t \in \text{Term}(\Sigma)_u$  where  $u \in U$  we denote the root symbol of  $t$  by  $\text{root}(t)$ . We say that two occurrences of symbols in a term are *nested* if the position [8, p.29] of one is a prefix of the position of the other. We define  $\mathcal{D}(\Sigma) := \{\text{root}(l) \mid l \rightarrow r \in R\}$ , the set of *defined symbols*, and

$\mathcal{C}(\Sigma) := \Sigma \setminus \mathcal{D}(\Sigma)$ , the set of *constructor symbols*. Then  $\mathcal{T}$  is called a *constructor TRS* if for every rewrite rule  $\rho \in R$ , the left-hand side is of the form  $f(t_1, \dots, t_n)$  with  $t_i \in \text{Term}(\mathcal{C}(\Sigma))$ ; then  $\rho$  is called a *defining rule for  $f$* . We call  $\mathcal{T}$  *exhaustive for  $f \in \Sigma$*  if every term  $f(t_1, \dots, t_n)$  with (possibly infinite) closed constructor terms  $t_i$  is a redex. Note that, stream constructor terms are inherently infinite.

A *stream TRS* is a finite  $\{S, D\}$ -sorted, orthogonal, constructor TRS  $\langle \Sigma, R \rangle$  such that  $\cdot' \in \Sigma_S$ , the *stream constructor symbol*, with arity  $D \times S \rightarrow S$  is the single constructor symbol in  $\Sigma_S$ . Elements of  $\Sigma_D$  and  $\Sigma_S$  are called the *data symbols* and the *stream symbols*, respectively. We let  $\Sigma_{\bar{S}} := \Sigma_S \setminus \{\cdot'\}$ , and, for all  $f \in \Sigma_{\bar{S}}$ , we assume, without loss of generality, that the stream arguments are in front:  $ar(f) \in S^{ar_s(f)} \times D^{ar_d(f)} \rightarrow S$ , where  $ar_s(f)$  and  $ar_d(f) \in \mathbb{N}$  are called the *stream arity* and the *data arity* of  $f$ , respectively. By  $\Sigma_{scon}$  we denote the set of symbols in  $\Sigma_{\bar{S}}$  with stream arity 0, called the *stream constant symbols*, and  $\Sigma_{sfun} := \Sigma_{\bar{S}} \setminus \Sigma_{scon}$  the set of symbols in  $\Sigma_{\bar{S}}$  with stream arity unequal to 0, called the *stream function symbols*. Note that stream constants may have a data arity  $> 0$  as for example in:  $\text{natsFrom}(n) \rightarrow n : \text{natsFrom}(s(n))$ . Finally, by  $R_{scon}$  we mean the defining rules for the symbols in  $\Sigma_{scon}$ .

**Definition 2.1.** A *stream specification*  $\mathcal{T}$  is a stream TRS  $\mathcal{T} = \langle \Sigma, R \rangle$  such that the following conditions hold:

- (i) There is a designated symbol  $M_0 \in \Sigma_{scon}$  with  $ar_d(M_0) = 0$ , the *root of  $\mathcal{T}$* .
- (ii)  $\langle \Sigma_D, R_D \rangle$  is a terminating,  $D$ -sorted TRS;  $R_D$  is called the *data layer of  $\mathcal{T}$* .
- (iii)  $\mathcal{T}$  is exhaustive (for all defined symbols in  $\Sigma = \Sigma_S \uplus \Sigma_D$ ).

Note that Def. 2.1 indeed imposes a hierarchical setup; in particular, stream dependent data functions are excluded by item (ii). Exhaustivity for  $\Sigma_D$  together with strong normalization of  $R_D$  guarantees that closed data terms rewrite to constructor normal forms, a property known as sufficient completeness [5].

We are interested in productivity of recursive stream specifications that make use of a library of ‘manageable’ stream functions. By this we mean a class of stream functions defined by a syntactic format with the property that their d-o lower bounds are computable and contained in a set of production moduli that is effectively closed under composition, pointwise infimum and where least fixed points can be computed. As such a format we define the class of flat stream specifications (Def. 2.2) for which d-o lower bounds are precisely the set of ‘periodically increasing’ functions (see Sec. 4). Thus only the stream function rules are subject to syntactic restrictions. No condition other than well-sortedness is imposed on the defining rules of stream constant symbols.

In the sequel let  $\mathcal{T} = \langle \Sigma, R \rangle$  be a stream specification. We define the relation  $\rightsquigarrow$  on rules in  $R_S$ : for all  $\rho_1, \rho_2 \in R_S$ ,  $\rho_1 \rightsquigarrow \rho_2$  ( $\rho_1$  *depends on*  $\rho_2$ ) holds if and only if  $\rho_2$  is the defining rule of a stream function symbol on the right-hand side of  $\rho_1$ . Furthermore, for a binary relation  $\rightarrow \subseteq A \times A$  on a set  $A$  we define  $(a \rightarrow) := \{b \in A \mid a \rightarrow b\}$  for all  $a \in A$ , and we denote by  $\rightarrow^+$  and  $\rightarrow^*$  the *transitive closure* and the *reflexive-transitive closure* of  $\rightarrow$ , respectively.

**Definition 2.2.** A rule  $\rho \in R_S$  is called *nesting* if its right-hand side contains nested occurrences of stream symbols from  $\Sigma_S^-$ . We use  $R_{nest}$  to denote the subset of nesting rules of  $R$  and define  $R_{\neg nest} := R_S \setminus R_{nest}$ , the set of *non-nesting rules*.

A rule  $\rho \in R_S$  is called *flat* if all rules in  $(\rho \rightsquigarrow^*)$  are non-nesting. A symbol  $f \in \Sigma_S^-$  is called *flat* if all defining rules of  $f$  are flat; the set of flat symbols is denoted  $\Sigma_{flat}$ . A stream specification  $\mathcal{T}$  is called *flat* if  $\Sigma_S^- \subseteq \Sigma_{flat} \cup \Sigma_{scon}$ , that is, all symbols in  $\Sigma_S^-$  are either flat or stream constant symbols.

See Fig. 2 and Ex. 5.5 for examples of flat stream specifications.

As the basis of d-o rewriting (see Def. 3.2) we define the data abstraction of terms as the results of replacing all data-subterms by the symbol  $\bullet$ .

**Definition 2.3.** Let  $\langle \Sigma \rangle := \{\bullet\} \uplus \Sigma_S$ . For stream terms  $s \in Term(\Sigma)_S$ , the *data abstraction*  $\langle s \rangle \in Term(\langle \Sigma \rangle)_S$  is defined by:

$$\langle \sigma \rangle = \sigma \quad \langle u : s \rangle = \bullet : \langle s \rangle \quad \langle f(s_1, \dots, s_n, u_1, \dots, u_m) \rangle = f(\langle s_1 \rangle, \dots, \langle s_n \rangle, \bullet, \dots, \bullet).$$

Based on this definition of data abstracted terms, we define the class of pure stream specifications, an extension of the equally named class in [3].

**Definition 2.4.** A stream specification  $\mathcal{T}$  is called *pure* if it is flat and if for every symbol  $f \in \Sigma_S^-$  the data abstractions  $\langle \ell \rangle \rightarrow \langle r \rangle$  of the defining rules  $\ell \rightarrow r$  of  $f$  coincide (modulo renaming of variables).

See Ex. 5.4 for an example of a pure stream function specification. Def. 2.4 generalizes the specifications called ‘pure’ in [3] in four ways concerning the defining rules of stream functions: First, the requirement of right-linearity of stream variables is dropped, allowing for rules like  $f(\sigma) \rightarrow g(\sigma, \sigma)$ . Second, ‘additional supply’ to the stream arguments is allowed. For instance, in a rule like  $\text{diff}(x : y : \sigma) \rightarrow \text{xor}(x, y) : \text{diff}(y : \sigma)$ , the variable  $y$  is ‘supplied’ to the recursive call of  $\text{diff}$ . Third, the use of non-productive stream functions is allowed now, relaxing an earlier requirement of [3] on stream function symbols to be ‘weakly guarded’, see Def. 5.1. Finally, defining rules for stream function symbols may use a restricted form of pattern matching as long as, for every stream function  $f$ , the d-o consumption/production behaviour (see Sec. 3) of all defining rules for  $f$  is the same.

**Definition 2.5.** A rule  $\rho \in R_S$  is called *friendly* if for all rules  $\gamma \in (\rho \rightsquigarrow^*)$  we have: (1)  $\gamma$  consumes in each argument at most one stream element, and (2) it produces at least one. The set of *friendly nesting rules*  $R_{fnest}$  is the largest extension of the set of friendly rules by non-nesting rules from  $R_S$  that is closed under  $\rightsquigarrow$ . A symbol  $f \in \Sigma_S^-$  is *friendly nesting* if all defining rules of  $f$  are friendly nesting. A stream specification  $\mathcal{T}$  is called *friendly nesting* if  $\Sigma_S^- \subseteq \Sigma_{fnest} \cup \Sigma_{scon}$ , that is, all symbols in  $\Sigma_S^-$  are either friendly nesting or stream constant symbols.

Note that, in particular, every flat stream specification is friendly nesting.

*Example 2.6.* The rules  $X \rightarrow 0 : f(X)$  and  $f(x : \sigma) \rightarrow x : f(\sigma)$  form a friendly nesting stream specification with an empty data layer.

**Definition 2.7.** Let  $\mathcal{A} = \langle \text{Term}(\Sigma)_S, \rightarrow \rangle$  be an abstract reduction system (ARS) on the set of terms over a stream TRS signature  $\Sigma$ . The *production function*  $\Pi_{\mathcal{A}} : \text{Term}(\Sigma)_S \rightarrow \overline{\mathbb{N}}$  of  $\mathcal{A}$  is defined for all  $s \in \text{Term}(\Sigma)_S$  by:

$$\Pi_{\mathcal{A}}(s) := \sup \{ n \in \mathbb{N} \mid s \rightarrow_{\mathcal{A}}^* u_1 : \dots : u_n : t \}.$$

We call  $\mathcal{A}$  *productive for a stream term*  $s$  if  $\Pi_{\mathcal{A}}(s) = \infty$ . A stream specification  $\mathcal{T}$  is called *productive* if  $\mathcal{T}$  is productive for its root  $M_0$ .

Note that in a stream specification  $\mathcal{T}$  it holds (since  $\mathcal{T}$  is an orthogonal rewriting system) that if  $\mathcal{T}$  is productive for a term  $s$ , then  $s$  rewrites in  $\mathcal{T}$  to an infinite constructor term  $u_1 : u_2 : u_3 : \dots$  as its unique infinite normal form.

### 3 Data-Oblivious Analysis

We formalize the notion of d-o rewriting and introduce the concept of d-o productivity. The idea is a quantitative reasoning where all knowledge about the concrete values of data elements during an evaluation sequence is ignored. For example, consider the following stream specification:

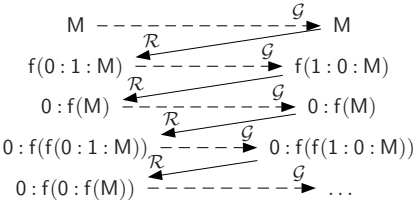
$$M \rightarrow f(0 : 1 : M) \quad (1) \quad f(0 : x : \sigma) \rightarrow 0 : 1 : f(\sigma) \quad (2) \quad f(1 : x : \sigma) \rightarrow x : f(\sigma)$$

The specification of  $M$  is productive:  $M \rightarrow^2 0 : 1 : f(M) \rightarrow^3 0 : 1 : 0 : 1 : f(f(M)) \rightarrow^* \dots$ . During the rewrite sequence (2) is never applied. Disregarding the identity of data, however, (2) becomes applicable and allows for the rewrite sequence:

$$M \rightarrow f(\bullet : \bullet : M) \xrightarrow{(2)} \bullet : f(M) \xrightarrow{*} \bullet : f(\bullet : f(\bullet : f(\dots))) ,$$

producing only one element. Hence from the perspective of a data-oblivious analysis there exists a rewrite sequence starting at  $M$  that converges to an infinite normal form which has only a stream prefix of length one. In terminology to be introduced in Def. 3.2 we will say that  $M$  is not ‘d-o productive’.

D-o term rewriting can be thought of as a two-player game between a *rewrite player*  $\mathcal{R}$  which performs the usual term rewriting, and an *opponent*  $\mathcal{G}$  which before every rewrite step is allowed to arbitrarily exchange data elements for (sort-respecting) data terms in constructor normal form. The opponent can either handicap or support the rewrite player. Respectively, the d-o lower bound on the production of a stream term  $s$  is the infimum of the production of  $s$  with respect to all possible strategies for the opponent  $\mathcal{G}$ .



**Fig. 3.** Data-oblivious rewriting

Fig. 3 depicts d-o rewriting of the above stream specification  $M$ ; by exchanging data elements, the opponent  $\mathcal{G}$  enforces the application of (2). The opponent can be modelled by an operation on stream terms, a function from stream terms to stream terms:  $\text{Term}(\Sigma)_S \rightarrow \text{Term}(\Sigma)_S$ . For our purposes it will be sufficient to consider strategies for  $\mathcal{G}$  with

the property that  $\mathcal{G}(s)$  is invariant under exchange of data elements in  $s$  for all terms  $s$  (see Prop. 3.4 below for a formal statement).

**Definition 3.1.** Let  $\mathcal{T} = \langle \Sigma, R \rangle$  be a stream specification. A *data-exchange function on  $\mathcal{T}$*  is a function  $\mathcal{G} : \text{Term}(\Sigma)_S \rightarrow \text{Term}(\Sigma)_S$  such that  $\llbracket \mathcal{G}(r) \rrbracket = \llbracket r \rrbracket$  for all  $r \in \text{Term}(\Sigma)_S$ , and  $\mathcal{G}(r)$  is in closed data-constructor normal form.

**Definition 3.2.** We define the ARS  $\mathcal{A}_{\mathcal{T}, \mathcal{G}} \subseteq \text{Term}(\Sigma)_S \times \text{Term}(\Sigma)_S$  for every data-exchange function  $\mathcal{G}$ , as follows:

$$\mathcal{A}_{\mathcal{T}, \mathcal{G}} := \{ \langle s, t \rangle \mid s, t \in \text{Term}(\Sigma), \mathcal{G}(s) \rightarrow_{\mathcal{T}} t \}.$$

Thus the steps  $s \rightarrow_{\mathcal{A}_{\mathcal{T}, \mathcal{G}}} t$  in  $\mathcal{A}_{\mathcal{T}, \mathcal{G}}$  are those of the form  $s \mapsto \mathcal{G}(s) \rightarrow_{\mathcal{T}} t$ .

The *d-o lower bound*  $\underline{do}_{\mathcal{T}}(s)$  on the production of a stream term  $s \in \text{Term}(\Sigma)_S$  is defined as follows:

$$\underline{do}_{\mathcal{T}}(s) := \inf \{ \Pi_{\mathcal{A}_{\mathcal{T}, \mathcal{G}}}(s) \mid \mathcal{G} \text{ a data-exchange function on } \mathcal{T} \}. \quad (*)$$

A stream specification  $\mathcal{T}$  is *d-o productive* if  $\underline{do}_{\mathcal{T}}(\mathbb{M}_0) = \infty$  holds.

**Proposition 3.3.** For  $\mathcal{T} = \langle \Sigma, R \rangle$  a stream specification and  $s \in \text{Term}(\Sigma)_S$ :

$$\underline{do}_{\mathcal{T}}(s) \leq \Pi_{\mathcal{T}}(s).$$

Hence *d-o productivity implies productivity*.

**Proposition 3.4.** The definition of the *d-o lower bound*  $\underline{do}_{\mathcal{T}}(s)$  of a stream term  $s$  in a stream specification  $\mathcal{T}$  in Def. 3.2 does not change if the quantification in  $(*)$  is restricted to data-exchange functions  $\mathcal{G}$  that factor as follows:

$$\mathcal{G} : \text{Term}(\Sigma) \xrightarrow{\langle \cdot \rangle} \text{Term}(\llbracket \Sigma \rrbracket) \xrightarrow{\mathcal{G}_{\bullet}} \text{Term}(\Sigma) \quad (\text{for some function } \mathcal{G}_{\bullet}) \quad (\dagger)$$

(data-exchange functions that are invariant under exchange of data elements).

*Proof (Sketch).* It suffices to prove that, for every term  $s \in \text{Term}(\Sigma)_S$ , and for every data-exchange function  $\mathcal{G}$  on  $\mathcal{T}$ , there exists a data-exchange function  $\mathcal{G}'$  on  $\mathcal{T}$  of the form  $(\dagger)$  such that  $\Pi_{\mathcal{A}_{\mathcal{T}, \mathcal{G}'}}(s) \leq \Pi_{\mathcal{A}_{\mathcal{T}, \mathcal{G}}}(s)$ . This can be shown by adapting  $\mathcal{G}$  in an infinite breadth-first traversal over  $\mathcal{R}(s)$ , the reduction graph of  $s$  in  $\mathcal{A}_{\mathcal{T}, \mathcal{G}}$ , thereby defining  $\mathcal{G}'$  as follows: if for a currently traversed term  $s$  there exists a previously traversed term  $s_0$  with  $\llbracket s_0 \rrbracket = \llbracket s \rrbracket$  and  $\mathcal{G}'(s_0) \neq \mathcal{G}(s)$ , then let  $\mathcal{G}'(s) := \mathcal{G}'(s_0)$ , otherwise let  $\mathcal{G}'(s) := \mathcal{G}(s)$ . Then the set of terms of the reduction graph  $\mathcal{R}'(s)$  of  $s$  in  $\mathcal{A}_{\mathcal{T}, \mathcal{G}'}$  is a subset of the terms in  $\mathcal{R}(s)$ .  $\square$

Let  $\mathcal{T}$  be a stream definition. As an immediate consequence of this proposition we obtain that, for all stream terms  $s_1, s_2 \in \text{Term}(\Sigma)$  in  $\mathcal{T}$ ,  $\underline{do}_{\mathcal{T}}(s_1) = \underline{do}_{\mathcal{T}}(s_2)$  holds whenever  $\llbracket s_1 \rrbracket = \llbracket s_2 \rrbracket$ . This fact allows to define d-o lower bounds directly on the data-abstractions of terms: For every term  $s \in \text{Term}(\llbracket \Sigma \rrbracket)$ , we let  $\underline{do}_{\mathcal{T}}(\llbracket s \rrbracket) := \underline{do}_{\mathcal{T}}(s)$  for an arbitrarily chosen  $s \in \text{Term}(\Sigma)_S$ . In order to reason about d-o productivity of stream constants (see Sec. 6), we now also introduce lower bounds on the d-o consumption/production behaviour of stream functions.



**Definition 3.5.** Let  $\mathcal{T} = \langle \Sigma, R \rangle$  be a stream specification,  $\mathbf{g} \in \Sigma_{\bar{S}}^{-}$ ,  $k = ar_s(\mathbf{g})$ , and  $\ell = ar_d(\mathbf{g})$ . The  $d$ -o lower bound  $\underline{do}_{\mathcal{T}}(\mathbf{g}) : \mathbb{N}^k \rightarrow \overline{\mathbb{N}}$  of  $\mathbf{g}$  is:

$$\underline{do}_{\mathcal{T}}(\mathbf{g})(n_1, \dots, n_k) := \underline{do}_{\mathcal{T}}(\mathbf{g}(\underbrace{(\bullet^{n_1} : \sigma_1), \dots, (\bullet^{n_k} : \sigma_k)}_{\ell \text{ times}}, \underbrace{\bullet, \dots, \bullet}_{m \text{ times}})),$$

where  $\bullet^m : \sigma := \bullet : \dots : \bullet : \sigma$ .

Let  $\mathcal{T}$  be a stream specification, and  $f \in \Sigma_{sfun}$  a unary stream function symbol. By a  $d$ -o trace of  $f$  in  $\mathcal{T}$  we mean, for a given data-exchange function  $\mathcal{G}$ , and a closed infinite stream term  $r$  of the form  $u_0 : u_1 : u_2 : \dots$ , the production function  $\pi_{\rho} : \mathbb{N} \rightarrow \overline{\mathbb{N}}$  of a rewrite sequence  $\rho : s_0 = f(r) \rightarrow_{\mathcal{A}_{\mathcal{T}, \mathcal{G}}} s_1 \rightarrow_{\mathcal{A}_{\mathcal{T}, \mathcal{G}}} s_2 \rightarrow_{\mathcal{A}_{\mathcal{T}, \mathcal{G}}} \dots$ , where  $\pi_{\rho}$  is defined as follows: for all  $n \in \mathbb{N}$ ,  $\pi_{\rho}(n)$  is the supremum of the lengths of stream prefixes in those terms  $s_i$  until which during the steps of  $\rho$  less or equal to  $n$  stream elements of  $r$  within  $s_i$  have been consumed; more precisely,  $\pi_{\rho}(n)$  is the supremum of the number of leading ‘:’ symbols in terms  $s_i$  where  $i$  is such that no descendent [8, p. 390] of the position of the  $(n+1)$ -th symbol ‘:’ in  $s_0$  is in the pattern of a redex contracted during the first  $i$  steps of  $\rho$ .

As a consequence of the use of pattern matching on data in defining rules, even simple stream function specifications can exhibit a complex  $d$ -o behaviour, that is, possess large sets of  $d$ -o traces. Consider the specification  $h(0 : s) \rightarrow h(s)$  and  $h(1 : s) \rightarrow 1 : h(s)$ . Here  $n \mapsto 0$ , and  $n \mapsto n$  are  $d$ -o traces of  $h$ , as well as all functions  $h : \mathbb{N} \rightarrow \overline{\mathbb{N}}$  with the property  $\forall n \in \mathbb{N}. 0 \leq h(n+1) - h(n) \leq 1$ . As an example of a more complicated situation, consider the flat function specification:

$$\begin{aligned} f(\sigma) &\rightarrow g(\sigma, \sigma) \\ g(0 : y : \sigma, x : \tau) &\rightarrow 0 : 0 : g(\sigma, \tau) \\ g(1 : \sigma, x_1 : x_2 : x_3 : x_4 : \tau) &\rightarrow 0 : 0 : 0 : 0 : 0 : g(\sigma, \tau) \end{aligned}$$

Fig. 4 shows a (small) selection of the set of  $d$ -o traces for  $f$ , in particular the  $d$ -o traces that contribute to the  $d$ -o lower bound  $\underline{do}_{\mathcal{T}}(f)$ . In this example the lower bound  $\underline{do}_{\mathcal{T}}(f)$  is a superposition of multiple  $d$ -o traces of  $f$ . In general  $\underline{do}_{\mathcal{T}}(f)$  can even be a superposition of infinitely many  $d$ -o traces.

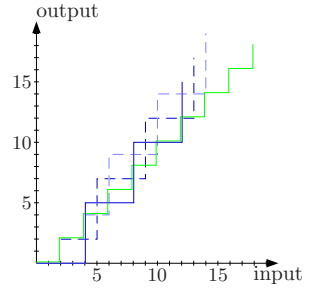


Fig. 4. Traces

## 4 The Production Calculus

As a means to compute the  $d$ -o production behaviour of stream specifications, we introduce a ‘production calculus’ with periodically increasing functions as its central ingredient.

We use  $\overline{\mathbb{N}} := \mathbb{N} \uplus \{\infty\}$ , the *extended natural numbers*, with the usual  $\leq$ ,  $+$ , and we define  $\infty - n := \infty$  for all  $n \in \mathbb{N}$ , and  $\infty - \infty := 0$ .

An infinite sequence  $\sigma \in X^\omega$  is *eventually periodic* if  $\sigma = \alpha\beta\beta\beta\dots$  for some  $\alpha \in X^*$  and  $\beta \in X^+$ . A function  $f : \mathbb{N} \rightarrow \overline{\mathbb{N}}$  is *eventually periodic* if the sequence  $\langle f(0), f(1), f(2), \dots \rangle$  is eventually periodic.

**Definition 4.1.** A function  $g : \mathbb{N} \rightarrow \overline{\mathbb{N}}$  is called *periodically increasing* if it is non-decreasing and the *derivative of  $g$* ,  $n \mapsto g(n+1) - g(n)$ , is eventually periodic. A function  $h : \overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}$  is called *periodically increasing* if its restriction to  $\mathbb{N}$  is periodically increasing and if  $h(\infty) = \lim_{n \rightarrow \infty} h(n)$ . Finally, a  $k$ -ary function  $i : (\overline{\mathbb{N}})^k \rightarrow \overline{\mathbb{N}}$  is called *periodically increasing* if  $i(n_1, \dots, n_k) = \min(i_1(n_1), \dots, i_k(n_k))$  for some unary periodically increasing functions  $i_1, \dots, i_k$ .

Periodically increasing (p-i) functions can be denoted by their value at 0 followed by a representation of their derivative. For example,  $031\overline{2}$  denotes the p-i function  $f : \mathbb{N} \rightarrow \mathbb{N}$  with values  $0, 3, 4, 6, 7, 9, \dots$ . We use a finer and more flexible notation over the alphabet  $\{-, +\}$  that will be useful in Sec. 5. For instance, we denote  $f$  as above by the ‘io-term’  $\langle +^0 - +^3, - +^1 - +^2 \rangle$ .

**Definition 4.2.** An *io-term* is a pair  $\langle \alpha, \beta \rangle$  with  $\alpha \in \{-, +\}^*$  and  $\beta \in \{-, +\}^+$ . The set of io-terms is denoted by  $\mathcal{I}$ , and we use  $\iota, \kappa$  to range over io-terms. For  $\iota \in \mathcal{I}$ , we define  $\llbracket \iota \rrbracket : \mathbb{N} \rightarrow \overline{\mathbb{N}}$ , the *interpretation of  $\iota \in \mathcal{I}$* , by:

$$\begin{aligned} \llbracket \langle -\alpha, \beta \rangle \rrbracket(0) &= 0 & \llbracket \langle +\alpha, \beta \rangle \rrbracket(n) &= 1 + \llbracket \langle \alpha, \beta \rangle \rrbracket(n) \\ \llbracket \langle -\alpha, \beta \rangle \rrbracket(n+1) &= \llbracket \langle \alpha, \beta \rangle \rrbracket(n) & \llbracket \langle \epsilon, \beta \rangle \rrbracket(n) &= \llbracket \langle \beta, \beta \rangle \rrbracket(n) \end{aligned}$$

for all  $n \in \mathbb{N}$ , and extend it to  $\overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}$  by adding  $\llbracket \iota \rrbracket(\infty) = \lim_{n \rightarrow \infty} \llbracket \iota \rrbracket(n)$ . We say that  $\iota$  *represents*  $\llbracket \iota \rrbracket$ . We use  $\alpha\overline{\beta}$  as a shorthand for  $\langle \alpha, \beta \rangle$ . Here  $\epsilon$  denotes the empty word and we stipulate  $\llbracket \langle \epsilon, +^p \rangle \rrbracket(n) = 1 + 1 + \dots = \infty$ .

It is easy to verify that, for every  $\iota \in \mathcal{I}$ , the function  $\llbracket \iota \rrbracket$  is periodically increasing. Furthermore, every p-i function is represented by an io-term. Subsequently, we write  $\underline{f}$  for the shortest io-term representing a p-i function  $f : \overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}$ . Of course we then have  $\llbracket \underline{f} \rrbracket = f$  for all p-i functions  $f$ .

**Proposition 4.3.** *Unary periodically increasing functions are closed under composition and minimum.*

In addition, these operations can be computed via io-term representations. In [2] we define computable operations  $\mathbf{comp} : \mathcal{I} \times \mathcal{I} \rightarrow \mathcal{I}$ , and  $\mathbf{fix} : \mathcal{I} \rightarrow \overline{\mathbb{N}}$  such that for all  $\iota, \kappa \in \mathcal{I}$ :  $\llbracket \mathbf{comp}(\iota, \kappa) \rrbracket = \llbracket \iota \rrbracket \circ \llbracket \kappa \rrbracket$  and  $\mathbf{fix}(\iota)$  is the least fixed point of  $\llbracket \iota \rrbracket$ .

We introduce a term syntax for the production calculus and rewrite rules for evaluating closed terms; these can be visualized by ‘pebbleflow nets’, see [3,2].

**Definition 4.4.** Let  $\mathcal{X}$  be a set. The set of *production terms*  $\mathcal{P}$  is generated by:

$$p ::= \underline{k} \mid x \mid \underline{f}(p) \mid \mu x.p \mid \min(p, p)$$

where  $x \in \mathcal{X}$ , for  $k \in \overline{\mathbb{N}}$ , the symbol  $\underline{k}$  is a *numeral* (a term representation) for  $k$ , and, for a unary p-i function  $f : \overline{\mathbb{N}} \rightarrow \overline{\mathbb{N}}$ ,  $\underline{f} \in \mathcal{I}$ , the io-term representing  $f$ . For every finite set  $P = \{p_1, \dots, p_n\} \subseteq \mathcal{P}$ , we use  $\min(p_1, \dots, p_n)$  and  $\min P$  as shorthands for the production term  $\min(p_1, \min(p_2, \dots, \min(p_{n-1}, p_n)))$ .

The *production*  $\llbracket p \rrbracket \in \overline{\mathbb{N}}$  of a closed production term  $p \in \mathcal{P}$  is defined by induction on the term structure, interpreting  $\mu$  as the least fixed point operator,  $\underline{f}$  as  $f$ ,  $\underline{k}$  as  $k$ , and  $\min$  as  $\min$ .

For faithfully modelling the d-o lower bounds of stream functions with stream arity  $r$ , we employ  $r$ -ary p-i functions, which we represent by  $r$ -ary gates. An  $r$ -ary *gate*, abbreviated by  $\mathbf{gate}(\iota_1, \dots, \iota_r)$ , is a production term context of the form  $\min(\iota_1(\square_1), \dots, \iota_r(\square_r))$ , where  $\iota_1, \dots, \iota_r \in \mathcal{I}$ . We use  $\gamma$  as a syntactic variable for gates. The *interpretation* of a gate  $\gamma = \mathbf{gate}(\iota_1, \dots, \iota_r)$  is defined as  $\llbracket \gamma \rrbracket(n_1, \dots, n_r) := \min(\llbracket \iota_1 \rrbracket(n_1), \dots, \llbracket \iota_r \rrbracket(n_r))$ . It is possible to choose unique gate representations  $\underline{f}$  of p-i functions  $f$  that are efficiently computable from other gate representations, see [2].

Owing to the restriction to (term representations of) periodically increasing functions in Def. 4.4 it is possible to calculate the production  $\llbracket p \rrbracket$  of terms  $p \in \mathcal{P}$ . For that purpose, we define a rewrite system which reduces any closed term to a numeral  $\underline{k}$ . This system makes use of the computable operations  $\mathbf{comp}$  and  $\mathbf{fix}$  on io-terms mentioned above.

**Definition 4.5.** The *rewrite relation*  $\rightarrow_{\mathbb{R}}$  on production terms is defined as the compatible closure of the following rules:

$$\begin{array}{ll} \iota_1(\iota_2(p)) \rightarrow \mathbf{comp}(\iota_1, \iota_2)(p) & \iota(\underline{k}) \rightarrow \llbracket \iota \rrbracket(\underline{k}) \\ \iota(\min(p_1, p_2)) \rightarrow \min(\iota(p_1), \iota(p_2)) & \mu x.x \rightarrow \underline{0} \\ \mu x.\min(p_1, p_2) \rightarrow \min(\mu x.p_1, \mu x.p_2) & \mu x.p \rightarrow p \quad \text{if } x \notin \mathbf{FV}(p) \\ \mu x.\iota(x) \rightarrow \underline{\mathbf{fix}(\iota)} & \min(\underline{k_1}, \underline{k_2}) \rightarrow \underline{\min(k_1, k_2)} \end{array}$$

The following theorem establishes the usefulness of  $\rightarrow_{\mathbb{R}}$ : the production  $\llbracket p \rrbracket$  of a production term  $p$  can always be computed by reducing  $p$  according to  $\rightarrow_{\mathbb{R}}$ , thereby obtaining a normal form that is a numeral after finitely many steps.

**Theorem 4.6.** *The rewrite relation  $\rightarrow_{\mathbb{R}}$  is confluent, terminating and production preserving, that is,  $p \rightarrow_{\mathbb{R}} p'$  implies  $\llbracket p \rrbracket = \llbracket p' \rrbracket$ . Every closed  $p \in \mathcal{P}$  has a numeral  $\underline{k}$  as its unique  $\rightarrow_{\mathbb{R}}$ -normal form, and it holds that  $\llbracket p \rrbracket = k$ .*

*Proof.* Termination of  $\rightarrow_{\mathbb{R}}$  is straightforward to show. Confluence of  $\rightarrow_{\mathbb{R}}$  follows by Newman's lemma since all critical pairs are convergent. For preservation of production of  $\rightarrow_{\mathbb{R}}$  it suffices to show this property for each of the rules. This is not difficult, except for the third rule (that distributes  $\mu x$  over  $\min$ ) for which preservation of production is an immediate consequence of Lem. 4.7 below, in view of the fact that  $(\overline{\mathbb{N}}, \leq)$  is a complete chain.  $\square$

A *complete lattice* is a partially ordered set in which every subset has a least upper bound and a greatest lower bound. A *complete chain* is a complete lattice on which the order is linear. As a consequence of the Knaster–Tarski theorem every order-preserving (non-decreasing) function  $f$  on a complete lattice has a least fixed point  $\mathbf{lfp}(f)$ . We use  $\wedge$  for the infix infimum operation.

**Lemma 4.7.** *Let  $\langle D, \leq \rangle$  be a complete chain. Then it holds that:*

$$\forall f, g : D \rightarrow D \text{ non-decreasing. } \text{lfp}(f \wedge g) = \text{lfp}(f) \wedge \text{lfp}(g) \quad (\circ)$$

*Proof.* Let  $\langle D, \leq \rangle$  be a complete chain, and let  $f, g : D \rightarrow D$  be non-decreasing. The inequality  $\text{lfp}(f \wedge g) \leq \text{lfp}(f) \wedge \text{lfp}(g)$  follows easily by using that, for every non-decreasing function  $h$  on  $D$ ,  $\text{lfp}(h)$  is the infimum of all pre-fixed points of  $h$ , that is, of all  $x \in D$  with  $h(x) \leq x$ . For the converse inequality, let  $x := \text{lfp}(f \wedge g)$ . Since  $x = (f \wedge g)(x) = f(x) \wedge g(x)$ , and  $D$  is linear, it follows that either  $f(x) = x$  or  $g(x) = x$ , and hence that  $x$  is either a fixed point of  $f$  or of  $g$ . Hence  $x \geq \text{lfp}(f)$  or  $x \geq \text{lfp}(g)$ , and therefore  $\text{lfp}(f \wedge g) = x \geq \text{lfp}(f) \wedge \text{lfp}(g)$ .  $\square$

We additionally mention that  $(\circ)$  holds in a complete lattice only if it is linear.

## 5 Translation into Production Terms

In this section we define a translation from stream constants in flat or friendly nesting specifications to production terms. In particular, the root  $M_0$  of a specification  $\mathcal{T}$  is mapped by the translation to a production term  $[M_0]$  with the property that if  $\mathcal{T}$  is flat (friendly nesting), then the d-o lower bound on the production of  $M_0$  in  $\mathcal{T}$  equals (is bounded from below by) the production of  $[M_0]$ .

### 5.1 Translation of Flat and Friendly Nesting Symbols

As a first step of the translation, we describe how for a flat (or friendly nesting) stream function symbol  $f$  in a stream specification  $\mathcal{T}$  a periodically increasing function  $[f]$  can be calculated that is (that bounds from below) the d-o lower bound on the production of  $f$  in  $\mathcal{T}$ .

Let us again consider the rules (i)  $f(\mathfrak{s}(x) : y : \sigma) \rightarrow a(\mathfrak{s}(x), y) : f(y : \sigma)$ , and (ii)  $f(0 : \sigma) \rightarrow 0 : \mathfrak{s}(0) : f(\sigma)$  from Fig. 2. We model the d-o lower bound on the production of  $f$  by a function from  $\overline{\mathbb{N}}$  to  $\overline{\mathbb{N}}$  defined as the unique solution for  $X_f$  of the following system of equations. We disregard what the concrete stream elements are, and therefore we take the infimum over all possible traces:

$$X_f(n) = \inf \{ X_{f,(i)}(n), X_{f,(ii)}(n) \}$$

where the solutions for  $X_{f,(i)}$  and  $X_{f,(ii)}$  are the d-o lower bounds of  $f$  assuming that the first rule applied in the rewrite sequence is (i) or (ii), respectively. The rule (i) consumes two elements, produces one element and feeds one element back to the recursive call. For rule (ii) these numbers are 1, 2, 0 respectively. Therefore we get:

$$\begin{aligned} X_{f,(i)}(n) &= \text{let } n' := n - 2, \text{ if } n' < 0 \text{ then } 0 \text{ else } 1 + X_f(n' + 1), \\ X_{f,(ii)}(n) &= \text{let } n' := n - 1, \text{ if } n' < 0 \text{ then } 0 \text{ else } 2 + X_f(n' + 0). \end{aligned}$$

The unique solution for  $X_f$  is  $n \mapsto n \dot{-} 1$ , represented by the io-term  $\overline{-\dot{-}1}$ .

In general, functions may have multiple arguments, which during rewriting may get permuted, duplicated or deleted. The idea is to track single arguments, and take the infimum over all branches in case an argument is duplicated.

For example, the rule  $\text{zip}(x : \sigma, \tau) \rightarrow x : \text{zip}(\tau, \sigma)$  with a permutation of the stream arguments, gives rise to the following specification:

$$\begin{aligned} X_{\text{zip},1}(n) &= \text{let } n' := n - 1, \text{ if } n' < 0 \text{ then } 0 \text{ else } 1 + X_{\text{zip},2}(n') \\ X_{\text{zip},2}(n) &= \text{let } n' := n - 0, \text{ if } n' < 0 \text{ then } 0 \text{ else } 1 + X_{\text{zip},1}(n'), \end{aligned}$$

and duplication of arguments like in the rule  $f(x : \sigma) \rightarrow g(\sigma, x : \sigma)$  yields:

$$X_{f,1}(n) = \text{let } n' := n - 1, \text{ if } n' < 0 \text{ then } 0 \text{ else } \inf \{X_{g,1}(n'), X_{g,2}(1 + n')\}.$$

For a recursion variable  $X$  let  $\langle X \rangle$  be the unique solution for  $X$ . The intuition behind the recursion variables is as follows. Let  $f$  be a flat stream function symbol with stream arity  $k$ . Then the solution  $\langle X_f \rangle$  for  $X_f$  models the d-o lower bound on the production of  $f$ , that is,  $\langle X_f \rangle = \underline{do}_{\mathcal{T}}(f)$ . Furthermore, the variables  $X_{f,i}$  for  $1 \leq i \leq k$  describe how the consumption from the  $i$ -th argument of  $f$  ‘retards’ the production of  $f$ , more precisely,  $\langle X_{f,i} \rangle = \lambda n. \underline{do}_{\mathcal{T}}(f(\bullet^\infty, \dots, \bullet^\infty, \bullet^n, \bullet^\infty, \dots, \bullet^\infty))$ .

Finally, consider  $h(x : \sigma) \rightarrow Y, Y \rightarrow 0 : Z$  and  $Z \rightarrow Z$ , a specification illustrating the case of deletion of stream arguments. To translate stream functions like  $h$  we extend the translation of flat stream functions to include flat stream constants. To cater for the case that there are no stream arguments or all stream arguments get deleted during reduction, we introduce fresh recursion variables  $X_{f,\star}$  for every stream symbol  $f$ . The variable  $X_{f,\star}$  expresses the production of  $f$  assuming infinite supply in each argument, that is,  $\langle X_{f,\star} \rangle = \underline{do}_{\mathcal{T}}(f(\bullet^\infty, \dots, \bullet^\infty))$ .

Therefore in the definition of the translation of stream functions, we need to distinguish the cases according to whether a symbol is weakly guarded or not.

**Definition 5.1.** We define the *dependency relation*  $\dashv\!\!\dashv$  between symbols in  $\Sigma_S^-$  by  $\dashv\!\!\dashv := \{(f, g) \in \Sigma_S^- \times \Sigma_S^- \mid f(\mathbf{s}, \mathbf{u}) \rightarrow g(\mathbf{t}, \mathbf{v}) \in R_S\}$  (remember that ‘ $\cdot$ ’  $\notin \Sigma_S^-$ ). We say that a symbol  $f \in \Sigma_S^-$  is *weakly guarded* if  $f$  is strongly normalising with respect to  $\dashv\!\!\dashv$  and *unguarded*, otherwise.

The translation of a stream function symbol is defined as the unique solution of a (potentially infinite) system of defining equations where the unknowns are functions. More precisely, for each symbol  $f \in \Sigma_{f_{\text{nest}}} \supseteq \Sigma_{f_{\text{fun}}}$  of a flat or friendly nesting stream specification, this system has a p-i function  $[f]$  as a solution for  $X_f$ , which is unique among the continuous functions. In [2] we present an algorithm that effectively calculates these solutions in the form of gates.

**Definition 5.2.** Let  $\langle \Sigma, R \rangle$  be a stream specification. For each flat or friendly nesting symbol  $f \in \Sigma_{f_{\text{nest}}} \supseteq \Sigma_{f_{\text{flat}}}$  with arities  $k = ar_s(f)$  and  $\ell = ar_d(f)$  we

define  $[f] : \overline{\mathbb{N}}^k \rightarrow \overline{\mathbb{N}}$ , the *translation* of  $f$ , as  $[f] := \langle X_f \rangle$  where  $\langle X_f \rangle$  is the unique solution for  $X_f$  of the following system of equations:

For all  $n_1, \dots, n_k \in \overline{\mathbb{N}}$ ,  $i \in \{1, \dots, k\}$ , and  $n \in \mathbb{N}$ :

$$\begin{aligned} X_f(n_1, \dots, n_k) &= \inf \{ X_{f, \star}, X_{f, 1}(n_1), \dots, X_{f, k}(n_k) \}, \\ X_{f, \star} &= \begin{cases} \inf \{ X_{f, \star, \rho} \mid \rho \text{ a defining rule of } f \} & \text{if } f \text{ is weakly guarded,} \\ 0 & \text{if } f \text{ is unguarded,} \end{cases} \\ X_{f, i}(n) &= \begin{cases} \inf \{ X_{f, i, \rho}(n) \mid \rho \text{ a defining rule of } f \} & \text{if } f \text{ is weakly guarded,} \\ 0 & \text{if } f \text{ is unguarded.} \end{cases} \end{aligned}$$

We write  $\mathbf{u}_i : \sigma_i$  for  $u_{i,1} : \dots : u_{i,p} : \sigma_i$ , and  $|\mathbf{u}_i|$  for  $p$ . For  $X_{f, \star, \rho}$  and  $X_{f, i, \rho}$  we distinguish the possible forms the rule  $\rho$  can have. If  $\rho$  is nesting, then  $X_{f, \star, \rho} = \infty$ , and  $X_{f, i, \rho}(n) = n$  for all  $n \in \overline{\mathbb{N}}$ . Otherwise,  $\rho$  is non-nesting and of the form:

$$f((\mathbf{u}_1 : \sigma_1), \dots, (\mathbf{u}_k : \sigma_k), v_1, \dots, v_\ell) \rightarrow w_1 : \dots : w_m : s,$$

where either (a)  $s \equiv \sigma_j$ , or (b)  $s \equiv \mathbf{g}((\mathbf{u}'_1 : \sigma_{\phi(1)}), \dots, (\mathbf{u}'_{k'} : \sigma_{\phi(k')}), v'_1, \dots, v'_{\ell'})$  with  $k' = \text{ar}_s(\mathbf{g})$ ,  $\ell' = \text{ar}_d(\mathbf{g})$ , and  $\phi : \{1, \dots, k'\} \rightarrow \{1, \dots, k\}$ . Then we add:

$$\begin{aligned} X_{f, \star, \rho} &= \begin{cases} \infty & \text{case (a)} \\ m + X_{\mathbf{g}, \star} & \text{case (b)} \end{cases} \\ X_{f, i, \rho}(n) &= \text{let } n' := n - |\mathbf{u}_i|, \text{ if } n' < 0 \text{ then } 0 \text{ else} \\ &\quad m + \begin{cases} n' & \text{case (a), } i = j \\ \infty & \text{case (a), } i \neq j \\ \inf \{ X_{\mathbf{g}, \star}, X_{\mathbf{g}, j}(n' + |\mathbf{u}'_j|) \mid j \in \phi^{-1}(i) \} & \text{case (b).} \end{cases} \end{aligned}$$

**Proposition 5.3.** *Let  $\mathcal{T}$  be a stream specification, and  $f \in \Sigma_{f\text{nest}} \supseteq \Sigma_{f\text{flat}}$  a stream function symbol with  $k = \text{ar}_s(f)$ . The system of recursive equations described in Def. 5.2 has a  $k$ -ary  $p$ - $i$  function as its unique solution for  $X_f$ , which we denote by  $[f]$ . Furthermore, the gate representation  $\underline{[f]}$  of  $[f]$  can be computed.*

Concerning non-nesting rules on which defining rules for friendly nesting symbols depend via  $\sim$ , this translation uses the fact that their production is bounded below by ‘min’. These bounds are not necessarily optimal, but can be used to show productivity of examples like Ex. 2.6.

*Example 5.4.* Consider a pure stream specification with the function layer:

$$\begin{aligned} f(x : \sigma) &\rightarrow x : \mathbf{g}(\sigma, \sigma, \sigma), \\ \mathbf{g}(x : y : \sigma, \tau, v) &\rightarrow x : \mathbf{g}(y : \tau, y : v, y : \sigma). \end{aligned}$$

The translation of  $f$  is  $[f]$ , the unique solution for  $X_f$  of the system:

$$\begin{aligned}
X_f(n) &= \inf \{X_{f,\star}, X_{f,1}(n)\} \\
X_{f,1}(n) &= \text{let } n' := n - 1 \\
&\quad \text{if } n' < 0 \text{ then } 0 \text{ else } 1 + \inf \{X_{g,\star}, X_{g,1}(n'), X_{g,2}(n'), X_{g,3}(n')\} \\
X_{f,\star} &= 1 + X_{g,\star} \\
X_{g,1}(n) &= \text{let } n' := n - 2, \text{ if } n' < 0 \text{ then } 0 \text{ else } 1 + \inf \{X_{g,\star}, X_{g,3}(1 + n')\} \\
X_{g,2}(n) &= 1 + \inf \{X_{g,\star}, X_{g,1}(1 + n)\} \\
X_{g,3}(n) &= 1 + \inf \{X_{g,\star}, X_{g,2}(1 + n)\} \\
X_{g,\star} &= 1 + X_{f,\star}
\end{aligned}$$

An algorithm for solving such systems of equations is described in [2]; here we solve the system directly. Note that  $X_{f,\star} = X_{g,\star} = \infty$ , and therefore  $X_{g,3}(n) = 1 + X_{g,2}(n + 1) = 2 + X_{g,1}(n + 2) = 3 + X_{g,3}(n)$ , hence  $\forall n \in \mathbb{N}. X_{g,3}(n) = \infty$ . Likewise we obtain  $X_{g,2}(n) = \infty$  if  $n \geq 1$  and 1 for  $n = 0$ , and  $X_{g,1}(n) = \infty$  if  $n \geq 2$  and 0 for  $n \leq 1$ . Then it follows that  $[f](0) = 0$ ,  $[f](1) = [f](2) = 1$ , and  $[f](n) = \infty$  for all  $n \geq 2$ , represented by the gate  $\underline{[f]} = \text{gate}(-+-\overline{-})$ . The gate corresponding to  $g$  is  $\underline{[g]} = \text{gate}(-\overline{-}\overline{-}, +-\overline{-}, \overline{-})$ .

*Example 5.5.* Consider a flat stream function specification with the following rules which use pattern matching on the data constructors 0 and 1:

$$f(0 : \sigma) \rightarrow g(\sigma) \quad f(1 : x : \sigma) \rightarrow x : g(\sigma) \quad g(x : y : \sigma) \rightarrow x : y : g(\sigma)$$

denoted  $\rho_{f_0}$ ,  $\rho_{f_1}$ , and  $\rho_g$ , respectively. Then,  $[f]$  is the solution for  $X_{f,1}$  of:

$$\begin{aligned}
X_f(n) &= \inf \{X_{f,\star}, X_{f,1}(n)\} \\
X_{f,1}(n) &= \inf \{X_{f,1,\rho_{f_0}}(n), X_{f,1,\rho_{f_1}}(n)\} \\
X_{f,1,\rho_{f_0}}(n) &= \text{let } n' := n - 1, \text{ if } n' < 0 \text{ then } 0 \text{ else } \{X_{g,\star}, X_{g,1}(n')\} \\
X_{f,1,\rho_{f_1}}(n) &= \text{let } n' := n - 2, \text{ if } n' < 0 \text{ then } 0 \text{ else } 1 + \{X_{g,\star}, X_{g,1}(n')\} \\
X_{f,\star} &= \inf \{X_{g,\star}, 1 + X_{g,\star}\} \\
X_{g,1}(n) &= \text{let } n' := n - 2, \text{ if } n' < 0 \text{ then } 0 \text{ else } 2 + \{X_{g,\star}, X_{g,1}(n')\} \\
X_{g,\star} &= 2 + X_{g,\star}.
\end{aligned}$$

As solution we obtain an overlapping of both traces  $\underline{[f]}_{1,\rho_{f_0}}$  and  $\underline{[f]}_{1,\rho_{f_1}}$ , that is,  $\underline{[f]}_1(n) = n \div 2$  represented by the gate  $\underline{[f]} = \text{gate}(-\overline{-}\overline{-})$ .

The following lemma states that the translation  $[f]$  of a flat stream function symbol  $f$  (as defined in Def. 5.2) is the d-o lower bound on the production function of  $f$ . For friendly nesting stream symbols  $f$  it states that  $[f]$  pointwisely bounds from below the d-o lower bound on the production function of  $f$ .

**Lemma 5.6.** *Let  $\mathcal{T}$  be a stream specification, and let  $f \in \Sigma_{f_{\text{nest}}} \supseteq \Sigma_{\text{flat}}$ .*

- (i) *If  $f$  is flat, then:  $[f] = \underline{d\mathcal{O}}_{\mathcal{T}}(f)$ . Hence,  $\underline{d\mathcal{O}}_{\mathcal{T}}(f)$  is periodically increasing.*
- (ii) *If  $f$  is friendly nesting, then it holds:  $[f] \leq \underline{d\mathcal{O}}_{\mathcal{T}}(f)$  (pointwise inequality).*

## 5.2 Translation of Stream Constants

In the second step, we now define a translation of stream constants in a flat or friendly nesting stream specification into production terms under the assumption that gate translations for the stream functions are given. 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 using gate translations of the stream function symbols from a given family of gates; whenever a stream constant is met that has been unfolded before, the translation stops after establishing a binding to a  $\mu$ -binder created earlier.

**Definition 5.7.** Let  $\mathcal{T}$  be a stream specification,  $M \in \Sigma_{scon}$ , and  $\mathcal{F} = \{\gamma_f\}_{f \in \Sigma_{sfun}}$  a family of gates. The *translation*  $[M]^\mathcal{F} \in \mathcal{P}$  of  $M$  with respect to  $\mathcal{F}$  is defined by  $[M]^\mathcal{F} := [M]^\mathcal{F}_\emptyset$ , where, for every  $M \in \Sigma_{scon}$  and every  $\alpha \subseteq \Sigma_{scon}$  we define:

$$[M(\mathbf{u})]^\mathcal{F}_\alpha := [M]^\mathcal{F}_\alpha := \begin{cases} \mu M. \min \{ [r]^\mathcal{F}_{\alpha \cup \{M\}} \mid M(\mathbf{v}) \rightarrow r \in R \} & \text{if } M \notin \alpha \\ M & \text{if } M \in \alpha \end{cases}$$

$$[u : s]^\mathcal{F}_\alpha := +\overline{+}([s]^\mathcal{F}_\alpha)$$

$$[f(s_1, \dots, s_{ar_s(f)}, u_1, \dots, u_{ar_u(f)})]^\mathcal{F}_\alpha := \gamma_f([s_1]^\mathcal{F}_\alpha, \dots, [s_{ar_s(f)}]^\mathcal{F}_\alpha)$$

*Example 5.8.* As an example we translate Pascal's triangle, see Fig. 2. The translation of the stream function symbols is  $\mathcal{F} = \{[f] = \text{gate}(\overline{+})\}$ , cf. page 90. Hence we obtain  $[P]^\mathcal{F} = \mu P. +\overline{+}(\overline{+}(\overline{+}(\overline{+}(P)))$  as the translation of  $P$ .

The following lemma is the basis of our main results in Sec. 6. It entails that if we use gates that represent d-o optimal lower bounds on the production of the stream functions, then the translation of a stream constant  $M$  yields a production term that rewrites to the d-o lower bound of the production of  $M$ .

**Lemma 5.9.** *Let  $\mathcal{T} = \langle \Sigma, R \rangle$  be a stream specification, and  $\mathcal{F} = \{\gamma_f\}_{f \in \Sigma_{sfun}}$  a family of gates. If  $[[\gamma_f]] = \underline{d}\mathcal{O}_\mathcal{T}(f)$  for all  $f \in \Sigma_{sfun}$ , then for all  $M \in \Sigma_{scon}$ :  $[[M]^\mathcal{F}] = \underline{d}\mathcal{O}_\mathcal{T}(M)$ . Hence,  $\mathcal{T}$  is d-o productive if and only if  $[[[M_0]^\mathcal{F}]] = \infty$ .*

*If  $[[\gamma_f]] \leq \underline{d}\mathcal{O}_\mathcal{T}(f)$  for all  $f \in \Sigma_{sfun}$ , then for all  $M \in \Sigma_{scon}$ :  $[[[M]^\mathcal{F}]] \leq \underline{d}\mathcal{O}_\mathcal{T}(M)$ . Consequently,  $\mathcal{T}$  is d-o productive if  $[[[M_0]^\mathcal{F}]] = \infty$ .*

## 6 Deciding Data-Oblivious Productivity

In this section we assemble our results concerning decision of d-o productivity, and automatable recognition of productivity. We define methods:

- (DOP) for deciding d-o productivity of flat stream specifications,
- (DP) for deciding productivity of pure stream specifications, and
- (RP) for recognising productivity of friendly nesting stream specifications,

that proceed in the following steps:

- (i) Take as input a (DOP) flat, (DP) pure, or (RP) friendly nesting stream specification  $\mathcal{T} = \langle \Sigma, R \rangle$ .



- (ii) Translate the stream function symbols into gates  $\mathcal{F} := \{\llbracket f \rrbracket\}_{f \in \Sigma_{sfun}}$  (Def. 5.2).
- (iii) Construct the production term  $[M_0]^{\mathcal{F}}$  with respect to  $\mathcal{F}$  (Def. 5.7).
- (iv) Compute the production  $k$  of  $[M_0]^{\mathcal{F}}$  using  $\rightarrow_{\mathcal{R}}$  (Def. 4.5).
- (v) Give the following output:
  - (DOP) “ $\mathcal{T}$  is d-o productive” if  $k = \infty$ , else “ $\mathcal{T}$  is not d-o productive”.
  - (DP) “ $\mathcal{T}$  is productive” if  $k = \infty$ , else “ $\mathcal{T}$  is not productive”.
  - (RP) “ $\mathcal{T}$  is productive” if  $k = \infty$ , else “don’t know”.

Note that all of these steps are automatable (cf. our productivity tool, Sec. 7).

Our main result states that d-o productivity is decidable for flat stream specifications. Since d-o productivity implies productivity (Prop. 3.3), we obtain a computable, d-o optimal, sufficient condition for productivity of flat stream specifications, which cannot be improved by any other d-o analysis. Second, since for pure stream specifications d-o productivity and productivity are the same, we get that productivity is decidable for them.

**Theorem 6.1.** (i) DOP decides d-o productivity of flat stream specifications, (ii) DP decides productivity of pure stream specifications.

*Proof.* Let  $k$  be the production of the term  $[M_0]^{\mathcal{F}} \in \mathcal{P}$  in step (iv) of DOP/DP.

- (i) By Lem. 5.6 (i), Lem. 5.9, and Thm. 4.6 we find:  $k = \underline{do}_{\mathcal{T}}(M_0)$ .
- (ii) For pure specifications we additionally note:  $\Pi_{\mathcal{T}}(M_0) = \underline{do}_{\mathcal{T}}(M_0)$ .  $\square$

Third, we obtain a computable, sufficient condition for productivity of friendly nesting stream specifications.

**Theorem 6.2.** A friendly nesting (flat) stream specification  $\mathcal{T}$  is productive if the algorithm RP(DOP) recognizes  $\mathcal{T}$  as productive.

*Proof.* By Lem. 5.6 (ii), Lem. 5.9, and Thm. 4.6:  $k \leq \underline{do}_{\mathcal{T}}(M_0) \leq \Pi_{\mathcal{T}}(M_0)$ .  $\square$

*Example 6.3.* We illustrate the decision of d-o productivity by means of Pascal’s triangle, Fig. 2. We reduce  $[P]^{\mathcal{F}}$ , the translation of  $P$ , to  $\rightarrow_{\mathcal{R}}$ -normal form:

$$[P]^{\mathcal{F}} = \mu P. + \overline{+} (+ \overline{+} (- \overline{+} (P))) \rightarrow_{\mathcal{R}}^* \mu P. + + \overline{+} (P) \rightarrow_{\mathcal{R}} \underline{\infty}$$

Hence  $\underline{do}_{\mathcal{T}}(P) = \infty$ , and  $P$  is d-o productive and therefore productive.

## 7 Conclusion and Further Work

In order to formalize quantitative approaches for recognizing productivity of stream specifications, we defined the notion of d-o rewriting and investigated d-o productivity. For the syntactic class of flat stream specifications (that employ pattern matching on data), we devised a decision algorithm for d-o productivity. In this way we settled the productivity recognition problem for flat stream specifications from a d-o perspective. For the even larger class including friendly nesting stream function rules, we obtained a computable sufficient condition for productivity. For the subclass of pure stream specifications (a substantial extension of the class given in [3]) we showed that productivity and d-o productivity

coincide, and thereby obtained a decision algorithm for productivity of pure specifications.

We have implemented in Haskell the decision algorithm for d-o productivity. This tool, together with more information including a manual, examples, our related papers, and a comparison of our criteria with those of [4,7,1] can be found at our web page <http://infinity.few.vu.nl/productivity>. The reader is invited to experiment with our tool.

It is not possible to obtain a d-o optimal criterion for non-productivity of flat specifications in an analogous way to how we established such a criterion for productivity. This is because the d-o upper bound  $\overline{do}_{\tau}(f)$  on the production of a stream function  $f$  in flat stream specifications is not in general a periodically increasing function. For example, for the following stream function specification:

$$f(x : \sigma, \tau) \rightarrow x : f(\sigma, \tau) , \quad f(\sigma, y : \tau) \rightarrow y : f(\sigma, \tau) ,$$

it holds that  $\overline{do}(f)(n_1, n_2) = n_1 + n_2$ , which is not p-i. While this example is not orthogonal,  $\overline{do}(f)$  is also not p-i for the following similar orthogonal example:

$$f(0 : x : \sigma, y : \tau) \rightarrow x : f(\sigma, \tau) , \quad f(1 : \sigma, x : y : \tau) \rightarrow y : f(\sigma, \tau) .$$

Currently we are developing a method that goes beyond a d-o analysis, one that would, e.g., prove productivity of the example **B** given in the introduction. Moreover, we study a refined production calculus that accounts for the delay of evaluation of stream elements, in order to obtain a faithful modelling of lazy evaluation, needed for example for **S** on page 82, where the first element depends on a ‘future’ expansion of **S**.

*Acknowledgement.* We thank Jan Willem Klop, Carlos Lombardi, Vincent van Oostrom, and Roel de Vrijer for useful discussions, and the anonymous referees for their comments and suggestions.

## 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. Endrullis, J., Grabmayer, C., Hendriks, D.: Data-Oblivious Stream Productivity. Technical report (2008), <http://arxiv.org/pdf/0806.2680>
3. Endrullis, J., Grabmayer, C., Hendriks, D., Isihara, A., Klop, J.W.: Productivity of Stream Definitions. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) *FCT 2007*. LNCS, vol. 4639, pp. 274–287. Springer, Heidelberg (2007)
4. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: *POPL 1996*, pp. 410–423 (1996)
5. Kapur, D., Narendran, P., Rosenkrantz, D.J., Zhang, H.: Sufficient-Completeness, Ground-Reducibility and their Complexity. *Acta Informatica* 28(4), 311–350 (1991)
6. Sijtsma, B.A.: On the Productivity of Recursive List Definitions. *ACM Transactions on Programming Languages and Systems* 11(4), 633–649 (1989)
7. Telford, A., Turner, D.: Ensuring Streams Flow. In: *AMAST*, pp. 509–523 (1997)
8. Terese: Term Rewriting Systems. *Cambridge Tracts in Theoretical Computer Science*, vol. 55. Cambridge University Press, Cambridge (2003)