Streams are Forever*

Jörg Endrullis Dimitri Hendriks Jan Willem Klop

VU University Amsterdam

Abstract

Just like diamonds, streams are forever. They are also ubiquitous, arising in theoretical computer science (formal languages and functional programming), mathematics (number theory), and engineering (signal processing). Also in computing applications streams are important. Here one can think of the streams of data queries flowing into search engines, or streams of financial data processed by financial organizations.

In this paper, we describe a number of aspects of streams that we have encountered and studied during the last years.

1 Introduction

Infinite streams, also called infinite sequences, infinite words, or ω-words, are the subject of study in several disciplines, ranging from the foundations of mathematics (choice sequences in intuitionistic logic), to engineering applications in signal processing. In this paper we will not be concerned with deep philosophical or logical aspects of infinite sequences. We will be interested mostly in specifications of streams, their classification, and their comparison.

A landmark was the work [55] of Axel Thue, who devised in 1906 infinite sequences of symbols avoiding certain simple patterns such as squares *ww* or cubes *www* where *w* is a finite word. He introduced the cubefree sequence $\mathsf{M} = 0110100110010110\cdots$, now known as the Thue-Morse sequence. This sequence turned out to be ubiquitous indeed, see [2], and was rediscovered by Marston Morse in 1921 in the mathematical context of dynamical systems and ergodic theory [45]. The Thue-Morse sequence is also known to be an automatic sequence (see [3]), and in particular it is a (purely) morphic sequence or D0L sequence. In the terminology of [49] the sequence is obtained by a 'substitution', another word

*This research has been funded by the Netherlands Organization for Scientific Research (NWO) under grant numbers 639.021.020 and 612.000.934.

for morphism. Later we will encounter several definitions of the Thue-Morse sequence, and also explain some of the key words just used denoting families of streams.

So infinite streams (we will often call them just streams) arise in theoretical computer science, in particular the areas of formal languages and combinatorics, and also in mathematics, with applications in dynamical systems and number theory. They also appear in the more practical side of computer science where functional programming languages reside (see e.g. [53]).

A word on methodology. Streams can be approached from various directions, originating from deep and established mathematical areas such as the theory of numeration systems (there have been several workshops devoted to this theme in recent years) concerned with methods to denote numbers, and also from relatively young developments in theoretical computer science, specifically infinitary term rewriting and coalgebra. The first is an outgrowth of the classical lambda calculus, while the second saw the light with the emergence of theories about infinite processes with communication, based on non-wellfounded set theory. Our point of view is that of infinitary term rewriting, to which we devote a nutshell introduction at the end of the paper.

Let us now give a short survey of the topics we aim to address. In Section 2 we start with presenting some of the well-known families of streams, exemplified by some famous streams. Thus we meet morphic sequences, automatic sequences, Toeplitz words, sturmian sequences, and sequences obtained by periodically iterated morphisms.

In Section 3 we briefly discuss the well-known relation between streams and fractals via turtle graphics. Even though it is well-known that some streams give rise via some turtles to some well-defined fractals, the complete correspondence between streams and fractals is not at all completely clear; at the end of the paper we devote a further question about such a correspondence.

The next point to visit in our guided tour is a discussion of possible ways to compare streams as to their complexity, Section 4. Here we do not mean the well-known logical complexity in terms of the arithmetical or analytical hierarchy. But there are two notions of comparing complexity of streams that spring to mind, namely Kolmogorov complexity and subword complexity. We argue that both have their drawbacks, and propose a way of comparing streams that is simple but did not yet receive attention. This comparison is analogous to comparing the intrinsic difficulty of sets of natural numbers (which are just streams over $\{0, 1\}$), by means of Turing degrees. There two objects are equivalent iff they can be transformed back and forth into each other by Turing Machines. For the purpose of comparing streams, a similar procedure can be adopted, except that Turing Machines now are much too powerful; all interesting streams are computable, and thus equivalent with respect to transforming them into one another

via Turing Machines. The appropriate transformation tool, generalizing several well-known transformation notions between streams, seems to be that of a finite state transducer (FST), a generalization of a Mealy machine. A Mealy machine transforms an input stream in a letter-to-letter fashion. FSTs are more general in that they transform letters to words. In this way we obtain an interesting hierarchy of degrees of streams, presenting us with a plethora of challenging questions.

After having discussed these degrees of streams, in Section 5 we turn to a specific question, which was raised by Larry Moss. It is concerned with a case study, namely how to decide the equivalence of streams that can be defined using very restricted means, only using the stream operator known as 'zip'. The famous Thue-Morse stream can be defined this way. Solving this question (see [32]), provided an additional bonus; it yielded an alternative characterization of automatic sequences. Moreover, a slight generalisation of the zip specification format resulted in some quite challenging open questions, where we find ourselves on the sharp edge of the decidability property.

In Section 6 we consider infinite words generated by periodically iterated morphisms, and a question raised by Lepistö and Karhumäki, asking for the subword complexity of streams obtainable by iterating morphisms. The upshot is that this way of specifying streams yields maximally complex streams in that their subword complexity is exponential. The technique to establish this is interesting, using Conway's intriguing programming language called Fractran of which a short exposition is given.

In Section 7 we discuss a problem that we have postponed somewhat, but that actually confronts us in the very beginning of studying streams, at least if we adopt the mind set of functional programming applications. This is the well-known problem of productivity, put on the map by Dijkstra, studied by his student Sijtsma [51], and approached forcefully by the functional programming community in the two seminal papers [33] and [53]. We outline the problem, and sketch progress made by the present authors.

We conclude our guided tour along streams with a sample of questions in Section 8.

2 Families of Streams

In the landscape of streams there are many families, characterised by shared principles or formats that are used in stream definitions or in the generation of streams. We give a brief listing of the main ones. An overview of some inclusion relations between these families is displayed in Figure 1.

We use standard terminology and notation; for example, see Allouche and Shallit [3] or Lothaire [43]. We use $\mathbb{N} = \{0, 1, 2, \ldots\}$ for the set of natural numbers.

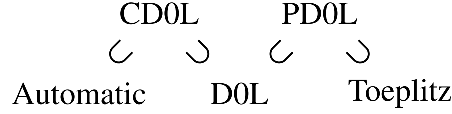


Figure 1: Some inclusions between stream families.

Let Σ be a finite alphabet. We denote by Σ^* the set of all finite words over Σ, and by ε the empty word. The set of infinite words over Σ is $\Sigma^\omega = \{\boldsymbol{x} \mid \boldsymbol{x} : \mathbb{N} \to \Sigma\}$. On the set of all words $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ we define the metric d for all $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^\infty$ by $\mathrm{d}(\boldsymbol{u}, \boldsymbol{v}) = 2^{-n}$, where n is the length of the longest common prefix of $\boldsymbol{u}$ and $\boldsymbol{v}$. For a word $w \in \Sigma^\infty$ and $n \in \mathbb{N}$, we write $w(n)$ for the n-th letter of w (counting from zero). We write $|x|$ for the length of a word $x \in \Sigma^*$. We call a word $v \in \Sigma^*$ a *factor* or *subword* of $x \in \Sigma^\infty$ if $x = uvy$ for some $u \in \Sigma^*$ and $y \in \Sigma^\infty$.

2.1 Morphic or CD0L Sequences

A widely studied family is that of the *morphic sequences*. Given alphabets Σ and Γ, a *morphism* is a map $h : \Sigma^* \to \Gamma^*$ such that $h(\varepsilon) = \varepsilon$ and $h(uv) = h(u)h(v)$ for all words $u, v \in \Sigma^*$. For $k \in \mathbb{N}$, a morphism h is called *k-uniform* if $|h(a)| = k$ for all $a \in \Sigma$; h is a *coding* if it is 1-uniform.

Let $s \in \Sigma^*$, $h : \Sigma^* \to \Sigma^*$ a morphism, and $c : \Sigma \to \Sigma$ a coding. If in the metric space $\langle \Sigma^\infty, \mathrm{d} \rangle$ the limit $h^\omega(s) = \lim_{i\to\infty} h^i(s)$ exists, then

(i) $h^\omega(s)$ is called *purely morphic* or *D0L word*, and

(ii) $c(h^\omega(s))$ is called a *morphic* or *CD0L word*.

For example, the Thue-Morse sequence M is obtained by iterating the morphism h defined by $h(0) = 01$, $h(1) = 10$ on the starting word 0, i.e., $\mathsf{M} = h^\omega(0)$.

2.2 Periodically Iterated Morphisms

Instead of repeatedly applying a single morphism, one may alternate several morphisms from a given (finite) set in a periodic fashion. This gives rise to what are called PD0L sequences [15, 16, 42, 10], which form a generalization of D0L sequences.

Let $H = \langle h_0, \ldots, h_{p-1} \rangle$ be a tuple of morphisms $h_i : \Sigma^* \to \Sigma^*$. We define the map $H : \Sigma^* \to \Sigma^*$ as follows:

$$H(a_0 a_1 \cdots a_n) = u_0 u_1 \cdots u_n$$
$$\text{where } u_i = h_k(a_i), \text{ with } k \equiv i \pmod{p} \text{ and } k \in \mathbb{N}_{<p}.$$

For $s \in \Sigma^*$, if in the metric space $\langle \Sigma^\infty, \mathrm{d}\rangle$ the limit $H^\omega(s) = \lim_{i\to\infty} H^i(s)$ exists, we call $H^\omega(s)$ a *PD0L word*.

A famous example generated by such a procedure is the Kolakoski word [39]

$$\mathsf{K} = 1\,22\,11\,2\,1\,22\,1\,22\,11\,2\,11\,22\,1\,2\,11\,2\,1\,22\,11\,2\,\cdots$$

which is defined such that $\mathsf{K}(0) = 1$ and $\mathsf{K}(n)$ equals the length of the n-th run of K; here by a 'run' we mean a maximal subsequence of consecutive identical symbols. The Kolakoski word can be generated by alternating two morphisms on the starting word 12, h_0 for the even positions and h_1 for the odd positions, defined as follows:

$$h_0 : \begin{array}{l} 1 \to 1 \\ 2 \to 11 \end{array} \qquad h_1 : \begin{array}{l} 1 \to 2 \\ 2 \to 22 \end{array}$$

So for $H = \langle h_0, h_1\rangle$, we have $\mathsf{K} = H^\omega(12)$, and the first few iterations are

$$\begin{array}{lcllcl} H^0(12) & = & & = & 12 \\ H^1(12) & = & h_0(1)\,h_1(2) & = & 12\mathbf{2} \\ H^2(12) & = & h_0(1)\,h_1(2)\,h_0(2) & = & 122\mathbf{11} \\ H^3(12) & = & h_0(1)\,h_1(2)\,h_0(2)\,h_1(1)\,h_0(1) & = & 12211\mathbf{21} \end{array}$$

It is known that the Kolakoski word is not a D0L word [16], i.e, cannot be generated by iterating a single morphism. However it is an open problem whether it is a CD0L word, i.e., the image under a coding of a D0L word. Another famous open problem about the Kolakoski word is whether the letter frequency exists and is indeed $\frac{1}{2}$, as computer experiments seem to support [44].

Some other open problems concerning PD0L words were recently solved by the first two authors, see further Section 6.

2.3 Toeplitz Words

A subclass of the class of PD0L sequences is formed by Toeplitz words [35]. A Toeplitz word T_x over an alphabet Σ is generated by a seed word $x \in \Sigma(\Sigma \cup \{?\})^*$ with $? \notin \Sigma$, as follows. For $\boldsymbol{u} \in \Sigma^\omega$ let $x^\omega[\boldsymbol{u}]$ denote the sequence obtained by replacing the subsequence of ?'s in x^ω by $\boldsymbol{u}$. Then T_x is the unique solution for $\boldsymbol{u}$ in the equation

$$\boldsymbol{u} = x^\omega[\boldsymbol{u}]\,.$$

So in order to construct T_x, we start with the periodic x^ω and then replace its subsequence of ?'s by the sequence T_x under construction. For example, if $x =$ 101?, then

$$x^\omega = 101?101?101?101?101?101?101?\cdots$$

$$\mathsf{T}_x = 101\underline{1}101\underline{0}101\underline{1}101\underline{1}101\underline{1}101\underline{0}101\underline{1}\cdots$$

where the underlined subsequence is identical to the whole sequence. This sequence T_x is known as the period doubling sequence [3, Example 6.4.3], which also is the sequence of first differences (modulo 2) of the Thue-Morse sequence M.

2.4 Automatic Sequences

An important subfamily of the morphic sequences is that of the automatic sequences, to which the beautiful monograph [3] is devoted. One way to characterize automatic sequences is that they can be obtained by iterating a uniform morphism, and apply a coding afterwards.

The standard definition of automatic sequences is via deterministic finite automata with output (DFAOs) that produce an element of a sequence when fed the index of the element as input. As an example we consider again the Thue-Morse sequence M. The n-th element $\mathsf{M}(n)$ is the parity of the number of 1's in $(n)_2$, the binary representation of n. This is realized by the automaton displayed in Figure 2.

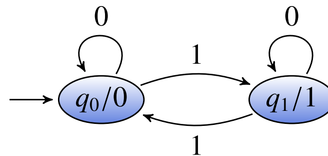


Figure 2: DFAO generating the Thue-Morse sequence.

The automaton has states $\{q_0, q_1\}$, initial state q_0, input alphabet $\{0, 1\}$ and output alphabet $\{0, 1\}$. The output letter assigned to q_0 is 0 and to q_1 is 1 (indicated by *state/output* in the states of the automaton). The automaton generates the Thue-Morse sequence $0110100110010110\cdots$ as follows. The n-th letter of the sequence is the output of the automaton when reading $(n)_2$, the base-2 expansion of n. For example, for input $(3)_2 = 11$ the automaton ends in state q_0 with output 0, and for input $(4)_2 = 100$ in state q_1 with output 1.

The automaton of Figure 2 is called a *deterministic finite state automaton with output (DFAO)*. For $k \geq 2$, a k-DFAO is an automaton over the input alphabet $\mathbb{N}_{<k} = \{0, 1, \ldots, k-1\}$. An infinite sequence $w \in \Delta^\omega$ is called *k-automatic* if there exists a k-DFAO such that for every $n \in \mathbb{N}$ the output of the automaton when reading the word $(n)_k \in \mathbb{N}_{<k}^*$ is $w(n)$, with $(n)_k$ the base-k expansion of n.

2.5 Sturmian Sequences

Much studied in mathematics, for its implications for number theory, is the family of sequences known as *sturmian sequences*. Sturmian sequences can be ob-

tained in a well-known direct geometrical way ('rotation sequences' or 'cutting sequences'), namely by intersections with the unit grid in the plane and a straight line from the origin. The most famous example here is the Fibonacci word F, $01001010\cdots$, obtained from the straight line from the origin with slope $\frac{1}{\varphi} = \varphi - 1$, with $\varphi = \frac{1+\sqrt{5}}{2}$ the golden ratio, see Figure 3.

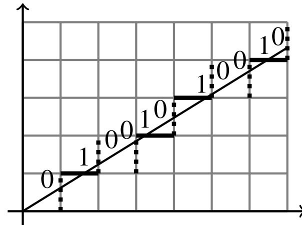


Figure 3: The Fibonacci stream $\mathsf{F} = 01001010\cdots$.

We note that the Fibonacci word is also a morphic sequence, $\mathsf{F} = h^{\omega}(0)$ with the morphism h given by $0 \to 01$, $1 \to 0$.

3 Streams and Fractals

We will now go back some thirty or forty years in the history, to the time that Seymour Papert developed his educational enterprise 'turtle graphics' or 'turtle geometry' together with the Logo programming language. It was meant to facilitate Papert's daughter with her experiments relating art and mathematics. We will use turtles in the form of finite state transducers (see Section 4), with the stipulation that the output alphabet consists of graphical instructions such as 'draw one straight line unit', 'turn write head over $\frac{\pi}{3}$', etc. We call this a 'smart' turtle as it has some memory being a finite state transducer. Now we can transform a stream into a fractal. To do this properly, we have to re-scale the drawn figure when it grows too large for the screen or page, and apply the Hausdorff metric on these successive approximations. For instance the fractal in Figure 4 (left) is generated from the Mephisto Waltz sequence $001001110001001110\cdots$ (obtained by iterating the morphism $0 \to 001$, $1 \to 110$ on starting letter 0). An interesting surprise in 2005 by Ma and Holdener was their discovery that the Thue-Morse sequence yields when drawn by a suitable turtle, the snowflake of Helge von Koch (also from 1906). As pointed out by J.-P. Allouche, this discovery was actually anticipated in the early 1980's by M. Dekking in the mathematical framework of iterated exponential sums. Not all streams and all smart turtles yield a 'decent' fractal. Some patterns are chaotic, as for example the notoriously difficult Kolakoski stream $1\,22\,11\,2\,1\,22\,1\,22\,11\,2\,11\ldots$ which is the sequence of its own

Figure 4: A turtle trajectory for the streams Mephisto Waltz (left), Kolakoski (middle) and Fibonacci (right).

run-lengths (see also Section 2). Other streams yield aesthetically pleasing patterns such as the Fibonacci word $\mathsf{F} = 0100101001001\cdots$. We are interested in the relation of the hierarchy of streams with fractals, and in particular the question whether properties of fractals can be employed to distinguish 'degrees of streams', a notion that we introduce in Section 4 below.

An amusing puzzle is to derive a sequence by looking at a fractal curve. In Figure 5 we have displayed the initial approximations of the Sierpiński arrowhead

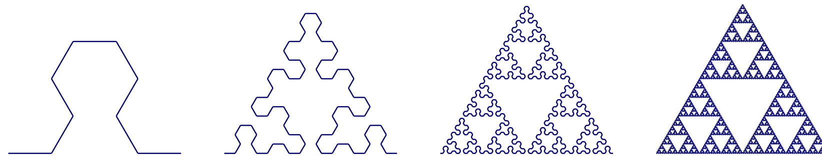


Figure 5: Initial approximations of the Sierpiński arrowhead curve [48].

curve [48]. The question arises: what is the sequence behind this fractal curve? In other words, interpreting 0 and 1 as turtle drawing instructions e.g. as follows:

0: move forward one unit length and turn to the left $\pi/3$ radials, and

1: move forward one unit length and turn to the right $\pi/3$ radials,

the search is for the sequence which generates the curve, in the limit using the Hausdorff metric.

To construct the sequence, we consider Figure 5. The first iteration of the construction, the arrowhead shape, corresponds to the word $w_1 = 00111100$. The second iteration is obtained from

$$w_2 = w_1\, 0\, \overline{w_1}\, 0\, w_1\, 1\, \overline{w_1}\, 1\, w_1\, 1\, \overline{w_1}\, 1\, w_1\, 0\, \overline{w_1}\, 0$$

where $\overline{w_1} = 11000011$, the mirrored arrowhead. Note that w_1 and $\overline{w_1}$ alternate, and the word filled in-between is w_1 itself. This construction clearly resembles

the construction of Toeplitz words, as introduced in Section 2.3, and we find that the Toeplitz word generated by the pattern

$$x = 0\,0\,1\,1\,1\,1\,0\,0\,?\,1\,1\,0\,0\,0\,0\,1\,1\,?$$

is the desired sequence $\mathsf{T}_x = 001111000110000110001111001110000111\cdots$, which we call the *Sierpiński sequence* $\mathsf{S} = \mathsf{T}_x$.

4 Degrees of Streams

We now consider a novel approach [29] to comparing the complexity of streams, namely in terms of reducibility by finite state transducers (FSTs). This gives rise to a hierarchy of stream 'degrees' somewhat analogous to the recursion-theoretic degrees of unsolvability (see further Shoenfield [50]). It is the structure and properties of this partial order of degrees that we are interested in. As we shall see, this hierarchy exhibits a variety of unique properties that set it apart from the usual complexity measures for streams.

We explain the notion of finite state transducers, and introduce the partial order of stream degrees. We then discuss and motivate this hierarchy, and compare it to the common complexity measures for streams. Then we sketch a few initial results, and collect some open questions.

Finite State Transducers. A *finite state transducer (FST)* is a finite automaton which reads the input stream letter by letter, in each step producing an output word and changing its state. An example of an FST is depicted in Figure 6, where we write '$a|w$' along the transitions to indicate that the input letter is a and the output word is w. This FST computes the first difference of the input stream. For example, it reduces the Thue-Morse sequence M to $\mathsf{T} = 1011101010111011\cdots$, the period doubling sequence (see Section 2.3).

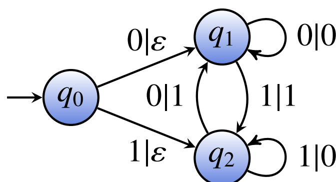


Figure 6: A finite state transducer realizing the difference Δ of consecutive bits modulo 2: $\Delta(abw) = (b - a\ (mod\ 2))\Delta(bw)$ for all $a, b \in \{0, 1\}$ and $w \in \{0, 1\}^\omega$.

The working of an FST on a stream is intuitively clear and it is easy to render it formally. At the end of the paper, in our nutshell introduction to infinitary rewriting, we show how the working of an FST on a stream can alternatively be phrased as infinitary rewriting to the output normal form.

4.1 A Hierarchy of Streams

Finite state transducers transform streams to streams (or finite words[1]). Thereby transducers give rise to a preorder $\trianglerighteq$ on the set of all streams: for streams u and v, we define $u \trianglerighteq v$, u is *reducible* to v, by:

$$u \trianglerighteq v \quad \iff \quad \text{there is an FST that transforms } u \text{ into } v$$

We write $u \trianglelefteq\trianglerighteq v$ if both a forth and a back transformation is possible, that is, $\trianglelefteq\trianglerighteq \;=\; \trianglelefteq \cap \trianglerighteq$. It is easily checked that $\trianglelefteq\trianglerighteq$ forms an equivalence relation, and we refer to the equivalence classes of $\trianglelefteq\trianglerighteq$ as *degrees*. The reducibility relation $\trianglerighteq$ induces a partial order on the degrees.

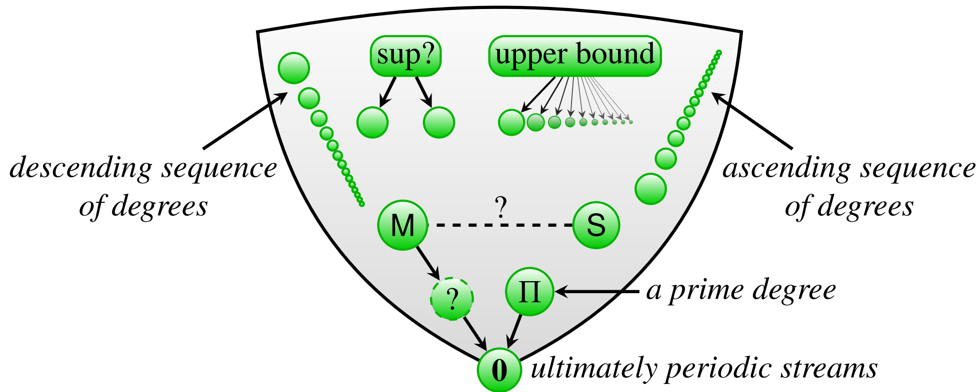


Figure 7: The partial order of stream degrees. Question marks indicate open problems. Here M *is the Thue-Morse sequence,* S *is the Sierpiński sequence, and* $\Pi = 11010010001\cdots$ *is the sequence of 'rarified ones' defined in* (1) *below.*

Figure 7 sketches a few initial results (obtained in [29]) and a few open questions. The preliminary results are: the hierarchy is not dense, not well-founded, there exist no maximal degrees, and a set of degrees has an upper bound if and only if the set is countable. The morphic degrees, and the computable degrees form interesting subhierarchies. The subhierarchy of computable degrees turns out to have a maximum degree. An interesting notion is that of a 'prime degree', an indivisible or minimal degree, see below.

Remark 4.1. We emphasize that it is important that the output given by a state transition is allowed to be a *word* over the output alphabet, and not just a single

[1]The result of the transformation is finite if the transducer outputs the empty word ε for almost all letters of the input stream. We are interested in streams only since the set of finite words would merely entail two spurious extra sub-bottom degrees of our hierarchy, one for finite non-empty words and one for the empty word.

letter or the empty word ε, although that may also be the case. Finite state transducers generalize the class of Mealy machines; the latter are restricted to output of precisely one letter in each step. For instance, the transducer shown in Figure 6 is not a Mealy machine, and there exists no Mealy machine implementing this transformation. Belov [8] independently studied the hierarchy arising from Mealy machines; this hierarchy however, does not have the nice properties that we envisage. In particular, the equivalence induced by Mealy machine transduction (forth and back) is not invariant under insertion or removal of finite (possibly scattered) subwords of a stream.

0

The bottom degree $\mathbf{0}$ is formed by the ultimately periodic streams, that is, all streams σ of the form $\sigma = \tau\gamma\gamma\gamma\ldots$ for finite τ, γ. Every stream can be reduced to any ultimately periodic stream $\sigma = \tau\gamma\gamma\gamma\ldots$ by an FST of the form displayed in Figure 8 consisting of just two states.

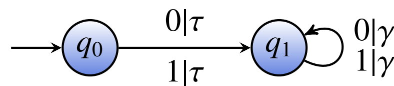


Figure 8: An FST reducing any stream to the ultimately periodic stream $\tau\gamma\gamma\gamma\cdots$.

upper bound

Every pair of degrees σ and τ has an upper bound. This upper bound can be constructed as follows: pick streams $u \in \sigma$ and $v \in \tau$. Let the stream $\mathsf{zip}(u, v)$ be obtained by alternatingly interleaving the elements of the streams u and v, that is:

$$\mathsf{zip}(u(0)\, u(1)\, u(2) \ldots,\; v(0)\, v(1)\, v(2) \ldots) = u(0)\, v(0)\, u(1)\, v(1)\, u(2)\, v(2) \ldots$$

Then the stream $\mathsf{zip}(u, v)$ that can be reduced to both u and v:

$$\mathsf{zip}(u, v) \rhd u \qquad\qquad \mathsf{zip}(u, v) \rhd v$$

by the FSTs shown in Figures 9 and 10, respectively. As a consequence, the degree of the stream $\mathsf{zip}(u, v)$ is an upper bound for the degrees σ and τ (the degrees of of u and v).

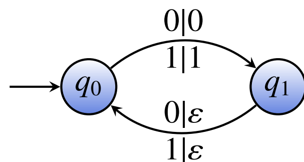


Figure 9: FST for $\mathsf{zip}(u, v) \rhd u$.

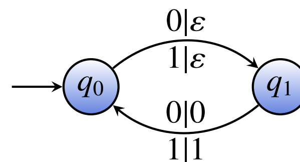
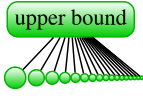


Figure 10: FST for $\mathsf{zip}(u, v) \rhd v$.



As the set of all FSTs is countable, it follows that every degree is countable, and that every degree can only have countably many degrees below it. Let $\{w_0, w_1, w_2, \ldots\}$ be a countable set of streams. Then we can obtain an upper bound by interleaving the streams in the following way:

$$\mathsf{zip}(w_0, \mathsf{zip}(w_1, \mathsf{zip}(w_2, \ldots))) \;,$$

see also Figure 11. As an immediate consequence we obtain that a set of degrees

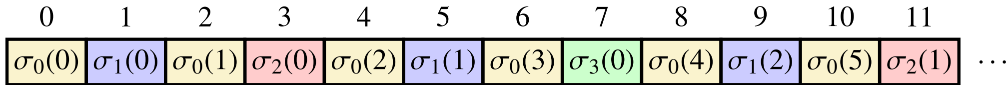
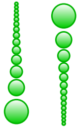


Figure 11: Zipping a countably infinite family of streams.

has an upper bound if and only if the set is countable.



There are no maximum degrees, i.e., for each degree a strictly larger one can be constructed. First note that the set of all degrees is uncountable as every degree is countable but there are uncountably many streams. Let σ be a degree. We show that σ is not maximal. Since there are uncountably many degrees, we can find a degree τ such that $\sigma \not\trianglerighteq \tau$, and then their upper bound is higher than σ. This establishes a non-constructive argument that every degree start an infinite ascending chain. A constructive example is the following [29]:

$$\begin{aligned}
&\ldots \\
\underset{\neq}{\rhd}\; A_3 &= 1(10)^3\, 1(100)^3\, 1(10000)^3\, 1(100000000)^3 \ldots \\
\underset{\neq}{\rhd}\; A_2 &= 1(10)^2\, 1(100)^2\, 1(10000)^2\, 1(100000000)^2 \ldots \\
\underset{\neq}{\rhd}\; A_1 &= 110\, 1100\, 110000\, 1100000000 \ldots \\
\underset{\neq}{\rhd}\; A_0 &= 111111 \ldots
\end{aligned}$$

forms an infinite ascending chain. For an infinite descending chain, consider:

$$\begin{aligned}
D_0 &= 10^{2^0} 10^{2^1} 10^{2^2} 10^{2^3} 10^{2^4} 10^{2^5} 10^{2^6} \ldots \\
\underset{\neq}{\rhd}\; D_1 &= 10^{2^0} 10^{2^2} 10^{2^4} 10^{2^6} 10^{2^8} 10^{2^{10}} 10^{2^{12}} \ldots \\
\underset{\neq}{\rhd}\; D_2 &= 10^{2^0} 10^{2^4} 10^{2^8} 10^{2^{12}} 10^{2^{16}} 10^{2^{20}} 10^{2^{24}} \ldots \\
\underset{\neq}{\rhd}\; &\ldots
\end{aligned}$$

Note that $A_1 \lhd\!\rhd D_0$ and so their degree has an infinite ascending chain as well as an infinite descending chain.

Prime Degrees. An interesting notion that suggests itself is that of a *prime degree*: a stream σ is prime if there exists no stream τ whose degree is strictly intermediate between that of σ and the bottom degree $\mathbf{0}$. Thus the prime degrees reduce only to $\mathbf{0}$ or themselves. An example of a stream of prime degree is the following sequence called *rarified ones* (see [29]):

$$\Pi = 1101001000100001000001\ldots$$

which can be obtained as the image of the fixed point of the morphism

$$a \mapsto a1 \qquad 1 \mapsto 01 \qquad 0 \mapsto 0 \tag{1}$$

on the starting word a, under the coding $a \mapsto 1$, $0 \mapsto 0$, $1 \mapsto 1$; thus Π is a morphic stream. Intuitively, the information content of the stream Π is 'indivisible': whatever FST is applied on this stream, either the result is eventually periodic (the structure is entirely destroyed), or there is enough structure left for an FST to reconstruct the original stream.

4.2 Motivation

Finite state automata and finite state transducers are ubiquitous in computer science and computational linguistics. Surprisingly, very little is known about the reducibility relation that finite state transducers induce on streams: *Given streams u and v, is there an FST reducing u to v?* This is a challenging question, especially for the case of proving non-reducibility, for example of morphic streams. At present there are no methods available for this problem.

Except for the importance of finite state transducers, the study of the FST-hierarchy is intriguing due to its unique characteristics that set it apart from all existing approaches to measuring stream complexity. The hierarchy is *robust* and *fine grained* at the same time:

(i) It is *robust* under the insertion and removal of arbitrary finite amount of elements in the streams (for example, cutting-off of prefixes), and change of encoding. In some sense, the FST-hierarchy classifies streams by their *invariant infinite patterns*.

(ii) Thus on the one hand we have a fairly general notion of transformation, but on the other hand it is not too strong, leading to a *fine grained* structure of degrees. By contrast, allowing Turing Machines for the transformations would be far too strong. Most of the interesting streams are computable, and with Turing Machines any pair of such computable streams could be transformed into each other, and our endeavour would trivialize. The hierarchy of degrees arising from Turing Machines is the well-known recursion

theoretic degrees of unsolvability that classify the intrinsic 'difficulty' of a set of natural numbers (Shoenfield [50]).

4.3 Comparison with Standard Complexity Measures

The hierarchy of stream degrees is very different from the common approaches for measuring the complexity of streams: (i) the recursion-theoretic hierarchy identifies all computable streams, (ii) Kolmogorov complexity can be increased arbitrarily by the insertion of elements, and (iii) subword complexity can be trivial (linear) even for non-computable streams. The subword complexity of a stream σ is a function $\xi : \mathbb{N} \to \mathbb{N}$ such that $\xi(n)$ is the number of subwords of length n occurring in σ. The Kolmogorov complexity $\mathcal{K}(\sigma) \in \mathbb{N}$ of a computable stream σ is the length of the shortest program computing σ.

4.4 Open Questions

The study of the hierarchy concerns the very core of finite state transducers: the transducibility relation they induce on streams. Surprisingly, although finite automata and finite state transducers are ubiquitous in computer science, only little is known about this transducibility relation. Given streams σ and τ, there are hardly any methods available to determine whether $\sigma \trianglerighteq \tau$, that is, can σ be reduced to τ via some FST? This is challenging especially for the case of proving that σ cannot be transduced to τ. In special cases, pumping lemma arguments may work, but they fail for basically all streams that we are interested in (e.g. morphic streams). If σ is morphic and τ not, then we can make use of a famous theorem by Dekking stating that morphic streams are closed under FST transduction. In all other cases, we are lost.

We mention a few intriguing open questions:

(i) What is the structure of the partial order of degrees?

(ii) How many prime degrees exist?

(iii) Is the degree of the Thue-Morse sequence M prime?

(iv) How do the degrees of some well-known streams compare? For example, are M and S of the same degree? Here S is the *Sierpiński* stream, see Figure 5.

(v) How to prove non-reducibility $\sigma \ntrianglerighteq \tau$ (e.g. for morphic streams σ, τ)?

(vi) Do the structures displayed in Figure 12 exist?

Figure 12: Possible structures in the FST-hierarchy: a diamond, and a line. The arrows $S \to T$ mean $S \gneq T$. Using transitivity of $\rhd$ we leave some arrows implicit. Moreover, we assume that if S is a degree and $S \rhd T$, then T is depicted as well. In particular there are no intermediate degrees between two displayed nodes connected by an arrow.

(vii) What is the structure of the partial order of degrees restricted to different families of streams? E.g. the families of computable or morphic streams?

5 Zip Goes a Million

Elias Howe, the inventor of the sewing machine, patented in 1851 an 'automatic, continuous clothing closure', later known as zip or zipper. The name is an onomatopoeia, a sound imitation. Apart from its use as a coding device for American postal services around 1950, 'zip' made its entrance also in the world of light culture, giving rise in 1919 to the Broadway play *Zip! goes a million*, and a 1954 remake as a London musical *Zip goes a million*, the former unsuccessful, but the latter quite successful. The plot was based on the book *Brewster's millions* from 1902, describing the problem how to loose \$1000 000 in order to gain \$7000 000.

In this section we are concerned with the more serious culture of stream specifications, and we will endeavour to describe how the stream operator 'zip', in the literature known as *perfect shuffle*, can profitably be used to give an elegant alternative definition of automatic sequences, and how its use suggests some further challenging questions. First, we will consider zip-specifications, to be followed by an excursion to mix-automatic sequences.

5.1 Destructing Automatic Sequences

For $i, k \in \mathbb{N}$, we define *projection functions* $\pi_{i,k} : \Delta^\omega \to \Delta^\omega$ by the equations:

$$\pi_{0,k}(x : \sigma) = x : \pi_{k-1,k}(\sigma) \qquad\qquad \pi_{i+1,k}(x : \sigma) = \pi_{i,k}(\sigma) \tag{2}$$

So $\pi_{i,k}(\sigma)$ is an arithmetic subsequence of σ:

$$\pi_{i,k}(\sigma) = \sigma(i)\ \sigma(i + k)\ \sigma(i + 2k) \cdots$$

The functions $\pi_{0,k}, \ldots, \pi_{k-1,k}$ are the *destructors* of zip_k, that is, we have

$$\pi_{i,k}(\mathsf{zip}_k(\sigma_0, \ldots, \sigma_{k-1})) = \sigma_i \qquad\qquad (0 \le i < k) \tag{3}$$

Definition 5.1. Let Φ be a set of stream functions $\Delta^\omega \to \Delta^\omega$, and $\sigma \in \Delta^\omega$ a stream. The Φ-*derivatives of* σ are the smallest set $D \subseteq \Delta^\omega$ such that: $\sigma \in D$, and $\phi(\tau) \in D$ whenever $\tau \in D$ and $\phi \in \Phi$.

The derivatives provide an elegant, iterative way of defining the k-kernels [3]. The k-*kernel* of a sequence σ is the set of $\{\pi_{0,k}, \ldots, \pi_{k-1,k}\}$-derivatives of σ. Then the well-known characterization of k-automatic sequences via finite k-kernels can be phrased as follows:

Theorem 5.1. *Let $k \in \mathbb{N}$. A stream $\sigma \in \Delta^\omega$, is k-automatic if and only if the set of $\{\pi_{0,k}, \ldots, \pi_{k-1,k}\}$-derivatives of σ is finite.*

This characterization can be generalized to other sets of derivatives:

Theorem 5.2. *Let $k > 1$. Let Φ be a finite set of projection functions π_{i,k^n} with $i, n \in \mathbb{N}$ and $n \geq 1$ such that the set $\mathbb{N} \setminus \{a + nb \mid \pi_{a,b} \in \Phi,\ n \in \mathbb{N}\}$ is finite. Then $\sigma \in \Delta^\omega$ is k-automatic if and only if the set of Φ-derivatives of σ is finite.*

For example, the set of $\{\pi_{1,k}, \ldots, \pi_{k,k}\}$-derivatives is finite if and only if the sequence is k-automatic, see further [27]. The increased flexibility in choosing the projection functions Φ can help to simplify proofs and disproofs of k-automaticity. In the terminology of [40], the derivatives $\{\pi_{0,k}, \ldots, \pi_{k-1,k}\}$ are a *complete set of cooperations*, forming (together with head) a *cobasis* for streams. This cobasis gives rise to a *final coalgebra* for automatic sequences, see [41, 32].

5.2 Automatic Sequences via Zip-Specifications

Automatic sequences can be defined via term rewriting, or equational specifications in a restricted format: *zip-specifications*, see further [32]. For example, the Thue–Morse sequence is obtained by the succinct zip-specification

$$\mathsf{M} = 0 : \mathsf{X} \qquad \mathsf{X} = 1 : \mathsf{zip}(\mathsf{X}, \mathsf{Y}) \qquad \mathsf{Y} = 0 : \mathsf{zip}(\mathsf{Y}, \mathsf{X}) \tag{4}$$

(We obtain a term rewriting system by orienting the equations from left to right.) Here the function zip interleaves the elements of two streams alternatingly, also known as *perfect shuffle*:

$$\mathsf{zip}(\sigma_0 : \sigma_1 : \sigma_2 : \ldots, \tau_0 : \tau_1 : \tau_2 : \ldots) = \sigma_0 : \tau_0 : \sigma_1 : \tau_1 : \sigma_2 : \tau_2 : \ldots$$

As a term rewriting rule, 'zip' can be defined as follows:

$$\mathsf{zip}(x : \sigma, \tau) = x : \mathsf{zip}(\tau, \sigma)$$

Naturally, the function zip can be generalized to interleaving k arguments:

$$\mathsf{zip}_k(x_1 : s_1, s_2, \ldots, s_k) = x_1 : \mathsf{zip}_k(s_2, \ldots, s_k, s_1)$$

A *zip-k specification* consists of recursion equations $X_1 = r_1, \ldots, X_n = r_n$ such that $X_1, \ldots, X_n$ are recursion variables, and the right-hand sides $r_1, \ldots, r_n$ are terms inductively generated by the following grammar:

$$G_k ::= X_i \mid a : G_k \mid \mathsf{zip}_k(G_k, \ldots, G_k) \qquad (1 \leq i \leq n, a \in \Sigma)$$

That is, the right-hand sides of the equations are built from recursion variables, prefixing a symbol at the head of a stream, and the stream function zip_k.

It turns out, that the class of streams definable by zip-k specifications is precisely the class of k-automatic sequences [32]. Thus zip-specifications provide a term rewriting syntax for automatic sequences.

Let us explain why zip-specifications give rise to automatic sequences at the example of the specification (4). We show that the sequence is 2-automatic by computing the 2-kernel, the set of $\{\pi_{0,2}, \pi_{1,2}\}$-derivatives. To this end, we compute the derivatives symbolically using the equations (2) and (3), and unfolding stream constants defined by (4) whenever necessary. For better readability we write even for $\pi_{0,2}$ and odd for $\pi_{1,2}$. We obtain:

$$\begin{aligned} \mathsf{even}(\mathsf{M}) &= \mathsf{even}(0 : \mathsf{X}) \\ &= 0 : \mathsf{odd}(\mathsf{X}) \\ &= 0 : \mathsf{odd}(1 : \mathsf{zip}(\mathsf{X}, \mathsf{Y})) \\ &= 0 : \mathsf{even}(\mathsf{zip}(\mathsf{X}, \mathsf{Y})) \\ &= 0 : \mathsf{X} \\ &= \mathsf{M} \end{aligned} \qquad \begin{aligned} \mathsf{odd}(\mathsf{M}) &= \mathsf{odd}(0 : \mathsf{X}) \\ &= \mathsf{even}(\mathsf{X}) \\ &= \mathsf{even}(1 : \mathsf{zip}(\mathsf{X}, \mathsf{Y})) \\ &= 1 : \mathsf{odd}(\mathsf{zip}(\mathsf{X}, \mathsf{Y})) \\ &= 1 : \mathsf{Y} \end{aligned}$$

Likewise $\mathsf{even}(1 : \mathsf{Y}) = 1 : \mathsf{Y}$ and $\mathsf{odd}(1 : \mathsf{Y}) = \mathsf{M}$. Thus the set of derivatives is finite, namely $\{\mathsf{M},\ 1 : \mathsf{Y}\}$, and by Theorem 5.1, the stream M is 2-automatic.

The derivatives of a stream specification can be illustrated using a graph. The nodes of the graph are the $\{\mathsf{even}, \mathsf{odd}\}$-derivatives. The edges represent the derivative relation and are labeled with even and odd, correspondingly. The graph corresponding to the $\{\mathsf{even}, \mathsf{odd}\}$-derivatives of M is shown in Figure 13. If we replace in this graph 'even' by 0 and 'odd' by 1, we obtain a 2-DFAO that generates the Thue–Morse sequence M as automatic sequence.

5.3 Mix-Automatic Sequences

The correspondence of k-automatic sequences with zip-k specifications, leads to the natural question: *What class of sequences is obtained when allowing zips of*

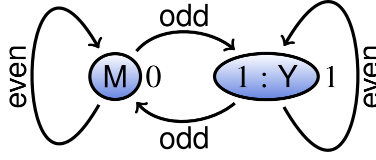


Figure 13: The ⟨head, even, odd⟩*-coalgebra induced by* M.

different arities in the same specification? For example:

$$\mathsf{Z} = \mathsf{zip}_2(0 : \mathsf{Z}, \mathsf{Y}) \qquad \mathsf{Y} = 1 : \mathsf{zip}_3(\mathsf{Z}, \mathsf{Y}, 0 : \mathsf{Z})$$

We call the arising class of sequences *mix-automatic*. It forms a proper extension of the class of automatic sequences [32], and is in contrast to automatic sequences, closed under zipping (perfect shuffle). For example, zipping a 2-automatic and a 3-automatic sequence, both not ultimately periodic, yields a non-automatic (yet mix-automatic) sequence. While automatic sequences have at most linear subword complexity, and morphic sequences at most quadratic, the subword complexity of mix-automatic sequences can exceed an arbitrary polynomial, see [22].

Mix-automatic sequences can be defined via a generalization of k-DFAOs allowing that the input alphabet depends on the current state. We call these automata *mix-DFAOs*. Let us consider the example of a mix-DFAO shown in Figure 14.

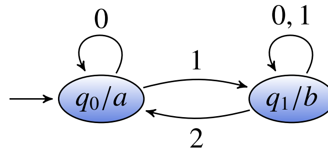


Figure 14: An example of a mix-DFAO.

The state q_0 has two outgoing edges, reflecting the input alphabet $\{0, 1\}$, while q_1 has three outgoing edges, reflecting the input alphabet $\{0, 1, 2\}$.

Dynamic Radix Numeration Systems. The numeration systems these automata operate on are no longer the standard base-k representation. The corresponding number systems are called *dynamic radix numeration systems*, a generalization of *mixed radix* numeration systems [38]. In this number representation, the base for each digit is determined depending on the lower-significance digits. Thus we let the automata read from the least to the most significant digit (from right to left). We write $(n)_M$ for the number representation of n that serves as input for the automaton M. For M the automaton from Figure 14, the representations of the first eight numbers are

$$(0)_M = \varepsilon \qquad (2)_M = 1_2 0_2 \qquad (4)_M = 1_2 0_2 0_2 \qquad (6)_M = 1_3 1_2 0_2$$

$(1)_M = 1_2$ $(3)_M = 1_3 1_2$ $(5)_M = 2_3 1_2$ $(7)_M = 1_3 0_3 1_2$

where a subscript b (not part of the number representation) in d_b indicates the base employed for d. Let us explain this at the example $(17)_M = 1_2 0_2 2_3 1_2$. Knowing the base for each digit, we can reconstruct the value of the representation as follows: $17 = 1 \cdot 2 \cdot 3 \cdot 2 + 0 \cdot 3 \cdot 2 + 2 \cdot 2 + 1$ where each digit is multiplied with the product of the bases of the lower digits. Given just the representation 1021, the base of each of the digits is determined by the input alphabet of the state of the automaton reading the digit. The states q_0 and q_1 of M have input alphabets $\{0, 1\}$ and $\{0, 1, 2\}$ and thus expect the input in base 2 and 3, respectively. When reading 1021 (right to left) the automaton M visits the states q_0, q_1, q_0, q_0 and q_1. Annotating the input digits with the state of the automaton when reading the digit, we obtain $1_{q_0} 0_{q_0} 2_{q_1} 1_{q_0}$, and mapping states to their expected base yields $1_2 0_2 2_3 1_2$.

Let M be a mix-DFAO. Then every $n \in \mathbb{N}$ has a unique representation $(n)_M = d_m \cdots d_0$ (without leading zeros). This representation can be computed as follows. Assume that we have determined the value of the digits $d_{i-1} \cdots d_0$ with corresponding bases $b_{i-1} \cdots b_0$. The base b_i of digit d_i is determined by the input alphabet of the state of the automaton after reading $d_{i-1} \cdots d_0$ (right to left), and digit d_i is the remainder of the division of $n - \sum_{0 \le j < i} d_j (b_{j-1} \cdots b_1 \cdot b_0)$ by b_i.

A mix-DFAO M gives rise to a mix-automatic sequence $w \in \Delta^\omega$ as follows: for every $n \in \mathbb{N}$, $w(n)$ is the output of M when reading $(n)_M$.

5.4 Research Questions

The concept of mix-automatic sequences and dynamic-radix numeration systems are very recent, and many interesting questions remain. We highlight three particularly intriguing, and challenging questions:

(1) (J.-P. Allouche) Characterize the intersection of mix-automatic and morphic sequences. (Note that at least all automatic sequences are in.)

(2) Is the following problem decidable: Given two mix-DFAOs, do they generate the same sequence?

(3) Can Cobham's Theorem (below) be generalized to mix-automatic sequences?

Theorem 5.3 (Cobham's Theorem [11])**.** *Let $k, \ell \ge 2$ be multiplicatively independent (i.e., $k^a \neq \ell^b$, for all $a, b > 0$), and let $w \in \Delta^\omega$ be both k- and ℓ-automatic. Then w is ultimately periodic.*

For more details, we refer to [32, 22].

6 Periodically Iterated Morphisms

Infinite words obtained from periodically iterating multiple morphisms, so-called PD0L words, have been studied in [15, 16, 42, 10]. In particular, Lepistö [42] shows that for all $r \in \mathbb{R}$ there is a PD0L word whose subword complexity is in $\Omega(n^r)$. Subword complexity [31, 1, 3] is a natural characteristic of streams. The subword complexity of a stream $\boldsymbol{u}$ is a function $\mathbb{N} \to \mathbb{N}$ mapping n to the number of n-length words that occur in $\boldsymbol{u}$. It is well-known that CD0L words can have at most quadratic subword complexity [19]. Hence from Lepistö's result [42] it follows that there are PD0L words that are not CD0L. Cassaigne and Karhumäki [10] show that all Toeplitz words are PD0L words, and that some of them have subword complexity in $\Omega(n^r)$ for $r > 2$, thus forming an alternative proof of what was established in [42]. We note that, conversely, the existence of CD0L words that are not PD0L words was shown in [15].

Recently, the first two authors [27] have shown that PD0L words can even exhibit exponential subword complexity, answering a question raised by Lepistö [42], Cassaigne and Karhumäki [10] on the existence of such words. Another open problem concerned the decidability of the first-order theories of PD0L words [46]; from [27] it follows that is already undecidable whether a certain letter occurs in a PD0L word. This stands in contrast to the situation for D0L words (purely morphic words), which are known to have at most quadratic subword complexity, and for which the monadic theory is decidable.

Actually, [27] shows a stronger result, which can be paraphrased by

Periodically iterated morphisms are Turing-complete.

The proof of this result is based on an encoding of Fractran [12, 13] programs as PD0L systems. Fractran is a very simple, yet Turing complete programming language invented by John Horton Conway. We will here describe the essence of this encoding. In particular we show how the halting problem of Fractran programs is translated to the productivity problem for erasing PD0L systems, i.e., systems where morphisms are allowed to map letters to the empty word ε. A PD0L system is called *productive* if it generates an infinite word. Productivity is discussed in the wider context of term rewriting in Section 7. In [27] this construction is extended to non-erasing PD0L systems that also record the output of the Fractran program, so that any computable stream can be 'embedded' in a PD0L word. Let us first give a nutshell introduction to Fractran.

6.1 Fractran

A Fractran program F is a finite list of fractions

$$F = \frac{n_1}{d_1}, \dots, \frac{n_k}{d_k} \tag{5}$$

with n_i, d_i positive integers. Let $f_i = \frac{n_i}{d_i}$. The action of F on an input integer $N \geq 1$ is to multiply N by the first 'applicable' fraction f_i, that is, the fraction f_i with i the least index such that the product $N' = N \cdot f_i$ is an integer again, and then to continue with N'. The program halts if there is no applicable fraction for the current integer N. For example, we consider the program

$$F = \frac{5}{2 \cdot 3}, \frac{1}{2}, \frac{1}{3}$$

and the run of F on input $N = 2^3 3^5$:

$$2^3 3^5 \to 2^2 3^4 5^1 \to 2^1 3^3 5^2 \to 2^0 3^2 5^3 \to 2^0 3^1 5^3 \to 2^0 3^0 5^3 .$$

Each multiplication by $\frac{5}{6}$ decrements the exponents of 2 and 3 while incrementing the exponent of 5. Once $\frac{5}{6}$ is no longer applicable, i.e., when one of the exponents of 2 and 3 in the prime factorization of the current integer N equals 0, the other is set to 0 as well. Hence, executing F on $N = 2^a\, 3^b$ halts after $\max(a, b)$ steps with $5^{\min(a,b)}$.

Thus the prime numbers that occur as factors in the numerators and denominators of a Fractran program can be regarded as registers, and if the current working integer is $N = 2^a\, 3^b\, 5^c\, \dots$ we can say that register 2 holds a, register 3 holds b, and so on.

The real power of Fractran comes from the use of prime exponents as *states*. To explain this, we temporarily let programs consist of multiple lines of the form

$$\alpha : \ \frac{n_1}{d_1} \to \alpha_1,\ \frac{n_2}{d_2} \to \alpha_2,\ \dots,\ \frac{n_m}{d_m} \to \alpha_m \tag{6}$$

forming the instructions for the program in state α: multiply N with the first applicable fraction $\frac{n_i}{d_i}$ and proceed in state α_i, or terminate if no fraction is applicable. We call the states $\alpha_1, \dots, \alpha_m$ in (6) the *successors* of α, and we say a state is *looping* if it is its own successor.

The program P_{add} given by the lines

$$\alpha : \frac{2 \cdot 5}{3} \to \alpha,\ \frac{1}{1} \to \beta \qquad \text{and} \qquad \beta : \frac{3}{5} \to \beta$$

realizes addition; running P_{add} in state α on $N = 2^a 3^b$ terminates in state β with $2^{a+b} 3^b$.

A program with n lines is called a *Fractran-n program*. A flat list of fractions $f_1, \ldots, f_k$ now is a shorthand for the Fractran-1 program $\alpha : f_1 \to \alpha,\ f_2 \to \alpha,\ \ldots,\ f_k \to \alpha$. Conway [13] explains how every Fractran-n program ($n \geq 2$) can be compiled into a Fractran-1 program, using the following steps:

(i) For every looping state α, introduce a 'mirror' state $\reflectbox{\ensuremath{\alpha}}$, substitute $\reflectbox{\ensuremath{\alpha}}$ for all occurrences of α in the right-hand sides of its program line, and add the line

$$\reflectbox{\ensuremath{\alpha}} : \frac{1}{1} \to \alpha$$

(ii) Replace state identifiers α by 'fresh' prime numbers.

(iii) For every line of the form (6) append the following fractions:

$$\frac{n_1 \cdot \alpha_1}{d_1 \cdot \alpha},\ \frac{n_2 \cdot \alpha_2}{d_2 \cdot \alpha},\ \ldots,\ \frac{n_k \cdot \alpha_m}{d_m \cdot \alpha}$$

(preserving the order) to the list of fractions constructed so far.

We explain these steps on the program P_{add}. Step (i) of splitting loops, results in

$$\alpha : \frac{2 \cdot 5}{3} \to \reflectbox{\ensuremath{\alpha}},\ \frac{1}{1} \to \beta \qquad\qquad \beta : \frac{3}{5} \to \reflectbox{\ensuremath{\beta}}$$

$$\reflectbox{\ensuremath{\alpha}} : \frac{1}{1} \to \alpha \qquad\qquad \reflectbox{\ensuremath{\beta}} : \frac{1}{1} \to \beta\,.$$

In step (ii), we introduce 'fresh' primes to serve as state indicators; for example, $\langle \alpha, \reflectbox{\ensuremath{\alpha}}, \beta, \reflectbox{\ensuremath{\beta}} \rangle = \langle 7, 11, 13, 17 \rangle$. Third, we replace lines by fractions, to obtain the program

$$F_{\mathsf{add}} = \frac{2 \cdot 5 \cdot \reflectbox{\ensuremath{\alpha}}}{3 \cdot \alpha},\ \frac{\alpha}{\reflectbox{\ensuremath{\alpha}}},\ \frac{\beta}{\alpha},\ \frac{3 \cdot \reflectbox{\ensuremath{\beta}}}{5 \cdot \beta},\ \frac{\beta}{\reflectbox{\ensuremath{\beta}}}\,.$$

Then indeed the run of F_{add} on $2^a 3^b \alpha$ ends in $2^{a+b} 3^b \beta$.

For 'sensible' programs any state indicator has value 0 ('off') or 1 ('on'), and the program is always in exactly one state at a time. Hence, if a program F uses primes $r_1, \ldots, r_p$ for storage, and primes $\alpha_1, \ldots, \alpha_q$ for control, at any instant the entire *configuration* of F (= register contents + state) is uniquely represented by the current working integer N

$$N = r_1^{e_1}\, r_2^{e_2}\, \cdots\, r_p^{e_p}\, \alpha_j$$

for some integers $e_1, \ldots, e_p \geq 0$ and $1 \leq j \leq q$.

The reason to employ two state indicators α and ɒ to break self-loops in step (i), is that each state indicator is consumed whenever it is tested, and so we need a secondary indicator ɒ to say “continue in the current state”. This secondary indicator ɒ is swapped back to the primary indicator α in the next instruction, and the loop continues.

The halting problem for Fractran programs is undecidable (implicit in [13], explicit in [21, 32]).

Proposition 6.1. *The input-2 halting problem for Fractran programs, that is, deciding whether a program halts for the starting integer $N = 2$, is Σ^0_1-complete.*

6.2 Encoding Fractran Programs as PD0L Systems

Definition 6.1. Let $H = \langle h_0, \ldots, h_{p-1} \rangle$ be a tuple of morphisms $h_i : \Sigma^* \to \Sigma^*$. We define the map $H : \Sigma^* \to \Sigma^*$ as follows:

$$H(a_0 a_1 \cdots a_n) = u_0 u_1 \cdots u_n$$
$$\text{where } u_i = h_k(a_i), \text{ with } k \equiv i \pmod{p} \text{ and } k \in \mathbb{N}_{<p}.$$

If $s \in \Sigma^*$ is such that $s \le H(s)$, then the triple $\mathcal{H} = \langle \Sigma, H, s \rangle$ is called a *PD0L system*. Then in the metric space $\langle \Sigma^\infty, \mathrm{d} \rangle$ the limit

$$H^\omega(s) = \lim_{i \to \infty} H^i(s)$$

exists, and we call $H^\omega(s)$ the *PD0L word* generated by $\mathcal{H}$. We say that $\mathcal{H}$ is *productive* if $H^\omega(s)$ is infinite, and $\mathcal{H}$ is *erasing* if some of its morphisms h_i are erasing. If x is a PD0L word generated by p morphisms, and $x = uvy$ for some $u, v \in \Sigma^*$ and $y \in \Sigma^\infty$, we say that the factor v of x occurs at *morphism index i* when $i \in \mathbb{N}_{<p}$ and $i \equiv |u| \pmod{p}$.

Example 6.1. Cassaigne and Karhumäki [10] show that all Toeplitz words are PD0L words. Consider for example the Toeplitz word $\mathsf{T}_y = 121211221112221\cdots$ generated by the seed word $y = 12???$ (see Section 2.3). Then we have $\mathsf{T}_y = H^\omega(1)$ where $H = \langle h_0, h_1, h_2 \rangle$ and $h_0(a) = 12a$ and $h_1(a) = h_2(a) = a$ for all $a \in \{1, 2\}$. Moreover, from [10, Theorem 5] it follows that the subword complexity of T_y is in $\Theta(n^r)$ with $r = \frac{\log 5}{\log 5 - \log 3} \simeq 3.15066$, thus forming an alternative proof (since morphic words have quadratic subword complexity at most) of what was established by Lepistö [42]: there are PD0L words that are not morphic.

In [27] it is shown that the problem of deciding productivity of erasing PD0L systems is undecidable. The idea is to encode a given Fractran program F as a PD0L system $\mathcal{H}_F = \langle \Sigma, H, s \rangle$ such that $H^\omega(s)$ is infinite if and only if F does not terminate on input 2.

Definition 6.2. Let $F = \frac{n_1}{d}, \dots, \frac{n_k}{d}$ be a Fractran program (every program can be brought into this form by taking d the least common denominator of the fractions). We define the PD0L system $\mathcal{H}_F = \langle \Gamma, H, \mathsf{s} \rangle$ with

$$\Gamma = \left\{ \mathsf{s}, \llcorner\!\lrcorner, \mathsf{a}, \mathsf{A}, \mathsf{b}, \mathsf{B} \right\} \quad \text{and} \quad H = \langle h_0, \dots, h_{d-1} \rangle ,$$

where for every i with $0 \leq i < d$ the morphism $h_i : \Gamma^* \to \Gamma^*$ is defined by

$$\begin{aligned}
h_i(\mathsf{s}) &= \mathsf{s} \llcorner\!\lrcorner^{d-1} \mathsf{a}\,\mathsf{a}\,\mathsf{b} \llcorner\!\lrcorner^{d-1} \\
h_i(\llcorner\!\lrcorner) &= \varepsilon \\
h_i(\mathsf{a}) &= \begin{cases} \mathsf{A} \llcorner\!\lrcorner^{d-1} & \text{if } i = d-1 \\ \varepsilon & \text{otherwise} \end{cases} \\
h_i(\mathsf{b}) &= \mathsf{B} \llcorner\!\lrcorner^{d-1-i} \\
h_i(\mathsf{A}) &= \begin{cases} \mathsf{a}^{n_{\psi(i)}} & \text{if } \psi(i) \text{ is defined} \\ \varepsilon & \text{otherwise} \end{cases} \\
h_i(\mathsf{B}) &= \begin{cases} \mathsf{a}^{i \cdot \frac{n_{\psi(i)}}{d}} \ \mathsf{b} \llcorner\!\lrcorner^{d-1} & \text{if } \psi(i) \text{ is defined} \\ \varepsilon & \text{otherwise} \end{cases}
\end{aligned}$$

In [27] it is shown that productivity of the PD0L system $\mathcal{H}_F$ coincides with F running forever on input 2. Here we give some intuitive explanation, and illustrate the working of $\mathcal{H}_F$ on an example program F.

Let F be a Fractran program with common denominator d, and (finite or infinite) run $N_0, N_1, N_2, \dots$. Let $q_i \in \mathbb{N}$ and $r_i \in \mathbb{N}_{<d}$ such that $N_i = q_i d + r_i$, for all $i \geq 0$. We let x_n be the 'contribution' of the iteration H^{n+1}, i.e., x_n is such that $H^{n+1}(\mathsf{s}) = H^n(\mathsf{s}) x_n$. Then $H^{\omega}(\mathsf{s}) = \mathsf{s}\, x_0 x_1 x_2 \cdots$. We will display $H^{\omega}(\mathsf{s})$ in separate lines each corresponding to an x_n. The computation of the word $H^{\omega}(\mathsf{s})$ proceeds in two alternating phases: the transition from even to odd lines corresponds to division by d, and the transition from odd to even lines corresponds to multiplication by the currently applicable fraction $\frac{n_{\psi(N_i)}}{d}$. These phases are indicated by the use of lower- and uppercase letters, that is, $x_{2n} \in \{\llcorner\!\lrcorner, \mathsf{a}, \mathsf{b}\}^*$ and $x_{2n+1} \in \{\llcorner\!\lrcorner, \mathsf{A}, \mathsf{B}\}^*$, as can be seen from the definition of the morphisms. Now the intuition behind the alphabet symbols (in view of the defining rules of the morphisms) can be described as follows. We use s as the starting symbol, and the symbol $\llcorner\!\lrcorner$ is used to shift the morphism index of subsequent letters.

In every even line x_{2i}

(i) there is precisely one block of a's; this block occurs at morphism index 0 and is of length N_i, representing the current value N_i in the run of F;

(ii) b is a special marker for the end of a block of a's, and so is positioned at morphism index r_i, the remainder of dividing N_i by d.

In every odd line x_{2i+1}

(iii) the number of A's corresponds to the quotient q_i, and every occurrence of A is positioned at morphism index r_i;

(iv) B (also at morphism index r_i) takes care of the multiplication of the remainder r_i with $\frac{n_{\psi(N_i)}}{d}$. Then $\psi(N_i) = \psi(r_i)$ ensures that the morphism can select the right fraction to multiply with.

We illustrate the encoding by means of an example.

Example 6.2. Consider the Fractran program $\frac{9}{2}, \frac{5}{3}$, or equivalently $F = \frac{27}{6}, \frac{10}{6}$ and its finite run $2, 9, 15, 25$. Following Definition 6.2 we construct the PD0L system $\mathcal{H}_F = \langle \Gamma, H, \mathsf{s} \rangle$ with $H = \langle h_0, \ldots, h_5 \rangle$ and

$$
\begin{array}{ll}
h_i(\mathsf{s}) = \mathsf{s}\ \sqcup^5\ \mathsf{a}\,\mathsf{a}\,\mathsf{b}\ \sqcup^5 & \\
h_i(\sqcup) = \varepsilon & h_i(\mathsf{b}) = \mathsf{B}\ \sqcup^{5-i} \\
h_0(\mathsf{a}) = \ldots = h_4(\mathsf{a}) = \varepsilon & h_0(\mathsf{B}) = \mathsf{b}\ \sqcup^5 \\
h_5(\mathsf{a}) = \mathsf{A}\ \sqcup^5 & h_2(\mathsf{B}) = \mathsf{a}^9\ \mathsf{b}\ \sqcup^5 \\
h_0(\mathsf{A}) = h_2(\mathsf{A}) = h_4(\mathsf{A}) = \mathsf{a}^{27} & h_4(\mathsf{B}) = \mathsf{a}^{18}\ \mathsf{b}\ \sqcup^5 \\
h_3(\mathsf{A}) = \mathsf{a}^{10} & h_3(\mathsf{B}) = \mathsf{a}^5\ \mathsf{b}\ \sqcup^5 \\
h_1(\mathsf{A}) = h_5(\mathsf{A}) = \varepsilon & h_1(\mathsf{B}) = h_5(\mathsf{B}) = \varepsilon
\end{array}
$$

for $i \in \mathbb{N}_{<6}$. Then $H^\omega(\mathsf{s})$ is finite and the stepwise computation of this fixed point can be displayed as follows. To ease reading, we write below each letter its morphism index. Let z_n denote the morphism index of x_n. The word $H^\omega(\mathsf{s}) = \mathsf{s}\, x_0 x_1 \cdots$ is broken into lines in such a way that every line x_{n+1} is the image of the previous line x_n under H_{z_n} (except for the line x_0, which is the tail of the image of s under H_0).

$$
\begin{array}{rl}
 & \underset{0}{\mathsf{s}} \\
x_0 = & \underset{1}{\sqcup^5}\ \underset{0}{\mathsf{a}}\ \underset{1}{\mathsf{a}}\ \underset{2}{\mathsf{b}}\ \underset{3}{\sqcup^5} \\
x_1 = & \underset{2}{\mathsf{B}}\ \underset{3}{\sqcup^3} \\
x_2 = & \underset{0}{\mathsf{a}}\ \underset{1}{\mathsf{a}}\ \underset{2}{\mathsf{a}}\ \underset{3}{\mathsf{a}}\ \underset{4}{\mathsf{a}}\ \underset{5}{\mathsf{a}}\ \underset{0}{\mathsf{a}}\ \underset{1}{\mathsf{a}}\ \underset{2}{\mathsf{a}}\ \underset{3}{\mathsf{b}}\ \underset{4}{\sqcup^5} \\
x_3 = & \underset{3}{\mathsf{A}}\ \underset{4}{\sqcup^5}\ \underset{3}{\mathsf{B}}\ \underset{4}{\sqcup^2} \\
x_4 = & \underset{0}{\mathsf{a}}\ \underset{1}{\mathsf{a}}\ \underset{2}{\mathsf{a}}\ \underset{3}{\mathsf{a}}\ \underset{4}{\mathsf{a}}\ \underset{5}{\mathsf{a}}\ \underset{0}{\mathsf{a}}\ \underset{1}{\mathsf{a}}\ \underset{2}{\mathsf{a}}\ \underset{3}{\mathsf{a}}\ \underset{4}{\mathsf{a}}\ \underset{5}{\mathsf{a}}\ \underset{0}{\mathsf{a}}\ \underset{1}{\mathsf{a}}\ \underset{2}{\mathsf{a}}\ \underset{3}{\mathsf{b}}\ \underset{4}{\sqcup^5} \\
x_5 = & \underset{3}{\mathsf{A}}\ \underset{4}{\sqcup^5}\ \underset{3}{\mathsf{A}}\ \underset{4}{\sqcup^5}\ \underset{3}{\mathsf{B}}\ \underset{4}{\sqcup^2}
\end{array}
$$

$$
\begin{aligned}
x_6 &= \underset{0}{\mathsf{a}}\,\underset{1}{\mathsf{a}}\,\underset{2}{\mathsf{a}}\,\underset{3}{\mathsf{a}}\,\underset{4}{\mathsf{a}}\,\underset{5}{\mathsf{a}}\,\underset{0}{\mathsf{a}}\,\underset{1}{\mathsf{a}}\,\underset{2}{\mathsf{a}}\,\underset{3}{\mathsf{a}}\,\underset{4}{\mathsf{a}}\,\underset{5}{\mathsf{a}}\,\underset{0}{\mathsf{a}}\,\underset{1}{\mathsf{a}}\,\underset{2}{\mathsf{a}}\,\underset{3}{\mathsf{a}}\,\underset{4}{\mathsf{a}}\,\underset{5}{\mathsf{a}}\,\underset{0}{\mathsf{a}}\,\underset{1}{\mathsf{a}}\,\underset{2}{\mathsf{a}}\,\underset{3}{\mathsf{a}}\,\underset{4}{\mathsf{a}}\,\underset{5}{\mathsf{a}}\,\underset{0}{\mathsf{a}}\,\underset{1}{\mathsf{b}}\,\underset{2}{\sqcup}^{5} \\
x_7 &= \underset{1}{\mathsf{A}}\,\underset{2}{\sqcup}^{5}\,\underset{1}{\mathsf{A}}\,\underset{2}{\sqcup}^{5}\,\underset{1}{\mathsf{A}}\,\underset{2}{\sqcup}^{5}\,\underset{1}{\mathsf{A}}\,\underset{2}{\sqcup}^{5}\,\underset{1}{\mathsf{B}}\,\underset{2}{\sqcup}^{4} \\
x_8 &= \varepsilon
\end{aligned}
$$

Lemma 6.2 ([27])**.** *For all Fractran programs F, the PD0L system $\mathcal{H}_F$ is productive if and only if F does not terminate on input* 2*.*

Hence, by Lemma 6.2 and Proposition 6.1, deciding productivity of PD0L systems is undecidable.

Theorem 6.3 ([27])**.** *The following problem is Π^0_1-complete: Given a PD0L system $\mathcal{H}$, is $\mathcal{H}$ is productive?* □

7 The Productivity Question

So far, we have met in our guided tour through the landscape of streams some of the important families, some famous members of these families, and we have considered some important properties of streams, at the same time outlining some tough questions that confront us. There is one basic issue that we still have been silent about (although it is in fact already present in the previous section), but is of paramount importance in stream specifications, in particular when we view the stream landscape from the point of view of functional programming applications. The issue was considered by Dijkstra, who also coined the word 'productivity' of a stream definition. Two important papers gave initial approaches to this important problem of productivity: Hughes, Pareto and Sabry [33], and Telford and Turner [53]. In this section we explain the problem, and briefly mention the realization of solving this problem in an automated way for a restricted yet extensive class of stream specifications.

An important aspect of program correctness is termination. When dealing with programs that construct or process infinite objects, one cannot require termination; instead such programs need to be *productive*. The concept of *productivity* captures the intuitive notion of unlimited progress, of programs producing values indefinitely, programs immune to deadlock. For instance, control programs, such as an operating system, are not supposed to be terminating, but we want them to be productive. In other words we want the system to keep responding to user input and produce output, thus generating an infinite stream of outputs. Thus productivity guarantees *deadlock-* and *lifelock-freeness*. In lazy functional programming languages such as Miranda, Clean or Haskell, usage of infinite structures such as streams, 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. This holds both for terminating programs that lazily evaluate only finite parts of the infinite structures, as well as for non-terminating programs that directly construct or process infinite objects (like streams of sensor data or user inputs). Since productivity is undecidable in general, the challenge is to find ever stronger methods for proving productivity.

We study termination and productivity in the framework of term rewriting. For example, the Thue-Morse sequence can be defined as follows:

$$\mathsf{M} = 0 : \mathsf{zip}(\mathsf{inv}(\mathsf{M}), \mathsf{tail}(\mathsf{M})) \tag{7}$$

$$\mathsf{tail}(x : \sigma) = \sigma \qquad \mathsf{inv}(0 : \sigma) = 1 : \mathsf{inv}(\sigma)$$

$$\mathsf{zip}(x : \sigma, \tau) = x : \mathsf{zip}(\tau, \sigma) \qquad \mathsf{inv}(1 : \sigma) = 0 : \mathsf{inv}(\sigma)$$

A stream specification is called *productive* if not only can the specification be evaluated continually to build up a unique infinite normal form, but the resulting infinite expression is also meaningful in the sense that it is a *constructor normal form* which allows to read off consecutively individual elements of the stream. We emphasize that productivity is not just unique solvability, but also the potential to obtain the solution by evaluation. For example, the specification $\mathsf{Z} = \mathsf{z}(\mathsf{Z})$ with $\mathsf{z}(x : \sigma) = 0 : \mathsf{z}(\sigma)$, has a unique solution (the stream of zeros), but it cannot be evaluated to obtain this solution.

For specification (7) of the Thue-Morse sequence, it is relatively easy to convince oneself that the definition is productive. Productivity of an arbitrary stream specification is however non-trivial, even undecidable. To get a feeling of the non-triviality, consider the stream specification

$$\mathsf{J} = 0 : 1 : \mathsf{even}(\mathsf{J}) \qquad \mathsf{even}(x : \sigma) = x : \mathsf{odd}(\sigma) \qquad \mathsf{odd}(x : \sigma) = \mathsf{even}(\sigma) \tag{8}$$

This specification 'deadlocks'; it produces only 4 entries and then starts an infinite idling sequence of function calls: $\mathsf{J} = 0 : 1 : 0 : 0 : \mathsf{even}(\mathsf{even}(\mathsf{even}(\ldots)))$.

The notion of productivity was first mentioned by Dijkstra [18]. Since then several papers [56, 51, 14, 33, 52, 9] have been devoted to criteria ensuring productivity, and more recently [23, 24, 20, 21, 57, 58, 25, 17, 26, 34]. Technically, the common essence of most of these approaches is a *data-oblivious* analysis, that is, a quantitative analysis of productivity where the concrete values of the elements in the stream are ignored. We have adopted and elaborated this approach in [23, 24, 20, 21]. The recent trend is towards a *data-aware* analysis of productivity via a transformation to termination of term rewriting systems, see further [25, 57, 58, 26]. Let us highlight two of the papers, namely [20] and [26].

In [20] we develop a *data-oblivious* method for proving productivity. This method is provably optimal among all data-oblivious approaches, meaning that

no data-oblivious approach can recognize more specifications as productive. Here 'data-oblivious' refers to a quantitative analysis of productivity where the concrete values of the elements in the stream are ignored. Previously, all techniques for recognizing productivity have been data-oblivious.

In [26] we have devised a sound and complete transformation of productivity to context-sensitive termination, thereby making the power of termination provers available for proving *data-aware* productivity. For proving termination of TRSs automatically, many powerful techniques and tools have been developed, to wit AProVE, Jambox, Matchbox, MuTerm, Torpa, TTT, etc.

$$
\begin{aligned}
[\mathsf{M}] &= \mu M.\bullet([\mathsf{zip}]([\mathsf{inv}](M), [\mathsf{tail}](M))) \\
&= \mu M.\bullet(\mathsf{inf}(\overline{-++}(\overline{-+}(M)), \overline{+-+}(-\overline{-+}(M)))) \\
&\twoheadrightarrow_{\mathsf{R}} \mu M.\bullet(\mathsf{inf}(\overline{-++}(M), +-\overline{-++}(M))) \\
&\twoheadrightarrow_{\mathsf{R}} \mu M.+\overline{-+}(\mathsf{inf}(\overline{-++}(M), +-\overline{-++}(M))) \\
&\twoheadrightarrow_{\mathsf{R}} \mu M.\mathsf{inf}(+\overline{-+}(\overline{-++}(M)), +\overline{-+}(+-\overline{-++}(M))) \\
&\twoheadrightarrow_{\mathsf{R}} \mu M.\mathsf{inf}(\overline{+-+}(M), ++-\overline{-++}(M)) \\
&\twoheadrightarrow_{\mathsf{R}} \mathsf{inf}(\mu M.\overline{+-+}(M), \mu M.++-\overline{-++}(M)) \\
&\twoheadrightarrow_{\mathsf{R}} \mathsf{inf}(\mathsf{src}(\infty), \mathsf{src}(\infty)) \\
&\twoheadrightarrow_{\mathsf{R}} \mathsf{src}(\infty)
\end{aligned}
$$

Figure 15: Output of our productivity decision tool, establishing productivity of Definition (7)*. Here μ symbolizes recursion, the pebble $\bullet$ an abstract element,* inf *the infimum,* $\{-, +\}$ *code input and output, and overlining means periodic iteration. The actual computation uses a confluent and terminating rewrite relation* $\twoheadrightarrow_{\mathsf{R}}$.

$$
\begin{aligned}
[\mathsf{J}] &= \mu J.\bullet(\bullet(\overline{-+-}(J))) \\
&\twoheadrightarrow_{\mathsf{R}} \mu J.+\overline{-+}(+\overline{-+}(\overline{-+-}(J))) \\
&\twoheadrightarrow_{\mathsf{R}} \mu J.++\overline{-+}(\overline{-+-}(J)) \\
&\twoheadrightarrow_{\mathsf{R}} \mu J.++\overline{-+-}(J) \\
&\twoheadrightarrow_{\mathsf{R}} \mathsf{src}(4)
\end{aligned}
$$

Figure 16: For the specification (8) *we obtain that* J *is not productive; only 4 elements can be evaluated.*

We have implemented our data-oblivious techniques [23, 24, 20] in the productivity prover *ProPro* (`http://infinity.few.vu.nl/productivity/`). Figure 15 gives a script of the tool's computation for specification (7) of the Thue-Morse

sequence, testifying that its definition outputs ∞ entries, and therefore is productive. Applied on (8) the answer is 4 (see Figure 16) , hence not productive.

8 More Questions

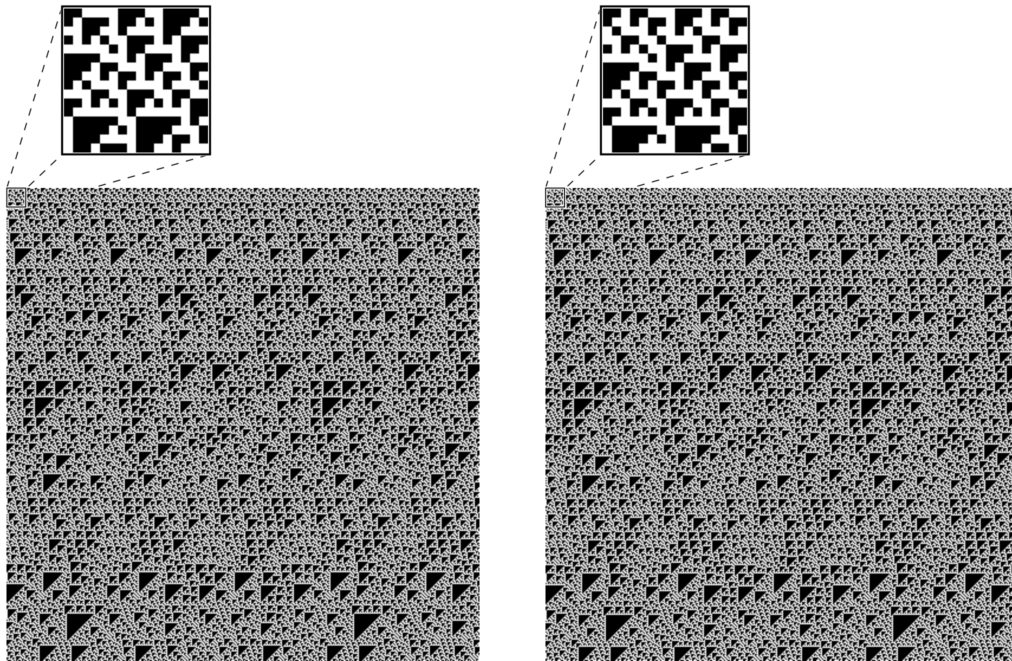


Figure 17: Comparing the 'fingerprints' $\Delta^{\omega}(\mathsf{S})$ *and* $\Delta^{\omega}(\mathsf{W})$ *of the Sierpiński sequence* S *(left), and the Mephisto Waltz* W*. We find that* $\Delta^{2}(\mathsf{S}) = \Delta^{3}(\mathsf{W})$*!*

(i) The productivity question for stream definitions has everything to do with the input-output behavior of the various operations on streams (like zip). Thus a stream specification is reminiscent to a system of communicating processes. There are tools (like μ-CRL) that analyze deadlock-freeness of process specifications. Can the stream productivity question profit from the tools technology for process communication? We expect that it can.

(ii) Connections with λ-calculus and infinitary λ-calculus are interesting. It is clear that stream operations like zip, even, etc., as expressible in term rewriting, are definable in λ-calculus. The resulting λ-term M that is the translation of the stream specification then has a Böhm Tree $\mathsf{BT}(M)$ that is precisely the stream to be defined. It is total (without bottoms $\perp$) iff the stream specification is productive. In fact, we may call λ-term M with total

$\mathsf{BT}(M)$ *productive*. The study of productive λ-terms seems worthwhile in itself.

(iii) What streams give rise to fractals? As we have seen, some streams give rise, with suitable 'turtle' instructions, to well-defined fractals such as the snowflake, the arrowhead curve, the dragon curve, the Cesàro fractal, and so on. But other streams, like Kolakoski, seem to generated chaotic patterns that do not converge (in the Hausdorff metric, after re-scaling successive approximations) to a well-defined fractal curve. Are there criteria (and definitions for what constitutes a 'well-defined fractal') that guarantee the convergence via some smart turtle to a fractal?

(iv) FSTs are highly useful in many fields. An interesting question is whether they can be decomposed in some elementary, 'prime', FSTs (with respect to the composition of FSTs, which is intuitively clear, feeding the output of the first FST as input of the second; this is a wreath product). A theory that was discovered in the sixties by Rhodes and Krohn seems to indicate that indeed there is a prime factorization theorem for FSTs. For Mealy Machines a prime factorization was obtained by Rhodes and Krohn, see [4]. Generalizing this to the more general FSTs should be a corollary which we expect to be straightforward (but not entirely trivial).

(v) Fingerprints of streams over $\{0, 1\}$. We wonder what information about the complexity of streams can be derived from their 'fingerprints'. What we record in the fingerprint is the stream of first differences (as realized by the FST Δ depicted in Figure 6), and this repeated ad infinitum. Being two-dimensional, the black and white patterns that arise in this way, are much easier 'processed' than the 'patterns' in the original stream, the top row. In this way, we observed [28, 29] the relation between the Sierpiński stream S and the Mephisto Waltz W (both sequences are defined in Section 3), see Figure 17. For the Fibonacci word F, a sturmian stream, the fingerprint reveals patterns of a rather 'quiet nature'. Is this always so for sturmian streams? Note that looking 'deep' in the fingerprint, amounts to looking far away to the right in the stream.

9 Background: Infinitary Term Rewriting

We give a brief introduction to infinitary rewriting. Actually, what we present here is only a prefix of a more extensive theory in which reductions can have any countable ordinal length. Here we will be satisfied with reductions up to length ω.

For the full framework of infinitary rewriting we refer to [37, 54, 7, 30], for an introduction to finitary rewriting to [36, 54, 5, 6].

A *signature* Σ is a set of symbols f each having a fixed arity $\#f \in \mathbb{N}$. (Earlier we used Σ to denote an alphabet of letters; we now overload the use of Σ somewhat, but this will not cause confusion.) Let $\mathcal{X}$ be an infinite set of variables such that $\mathcal{X} \cap \Sigma = \varnothing$. Then the set of *finite terms* $Ter(\Sigma, \mathcal{X})$ over Σ and $\mathcal{X}$ is inductively defined by the grammar:

$$T ::= x \mid f(\underbrace{T, \ldots, T}_{\#f \text{ times}}) \qquad\qquad (x \in \mathcal{X},\ f \in \Sigma) \tag{9}$$

We obtain the set of (finite and) *infinite terms* $Ter^\infty(\Sigma, \mathcal{X})$ by interpreting this grammar coinductively. That is, $Ter^\infty(\Sigma, \mathcal{X})$ is the largest set T such that every term $t \in T$ is either a variable $t \in \mathcal{X}$, or $t = f(t_1, \ldots, t_n)$ with $n = \#f$ and $t_1, \ldots, t_n \in T$.

The equality on infinite terms is *bisimilarity* $\underline{\leftrightarrow} \subseteq Ter^\infty(\Sigma, \mathcal{X}) \times Ter^\infty(\Sigma, \mathcal{X})$ which is defined as the largest relation R such that $s\ R\ t$ implies that $s = t \in \mathcal{X}$, or $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ such that $s_1\ R\ t_1, \ldots, s_n\ R\ t_n$. We consider bisimilar terms $s \underline{\leftrightarrow} t$ as identical.

Remark 9.1. Alternatively, the infinite terms arise from the set of finite terms, $Ter(\Sigma, \mathcal{X})$, by metric completion, using the well-known distance function d such that for $t, s \in Ter(\Sigma, \mathcal{X})$, $\mathrm{d}(t, s) = 2^{-n}$ if the n-th level of the terms t, s (viewed as labeled trees) is the first level where a difference appears, in case t and s are not identical; furthermore, $\mathrm{d}(t, t) = 0$. It is standard that this construction yields $\langle Ter(\Sigma, \mathcal{X}), \mathrm{d}\rangle$ as a metric space. Now infinite terms are obtained by taking the completion of this metric space, and they are represented by infinite trees. We will refer to the complete metric space arising in this way as $\langle Ter^\infty(\Sigma, \mathcal{X}), \mathrm{d}\rangle$, where $Ter^\infty(\Sigma, \mathcal{X})$ is the set of finite and infinite terms over Σ.

Let $t \in Ter^\infty(\Sigma, \mathcal{X})$ be a finite or infinite term. The set of *variables* $\mathcal{V}ar(t) \subseteq \mathcal{X}$ *of* t, and the set of *positions* $\mathcal{P}os(t) \subseteq \mathbb{N}^*$ *of* t are defined coinductively by:

$$\mathcal{V}ar(x) = \{x\} \qquad \mathcal{V}ar(f(t_1, \ldots, t_n)) = \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n)$$
$$\mathcal{P}os(x) = \{\epsilon\} \qquad \mathcal{P}os(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{ip \mid 1 \le i \le n,\ p \in \mathcal{P}os(t_i)\}$$

For $p \in \mathcal{P}os(t)$, the *subterm* $t|_p$ *of* t *at position* p is defined by:

$$t|_\epsilon = t \qquad\qquad f(t_1, \ldots, t_n)|_{ip} = t_i|_p$$

A *substitution* σ is a map $\sigma : \mathcal{X} \to Ter^\infty(\Sigma, \mathcal{X})$. We extend the domain of substitutions σ to terms $Ter^\infty(\Sigma, \mathcal{X})$ as follows:

$$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$$

We write $\{ x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \}$ for the substitution σ defined by $\sigma(x_1) = t_1, \ldots,$ $\sigma(x_n) = t_n$ and $\sigma(y) = y$ for every $y \in \mathcal{X} \setminus \{ x_1, \ldots, x_n \}$.

A *context* C is a term $Ter^\infty(\Sigma, \mathcal{X} \cup \{\Box\},)$ containing precisely one occurrence of $\Box$, that is, there is precisely one position $p \in \mathcal{P}os(C)$ such that $C|_p = \Box$. For a context C and a term t, we write $C[t]$ for the term $\{ \Box \mapsto t \}(C)$.

A *rewrite rule* $\ell \to r$ over Σ and $\mathcal{X}$ is a pair $(\ell, r) \in Ter(\Sigma, \mathcal{X}) \times Ter(\Sigma, \mathcal{X})$ of finite terms such that the left-hand side ℓ is not a variable ($\ell \notin \mathcal{X}$), and all variables in the right-hand side r occur in ℓ ($\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell)$). A *term rewrite system (TRS)* $\mathcal{R}$ over Σ and $\mathcal{X}$ is a set of rewrite rules over Σ and $\mathcal{X}$. We define a binary relation $\to_{\mathcal{R}}$, the *rewrite steps*, to consist of all pairs

$$C[\sigma(\ell)] \to_{\mathcal{R}} C[\sigma(r)]$$

for contexts C, rule $\ell \to r \in \mathcal{R}$, and substitution $\sigma : \mathcal{X} \to Ter^\infty(\Sigma, \mathcal{X})$. Furthermore, we write $\to_{\mathcal{R},p}$ whenever additionally $C|_p = \Box$. We drop the subscript $\mathcal{R}$ in $\to_{\mathcal{R}}$ and $\to_{\mathcal{R},p}$ whenever $\mathcal{R}$ is clear from the context. The notion of *normal form*, which now may be an infinite term, is unproblematic: it is a term without a redex occurrence. A *finite rewrite sequence from s to t*, denoted $s \twoheadrightarrow t$, is a sequence is a sequence $s = s_0 \to s_1 \to \ldots \to s_n = t$.

An *infinitary rewrite sequence from s to t*, denoted $s \twoheadrightarrow\!\!\!\!\!\!\!\rightarrow t$, is either a finite rewrite sequence $s \twoheadrightarrow t$, or an infinite sequence $s_0 \to_{p_0} s_1 \to_{p_1} s_2 \to_{p_2} \ldots$ of rewrite steps such that the following conditions hold:

(i) the distance $\mathrm{d}(s_i, t)$ tends to 0 for $i \to \infty$ and, moreover,

(ii) the depth of the rewrite action, that is, the length of the position p_i, tends to infinity for $i \to \infty$.

Note that item (i) requires *Cauchy convergence* of the sequence of terms. The requirement (ii) is *strong convergence*, which in addition requires that the depth of the redexes contracted in the successive steps tends to infinity. So this rules out the possibility that the action of redex contraction stays confined at the top, or stagnates at some finite level of depth.

Example 9.2. Let us give a simple example of infinitary term rewriting that moreover explains the working of an FST on a stream. Consider the FST in Figure 6. We 'translate' this FST in the TRS with rules:

$$\begin{array}{lll} q(0 : s) \to q_0(s) & q_0(0 : s) \to 0 : q_0(s) & q_1(0 : s) \to 1 : q_0(s) \\ q(1 : s) \to q_1(s) & q_0(1 : s) \to 1 : q_1(s) & q_1(1 : s) \to 0 : q_1(s) \end{array}$$

Here $0, 1$ are 0-ary constants, q, q_0, q_1 are unary function symbols, ':' is a binary stream constructor, s is a variable (for streams).

Now the transformation of the Thue-Morse sequence M to the Toeplitz word $\mathsf{T} = \mathsf{T}_{101?}$ (see Section 2.3) proceeds by the following infinite reduction sequence, where we omit the infix ':' symbols:

$$\begin{aligned} q(\mathsf{M}) &= q(0110100110010110\ldots) \\ &\to q_0(110100110010110\ldots) \\ &\to 1\, q_1(10100110010110\ldots) \\ &\to 10\, q_1(0100110010110\ldots) \\ &\to 101\, q_0(100110010110\ldots) \\ &\to 1011\, q_1(00110010110\ldots) \\ &\twoheadrightarrow \mathsf{T} \end{aligned}$$

The term T is an infinite normal form. Note that indeed the depth of the contracted redexes tends to infinity during the reduction.

It is not hard to see that we can easily extend such rewrite sequences to length beyond ω; e.g. $q(q(\mathsf{M})) \twoheadrightarrow q(\mathsf{T}) \twoheadrightarrow 11001\ldots$, which yields an infinite normal form after $\omega \cdot 2$ steps. In Figure 17 we already considered the 'fingerprint' of a stream s, being the matrix with rows $q^n(s)$, or in our earlier notation $\Delta^n(s)$.

Remark 9.3. Without wanting to go deep in the theory of infinitary rewriting, let us mention some facts. There are infinitary counterparts CR^∞, UN^∞, WN^∞ and SN^∞ for the well-known finitary properties CR (confluence or Church–Rosser property), UN (unique normal form property), WN (weak normalization), and SN (strong normalization). In general CR^∞ does not hold for first-order orthogonal term rewriting systems, but for TRSs as in the example, without collapsing rules, it does. Moreover, UN^∞ always holds for first-order orthogonal TRSs. For the example TRS also WN^∞, SN^∞ and UN^∞ hold.

References

[1] J.-P. Allouche. Sur la Complexité des Suites Infinies. *Journées Montoises*, 1(2):133–143, 1994.

[2] J.-P. Allouche and J. Shallit. The Ubiquitous Prouhet–Thue–Morse Sequence. In *Proc. Conf. on Sequences and Their Applications (SETA 1998)*, pages 1–16. Springer, 1999.

[3] J.-P. Allouche and J. Shallit. *Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press, New York, 2003.

[4] M. A. Arbib, K. Krohn, and J. L. Rhodes. *Algebraic Theory of Machines, Languages, and Semi-Groups*. Academic Press, 1968.

[5] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[6] H. P. Barendregt. The Type Free Lambda Calculus. In J. Barwise, editor, *Handbook of mathematical Logic*, pages 1091–1132. North-Holland, 1977.

[7] H. P. Barendregt and J. W. Klop. Applications of Infinitary Lambda Calculus. *Information of Computation*, 207(5):559–582, 2009.

[8] A. Belov. Some Algebraic Properties of Machine Poset of Infinite Words. *ITA*, 42(3):451–466, 2008.

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

[10] J. Cassaigne and J. Karhumäki. Toeplitz words, Generalized Periodicity and Periodically Iterated Morphisms. *European Journal of Combinatorics*, 18(5):497–510, 1997.

[11] A. Cobham. On the Base-Dependence of Sets of Numbers Recognizable by Finite Automata. *Mathematical Systems Theory*, 3(2):186–192, 1969.

[12] J. H. Conway. Unpredictable Iterations. In *Proceedings of the 1972 Number Theory Conference*, pages 49–52, 1972.

[13] J. H. Conway. Fractran: A Simple Universal Programming Language for Arithmetic. In T.M. Cover and B. Gopinath, editors, *Open Problems in Communication and Computation*, pages 4–26. Springer, 1987.

[14] Th. Coquand. Infinite Objects in Type Theory. In H. Barendregt and T. Nipkow, editors, *TYPES*, volume 806, pages 62–78. Springer, Berlin, 1994.

[15] K. Culik II and J. Karhumäki. Iterative Devices Generating Infinite Words. In *Proc. 9th Ann. Symp. on Theoretical Aspects of Computer Science (STACS 1992)*, volume 577 of *Lecture Notes in Computer Science*, pages 531–543. Springer, 1992.

[16] K. Culik II, J. Karhumäki, and A. Lepistö. Alternating Iteration of Morphisms and Kolakovski [*sic*] Sequence. In G. Rozenberg and A. Salomaa, editors, *Lindermayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 93–106. Springer, 1992.

[17] N. A. Danielsson. Beating the Productivity Checker Using Embedded Languages. In *Proc. Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR 2010)*, volume 43 of *EPTCS*, pages 29–48, 2010.

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

[19] A. Ehrenfeucht, K. P. Lee, and G. Rozenberg. Subword Complexities of Various Classes of Deterministic Developmental Languages Without Interaction. *Theoretical Computer Science*, 1:59–75, 1975.

[20] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-Oblivious Stream Productivity. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Quatar*, pages 79–96, 2008.

[21] J. Endrullis, C. Grabmayer, and D. Hendriks. Complexity of Fractran and Productivity. In R.A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, pages 371–387, 2009.

[22] J. Endrullis, C. Grabmayer, and D. Hendriks. Mix-Automatic Sequences. In *Proc. Conf. on Language and Automata Theory and Applications (LATA 2013)*, 2013. To appear.

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

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

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

[26] J. Endrullis and D. Hendriks. Lazy Productivity via Termination. *Theoretical Computer Science*, 412(28):3203–3225, 2011.

[27] J. Endrullis and D. Hendriks. On Periodically Iterated Morphisms. *CoRR*, abs/1207.2336, 2012.

[28] J. Endrullis, D. Hendriks, and J. W. Klop. Let's Make a Difference! In *Liber Amicorum for Roel de Vrijer*, pages 61–73. 2009. Available at `http://arxiv.org/abs/0911.1004`.

[29] J. Endrullis, D. Hendriks, and J. W. Klop. Degrees of Streams. *Journal of Integers*, 11B(A6):1–40, 2011. Proceedings of the Leiden Numeration Conference 2010.

[30] J. Endrullis, D. Hendriks, and J. W. Klop. Highlights in Infinitary Rewriting and Lambda Calculus. *Theoretical Computer Science*, 464:48–71, 2012.

[31] S. Ferenczi. Complexity of Sequences and Dynamical Systems. *Discrete Mathematics*, 206(1-3):145–154, 1999.

[32] C. Grabmayer, J. Endrullis, D. Hendriks, J. W. Klop, and L. S. Moss. Automatic Sequences and Zip-Specifications. In *Proc. Symp. on Logic in Computer Science (LICS 2012)*, pages 335–344. IEEE Computer Society, 2012.

[33] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL '96*, pages 410–423, 1996.

[34] G. Hutton and M. Jaskelioff. Representing Contractive Functions on Streams. Submitted to the Journal of Functional Programming, 2011.

[35] K. Jacobs and M. Keane. 0-1-Sequences of Toeplitz Type. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 13(2):123–131, 1969.

[36] J. W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, volume II, pages 1–116. Oxford University Press, 1992.

[37] J. W. Klop and R.C. de Vrijer. Infinitary Normalization. In S. Artemov, H. Barringer, A.S. d'Avila Garcez, L.C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 2, pages 169–192. College Publications, 2005.

[38] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 2nd edition, 1981.

[39] W. Kolakoski. Self Generating Runs. *Amer. Math. Monthly*, 72, 1965. Problem 5304.

[40] C. Kupke and J. J. M. M. Rutten. Complete Sets of Cooperations. *Information and Computation*, 208(12):1398–1420, 2010.

[41] C. Kupke and J. J. M. M. Rutten. On the Final Coalgebra of Automatic Sequences. In *Logic and Program Semantics - Essays Dedicated to Dexter Kozen on the Occasion of his 60th Birthday*, volume 7230 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2012.

[42] A. Lepistö. On the Power of Periodic Iteration of Morphisms. In *Proc. Colloquium on Automata, Languages and Programming (ICALP)*, volume 700, pages 496–506. Springer, 1993.

[43] M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.

[44] Th. Monteil. *Brute force Kolakoski*, 2007. Blog entry available at `http://www2.lirmm.fr/ monteil/blog/BruteForceKolakoski/`.

[45] M. Morse. Recurrent Geodesics on a Surface of Negative Curvature. In *Trans. Amer. Math. Soc.*, volume 22, pages 84–100, 1921.

[46] A. A. Muchnik, Y. L. Pritykin, and A. L. Semenov. Sequences Close to Periodic. *Russian Mathematical Surveys*, 64(5):805–871, 2009.

[47] T. Nagell, editor. *Selected mathematical papers of Axel Thue*. Universitetsforlaget, Oslo, 1977.

[48] H. Peitgen, H. Jürgens, and D. Saupe. *Chaos and Fractals*. New Frontiers of Science. Springer, 2nd edition, 2004.

[49] N. Pytheas Fogg. *Substitutions in Dynamics, Arithmetics and Combinatorics*, volume 1794 of *Lecture Notes in Mathematics*. Springer, 2002.

[50] J. R. Shoenfield. *Degrees of Unsolvability*. North-Holland, Elsevier, 1971.

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

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

[53] A. Telford and D. Turner. Ensuring the Productivity of Infinite Structures. Technical Report 14-97, The Computing Laboratory, Univ. of Kent at Canterbury, 1997.

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

[55] A. Thue. Über Unendliche Zeichenreihen. In *Norske vid. Selsk. Skr. Mat. Nat. Kl.*, volume 7, pages 1–22, 1906. Reprinted in [47, 139–158].

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

[57] H. Zantema and M. Raffelsieper. Stream Productivity by Outermost Termination. In *Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2009)*, volume 15 of *Electronic Proceedings in Theoreatical Computer Science*, pages 83–95, 2009.

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