Complexity of Fractran and Productivity

Jörg Endrullis[1], Clemens Grabmayer[2], and Dimitri Hendriks[1]

[1] Vrije Universiteit Amsterdam, Department of Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
joerg@few.vu.nl diem@cs.vu.nl
[2] Universiteit Utrecht, Department of Philosophy
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
clemens@phil.uu.nl

Abstract. In functional programming languages the use of infinite structures is common practice. For total correctness of programs dealing with infinite structures one must guarantee that every finite part of the result can be evaluated in finitely many steps. This is known as productivity. For programming with infinite structures, productivity is what termination in well-defined results is for programming with finite structures.
Fractran is a simple Turing-complete programming language invented by Conway. We prove that the question whether a Fractran program halts on all positive integers is Π^0_2-complete. In functional programming, productivity typically is a property of individual terms with respect to the inbuilt evaluation strategy. By encoding Fractran programs as specifications of infinite lists, we establish that this notion of productivity is Π^0_2-complete even for some of the most simple specifications. Therefore it is harder than termination of individual terms. In addition, we explore generalisations of the notion of productivity, and prove that their computational complexity is in the analytical hierarchy, thus exceeding the expressive power of first-order logic.

1 Introduction

For programming with infinite structures, productivity is what termination is for programming with finite structures. In lazy functional programming languages like Haskell, Miranda or Clean the use of data structures, whose intended semantics is an infinite structure, is common practice. Programs dealing with such infinite structures can very well be terminating. For example, consider the Haskell program implementing a version of Eratosthenes' sieve:

```
prime n = primes !! (n-1)
primes = sieve [2..]
sieve (n:xs) = n:(sieve (filter (\m -> m `mod` n /= 0) xs))
```

where `prime n` returns the n-th prime number for every $n \geq 1$. The function `prime` is terminating, despite the fact that it contains a call to the non-terminating function `primes` which, in the limit, rewrites to the infinite list of prime numbers in ascending order. To make this possible, the strategy with respect to which the terms are evaluated is crucial. Obviously, we cannot fully

evaluate `primes` before extracting the n-th element. For this reason, lazy functional languages typically use a form of outermost-needed rewriting where only needed, finite parts of the infinite structure are evaluated, see for example [13].

Productivity captures the intuitive notion of unlimited progress, of 'working' programs producing values indefinitely, programs immune to livelock and deadlock, like `primes` above. A recursive specification is called productive if not only can the specification be evaluated continually to build up an infinite normal form, but this infinite expression is also meaningful in the sense that it represents an infinite object from the intended domain. The study of productivity (of stream specifications in particular) was pioneered by Sijtsma [15]. More recently, a decision algorithm for productivity of stream specifications from an expressive syntactic format has been developed [6] and implemented [4].

We consider various variants of the notion of productivity and pinpoint their computational complexity in the arithmetical and analytical hierarchy. In functional programming, expressions are evaluated according to an inbuilt evaluation strategy. This gives rise to *productivity with respect to an evaluation strategy*. We show that this property is Π^0_2-complete (for individual terms) using a standard encoding of Turing machines into term rewriting systems. Next, we explore two generalisations of this concept: *strong* and *weak productivity*. Strong productivity requires every outermost-fair rewrite sequence to 'end in' a constructor normal form, whereas weak productivity demands only the existence of a rewrite sequence to a constructor normal form. As it turns out, these properties are of analytical complexity: Π^1_1 and Σ^1_1-complete, respectively.

Finally, we encode Fractran programs into stream specifications. In contrast to the encoding of Turing machines, the resulting specifications are of a very simple form and do not involve any computation on the elements of the stream. We show that the uniform halting problem of Fractran programs is Π^0_2-complete. (Although Turing-completeness of Fractran is folklore, the exact complexity has not yet been investigated before.) Consequently we obtain a strengthening of the earlier mentioned Π^0_2-completeness result for productivity.

Fractran [2] is a remarkably simple Turing-complete programming language invented by the mathematician John Horton Conway. A Fractran program is a finite list of fractions $\frac{p_1}{q_1}, \ldots, \frac{p_k}{q_k}$. Starting with a positive integer n_0, the algorithm successively calculates n_{i+1} by multiplying n_i with the first fraction that yields an integer again. The algorithm halts if there is no such fraction.

To illustrate the algorithm we consider an example of Conway from [2]:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}$$

We start with $n_0 = 2$. The leftmost fraction which yields an integer product is $\frac{15}{2}$, and so $n_1 = 2 \cdot \frac{15}{2} = 15$. Then we get $n_2 = 15 \cdot \frac{55}{1} = 825$, etcetera. By successive application of the algorithm, we obtain the following infinite sequence:

$$2, 15, 825, 725, 1925, 2275, 425, 390, 330, 290, 770, \ldots$$

Apart from 2^1, the powers of 2 occurring in this infinite sequence are $2^2, 2^3, 2^5, 2^7, 2^{11}, 2^{13}, 2^{17}, 2^{19}, \ldots$, where the exponents form the sequence of primes.

We translate Fractran programs to streams specifications in such a way that the specification is productive if and only if the program halts on all $n_0 > 1$. Let us define the target format of this translation: the *lazy stream format* (LSF). LSF consists of stream specifications of the form $\mathsf{M} \to C[\mathsf{M}]$ where C is a context built solely from: one data element $\bullet$, the stream constructor ':', the functions $\mathsf{head}(x : \sigma) \to x$ and $\mathsf{tail}(x : \sigma) \to \sigma$, unary stream functions mod_n, and k-ary stream functions zip_k with the following defining rules, for every $n, k \geq 1$:

$$\begin{aligned} \mathsf{mod}_n(\sigma) &\to \mathsf{head}(\sigma) : \mathsf{mod}_n(\mathsf{tail}^n(\sigma)) \\ \mathsf{zip}_k(\sigma_1, \sigma_2 \ldots, \sigma_k) &\to \mathsf{head}(\sigma_1) : \mathsf{zip}_k(\sigma_2, \ldots, \sigma_k, \mathsf{tail}(\sigma_1)) \end{aligned} \tag{LSF}$$

By reducing the uniform halting problem of Fractran programs to productivity of LSF, we get that productivity for LSF is Π^0_2-complete.

This undecidability result stands in sharp contrast to the decidability of productivity for the *pure stream format* (PSF, [6]). Let us elaborate on the difference between these two formats. Examples of specifications in PSF are:

$$\mathsf{J} \to 0 : 1 : \mathsf{even}(\mathsf{J}) \qquad \text{and} \qquad \mathsf{Z} \to 0 : \mathsf{zip}(\mathsf{even}(\mathsf{Z}), \mathsf{odd}(\mathsf{Z})) \,,$$

including the defining rules for the stream functions involved:

$$\mathsf{even}(x : \sigma) \to x : \mathsf{odd}(\sigma) \,, \quad \mathsf{odd}(x : \sigma) \to \mathsf{even}(\sigma) \,, \quad \mathsf{zip}(x : \sigma, \tau) \to x : \mathsf{zip}(\tau, \sigma) \,,$$

where zip 'zips' two streams alternatingly into one, and even (odd) returns a stream consisting of the elements at its even (odd) positions. The specification for Z produces the stream $0 : 0 : 0 : \ldots$ of zeros, whereas the infinite normal form of J is $0 : 1 : 0 : 0 : \mathsf{even}^\omega$, which is not a constructor normal form.

Excluded from PSF is the observation function on streams $\mathsf{head}(x : \sigma) \to x$. This is for a good reason, as we shall see shortly. PSF is essentially layered: data terms (terms of sort `data`) cannot be built using stream terms (terms of sort `stream`). As soon as *stream dependent* data functions are admitted, the complexity of the productivity problem of such an extended format is increased. Indeed, as our Fractran translation shows, productivity of even the most simple stream specifications is undecidable and Π^0_2-hard. The problem with stream dependent data functions is that they possibly create 'look-ahead': the evaluation of the 'current' stream element may depend on the evaluation of 'future' stream elements. To see this, consider an example from [15]:

$$\mathsf{S}_n \to 0 : \mathsf{S}_n(n) : \mathsf{S}_n$$

where for a term t of sort stream and $n \in \mathbb{N}$, we write $t(n)$ as a shorthand for $\mathsf{head}(\mathsf{tail}^n(t))$. If we take n to be an even number, then S_n is productive, whereas it is unproductive for odd n.

A hint for the fact that it is Π^0_2-hard to decide whether a lazy specification is productive already comes from a simple encoding of the Collatz conjecture (also known as the '$3x$+1-problem' [12]) into a productivity problem. Without proof we state: *the Collatz conjecture is true if and if only the following specification*

produces the infinite chain $\bullet : \bullet : \bullet : \ldots$ *of data elements* $\bullet$*:*

$$\mathsf{collatz} \to \bullet : \mathsf{zip}_2(\mathsf{collatz}, \mathsf{mod}_6(\mathsf{tail}^9(\mathsf{collatz}))) \tag{1}$$

In order to understand the operational difference between rules in PSF and rules in LSF, consider the following two rules:

$$\mathsf{read}(\sigma) \to \mathsf{head}(\sigma) : \mathsf{read}(\mathsf{tail}(\sigma)) \tag{2}$$

$$\mathsf{read}'(x : \sigma) \to x : \mathsf{read}'(\sigma) \tag{3}$$

The functions defined by these rules are extensionally equivalent: they both implement the identity function on fully developed streams. However, intensionally, or operationally, there is a difference. A term $\mathsf{read}'(s)$ is a redex only in case s is of the form $u : t$, whereas $\mathsf{read}(s)$ constitutes a redex for *every* stream term s, and so $\mathsf{head}(s)$ can be undefined. The 'lazy' rule (2) *postpones* pattern matching. Although in PSF we can define functions mod'_n and zip'_k extensionally equivalent to mod_n and zip_k, a pure version $\mathsf{collatz}'$ of $\mathsf{collatz}$ in (1) above (using mod'_6 and zip'_2 instead) can easily be seen to be not productive (it produces two data elements only), and to have no bearing on the Collatz conjecture.

Contribution and Overview. In Section 2 we show that the uniform halting problem of Fractran programs is Π^0_2-complete. This is the problem of determining whether a program terminates on all positive integers. Turing-completeness of a computational model does not imply that the uniform halting problem in the strong sense of termination on *all configurations* is Π^0_2-complete. For example, assume that we extend Turing machines with a special non-terminating state. Then the computational model obtained can still compute every recursive function. However, the uniform halting problem becomes trivial.

Our result is a strengthening of the result in [11] where it has been shown that the generalised Collatz problem (GCP) is Π^0_2-complete. This is because every Fractran program P can easily be translated into a Collatz function f such that the uniform halting problem for P is equivalent to the GCP for f. The other direction is not immediate, since Fractran programs form a strict subset of Collatz functions. We discuss this in more detail in Section 2.

In Section 3 we explore alternative definitions of productivity and make them precise in the framework of term rewriting. These can be highly undecidable: 'strong productivity' turns out to be Π^1_1-complete and 'weak productivity' is Σ^1_1-complete. Productivity of individual terms with respect to a computable strategy, which is the notion used in functional programming, is Π^0_2-complete.

In Section 4 we prove that productivity Π^0_2-complete even for specifications of the restricted LSF format. The new proof uses a simple encoding of Fractran programs P into stream specifications of the form $\mathsf{M}_P \to C[\mathsf{M}_P]$, in such a way that M_P is productive if and only if the program P halts on all inputs. The resulting stream specifications are very simple compared to the ones resulting from encoding of Turing machines employed in Section 3. Whereas the Turing machine encoding essentially uses calculations on the elements of the list, the specifications obtained from the Fractran encoding contain no operations on the list elements. In particular, the domain of data elements is a singleton.

Related Work. In [3] undecidability of different properties of first-order TRSs is analysed. While the standard properties of TRSs turn out to be either Σ^0_1- or Π^0_2-complete, the complexity of the dependency pair problems [1] is essentially analytical: it is shown to be Π^1_1-complete. We employ the latter result as a basis for our Π^1_1- and Σ^1_1-completeness results for productivity.

Roşu [14] shows that equality of stream specifications is Π^0_2-complete. We remark that this result can be obtained as a corollary of our translation of Fractran programs P to stream specifications M_P. Stream specifications M_P have the stream $\bullet : \bullet : \ldots$ as unique solutions if and only if they are productive. Thus Π^0_2-completeness of productivity of these specifications implies Π^0_2-completeness of the stream equality problem $\mathsf{M}_P = \bullet : \bullet : \ldots$.

One of the reviewers pointed us to recent work [7] of Grue Simonsen (not available at the time of writing) where Π^0_2-completeness of productivity of orthogonal stream specifications is shown. Theorem 3.5 below can be seen as a sharpening of that result in that we consider general TRSs and productivity with respect to arbitrary evaluation strategies. For orthogonal systems the evaluation strategy is irrelevant as long as it is outermost-fair. Moreover we further strengthen the result on orthogonal stream specifications by restricting the format to LSF.

2 Fractran

The one step computation of a Fractran program is a partial function.

Definition 2.1. Let $P = \frac{p_1}{q_1}, \ldots, \frac{p_k}{q_k}$ be a Fractran program. The partial function $f_P : \mathbb{N} \rightharpoonup \mathbb{N}$ is defined for all $n \in \mathbb{N}$ by:

$$f_P(n) = \begin{cases} n \cdot \frac{p_i}{q_i} & \text{where } \frac{p_i}{q_i} \text{ is the first fraction of } P \text{ such that } n \cdot \frac{p_i}{q_i} \in \mathbb{N}, \\ \text{undefined} & \text{if no such fraction exists.} \end{cases}$$

We say that P *halts* on $n \in \mathbb{N}$ if there exists $i \in \mathbb{N}$ such that $f^i_P(n) = \text{undefined}$. For $n, m \in \mathbb{N}$ we write $n \to_P m$ whenever $m = f_P(n)$.

The Fractran program for generating prime numbers, that we discussed in the introduction, is non-terminating for all starting values n_0, because the product of any integer with $\frac{55}{1}$ is an integer again. However, in general, termination of Fractran programs is undecidable.

Theorem 2.2. *The uniform halting problem for Fractran programs, that is, deciding whether a program halts for every starting value $n_0 \in \mathbb{N}_{>0}$, is Π^0_2-complete.*

A related result is obtained in [11] where it is shown that the generalised Collatz problem (GCP) is Π^0_2-complete, that is, the problem of deciding for a Collatz function f whether for every integer $x > 0$ there exists $i \in \mathbb{N}$ such that $f^i(x) = 1$. A Collatz function f is a function $f : \mathbb{N} \to \mathbb{N}$ of the form:

$$f(n) = \begin{cases} a_0 \cdot n + b_0, & \text{if } n \equiv 0 \pmod{p} \\ \vdots & \vdots \\ a_{p-1} \cdot n + b_{p-1}, & \text{if } n \equiv p-1 \pmod{p} \end{cases}$$

for some $p \in \mathbb{N}$ and rational numbers a_i, b_i such that $f(n) \in \mathbb{N}$ for all $n \in \mathbb{N}$.

The result of [11] is an immediate corollary of Theorem 2.2. Every Fractran program P is a Collatz function f'_P where f'_P is obtained from f_P (see Definition 2.1) by replacing undefined with 1. We obtain the above representation of Collatz functions simply by choosing for p the least common multiple of the denominators of the fractions of P. We call a Fractran program P *trivially immortal* if P contains a fraction with denominator 1 (an integer). Then for all not trivially immortal P, P halts on all inputs if and only for all $x > 0$ there exists $i \in \mathbb{N}$ such that $f_P^i(x) = 1$. Using our result, this implies that GCP is Π^0_2-hard.

Theorem 2.2 is a strengthening of the result in [11] since Fractran programs are a strict subset of Collatz functions. If Fractran programs are represented as Collatz functions directly, for all $0 \leq i < p$ it holds either $b_i = 0$, or $a_i = 0$ and $b_i = 1$. Via such a translation Fractran programs are, e.g., not able to implement the famous Collatz function $C(2n) = n$ and $C(2n+1) = 6n+4$ (for all $n \in \mathbb{N}$), nor an easy function like $f(2n) = 2n+1$ and $f(2n+1) = 2n$ (for all $n \in \mathbb{N}$).

For the proof of Theorem 2.2 we devise a translation from Turing machines to Fractran programs ([11] uses register machines) such that the resulting Fractran program halts on all positive integers ($n_0 \geq 1$) if and only if the Turing machine is terminating on all configurations. That is, we reduce the uniform halting problem of Turing machines to the uniform halting problem of Fractran programs.

We briefly explain why we employ the uniform halting problem instead of the problem of totality (termination on all inputs) of Turing machines, also known as the initialised uniform halting problem. When translating a Turing machine M to a Fractran program P_M, start configurations (initialised configurations) are mapped to a subset $I_M \subseteq \mathbb{N}$ of Fractran inputs. Then from Π^0_2-hardness of the totality problem one can conclude Π^0_2-hardness of the question whether P_M terminates on all numbers from I_M. But this does not imply that the uniform halting problem for Fractran programs is Π^0_2-hard (termination on all natural numbers $n \in \mathbb{N}$). The numbers not in the target of the translation could make the problem both harder as well as easier. A situation where extending the domain of inputs makes the problem easier is: local confluence of TRSs is Π^0_2-complete for the set of ground terms, but only Σ^0_1-complete for the set of all terms [3].

To keep the translation as simple we restrict to unary Turing machines having only two symbols $\{0, 1\}$ in their tape alphabet, 0 being the blank symbol.

Definition 2.3. A unary *Turing machine* M is a triple $\langle Q, q_0, \delta\rangle$, where Q is a finite set of states, $q_0 \in Q$ the initial state, and $\delta : Q \times \{0,1\} \rightharpoonup Q \times \{0,1\} \times \{L, R\}$ a (partial) *transition function*. A *configuration* of M is a pair $\langle q, \mathit{tape}\rangle$ consisting of a state $q \in Q$ and the tape content $\mathit{tape} : \mathbb{Z} \to \{0,1\}$ such that the support $\{n \in \mathbb{Z} \mid \mathit{tape}(n) \neq 0\}$ is finite. The set of all configurations is denoted by $\mathcal{C}\mathit{onf}_M$. We define the relation $\to_M$ on the set of configurations $\mathcal{C}\mathit{onf}_M$ as follows: $\langle q, \mathit{tape}\rangle \to_M \langle q', \mathit{tape}'\rangle$ whenever:

- $\delta(q, \mathit{tape}(0)) = \langle q', f, L\rangle$, $\mathit{tape}'(1) = f$ and $\forall n \neq 0.\ \mathit{tape}'(n+1) = \mathit{tape}(n)$, or
- $\delta(q, \mathit{tape}(0)) = \langle q', f, R\rangle$, $\mathit{tape}'(-1) = f$ and $\forall n \neq 0.\ \mathit{tape}'(n-1) = \mathit{tape}(n)$.

We say that M *halts (or terminates) on a configuration* $\langle q, \mathit{tape}\rangle$ if the configuration $\langle q, \mathit{tape}\rangle$ does not admit infinite $\to_M$ rewrite sequences.

The *uniform halting problem* of Turing machines is the problem of deciding whether a given Turing machine M halts on all (initial or intermediate) configurations. The following theorem is a result of [8]:

Theorem 2.4. *The uniform halting problem for Turing machines is* Π_2^0*-complete.*

This result carries over to unary Turing machines using a simulation based on a straightforward encoding of tape symbols as blocks of zeros and ones (of equal length), which are admissible configurations of unary Turing machines.

We now give a translation of Turing machines to Fractran programs. Without loss of generality we restrict in the sequel to Turing machines $M = \langle Q, q_0, \delta\rangle$ for which $\delta(q, x) = \langle q', s', d'\rangle$ implies $q \neq q'$. In case M does not fulfil this condition then we can find an equivalent Turing machine $M' = \langle Q \cup Q_\#, q_0, \delta'\rangle$ where $Q_\# = \{q_\# \mid q \in Q\}$ and δ' is defined by $\delta'(q, x) = \langle p_\#, s, d\rangle$ and $\delta'(q_\#, x) = \langle p, s, d\rangle$ for $\delta(q, x) = \langle p, s, d\rangle$.

Definition 2.5. Let $M = \langle Q, q_0, \delta\rangle$ be a Turing machine. Let $tape_\ell$, h, $tape_r$, $tape'_\ell$, h', $tape'_r$, $m_{L,x}$, $m_{R,x}$, $copy_x$ and p_q for every $q \in Q$ and $x \in \{0, 1\}$ be pairwise distinct prime numbers. The intuition behind these primes is:

- $tape_\ell$ and $tape_r$ represent the tape left and right of the head, respectively,
- h is the tape symbol in the cell currently scanned by the tape head,
- $tape'_\ell$, h', $tape'_r$ store temporary tape content (when moving the head),
- $m_{L,x}$, $m_{R,x}$ execute a left or right move of the head on the tape, respectively,
- $copy_x$ copies the temporary tape content back to the primary tape, and
- p_q represent the states of the Turing machine.

The subscript $x \in \{0, 1\}$ is used to have two primes for every action: in case an action p takes more than one calculation step we cannot write $\frac{p\cdot\ldots}{p\cdot\ldots}$ since then p in numerator and denominator would cancel itself out. We define the Fractran program P_M to consist of the following fractions (listed in program order):

$$\frac{1}{p \cdot p'} \qquad \begin{array}{c}\text{for every } p, p' \in \{m_{L,0},\, m_{L,1},\, m_{R,0},\, m_{R,1},\, copy_0,\, copy_1\} \\ \text{every } p, p' \in \{p_q \mid q \in Q\} \text{ and } p, p' \in \{h, h'\}\end{array} \tag{4}$$

to get rid of illegal configurations,

$$\frac{m_{L,1-x} \cdot tape'_\ell}{m_{L,x} \cdot tape_\ell^2} \quad \frac{m_{L,1-x} \cdot tape_r'^2}{m_{L,x} \cdot tape_r} \quad \frac{m_{L,1-x} \cdot tape'_r}{m_{L,x} \cdot h'} \quad \frac{m_{L,1-x} \cdot h}{m_{L,x} \cdot tape_\ell} \quad \frac{copy_0}{m_{L,x}} \tag{5}$$

with $x \in \{0, 1\}$, for moving the head left on the tape,

$$\frac{m_{R,1-x} \cdot tape'_r}{m_{R,x} \cdot tape_r^2} \quad \frac{m_{R,1-x} \cdot tape_\ell'^2}{m_{R,x} \cdot tape_\ell} \quad \frac{m_{R,1-x} \cdot tape'_\ell}{m_{R,x} \cdot h'} \quad \frac{m_{R,1-x} \cdot h}{m_{R,x} \cdot tape_r} \quad \frac{copy_0}{m_{R,x}} \tag{6}$$

with $x \in \{0, 1\}$, for moving the head right on the tape,

$$\frac{copy_{1-x} \cdot tape_\ell}{copy_x \cdot tape'_\ell} \qquad \frac{copy_{1-x} \cdot tape_r}{copy_x \cdot tape'_r} \qquad \frac{1}{copy_x} \tag{7}$$

with $x \in \{0,1\}$, for copying the temporary tape back to the primary tape,

$$\frac{p_{q'} \cdot h'^{s'} \cdot m_{d,0}}{p_q \cdot h} \qquad \text{whenever } \delta(q,1) = \langle q', s', d\rangle \tag{8}$$

$$\frac{1}{p_q \cdot h} \qquad \text{(for termination) for every } q \in Q \tag{9}$$

$$\frac{p_{q'} \cdot h'^{s'} \cdot m_{d,0}}{p_q} \qquad \text{whenever } \delta(q,0) = \langle q', s', d\rangle \tag{10}$$

for the transitions of the Turing machine. Whenever we use variables in the rules, e.g. $x \in \{0,1\}$, then it is to be understood that instances of the same rule are immediate successors in the sequence of fractions (the order of the instances among each other is not crucial).

Example 2.6. Let $M = \langle Q, a_0, \delta\rangle$ be a Turing machine where $Q = \{a_0, a_1, b\}$, and the transition function is defined by $\delta(a_0,0) = \langle b,1,R\rangle$, $\delta(a_1,0) = \langle b,1,R\rangle$, $\delta(a_0,1) = \langle a_1,0,R\rangle$, $\delta(a_1,1) = \langle a_0,0,R\rangle$, $\delta(b,1) = \langle a_0,0,R\rangle$, and we leave $\delta(b,0)$ undefined. That is, M moves to the right, converting zeros into ones and vice versa, until it finds two consecutive zeros and terminates. Assume that M is started on the configuration $1b1001$, that is, the tape content 11001 in state b with the head located on the second 1. In the Fractran program P_M this corresponds to $n_0 = p_b \cdot \mathit{tape}_\ell^1 \cdot h^1 \cdot \mathit{tape}_r^{100}$ as the start value where we represent the exponents in binary notation for better readability. Started on n_0 we obtain the following calculation in P_M:

$$\begin{gathered}
p_b \cdot \mathit{tape}_\ell^1 \cdot h^1 \cdot \mathit{tape}_r^{100} \qquad \text{(configuration } 1b1001\text{)} \\
\to_{(8)} m_{R,0} \cdot p_{a_0} \cdot \mathit{tape}_\ell^1 \cdot \mathit{tape}_r^{100} \to^2_{(6;1^{\text{st}})} m_{R,0} \cdot p_{a_0} \cdot \mathit{tape}_\ell^1 \cdot \mathit{tape}_r'^{10} \\
\to_{(6;2^{\text{nd}})} m_{R,1} \cdot p_{a_0} \cdot \mathit{tape}_\ell'^{10} \cdot \mathit{tape}_r'^{10} \to_{(6;5^{\text{th}})} \mathit{copy}_0 \cdot p_{a_0} \cdot \mathit{tape}_\ell'^{10} \cdot \mathit{tape}_r'^{10} \\
\to^2_{(7;1^{\text{st}})} \to^2_{(7;2^{\text{nd}})} \to_{(7;3^{\text{rd}})} p_{a_0} \cdot \mathit{tape}_\ell^{10} \cdot \mathit{tape}_r^{10} \qquad \text{(configuration } 10a001\text{)} \\
\to_{(10)} m_{R,0} \cdot p_b \cdot \mathit{tape}_\ell^{10} \cdot h'^1 \cdot \mathit{tape}_r^{10} \to_{(6;1^{\text{st}})} m_{R,1} \cdot p_b \cdot \mathit{tape}_\ell^{10} \cdot h'^1 \cdot \mathit{tape}_r'^1 \\
\to^2_{(6;2^{\text{nd}})} m_{R,1} \cdot p_b \cdot \mathit{tape}_\ell'^{100} \cdot h'^1 \cdot \mathit{tape}_r'^1 \to_{(6;3^{\text{rd}}+5^{\text{th}})} \mathit{copy}_0 \cdot p_b \cdot \mathit{tape}_\ell'^{101} \cdot \mathit{tape}_r'^1 \\
\to^5_{(7;1^{\text{st}})} \to_{(7;2^{\text{nd}})} \to_{(7;3^{\text{rd}})} p_b \cdot \mathit{tape}_\ell^{101} \cdot \mathit{tape}_r^1 \qquad \text{(configuration } 101b01\text{)}
\end{gathered}$$

reaching a configuration where the Fractran program halts.

Definition 2.7. We translate configurations $c = \langle q, \mathit{tape}\rangle$ of Turing machines $M = \langle Q, q_0, \delta\rangle$ to natural numbers (input values for Fractran programs). We reuse the notation of Definition 2.5 and define:

$$n_c = \mathit{tape}_\ell^L \cdot p_q \cdot h^H \cdot \mathit{tape}_r^R$$

$$L = \sum_{i=0}^{\infty} 2^i \cdot \mathit{tape}(-1-i) \qquad H = \mathit{tape}(0) \qquad R = \sum_{i=0}^{\infty} 2^i \cdot \mathit{tape}(1+i)$$

Lemma 2.8. *For every Turing machine M and configurations c_1, c_2 we have:*

(i) if $c_1 \to_M c_2$ then $n_{c_1} \to^*_{P_M} n_{c_2}$, and
(ii) if c_1 is a $\to_M$ normal form then $n_{c_1} \to^*_{P_M}$ undefined.

Proofs of Lemma 2.8 and Theorem 2.2 can be found in [5].

3 What is Productivity?

A program is productive if it evaluates to a finite or infinite constructor normal form. This rather vague description leaves open several choices that can be made to obtain a more formal definition. We explore several definitions and determine the degree of undecidability for each of them. See [6] for more pointers to the literature on productivity.

The following is a productive specification of the (infinite) stream of zeros:

$$\mathsf{zeros} \to 0 : \mathsf{zeros}$$

Indeed, there exists only one maximal rewrite sequence from zeros and this ends in the infinite constructor normal form $0 : 0 : 0 : \ldots$. Here and later we say that a rewrite sequence $\rho : t_0 \to t_1 \to t_2 \to \ldots$ *ends in* a term s if either ρ is finite with its last term being s, or ρ is infinite and then s is the limit of the sequence of terms t_i, i.e. $s = \lim_{i\to\infty} t_i$. We consider only rewrite sequences starting from finite terms, thus all terms occurring in ρ are finite. Nevertheless, the limit s of the terms t_i may be an infinite term. Note that, if ρ ends in a constructor normal form, then every finite prefix will be evaluated after finitely many steps.

The following is a slightly modified specification of the stream of zeros:

$$\mathsf{zeros} \to 0 : \mathsf{id}(\mathsf{zeros}) \qquad\qquad \mathsf{id}(\sigma) \to \sigma$$

This specification is considered productive as well, although there are infinite rewrite sequences that do not even end in a normal form, let alone in a constructor normal form: e.g. by unfolding zeros only we get the limit term $0 : \mathsf{id}(0 : \mathsf{id}(0 : \mathsf{id}(\ldots)))$. In general, normal forms can only be reached by outermost-fair rewriting sequences. A rewrite sequence $\rho : t_0 \to t_1 \to t_2 \to \ldots$ is *outermost-fair* [16] if there is no t_n containing an outermost redex which remains an outermost redex infinitely long, and which is never contracted. For this reason it is natural to consider productivity of terms with respect to outermost-fair strategies.

What about stream specifications that admit rewrite sequences to constructor normal forms, but that also have divergent rewrite sequences:

$$\mathsf{maybe} \to 0 : \mathsf{maybe} \qquad \mathsf{maybe} \to \mathsf{sink} \qquad \mathsf{sink} \to \mathsf{sink}$$

This example illustrates that, for non-orthogonal stream specifications, reachability of a constructor normal form depends on the evaluation strategy. The term maybe is only productive with respect to strategies that always apply the first rule.

For this reason we propose to think of productivity as a property of individual terms *with respect to* a given rewrite strategy. This reflects the situation in functional programming, where expressions are evaluated according to an inbuilt strategy. These strategies are usually based on a form of outermost-needed rewriting with a priority order on the rules.

3.1 Productivity with respect to Strategies

For term rewriting systems (TRSs) [16] we now fix definitions of the notions of (history-free) strategy and history-aware strategy. Examples for the latter notion are outermost-fair strategies, which typically have to take history into account.

Definition 3.1. Let R be a TRS with rewrite relation $\to_R$.

A *strategy for* $\to_R$ is a relation $\rightsquigarrow \subseteq \to_R$ with the same normal forms as $\to_R$.

The *history-aware rewrite relation* $\to_{\mathcal{H},R}$ *for* R is the binary relation on $Ter(\Sigma) \times (R \times \mathbb{N}^*)^*$ that is defined by:

$$\langle s, h_s\rangle \to_{\mathcal{H},R} \langle t, h_s : \langle \rho, p\rangle\rangle \iff s \to t \text{ via rule } \rho \in R \text{ at position } p\,.$$

We identify $t \in Ter(\Sigma)$ with $\langle t, \epsilon\rangle$, and for $s, t \in Ter(\Sigma)$ we write $s \to_{\mathcal{H},R} t$ whenever $\langle s, \epsilon\rangle \to_{\mathcal{H},R} \langle t, h\rangle$ for some history $h \in (R \times \mathbb{N}^*)^*$. A *history-aware strategy for* R is a strategy for $\to_{\mathcal{H},R}$.

A strategy $\rightsquigarrow$ is *deterministic* if $s \rightsquigarrow t$ and $s \rightsquigarrow t'$ implies $t = t'$. A strategy $\rightsquigarrow$ is *computable* if the function mapping a term (a term/history pair) to its set of $\rightsquigarrow$-successors is a total recursive function, after coding into natural numbers.

Remark 3.2. Our definition of strategy for a rewrite relation follows [17]. For abstract rewriting systems, in which rewrite steps are first-class citizens, a definition of strategy is given in [16, Ch. 9]. There, history-aware strategies for a TRS R are defined in terms of 'labellings' for the 'abstract rewriting system' underlying R. While that approach is conceptually advantageous, our definition of history-aware strategy is equally expressive.

Definition 3.3. A *(TRS-indexed) family of strategies* $\mathcal{S}$ is a function that assigns to every TRS R a set $\mathcal{S}(R)$ of strategies for R. We call such a family $\mathcal{S}$ of strategies *admissible* if $\mathcal{S}(R)$ is non-empty for every orthogonal TRS R.

Now we give the definition of productivity with respect to a strategy.

Definition 3.4. A term t is called *productive with respect to a strategy* $\rightsquigarrow$ if all maximal $\rightsquigarrow$ rewrite sequences starting from t end in a constructor normal form.

In the case of non-deterministic strategies we require here that all maximal rewrite sequences end in a constructor normal form. Another possible choice could be to require only the existence of one such rewrite sequence (see Section 3.2). However, we think that productivity should be a practical notion. Productivity of a term should entail that arbitrary finite parts of the constructor normal form can indeed be evaluated. The mere requirement that a constructor normal form exists leaves open the possibility that such a normal form cannot be approximated to every finite precision in a computable way.

For orthogonal TRSs outermost-fair (or fair) rewrite strategies are the natural choice for investigating productivity because they guarantee to find (the unique) infinitary constructor normal form whenever it exists (see [16]).

Pairs and finite lists of natural numbers can be encoded using the well-known Gödel encoding. Likewise terms and finite TRSs over a countable set of variables can be encoded. A TRS is called *finite* if its signature and set of rules are finite. In the sequel we restrict to (families of) computable strategies, and assume that

strategies are represented by appropriate encodings.

Now we define the productivity problem in TRSs with respect to families of computable strategies, and prove a Π^0_2-completeness result.

PRODUCTIVITY PROBLEM with respect to a family $\mathcal{S}$ of computable strategies
Instance: Encodings of a finite TRS R, a strategy $\rightsquigarrow \in \mathcal{S}(R)$ and a term t.
Answer: 'Yes' if t is productive with respect to $\rightsquigarrow$, and 'No', otherwise.

Theorem 3.5. *For every family of admissible, computable strategies $\mathcal{S}$, the productivity problem with respect to $\mathcal{S}$ is Π^0_2-complete.*

Proof. A Turing machine is called *total* (encodes a total function $\mathbb{N} \to \mathbb{N}$) if it halts on all inputs encoding natural numbers. The problem of deciding whether a Turing machine is *total* is well-known to be Π^0_2-complete, see [9]. Let M be an arbitrary Turing machine. Employing the encoding of Turing machines into orthogonal TRSs from [10], we can define a TRS R_M that simulates M such that for every $n \in \mathbb{N}$ it holds: every reduct of the term $\mathsf{M}(\mathsf{s}^n(0))$ contains at most one redex occurrence, and the term $\mathsf{M}(\mathsf{s}^n(0))$ rewrites to 0 if and only if the Turing machine M halts on the input n. Note that the rewrite sequence starting from $\mathsf{M}(\mathsf{s}^n(0))$ is deterministic. We extend the TRS R_M to a TRS R'_M with the following rules:

$$\mathsf{go}(0, x) \to 0 : \mathsf{go}(\mathsf{M}(x), \mathsf{s}(x))$$

and choose the term $t = \mathsf{go}(0, 0)$. Then R'_M is orthogonal and by construction every reduct of t contains at most one redex occurrence (consequently all strategies for R coincide on every reduct of t). The term t is productive if and only if $\mathsf{M}(\mathsf{s}^n(0))$ rewrites to 0 for every $n \in \mathbb{N}$ which in turn holds if and only if the Turing machine M is total. This concludes Π^0_2-hardness.

For Π^0_2-completeness let $\mathcal{S}$ be a family of computable strategies, R a TRS, $\rightsquigarrow \in \mathcal{S}(R)$ and t a term. Then productivity of t can be characterised as:

$$\forall d \in \mathbb{N}.\, \exists n \in \mathbb{N}.\, \text{every } n\text{-step } \rightsquigarrow\text{-reducts of } t \text{ is a constructor normal form up to depth } d \qquad (\star)$$

Since the strategy $\rightsquigarrow$ is computable and finitely branching, all n-step reducts of t can be computed. Obviously, if the formula $(\star)$ holds, then t is productive w.r.t. $\rightsquigarrow$. Conversely, assume that t is productive w.r.t. $\rightsquigarrow$. For showing $(\star)$, let $d \in \mathbb{N}$ be arbitrary. By productivity of t w.r.t. $\rightsquigarrow$, on every path in the reduction graph of t w.r.t. $\rightsquigarrow$ eventually a term with a constructor normal form up to depth d is encountered. Since reduction graphs in TRSs always are finitely branching, Koenig's lemma implies that there exists an $n \in \mathbb{N}$ such that all terms on depth greater or equal to n in the reduction graph of t are constructor prefixes of depth at least d. Since d was arbitrary, $(\star)$ has been established. Because $(\star)$ is a Π^0_2-formula, the productivity problem with respect to $\mathcal{S}$ also belongs to Π^0_2. □

Theorem 3.5 implies that productivity is Π^0_2-complete for orthogonal TRSs with respect to outermost-fair rewriting. To see this, apply the theorem to the family of strategies that assigns to every orthogonal TRS R the set of computable, outermost-fair rewriting strategies for R, and $\varnothing$ to non-orthogonal TRSs.

The definition of productivity with respect to computable strategies reflects the situation in functional programming. Nevertheless, we now investigate variants of this notion, and determine their respective computational complexity.

3.2 Strong Productivity

As already discussed, only outermost-fair rewrite sequences can reach a constructor normal form. Dropping the fine tuning device 'strategies', we obtain the following stricter notion of productivity.

Definition 3.6. A term t is called *strongly productive* if all maximal outermost-fair rewrite sequences starting from t end in a constructor normal form.

The definition requires all outermost-fair rewrite sequences to end in a constructor normal form, including non-computable rewrite sequences. This catapults productivity into a much higher class of undecidability: Π^1_1, a class of the analytical hierarchy. The analytical hierarchy continues the classification of the arithmetical hierarchy using second order formulas. The computational complexity of strong productivity therefore exceeds the expressive power of first-order logic to define sets from recursive sets.

A well-known result of recursion theory states that for a given computable relation $> \subseteq \mathbb{N} \times \mathbb{N}$ it is Π^1_1-hard to decide whether $>$ is well-founded, see [9]. Our proof is based on a construction from [3]. There a translation from Turing machines M to TRSs $\mathcal{R}oot_M$ (which we explain below) together with a term t_M is given such that: t_M is root-terminating (i.e., t_M admits no rewrite sequences containing an infinite number of root steps) if and only if the binary relation $>_M$ encoded by M is well-founded. The TRS $\mathcal{R}oot_M$ consists of rules for simulating the Turing machine M such that $M(x, y) \to^* \mathsf{T}$ iff $x >_M y$ holds (which basically uses a standard encoding of Turing machines, see [10]), a rule:

$$\mathsf{run}(\mathsf{T}, \mathsf{ok}(x), \mathsf{ok}(y)) \to \mathsf{run}(M(x, y), \mathsf{ok}(y), \mathsf{pickn})$$

and rules for randomly generating a natural number:

$$\mathsf{pickn} \to \mathsf{c}(\mathsf{pickn}) \qquad \mathsf{pickn} \to \mathsf{ok}(0(\triangleright)) \qquad \mathsf{c}(\mathsf{ok}(x)) \to \mathsf{ok}(\mathsf{S}(x))\,.$$

The term $t_M = \mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn})$ admits a rewrite sequence containing infinitely many root steps if and only if $>_M$ is not well-founded. More precisely, whenever there is an infinite decreasing sequence $x_1 >_M x_2 >_M x_3 >_M \ldots$, then t_M admits a rewrite sequence $\mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn}) \to^* \mathsf{run}(\mathsf{T}, \mathsf{ok}(x_1), \mathsf{ok}(x_2)) \to \mathsf{run}(M(x_1, x_2), \mathsf{ok}(x_2), \mathsf{pickn}) \to^* \mathsf{run}(\mathsf{T}, \mathsf{ok}(x_2), \mathsf{ok}(x_3)) \to^* \ldots$. We further note that t_M and all of its reducts contain exactly one occurrence of the symbol run, namely at the root position.

Theorem 3.7. *Strong productivity is Π^1_1-complete.*

Proof. For the proof of Π^1_1-hardness, let M be a Turing machine. We extend the TRS $\mathcal{R}oot_M$ from [3] with the rule $\mathsf{run}(x, y, z) \to 0{:}\mathsf{run}(x, y, z)$. As a consequence the term $\mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn})$ is strongly productive if and only if $>_M$ is well-founded (which is Π^1_1-hard to decide). If $>_M$ is not well-founded, then by the result in [3] t_M admits a rewrite sequence containing infinitely many root steps

which obviously does not end in a constructor normal form. On the other hand if $>_M$ is well-founded, then t_M admits only finitely many root steps with respect to $\mathcal{R}oot_M$, and thus by outermost-fairness the freshly added rule has to be applied infinitely often. This concludes Π^1_1-hardness.

Rewrite sequences of length ω can be represented by functions $r : \mathbb{N} \to \mathbb{N}$ where $r(n)$ represents the n-th term of the sequence together with the position and rule applied in step n. Then for all r (one universal $\forall r$ function quantifier) we have to check that r converges towards a constructor normal form whenever r is outermost-fair; this can be checked by a first order formula. We refer to [3] for the details of the encoding. Hence strong productivity is in Π^1_1. □

3.3 Weak Productivity

A natural counterpart to strong productivity is the notion of 'weak productivity': the existence of a rewrite sequence to a constructor normal form. Here outermost-fairness does not need to be required, because rewrite sequences that reach normal forms are always outermost-fair.

Definition 3.8. A term t is called *weakly productive* if there exists a rewrite sequence starting from t that ends in a constructor normal form.

For non-orthogonal TRSs the practical relevance of this definition is questionable since, in the absence of a computable strategy to reach normal forms, mere knowledge that a term t is productive does typically not help to find a constructor normal form of t. For orthogonal TRSs computable, normalising strategies exist, but then also all of the variants of productivity coincide (see Section 3.4).

Theorem 3.9. *Weak productivity is Σ^1_1-complete.*

Proof. For the proof of Σ^1_1-hardness, let M be a Turing machine. We exchange the rule $\mathsf{run}(\mathsf{T}, \mathsf{ok}(x), \mathsf{ok}(y)) \to \mathsf{run}(M(x,y), \mathsf{ok}(y), \mathsf{pickn})$ in the TRS $\mathcal{R}oot_M$ from [3] by the rule $\mathsf{run}(\mathsf{T}, \mathsf{ok}(x), \mathsf{ok}(y)) \to 0 : \mathsf{run}(M(x,y), \mathsf{ok}(y), \mathsf{pickn})$. Then we obtain that the term $\mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn})$ is weakly productive if and only if $>_M$ is not well-founded (which is Σ^1_1-hard to decide). This concludes Π^1_1-hardness.

The remainder of the proof proceeds analogously to the proof of Theorem 3.7, except that we now have an existential function quantifier $\exists r$ to quantify over all rewrite sequences of length ω. Hence weak productivity is in Σ^1_1. □

3.4 Discussion

For orthogonal TRSs all of the variants of productivity coincide. That is, if we restrict the first variant to computable outermost-fair strategies;A as already discussed, other strategies are not very reasonable. For orthogonal TRSs there always exist computable outermost-fair strategies, and whenever for a term there exists a constructor normal form, then it is unique and all outermost-fair rewrite sequences will end in this unique constructor normal form.

This raises the question whether uniqueness of the constructor normal forms should be part of the definition of productivity. We consider a specification of

the stream of random bits:

$$\mathsf{random} \to 0 : \mathsf{random} \qquad\qquad \mathsf{random} \to 1 : \mathsf{random}$$

Every rewrite sequence starting from random ends in a normal form. However, these normal forms are not unique. In fact, there are uncountably many of them. We did not include uniqueness of normal forms into the definition of productivity since non-uniqueness only arises in non-orthogonal TRSs when using non-deterministic strategies. However, one might want to require uniqueness of normal forms even in the case of non-orthogonal TRSs.

Theorem 3.10. *The problem of determining, for TRSs R and terms t in R, whether t has a unique (finite or infinite) normal form is Π^1_1-complete.*

Proof. For Π^1_1-hardness, we extend the TRS constructed in the proof of Theorem 3.9 by the rules: $\mathsf{start} \to \mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn})$, $\mathsf{run}(x, y, z) \to \mathsf{run}(x, y, z)$, $\mathsf{start} \to \mathsf{ones}$, and $\mathsf{ones} \to 1 : \mathsf{ones}$. Then start has a unique normal form if and only if $>_M$ is well-founded. For Π^1_1-completeness, we observe that the property can be characterised by a Π^1_1-formula: we quantify over two infinite rewrite sequences, and, in case both of them end in a normal form, we compare them. Note that consecutive universal quantifiers can be compressed into one. □

Let us consider the impact on computational complexity of taking up the condition of uniqueness of normal forms into the definition of productivity. Including uniqueness of normal forms without considering the strategy would increase the complexity of productivity with respect to a family of strategies to Π^1_1. However, we think that doing so would be contrary to the spirit of the notion of productivity. Uniqueness of normal forms should only be required for the normal forms reachable by the given (non-deterministic) strategy. But then the complexity of productivity remains unchanged, Π^0_2-complete. The complexity of strong productivity remains unaltered, Π^1_1-complete, when including uniqueness of normal forms. However, the degree of undecidability of weak productivity increases. From the proofs of Theorems 3.9 and 3.10 it follows that the property would then both be Σ^1_1-hard and Π^1_1-hard, then being in Δ^1_1.

4 Productivity for Lazy Stream Specifications is Π^0_2

In this section we strengthen the undecidability result of Theorem 3.5 by showing that the productivity problem is Π^0_2-complete already for a very simple format of stream specifications, namely the lazy stream format (LSF) introduced on page 3. We do so by giving a translation from Fractran programs into LSF and applying Theorem 2.2.

Definition 4.1. Let $P = \frac{p_1}{q_1}, \ldots, \frac{p_k}{q_k}$ be a Fractran program. Let d be the least common multiple of the denominators of P, that is, $d := \mathrm{lcm}(q_1, \ldots, q_k)$. Then for $n = 1, \ldots, d$ define $p'_n = p_i \cdot (d/q_i)$ and $b_n = n \cdot \frac{p_i}{q_i}$ where $\frac{p_i}{q_i}$ is the first fraction of P such that $n \cdot \frac{p_i}{q_i}$ is an integer, and we let p'_n and b_n be undefined if no such fraction exists. Then, the *stream specification induced by P* is a term rewriting

system $\mathcal{R}_P = \langle \Sigma_P, R_P \rangle$ with:

$$\Sigma_P = \{\bullet, :, \mathsf{head}, \mathsf{tail}, \mathsf{zip}_d, \mathsf{M}_P\} \cup \{\mathsf{mod}_{p'_n} \mid p'_n \text{ is defined}\}$$

and with R_P consisting of the following rules:

$$\mathsf{M}_P \to \mathsf{zip}_d(\mathsf{T}_1, \ldots, \mathsf{T}_d), \text{ where, for } 1 \leq n \leq d, \mathsf{T}_n \text{ is shorthand for:}$$

$$\mathsf{T}_n = \begin{cases} \mathsf{mod}_{p'_n}(\mathsf{tail}^{b_n - 1}(\mathsf{M}_P)) & \text{if } p'_n \text{ is defined,} \\ \bullet : \mathsf{mod}_d(\mathsf{tail}^{n-1}(\mathsf{M}_P)) & \text{if } p'_n \text{ is undefined.} \end{cases}$$

$$\mathsf{head}(x : \sigma) \to x \qquad \mathsf{mod}_k(\sigma) \to \mathsf{head}(\sigma) : \mathsf{mod}_k(\mathsf{tail}^k(\sigma))$$

$$\mathsf{tail}(x : \sigma) \to \sigma \qquad \mathsf{zip}_d(\sigma_1, \sigma_2 \ldots, \sigma_d) \to \mathsf{head}(\sigma_1) : \mathsf{zip}_d(\sigma_2, \ldots, \sigma_d, \mathsf{tail}(\sigma_1))$$

where x, σ, σ_i are variables.[1]

The rule for mod_n defines a stream function which takes from a given stream σ all elements $\sigma(i)$ with $i \equiv 0 \pmod{n}$, and results in a stream consisting of those elements in the original order. As we only need rules $\mathsf{mod}_{p'_n}$ whenever p'_n is defined we need d such rules at most.

If p'_n is undefined then it should be understood that $m \cdot p'_n$ is undefined. For $n \in \mathbb{N}$ let $\varphi(n)$ denote the number from $\{1, \ldots, d\}$ with $n \equiv \varphi(n) \pmod{d}$.

Lemma 4.2. *For every $n > 0$ we have $f_P(n) = \lfloor (n-1)/d \rfloor \cdot p'_{\varphi(n)} + b_{\varphi(n)}$.*

Proof. Let $n > 0$. For every $i \in \{1, \ldots, k\}$ we have $n \cdot \frac{p_i}{q_i} \in \mathbb{N}$ if and only if $\varphi(n) \cdot \frac{p_i}{q_i} \in \mathbb{N}$, since $n \equiv \varphi(n) \bmod d$ and d is a multiple of q_i. Assume that $f_P(n)$ is defined. Then $f_P(n) = n \cdot p'_{\varphi(n)}/d = (\lfloor (n-1)/d \rfloor \cdot d + ((n-1) \bmod d) + 1) \cdot p'_{\varphi(n)}/d$ $= \lfloor (n-1)/d \rfloor \cdot p'_{\varphi(n)} + \varphi(n) \cdot p_i/q_i = \lfloor (n-1)/d \rfloor \cdot p'_{\varphi(n)} + b_{\varphi(n)}$. Otherwise whenever $f_P(n)$ is undefined then $p'_{\varphi(n)}$ is undefined. □

Lemma 4.3. *Let P be a Fractran program. Then $\mathcal{R}_P$ is productive for M_P if and only if P is terminating on all integers $n > 0$.*

Proof. Let $\sigma(n)$ be shorthand for $\mathsf{head}(\mathsf{tail}^n(\sigma))$. It suffices to show for all $n \in \mathbb{N}$: $\mathsf{M}_P(n) \to^* \bullet$ if and only if P halts on n. For this purpose we show $\mathsf{M}_P(n) \to^+ \bullet$ whenever $f_P(n+1)$ is undefined, and $\mathsf{M}_P(n) \to^+ \mathsf{M}_P(f_P(n+1) - 1)$, otherwise. We have $\mathsf{M}_P(n) \to^* \mathsf{T}_{\varphi(n+1)}(\lfloor n/d \rfloor)$.

Assume that $f_P(n+1)$ is undefined. By Lemma 4.2 $p'_{\varphi(n+1)}$ is undefined, thus thus $\mathsf{M}_P(n) \to^* \bullet$ whenever $\lfloor n/d \rfloor = 0$, and otherwise we have:

$$\mathsf{M}_P(n) \to^* \mathsf{T}_{\varphi(n+1)}(\lfloor n/d \rfloor) \to^* \mathsf{mod}_d(\mathsf{tail}^{\varphi(n+1)-1}(\mathsf{M}_P))(\lfloor n/d \rfloor - 1) \to^* \mathsf{M}_P(n')$$

where $n' = (\lfloor n/d \rfloor - 1) \cdot d + \varphi(n+1) - 1 = n - d$. Clearly $n \equiv n' \pmod{d}$, and then $\mathsf{M}_P(n) \to^* \bullet$ follows by induction on n.

Assume that $f_P(n+1)$ is defined. By Lemma 4.2 $p'_{\varphi(n+1)}$ is defined and:

$$\mathsf{M}_P(n) \to^* \mathsf{T}_{\varphi(n+1)}(\lfloor n/d \rfloor) \to^* \mathsf{mod}_{p'_{\varphi(n+1)}}(\mathsf{tail}^{b_{\varphi(n+1)}-1}(\mathsf{M}_P))(\lfloor n/d \rfloor)$$

[1] Note that $\mathsf{mod}_d(\mathsf{tail}^{n-1}(\mathsf{zip}_d(\mathsf{T}_1, \ldots, \mathsf{T}_d)))$ equals T_n, and so, in case p'_n is undefined, we just have $\mathsf{T}_n = \bullet : \mathsf{T}_n$. In order to have the simplest TRS possible (for the purpose at hand), we did not want to use an extra symbol ($\bullet$) and rule $(\bullet) \to \bullet : (\bullet)$.

and hence $\mathsf{M}_P(n) \to^+ \mathsf{M}_P(n')$ with $n' = \lfloor n/d \rfloor \cdot p'_{\varphi(n+1)} + b_{\varphi(n+1)} - 1$. Then we have $n' = f_P(n+1) - 1$ by Lemma 4.2. □

Theorem 4.4. *The restriction of the productivity problem to stream specifications induced by Fractran programs and outermost-fair strategies is* Π^0_2*-complete.*

Proof. Since by Lemma 4.3 the uniform halting problem for Fractran programs can be reduced to the problem here, Π^0_2-hardness is a consequence of Theorem 2.2. Π^0_2-completeness follows from membership of the problem in Π^0_2, which can be established analogously as in the proof of Theorem 3.5. □

Note that Theorem 4.4 also gives rise to an alternative proof for the Π^0_2-hardness part of Theorem 3.5, the result concerning the computational complexity of productivity with respect to strategies.

References

1. T. Arts and J. Giesl. Termination of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. J.H. Conway. Fractran: A Simple Universal Programming Language for Arithmetic. In *Open Problems in Communication and Computation*, pages 4–26. Springer, 1987.
3. J. Endrullis, H. Geuvers, and H. Zantema. Degrees of Undecidabililty of TRS Properties. `http://arxiv.org/abs/0902.4723`, 2009.
4. J. Endrullis, C. Grabmayer, and D. Hendriks. ProPro: an Automated Productivity Prover. `http://infinity.few.vu.nl/productivity/`, 2008.
5. J. Endrullis, C. Grabmayer, and D. Hendriks. Complexity of Fractran and Productivity. `http://arxiv.org/abs/0903.4366`, 2009.
6. J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J.W. Klop. Productivity of Stream Definitions. In *FCT 2007*, number 4639 in LNCS, pages 274–287. Springer, 2007.
7. J. Grue Simonsen. The Π^0_2-Completeness of Most of the Properties of Rewriting Systems You Care About (and Productivity). In *RTA'09*, 2009. To appear.
8. G.T. Herman. Strong Computability and Variants of the Uniform Halting Problem. *Zeitschrift für Math. Logik und Grundlagen der Mathematik*, 17(1):115–131, 1971.
9. P.G. Hinman. *Recursion-Theoretic Hierarchies*. Springer, 1978.
10. J.W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.
11. S.A. Kurtz and J. Simon. The Undecidability of the Generalized Collatz Problem. In *TAMC '07*, volume 4484 of *LNCS*, pages 542–553. Springer, 2007.
12. J.C. Lagarias. The $3x + 1$ Problem and its Generalizations. *AMM*, 92(1):3–23, 1985.
13. S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice–Hall, 1987.
14. G. Roşu. Equality of Streams is a Π^0_2-complete Problem. In *ICFP*, pages 184–191, 2006.
15. B.A. Sijtsma. On the Productivity of Recursive List Definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.
16. Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
17. Y. Toyama. Strong Sequentiality of Left-Linear Overlapping Term Rewriting Systems. In *LICS*, pages 274–284. IEEE Computer Society Press, Los Alamitos, 1992.