Turing-Completeness of Polymorphic Stream Equation Systems

Christian Sattler and Florent Balestrieri

School of Computer Science, University of Nottingham
Nottingham, NG8 1BB, UK
`{cvs,fyb}@cs.nott.ac.uk`

Abstract

Polymorphic stream functions operate on the structure of streams, infinite sequences of elements, without inspection of the contained data, having to work on all streams over all signatures uniformly. A natural, yet restrictive class of polymorphic stream functions comprises those definable by a system of equations using only stream constructors and destructors and recursive calls. Using methods reminiscent of prior results in the field, we first show this class consists of exactly the computable polymorphic stream functions. Using much more intricate techniques, our main result states this holds true even for unary equations free of mutual recursion, yielding an elegant model of Turing-completeness in a severely restricted environment and allowing us to recover previous complexity results in a much more restricted setting.

1998 ACM Subject Classification F.1.1 Models of Computation; F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases stream functions, Turing-completeness, recursion

Digital Object Identifier 10.4230/LIPIcs.RTA.2012.256

Category Regular Research Paper

1 Introduction

Streams over some set $\mathbb{D}$ are the basic example of a polymorphic coinductive data type, having been forced to serve as case study for almost every technique dealing with infinite data structures. Although being well-explored coalgebraic objects [5, 3, 19], they have recently re-emerged in the specific setting of term rewriting [22, 6]. They are usually introduced as the coinductive data type $\mathrm{Str}_{\mathbb{D}}$ generated by the constructor $\cdot :: \cdot : \mathbb{D} \times \mathrm{Str}_{\mathbb{D}} \to \mathrm{Str}_{\mathbb{D}}$ (cons), appending an element to the front of a stream, and come with destructors $\mathrm{head} : \mathrm{Str}_{\mathbb{D}} \to \mathbb{D}$ and $\mathrm{tail} : \mathrm{Str}_{\mathbb{D}} \to \mathrm{Str}_{\mathbb{D}}$, selecting and removing the front element, respectively. Algebraically, they can be characterised as an infinite term model parameterised by the value type $\mathbb{D}$ modulo observational equivalence on $\mathbb{D}$.

Since this work is concerned with computability and partiality, we choose to work in a semantics of partial streams, adding a bottom element $\perp$ to the underlying data type. Technically, such streams are just functions of type $\mathbb{N} \to \mathbb{D}_{\perp}$. Note that with this terminology, if an element of a stream equals $\perp$, further elements can still be proper inhabitants of $\mathbb{D}$. Also, when speaking of computable (stream) functions, we always mean partial computable (stream) functions.

One of the simplest classes of functions on streams are the polymorphic stream functions, being parametric in the data type $\mathbb{D}$. This prohibits any kind of pattern matching or case distinction on the underlying data type, effectively restricting them to discarding, duplicating, and reordering of the input stream elements. This defines an indexing function which in the unary case has type $\mathbb{N} \to \mathbb{N}_\perp$, associating with each output stream index the corresponding input stream index to copy from, or $\perp$ if the output element is $\perp$. We can consider a polymorphic stream function f computable if this indexing function, denoted $\overline{f}$, is computable. What we call indexing function is a container morphism for streams in the terminology of Abott et al. [1].

We consider recursive stream equation systems for specifying polymorphic stream functions involving only stream constructors and destructors. As a representative example, consider the system

$$\begin{aligned} \mathrm{const}(s) &= \mathrm{head}(s) :: \mathrm{const}(s), \\ \mathrm{zip}_2(s,t) &= \mathrm{head}(s) :: \mathrm{zip}_2(t, \mathrm{tail}(s)), \\ \mathrm{hanoi}(s) &= \mathrm{zip}_2(\mathrm{hanoi}(\mathrm{tail}(s)), \mathrm{const}(s)). \end{aligned}$$

Through evaluation, which will be elaborated upon in the examples subsection, we find that

$$\mathrm{hanoi}(s) = \perp :: s(0) :: s(1) :: s(0) :: s(2) :: s(0) :: s(1) :: s(0) :: s(3) :: \ldots.$$

The corresponding indexing function is $\overline{\mathrm{hanoi}}(k) = \max\{v \text{ such that } 2^v \text{ divides } k\}$ where $\max\{\mathbb{N}\} := \perp$. To explain the naming, if $\mathbb{D}$ is instantiated with the set of disks of an infinite Tower of Hanoi and $s \in \mathrm{Str}_\mathbb{D}$ is a list of the disks sorted by increasing size, then $\mathrm{tail}(\mathrm{hanoi}(s))$ is a walkthrough for coinductively solving the puzzle, the k-th stream element being the disk to be moved in the k-th step, with the smallest disk always moving in the same direction [12].

The key point to stress is that polymorphism is a severe restriction. Constructing examples less trivial than the above seems out of reach: the reader is invited to try to encode the Fibonacci sequence as the indexing function of an equation in a polymorphic system.

Most proofs of undecidability and complexity results for stream equations, like the ones of Roşu [18] and Simonsen [20], use straightforward encodings of Turing machines, representing the infinite band of symbols as two streams, one each for the left and right side of the band relative to the head, with canonical rewrite rules pattern matching on the current symbol. This central dispatching mechanism is unavailable in our setting. Still, we are able to recover all of these results even in the unary setting as direct corollaries of Theorem 14.

As a first taste, Proposition 3 states that our limited systems are nevertheless still sufficient to define every computable polymorphic stream function. Although the construction requires some imagination, the simulation of counter machines is quite direct and mainly intended to give the reader some intuition for the long road towards the proof of our key result, Theorem 14, which improves upon this by restricting systems to unary stream functions without mutual recursion. This is the main contribution of our work, which seems surprising giving the crippled expressiveness of the syntax.

Endrullis et al. [7, 8, 9] strive to decompose rewriting into a stream layer and a data layer in such a way as to encapsulate just so much complexity into the data layer that the productivity of streams becomes decidable while still retaining usefulness of computation. Our work can be seen as a another extreme, eradicating the data layer and showing that polymorphic unary stream functions attain computational completeness. For example, our results imply that the lazy stream formats of Endrullis et al. [8] can actually be restricted to (general) unary stream functions with productivity still retaining Π^0_2-completeness (in the non-unary case, a hint of Proposition 3 can be found in their encoding of FRACTRAN-programs). We note that their notions of lazy stream specifications and data-oblivious analysis shares some points with our polymorphism restriction: choosing the unit type for the data type leaves no possibility of analysing the input. We also note that the flat stream specifications, for which the authors develop an algorithm for semi-deciding productivity, present an exception: we allow general nested calls.

See Simonsen [20] for a good survey of some of the complexity analysis on stream rewriting our developments generalise. We hope that our Turing-complete unary recursive systems, in their simplicity, may be used as a computational model in further reduction proofs (e.g. of complexity results) not only in rewriting theory. We conclude by remarking that all our proofs are constructive, i.e. algorithmically implementable.

2 Syntax and Semantics

2.1 Streams and Indexing Functions

Given a set $\mathbb{D}$, we denote $\mathbb{D}_\perp := \mathbb{D} \cup \{\perp\}$ the set together with a distinguished *bottom element* $\perp$, which is used to denotate partial or non-terminating computation on the object-level of stream reduction. We warn that on the meta-level, $\perp$ is in quoted form, i.e. treated as a normal element. Given a function $f : A \to B_\perp$, we adopt the convention of implicitly extending the domain of f to $A_\perp$ by setting $f(\perp) = \perp$.

The set of (partial) streams over $\mathbb{D}$ is defined as $\mathrm{Str}_\mathbb{D} := \mathbb{N} \to \mathbb{D}_\perp$. A stream s is *total* if $s(k) \neq \perp$ for all $k \in \mathbb{N}$. A *polymorphic stream function* is a family of functions

$$f_\mathbb{D} : \mathrm{Str}_\mathbb{D} \times \ldots \times \mathrm{Str}_\mathbb{D} \to \mathrm{Str}_\mathbb{D}$$

natural (or *parametric*) in the domain argument $\mathbb{D}$, i.e. for all sets $\mathbb{D}_1, \mathbb{D}_2$ and functions $g : \mathbb{D}_1 \to (\mathbb{D}_2)_\perp$, we have

$$(\mathrm{map}\, g) \circ f_{\mathbb{D}_1} = f_{\mathbb{D}_2} \circ (\mathrm{map}\, g, \ldots, \mathrm{map}\, g)$$

where $\mathrm{map}\, g : \mathrm{Str}_{\mathbb{D}_1} \to \mathrm{Str}_{\mathbb{D}_2}, s \mapsto g \circ s$. Such a function is called *total* if total arguments yield total results.

Even though a polymorphic stream function is an operation on streams, its parametricity property enables us to express it as a single stream on a particular domain. To see this, consider a stream function f of arity n. Let $I_n := \{0, \ldots, n-1\} \times \mathbb{N}$. In the following, we exploit the fact that the type of the argument of $f_\mathbb{D}$ is isomorphic to $I_n \to \mathbb{D}_\perp$. Define $\overline{f} : \mathbb{N} \to (I_n)_\perp$, $\overline{f} := f_{I_n}(j \mapsto (i,j))_{i \in \{0,\ldots,n-1\}}$. Essentially, this is f_{I_n} applied to the identity. Given an arbitrary domain $\mathbb{D}$ and stream arguments $s_0, \ldots, s_{n-1} \in \mathrm{Str}_\mathbb{D}$, define $g : I_n \to \mathbb{D}_\perp$, $(i,j) \mapsto s_i(j)$. By parametricity,

$$\begin{aligned} f_\mathbb{D}(s_0, \ldots, s_{n-1}) &= (f_\mathbb{D} \circ (\mathrm{map}\, g, \ldots, \mathrm{map}\, g))(j \mapsto (i,j))_{i \in \{0,\ldots,n-1\}} \\ &= ((\mathrm{map}\, g) \circ f_{I_n})(j \mapsto (i,j))_{i \in \{0,\ldots,n-1\}} \\ &= (\mathrm{map}\, g)(\overline{f}). \end{aligned}$$

This equation is the reason we call $\overline{f}$ the *indexing function* of f: for each $k \in \mathbb{N}$, the value of $f_\mathbb{D}(s_0, \ldots, s_{n-1})$ at stream position k is given by $s_i(j)$ if $\overline{f}(k) = (i,j)$, and $\perp$ if $\overline{f}(k) = \perp$, i.e.

$$f_\mathbb{D}(s_0, \ldots, s_{n-1}) = s_{\overline{f}(0)} :: s_{\overline{f}(1)} :: s_{\overline{f}(2)} :: \ldots$$

with the convention that $s_{(i,j)} := s_i(j)$ and $s_\perp = \perp$.

From this, it is clear that polymorphic stream functions and indexing functions of the same arity are in bijective correspondence. A polymorphic stream function is total if and only if its indexing function is total. We call it *computable* if its indexing function is a computable function. In the remainder of the article, we will identify I_1 with $\mathbb{N}$.

It is worth noting that indexing functions represent an inversion of the usual notions of input and output for stream functions. This contravariance is reflected in $\overline{f \circ g} = \overline{g} \circ \overline{f}$ for polymorphic $f_{\mathbb{D}}, g_{\mathbb{D}} : \mathrm{Str}_{\mathbb{D}} \to \mathrm{Str}_{\mathbb{D}}$, a fact heavily utilised in the rest of the article. Basic examples of polymorphic stream functions are the tail operation $\mathsf{tail}_{\mathbb{D}} : \mathrm{Str}_{\mathbb{D}} \to \mathrm{Str}_{\mathbb{D}}, s \mapsto i \mapsto s(i+1)$ with indexing function $\overline{\mathsf{tail}}(k) = k+1$ and the combined head-cons operation

$$\begin{aligned}(\mathrm{head}(\cdot) :: \cdot)_{\mathbb{D}} &: \mathrm{Str}_{\mathbb{D}} \times \mathrm{Str}_{\mathbb{D}} \to \mathrm{Str}_{\mathbb{D}},\\ (s,t) &\mapsto i \mapsto \begin{cases} s(0) & \text{if } i = 0,\\ t(i-1) & \text{else}\end{cases}\end{aligned}$$

with indexing function

$$(\overline{\mathrm{head}(\cdot) :: \cdot})(k) = \begin{cases} (0,0) & \text{if } k = 0,\\ (1,k-1) & \text{else.}\end{cases}$$

2.2 Stream Equation Systems

We now define a simple form of a recursive system of equations for specifying polymorphic stream functions. A *(stream equation) system* of size n is a family of *stream equations* $f_k(s_0, \ldots, s_{m_k-1}) = \sigma_k$ with $k \in \{0, \ldots, n-1\}$ where each σ_k is a *stream term* of the form

$\sigma ::=$	s_i	stream parameter with $i \in \{0, \ldots, m_k - 1\}$,
$\mid$	$\mathsf{tail}(\sigma)$	stream stripped of its first element,
$\mid$	$\mathrm{head}(\sigma) :: \sigma'$	first element of a stream prepended to stream (head-cons),
$\mid$	$f_j(\sigma_0, \ldots, \sigma_{m_j-1})$	recursive stream function call with $j \in \{0, \ldots, n-1\}$.

We explicitly state that there are no further restrictions such as guardedness on the form of the stream terms σ_k since we specifically deal with ill-defined equation using our (domain theoretic) partiality semantics. A system is called *unary* if all defined stream functions are unary, i.e. $m_k = 1$ for all k. By a canonical application of the Kleene fixed-point theorem [10], each stream equation system of size n gives rise to n corresponding polymorphic stream functions, the least fixpoint of the given system of equations with respect to the partial ordering on $\mathbb{D}_\perp$ generated by $\perp < d$ for $d \in \mathbb{D}$ when the tail and head-cons operations are interpreted according to the previous section. An equation in the system is called *productive* if the polymorphic stream function it defines is total. In what follows, we will usually use the same symbols to denote syntactic occurrences and their semantic counterparts as their meaning is always clear from the context.

We can also view such a system as an *executable specification*, giving rise to a notion of operational semantics allowing us to explicitly construct the least fixpoint alluded to above. For this, we formalise the computation of stream elements using a functional relation $\to$ on pairs $\sigma \; ! \; k$ of stream terms σ and indices $k \in \mathbb{N}$ by setting

$$\begin{aligned}\mathsf{tail}(\sigma) \; ! \; k &\to \sigma \; ! \; k+1,\\ \mathrm{head}(\sigma) :: \sigma' \; ! \; k &\to \begin{cases} \sigma \; ! \; 0 & \text{if } k = 0,\\ \sigma' \; ! \; k-1 & \text{else,}\end{cases}\\ f_j(\sigma_0, \ldots, \sigma_{n_j-1}) \; ! \; k &\to \rho[\sigma_i/s_i]_{i \in \{0,\ldots,n_j-1\}} \; ! \; k\end{aligned}$$

where $f_j(s_0, \ldots, s_{n_j-1}) = \rho$ is an equation in the system. This is effectively equivalent to introducing a rewriting system on constructs of the form $\mathrm{head}(\mathsf{tail}^k(\sigma))$ based on the rules

$$\begin{aligned}\mathrm{head}(\mathrm{head}(\sigma) :: \sigma') &\to \mathrm{head}(\sigma),\\ \mathsf{tail}(\mathrm{head}(\sigma) :: \sigma') &\to \sigma'\end{aligned}$$

with a deterministic outermost rewriting strategy. For a given domain $\mathbb{D}$, we can now define $f_{j,\mathbb{D}}$ as

$$f_{j,\mathbb{D}}(s_0,\dots,s_{m_j-1}) = \begin{cases} s_w(i) & \text{if } f_j(s_0,\dots,s_{m_j-1}) \ ! \ k \to^* s_w \ ! \ i, \\ \bot & \text{else} \end{cases}$$

for $j \in \{0,\dots,n-1\}$, verifying that this is indeed the smallest solution to the given specification and fulfills the parametricity property. From this, we can express the indexing function as

$$\overline{f_j}(k) = \begin{cases} (w,i) & \text{if } f_j(s_0,\dots,s_{m_j-1}) \ ! \ k \to^* s_w \ ! \ i, \\ \bot & \text{else.} \end{cases}$$

Retrospectively, this consolidates our definition of productivity with its usual connotation, namely that each finite prefix of a stream (or the result of a stream function called with productive arguments) be constructible through finite evaluation. It furthermore shows that in this restricted settings, what is usually denoted by productivity is equivalent to unique solvability for instantiations of the parameter $\mathbb{D}$ to sets containing more than one element, i.e. well-definedness. On a side note, this description of indexing functions makes explicit the obvious fact that the defined stream functions are always computable.

2.3 Examples

Using this syntax, we can specify the *interleaving* function zip_n for $n > 0$ as

$$\mathrm{zip}_n(s_0,\dots,s_{n-1}) = \mathrm{head}(s_0) :: \mathrm{zip}_n(s_1,\dots,s_{n-1},\mathrm{tail}(s_0)).$$

It is productive with indexing function $\overline{\mathrm{zip}_n}(k) = (k \bmod n, \lfloor k/n \rfloor)$. Let us prove this statement in detail by induction. In the base case, we have

$$\mathrm{zip}_n(s_0,\dots,s_{n-1}) \ ! \ 0 \to \mathrm{head}(s_0) :: \mathrm{zip}_n(s_1,\dots,s_{n-1},\mathrm{tail}(s_0)) \ ! \ 0 \to s_0 \ ! \ 0$$

proving $\overline{\mathrm{zip}_n}(0) = (0,0) = (0 \bmod n, \lfloor 0/n \rfloor)$. In the induction step, we have

$$\begin{aligned} \mathrm{zip}_n(s_0,\dots,s_{n-1}) \ ! \ k+1 &\to \mathrm{head}(s_0) :: \mathrm{zip}_n(s_1,\dots,s_{n-1},\mathrm{tail}(s_0)) \ ! \ k+1 \\ &\to \mathrm{zip}_n(s_1,\dots,s_{n-1},\mathrm{tail}(s_0)) \ ! \ k \\ &\to^* \begin{cases} \mathrm{tail}(s_0) \ ! \ \lfloor k/n \rfloor & \text{if } k \equiv -1 \mod n, \\ s_{(k \bmod n)+1} \ ! \ \lfloor k/n \rfloor & \text{else} \end{cases} \\ &\to^* s_{(k+1) \bmod n} \ ! \ \lfloor (k+1)/n \rfloor, \end{aligned}$$

proving $\overline{\mathrm{zip}_n}(k) = (k \bmod n, \lfloor k/n \rfloor)$ implies $\overline{\mathrm{zip}_n}(k) = ((k+1) \bmod n, \lfloor (k+1)/n \rfloor)$.

As a kind of inverse to interleaving, the *projection* function proj_n is defined as

$$\mathrm{proj}_n(s) = \mathrm{head}(s) :: \mathrm{proj}_n(\mathrm{tail}^n(s)).$$

It is easily shown to be productive with indexing function $\overline{\mathrm{proj}_n}(k) = n \cdot k$. For convenience, we also define shifted projections $\mathrm{proj}_{n,i}(s) := \mathrm{proj}_n(\mathrm{tail}^i(s))$ for $i < n$ with $\overline{\mathrm{proj}_{n,i}}(k) = n \cdot k + i$. Note that $\mathrm{proj}_{n,i,\mathbb{D}}(\mathrm{zip}_{n,\mathbb{D}}(s_0,\dots,s_{n-1})) = s_i$ as well as $s = \mathrm{zip}_{n,\mathbb{D}}\left((\mathrm{proj}_{n,i,\mathbb{D}}(s))_{i \in \{0,\dots,n-1\}}\right)$ for all domains $\mathbb{D}$ and $s, s_0, \dots, s_{n-1} \in \mathrm{Str}_{\mathbb{D}}$. We also have the *constant function* $\mathrm{const}(s) = \mathrm{head}(s) :: \mathrm{const}(s)$, repeating the first stream element of its argument, with $\overline{\mathrm{const}}(k) = 0$.

We are now ready to return to the example from the introduction. Recall that $\mathrm{hanoi}(s) = \mathrm{zip}_2(\mathrm{hanoi}(\mathrm{tail}(s)), \mathrm{const}(s))$. We will prove that $\overline{\mathrm{hanoi}}(2^v(2m+1)) = v$ by induction on v. In the base case, we have

$$\mathrm{hanoi}(s) \ ! \ 2k+1 \to \mathrm{zip}_2(\mathrm{hanoi}(\mathrm{tail}(s)), \mathrm{const}(s)) \ ! \ 2k+1 \to^* \mathrm{const}(s) \ ! \ k \to^* s \ ! \ 0,$$

proving $\overline{\mathrm{hanoi}}(2k+1) = 0$. In the induction step, we have

$$\begin{aligned}\mathrm{hanoi}(s) \ ! \ 2^{v+1}(2m+1) &\to \mathrm{zip}_2(\mathrm{hanoi}(\mathrm{tail}(s)), \mathrm{const}(s)) \ ! \ 2^{v+1}(2m+1)\\ &\to^* \mathrm{hanoi}(\mathrm{tail}(s)) \ ! \ 2^{v}(2m+1)\\ &\to^* \mathrm{tail}(s) \ ! \ v \to s \ ! \ v+1,\end{aligned}$$

proving $\overline{\mathrm{hanoi}}(2^v(2m+1)) = v$ implies $\overline{\mathrm{hanoi}}(2^{v+1}(2m+1)) = v+1$. Now, the interesting artifact is the first stream position,

$$\mathrm{hanoi}(s) \ ! \ 0 \to \mathrm{zip}_2(\mathrm{hanoi}(\mathrm{tail}(s)), \mathrm{const}(s)) \ ! \ 0 \to^+ \mathrm{hanoi}(\mathrm{tail}(s)) \ ! \ 0,$$

leading to an infinite loop

$$\mathrm{hanoi}(s) \ ! \ 0 \to^+ \mathrm{hanoi}(\mathrm{tail}(s)) \ ! \ 0 \to^+ \mathrm{hanoi}(\mathrm{tail}^2(s)) \ ! \ 0 \to^+ \dots,$$

showing that $\overline{\mathrm{hanoi}}(0) = \bot$.

After having established some intuition for computing indexing functions, let us prove a lemma which will be of much use further on.

▶ **Lemma 1.** *Given $h \geq 1$ and a stream equation of the form*

$$f(s) = \mathrm{head}(\mathrm{tail}^{a_0}(s)) :: \dots :: \mathrm{head}(\mathrm{tail}^{a_{h-1}}(s)) :: f(v(s)),$$

then $\overline{f}(k) = \overline{v}^{\lfloor k/h \rfloor}(a_{k \bmod h})$ for $k \in \mathbb{N}$.

Proof. Since this is our first technical result about indexing functions, we will be explicit in every detail. The proof is by induction on $k \in \mathbb{N}$. For $k < h$, we have

$$\begin{aligned}f(s) \ ! \ k &= \mathrm{head}(\mathrm{tail}^{a_0}(s)) :: \dots :: \mathrm{head}(\mathrm{tail}^{a_{h-1}}(s)) :: f(v(s)) \ ! \ k\\ &\to^k \mathrm{tail}^{a_k}(s) \ ! \ 0 \to^{a_k} s \ ! \ a_k,\end{aligned}$$

yielding $\overline{f}(k) = a_k = \overline{v}^0(a_k) = \overline{v}^{\lfloor k/h \rfloor}(a_{k \bmod h})$. For $k \geq h$, we have

$$f(s) \ ! \ k = \mathrm{head}(\mathrm{tail}^{a_0}(s)) :: \dots :: \mathrm{head}(\mathrm{tail}^{a_{h-1}}(s)) :: f(v(s)) \ ! \ k \to^h f(v(s)) \ ! \ k-h.$$

By induction hypothesis, it follows that

$$\begin{aligned}\overline{f}(k) = (\overline{f \circ v})(k-h) &= \overline{v}(\overline{f}(k-h))\\ &= \overline{v}(\overline{v}^{\lfloor (k-h)/h \rfloor}(a_{(k-h) \bmod h}))\\ &= \overline{v}(\overline{v}^{\lfloor k/h \rfloor - 1}(a_{k \bmod h})) = \overline{v}^{\lfloor k/h \rfloor}(a_{k \bmod h}),\end{aligned}$$

where we exploited contravariance of the indexing operation in the second step. ◀

3 Definability

Our first result consists of the insight that the above stream equations of simple form, incorporating only the stream constructor and destructors and recursion, already allow for the definition of every computable polymorphic stream function. In the following, we will only consider single argument functions since we can easily fuse multiple arguments into a single one using instances of zip and proj. The argument can be seen as a generalization of the idea expressed in the proof of Theorem 4.4 in an article by Endrullis et. al. [8] that recognising productivity of a form of system similar to ours (with a unit stream data type instead of abstract polymorphism) is of complexity Π^0_2. Since our key contribution concerns the even more general result of unary definability, the main purpose of the following exposition is to serve as a contrast to the next section.

Recall that for a single argument stream function, the indexing function has type $\mathbb{N} \to \mathbb{N}_\perp$. We will give our proof in the form of a reduction, transforming a counter machine [16] representing an arbitrary computable function $\phi : \mathbb{N} \to \mathbb{N}_\perp$ into a corresponding system with an equation having ϕ as indexing function.

▶ **Definition 2.** A *counter machine* is given by a tuple (L, I) where L is the length of the program and I is a list $I_0, \ldots, I_{L-1}$ of instructions $\mathrm{inc}(r)$ with $r \in \mathbb{N}$, denoting increment of register r, and $\mathrm{jzdec}(r, l)$ with $r \in \mathbb{N}$ and $l \in \{0, \ldots, L\}$, denoting a jump to instruction l if register r is zero and a decrement of r otherwise.

The semantics of such a machine is as follows: The *state* $(P, R) \in S := \{0, \ldots, L\} \times \mathbb{N}^{(\mathbb{N})}$ consists of the value P of its instruction pointer and the values R of its registers, where $\mathbb{N}^{(\mathbb{N})}$ denotes the set of functions $\mathbb{N} \to \mathbb{N}$ with finite support, i.e. with only finitely many values non-zero. Such a tuple is called *terminal* if $P = L$, denoting the machine has exited with output $R(1)$. For $R \in \mathbb{N}^{(\mathbb{N})}$ and $r, v \in \mathbb{N}$, the result of replacing the r-th entry of R with v will be denoted $R[r \leftarrow v]$. Representing execution, We define a functional *next* relation $\to$ on S on non-terminal elements by

$$(P, R) \to \begin{cases} (P+1, R[r \leftarrow R(r)+1]) & \text{if } I_P = \mathrm{inc}(r), \\ (P+1, R[r \leftarrow R(r)-1]) & \text{if } I_P = \mathrm{jzdec}(r, l), R(r) \neq 0, \\ (l, R) & \text{if } I_P = \mathrm{jzdec}(r, l), R(r) = 0. \end{cases}$$

The *result function* $\mathrm{result}_{(L,I)} : S \to \mathbb{N}_\perp$ is defined as

$$\mathrm{result}_{(L,I)}(t) = \begin{cases} R(1) & \text{for } t \to^* (L, R) \text{ terminal}, \\ \perp & \text{else.} \end{cases}$$

The *initial state* for a given input $i \in \mathbb{N}$ is given by $\mathrm{init}(i) := (0, (i, 0, \ldots))$. With this machinery, we can now define the associated computable function of the counter machine as $\phi_{(L,I)} = \mathrm{result}_{(L,I)} \circ \mathrm{init} : \mathbb{N} \to \mathbb{N}_\perp$.

▶ **Proposition 3.** *Given a computable function represented as a counter machine (L, I), there is a stream equation system defining a unary stream function with indexing function $\phi_{(L,I)}$.*

▶ **Definition 4.** Let $p_0 < p_1 < \ldots$ be all the primes. We encode a register state $R \in \mathbb{N}^{(\mathbb{N})}$ as a single positive integer using $\widehat{R} := \prod_{r \in \mathbb{N}, R(r) \neq 0} p_r^{R(r)}$.

Proof of Proposition 3. Our goal is to mutually define unary stream functions $f_0, \ldots, f_{L-1}$ such that for a sequence of machine states $(P, R) \to (P', R')$, we have $f_P \; ! \; \widehat{R} \to^+ f_{P'} \; ! \; \widehat{R'}$, effectively simulating the execution of the counter machine. It follows that whenever the counter machine terminates with $(P, R) \to^* (L, R')$ terminal, then $\overline{f_P} \; ! \; \widehat{R} \to^* \overline{f_L} \; ! \; \widehat{R'}$, and $\overline{f_P}(\widehat{R}) = \perp$ otherwise. Defining f_L to extract the value of register 1, i.e. the exponent of $p_1 = 3$, from the encoding we set

$$f_L(s) = \mathrm{zip}_3(f_L(\mathrm{tail}(s)), \mathrm{const}(s), \mathrm{const}(s)).$$

A straightforward induction on $R(1)$ shows that $\overline{f_L}(\widehat{R}) = R(1) = \mathrm{result}_{(L,I)}(L, R) \neq \perp$ for $R \in \mathbb{N}^{(\mathbb{N})}$. Together, this means $f_P(\widehat{R}) = \mathrm{result}_{(L,I)}(P, R)$ for any state $(P, R) \in S$.

For each instruction I_P with $P \in \{0, \ldots, L-1\}$, we mutually define a corresponding stream function f_P reproducing the action of I_P on the register encoding. If $I_P = \text{inc}(r)$, let

$$f_P(s) = \text{proj}_{p_r}(f_{P+1}(s))$$

and note that $f_P \;!\; \widehat{R} \to^+ f_{P+1} \;!\; p_r\widehat{R} = f_{P+1} \;!\; \widehat{R[r \leftarrow R(r)+1]}$ for $R \in \mathbb{N}^{(\mathbb{N})}$, simulating an increment. If $I_P = \text{jzdec}(r,l)$, let

$$f_P(s) = \text{zip}_{p_r}\left(\begin{cases} f_{P+1}(s) & \text{if i} = 0, \\ \text{proj}_{p_r,i}(f_l(s)) & \text{else} \end{cases}\right)_{i \in \{0,\ldots,p_r-1\}}.$$

Given $R \in \mathbb{N}^{(\mathbb{N})}$ with $R(r) \neq 0$, we know $R(r)$ is divisible by p_r and hence $f_P \;!\; \widehat{R} \to^+ f_{P+1} \;!\; \widehat{R}/p_r = f_{P+1} \;!\; \widehat{R[r \leftarrow R(r)-1]}$, simulating a decrement. For $R(r) = 0$, we have $f_P \;!\; \widehat{R} \to^+ f_l \;!\; p_r\lfloor\widehat{R}/p_r\rfloor + (\widehat{R} \bmod p_r) = f_l \;!\; \widehat{R}$, simulating a jump. Here, we exploited properties of the indexing functions of proj and zip noted earlier.

Finally, we need a stream function to produce the initial register encoding. This will be accomplished by $u(s) = \text{head}(\text{tail}(s)) :: u(\text{proj}_{p_0}(s))$. By Lemma 1, we have $\overline{u}(i) = \overline{\text{proj}_{p_0}}^i(1) = p_0^i = \widehat{(i, 0, \ldots)}$. Defining $q(s) = u(f_0(s))$, we have $\overline{q}(i) = \overline{f_0}(\widehat{(i, 0, \ldots)}) = \text{result}_{(L,I)}(\text{init}(i))$ for $i \in \mathbb{N}$, and thus $\overline{q} = \phi_{(L,I)}$. The equations for the stream functions $f_0, \ldots, f_L, u, q,$ const plus finitely many instances of zip and proj hence represent a stream equation system with the denotation of q having $\phi_{(L,I)}$ as indexing function. ◀

Note that Minsky [16] shows that counter machines with only two registers are enough to achieve Turing-completeness. However, when representing computable functions, this requires an extra level of encoding of input values and decoding of output values, which can also be achieved by defining suitable stream functions.

4 Unary Definability

The defining feature of the previous construction was its reliance on interleaving for implementing conditional execution, effectively dispatching different cases, identified by their residues of the register encoding modulo a prime, to arbitrarily different handlers. Noting that zip_p with p prime were the only non-unary stream functions in the construction, we are left to reflect on the computational consequences of only allowing unary stream functions to be defined. Note that allowing interleaving is synonymous to allowing non-unary stream functions since we can use interleaving to merge any number of stream arguments into a single one. In order to prove an even more general definability result in the unary setting, entirely different techniques need to be developed, separating conditional execution and unbounded looping into orthogonal concepts.

4.1 Collatz Functions and If-Programs

▶ **Definition 5.** A function $g : \mathbb{N} \to \mathbb{N}$ is called a *Collatz function* if there is $n > 0$ such that g is affine on each equivalence class modulo n, i.e. there are coefficients $a_i, b_i \in \mathbb{N}$ for $i = 0, \ldots, n-1$ such that $g(n \cdot q + i) = a_i \cdot q + b_i$ for $q \in \mathbb{N}$. In this case, n is called a *modulus* of g.

The naming stems from the famous conjecture first proposed by Collatz in 1937, asking whether the function collatz : $\mathbb{N} \to \mathbb{N}$, $n \mapsto n/2$ for n even, $n \mapsto 3n+1$ for n odd, will map each positive integer to 1 after finitely many applications. Despite its deceivingly simple form, it has been resisting all attempts of resolution [15]. In recent years, the equivalent of this conjecture for the above generalised notion of Collatz functions has been proved (algorithmically) undecidable [14].

▶ **Lemma 6.** *Given a Collatz function g, we can construct a non-mutually recursive unary system defining a stream function v such that $\overline{v} = g$.*

Before going into the details of the proof, note that, although it is quite clear that the above encoding of Collatz functions already enables an embedding of full computational power into unary stream equations, what is not at all obvious is whether we can actually define every computable unary stream function through a purely unary system.

Proof. Let modulus $n > 0$ and coefficients $a_i, b_i \in \mathbb{N}$ be as in the above definition. Define a stream function

$$\mathrm{add}(s) = \mathrm{head}(\mathrm{tail}^{n \cdot a_0 + 0}(s)) :: \ldots :: \mathrm{head}(\mathrm{tail}^{n \cdot a_{n-1} + (n-1)}(s)) :: \mathrm{add}(\mathrm{tail}^n(s)).$$

Lemma 1 shows that $\overline{\mathrm{add}}(k) = \lfloor \frac{k}{n} \rfloor \cdot n + (n \cdot a_{k \bmod n} + (k \bmod n)) = k + n \cdot a_{k \bmod n}$ for $k \in \mathbb{N}$. The role of this function is to act as a crude replacement conditional for the unavailable zip, adding different constants depending on the equivalence class of the stream index modulo n.

Next, define a stream function

$$u(s) = \mathrm{head}(\mathrm{tail}^{n \cdot b_0 + 0}(s)) :: \ldots :: \mathrm{head}(\mathrm{tail}^{n \cdot b_{n-1} + (n-1)}(s)) :: u(\mathrm{add}(s)).$$

Using Lemma 1, we derive $\overline{u}(n \cdot q + i) = \overline{\mathrm{add}}^q(n \cdot b_i + i) = (n \cdot b_i + i) + q \cdot (n \cdot a_i) = n \cdot g(k) + i$ for $q \in \mathbb{N}$. This function is an approximation to g, the only difference being that the output indices come pre-multiplied by n.

We fix this by defining a stream function

$$\mathrm{div}(s) = \underbrace{\mathrm{head}(s) :: \ldots :: \mathrm{head}(s)}_{n \text{ times}} :: \mathrm{div}(\mathrm{tail}(s)).$$

A trivial application of Lemma 1 verifies that $\overline{\mathrm{div}}(k) = \lfloor \frac{k}{n} \rfloor$ for $k \in \mathbb{N}$. Finally defining $v(s) = u(\mathrm{div}(s))$ yields the Collatz function semantics we want in that $\overline{v} = \overline{\mathrm{div}} \circ \overline{u} = g$. ◀

For instance, the original Collatz function would be encoded as collatz $= \overline{v}$ as follows:

$$\begin{aligned}
\mathrm{add}(s) &= \mathrm{head}(\mathrm{tail}^2(s)) :: \mathrm{head}(\mathrm{tail}^{13}(s)) :: \mathrm{add}(\mathrm{tail}^2(s)),\\
u(s) &= \mathrm{head}(s) :: \mathrm{head}(\mathrm{tail}^9(s)) :: u(\mathrm{add}(s)),\\
\mathrm{div}(s) &= \mathrm{head}(s) :: \mathrm{head}(s) :: \mathrm{div}(\mathrm{tail}(s)),\\
v(s) &= u(\mathrm{div}(s)).
\end{aligned}$$

For further illustration, let us evaluate stream position 3 of $v(s)$:

$$\begin{aligned}
v(s)\ !\ 3 &\to^* u(\mathrm{div}(s))\ !\ 3 \to^* u(\mathrm{add}(\mathrm{div}(s)))\ !\ 1 \to^* \mathrm{add}(\mathrm{div}(s))\ !\ 9\\
&\to^* \mathrm{add}(\mathrm{tail}^2(\mathrm{div}(s)))\ !\ 7 \to^* \ldots \to^* \mathrm{add}(\mathrm{tail}^8(\mathrm{div}(s)))\ !\ 1\\
&\to^* \mathrm{div}(s)\ !\ 21 \to^* \mathrm{div}(\mathrm{tail}(s))\ !\ 19 \to^* \ldots \to^* \mathrm{div}(\mathrm{tail}^{10}(s))\ !\ 1\\
&\to^* s\ !\ 10.
\end{aligned}$$

This is consistent with collatz$(3) = 3 \cdot 3 + 1 = 10$.

Note that this encoding is fundamentally different from the encoding of Collatz functions or Fractran program iterations used by Endrullis et al. [8], the essential difference being the unavailability of zip in the unary setting. The dispatch mechanism of interleaving, enabling differing treatment of stream positions based on the residue of their indices, makes the implementation of Collatz-like constructs rather straightforward. Since we are lacking even such basic conditional control flow mechanisms, we have to resort to highly indirect constructions such as the above.

The role of Collatz functions in our setting is to serve as an intermediate between indexing functions and the known world of computability. To make the latter link clearer, we will show how Collatz functions relate semantically to different register machine models under the prime factorization register encoding introduced in the previous section.

▶ **Definition 7.** The inductive set of IF-*programs* is generated by concatenation $\mathbf{A}_0 \dots \mathbf{A}_{n-1}$, increments $\operatorname{inc}(r)$, decrements $\operatorname{dec}(r)$, and conditional clauses $\operatorname{ifz}(r, \mathbf{A}, \mathbf{B})$, where $n \in \mathbb{N}$, $r \in \mathbb{N}$ designates a register, and $\mathbf{A}_0, \dots, \mathbf{A}_{n-1}, \mathbf{A}, \mathbf{B}$ are IF-programs.

Although it is quite clear intuitively what the semantic effects of running an IF-program $\mathbf{A}$ on some register state $R \in \mathbb{N}^{(N)}$ are, we will formally introduce an associated semantics function $\chi_{\mathbf{A}} : \mathbb{N}^{(\mathbb{N})} \to \mathbb{N}^{(\mathbb{N})}$ defined structurally as follows:

$$\begin{aligned}
\chi_{\mathbf{A}_0 \dots \mathbf{A}_{n-1}} &= \chi_{\mathbf{A}_{n-1}} \circ \dots \circ \chi_{\mathbf{A}_0}, \\
\chi_{\operatorname{inc}(r)}(R) &= R[r \leftarrow R(r) + 1], \\
\chi_{\operatorname{dec}(r)}(R) &= R[r \leftarrow \max(R(r) - 1, 0)], \\
\chi_{\operatorname{ifz}(r,\mathbf{A},\mathbf{B})}(R) &= \begin{cases} \chi_{\mathbf{A}}(R) & \text{if } R(r) = 0, \\ \chi_{\mathbf{B}}(R) & \text{else.} \end{cases}
\end{aligned}$$

Note that a decrement on a zero-valued register is ignored.

We will reuse the prime factorization register encoding $\widehat{\cdot} : \mathbb{N}^{(\mathbb{N})} \to \mathbb{N} \setminus \{0\}, \widehat{R} \mapsto \widehat{R} = \prod_{r \in \mathbb{N}, R(r) \neq 0} p_r^{R(r)}$ from the previous section. Translated to this setting, the semantics function of an IF-program $\mathbf{A}$ takes the form $\widehat{\chi_{\mathbf{A}}} := \widehat{\cdot} \circ \chi_{\mathbf{A}} \circ \widehat{\cdot}^{-1}$. Although this is an endofunction on the positive integers, to make the following treatment more uniform, we will extend it to the natural numbers by setting $\widehat{\chi_{\mathbf{A}}}(0) := 0$.

▶ **Lemma 8.** *Given an if-program* $\mathbf{A}$*, its semantics* $\widehat{\chi_{\mathbf{A}}} : \mathbb{N} \to \mathbb{N}$ *on the register encoding is a Collatz function.*

Proof. By induction on the structure of $\mathbf{A}$, noting that:

- The concatenation of finitely many Collatz functions of moduli $m_0 \cdot \dots \cdot m_{n-1}$ is a Collatz function of modulus $m_0 \cdot \dots \cdot m_{n-1}$.
- Given a register state $R \in \mathbb{N}^{(\mathbb{N})}$, increment of register r corresponds to multiplication of $\widehat{R}$ with p_r, a Collatz function of modulus 1.
- Decrement of register r corresponds to division of $\widehat{R}$ by p_r if the former is divisible by p_r, and no change otherwise. This is a Collatz function of modulus p_r.
- Let $\widehat{\chi_{\mathbf{A}}}$ and $\widehat{\chi_{\mathbf{B}}}$ be Collatz functions of moduli $m_{\mathbf{A}}$ and $m_{\mathbf{B}}$, respectively. A conditional clause $\operatorname{ifz}(r, \mathbf{A}, \mathbf{B})$ corresponds first to case distinction depending on whether $\widehat{R}$ is divisible by p_r and subsequent application of either $\widehat{\chi_{\mathbf{A}}}$ or $\widehat{\chi_{\mathbf{B}}}$. This is a Collatz function of modulus the least common multiple of $p_r, m_{\mathbf{A}}, m_{\mathbf{B}}$.

◀

We note that the Collatz function $\widehat{\chi}_{\mathbf{A}}$ in the previous lemma is special in that it is linear on each of its equivalence classes in the strict sense, i.e. with vanishing ordinate, corresponding to single multiplication with a fraction. Even though we do not make use of this fact in our developments, it shows the connection between IF-programs and the iteration steps of the FRACTRAN-programs of Conway [4], which are of equivalent expressive power.

4.2 Iteration-Programs and Their Encoding

Unsurprisingly, the expressive power of IF-programs by themselves is quite limited. To achieve computational completeness, we need an unbounded looping construct. The following definition intends to provide a minimal such model, enabling us to concentrate on the essential details of the conversion from Turing-complete programs to stream equation systems.

▶ **Definition 9.** An ITERATION-*program* $\mathbf{P}$ is a tuple $(\mathbf{Body_P}, \mathsf{input_P}, \mathsf{output_P}, \mathsf{loop_P})$ consisting of an IF-program $\mathbf{Body_P}$ called the *body* of $\mathbf{P}$ and designated and mutually distinct *input*, *output* and *loop* registers $\mathsf{input_P}, \mathsf{output_P}, \mathsf{loop_P} \in \mathbb{N}$.

The semantics of such a program is a computable function $\phi_{\mathbf{P}} : \mathbb{N} \to \mathbb{N}_{\perp}$ defined as follows: given an input $i \in \mathbb{N}$, the register state $R_0 \in \mathbb{N}^{(\mathbb{N})}$ is initialised with $R_0(\mathsf{input_P}) := i$, $R_0(\mathsf{loop_P} := 1$, and $R_0(r) := 0$ for $r \neq \mathsf{input_P}, \mathsf{loop_P}$. We iteratively execute the body of $\mathbf{P}$, yielding $R_{n+1} := \chi_{\mathbf{Body_P}}(R_n)$ for $n \in \mathbb{N}$. If there is n minimal such that $R_n(\mathsf{loop_P}) = 0$, then $\mathbf{P}$ is called *terminating* with *iteration count* $\mathrm{count}\, P(i) := n$ and output $\phi_{\mathbf{P}}(i) := R_n(\mathsf{output_P})$ for input i. Otherwise, $\mathrm{count}_{\mathbf{P}}(i) := \perp$ and $\phi_{\mathbf{P}}(i) := \perp$.

Intuitively, an ITERATION-program is just a WHILE-program [17] with a single top-level loop, a well-studied concept in theoretical computer science bearing resemblance to the normal form theorem for μ-recursive functions [13], [21] except that we do not even allow primitive recursion inside the loop.

▶ **Theorem 10.** *Given a computable function $\phi : \mathbb{N} \to \mathbb{N}_{\perp}$, there is an* ITERATION-*program $\mathbf{P}$ with semantics $\phi_{\mathbf{P}} = \phi$.*

Proof. This is a folklore theorem [11], see Böhm and Jacopini [2] and Perkowska [17] for more details. ◀

The reason behind our choice for this computationally complete machine model is that we already have the machinery to simulate a single execution of the body of such a machine via Collatz functions as indexing functions of stream equations using our prime factorization exponential encoding on the register state. In fact, another option would have been the FRACTRAN-programs of Conway [4], but we are in need of a more conceptual representation in light of what lies ahead of us.

We will now investigate how to translate a top-level unbounded looping construct into the recursive stream equation setting.

▶ **Lemma 11.** *Let $h \geq 1$ be given with a stream equation*

$$f(s) = \mathrm{head}(\mathrm{tail}^{a_0}(s)) :: \ldots :: \mathrm{head}(\mathrm{tail}^{a_{h-1}}(s)) :: \mathrm{tail}^h(u(f(v(s)))).$$

Fix $k \in \mathbb{N}$ and choose $c(k) \in \mathbb{N}$ minimal such that $d(k) := \overline{u}^{c(k)}(k) \in \{\perp, 0, \ldots, h-1\}$. If such a $c(k)$ exists and $d(k) \neq \perp$, then $\overline{f}(k) = \overline{v}^{c(k)}(a_{d(k)})$, otherwise $\overline{f}(k) = \perp$.

Proof. The proof is by induction on $c(k)$ if existent. At the base, $c(k) = 0$ is equivalent to $k < h$. In this case, $f(s)\ !\ k \to^k \mathrm{tail}^{a_k}(s)\ !\ 0 \to^{a_k} s\ !\ a_k$, i.e. $\overline{f}(k) = a_k = \overline{v}^{c(k)}(a_{d(k)})$.

Now assume $k \geq h$. Note that

$$f(s) \ ! \ k \to^h \mathrm{tail}^h(u(f(v(s)))) \ ! \ k - h \to^h u(f(v(s))) \ ! \ k.$$

If $\overline{u}(k) = \bot$, then $c(k) = 1$, $d(k) = \bot$, and $\overline{f}(k) = \bot$. In the remainder, we will assume $\overline{u}(k) \neq \bot$. Then, $f(s) \ ! \ k \to^+ f(v(s)) \ ! \ \overline{u}(k)$, and $\overline{f}(k) = \overline{v}(\overline{f}(\overline{u}(k)))$.

If $c(k)$ is defined, then $c(k) = c(\overline{u}(k)) + 1$ and we can apply the induction hypothesis: if $d(k) = d(\overline{u}(k)) \neq \bot$, then $\overline{v}(\overline{f}(\overline{u}(k))) = \overline{v}(\overline{v}^{c(\overline{u}(k))} a_{d(k)}) = \overline{v}^{c(k)}(a_{d(k)})$, otherwise $\overline{v}(\overline{f}(\overline{u}(k))) = \overline{v}(\bot) = \bot$.

If $c(k)$ is undefined, then so is $c(\overline{u}(k))$, and with a second induction we can construct an infinite sequence $f(s) \ ! \ k \to^+ f(v(s)) \ ! \ \overline{u}(k) \to^+ f(v^2(s)) \ ! \ \overline{u}^2(k) \to^+ \ldots$ showing non-termination and $\overline{f}(k) = \bot$. ◀

The inquiring reader will notice that this lemma can be seen as a generalization of Lemma 1 with u defined in a particular way, namely

$$u(s) = \underbrace{\mathrm{head}(s) :: \ldots :: \mathrm{head}(s)}_{h \text{ times}} :: s.$$

Using Lemmata 8 and 6, we can translate the encoded iteration step function $\widehat{\chi}_{\mathbf{Body_P}} : \mathbb{N} \to \mathbb{N}$ of an ITERATION-program $\mathbf{P}$ to an indexing function of a stream equation for some u. We would like to use this stream function u as it appears in Lemma 11 in a way such that the minimal choice of $c(k)$ corresponds to the iteration count of $\mathbf{P}$. Unfortunately, the equivalent of the stopping condition in the lemma, that the index be smaller than some constant h, corresponds to $\widehat{R} < h$ for the register state $R \in \mathbb{N}^{(\mathbb{N})}$, a statement which does not have a natural meaning for the registers of R individually, forestalling us from expressing the condition $p_{\mathsf{loop_P}} \mid \widehat{R}$ corresponding to the termination condition $R(\mathsf{loop_P}) = 0$. A second problem comes from our desire to somehow extract the value of $R(\mathsf{output_P})$ after termination. But since at this point of time $\widehat{R}$ is limited to a finite set of values, there is no direct way of realising this.

What we can do is extract the iteration count for particularly nicely behaving programs.

▶ **Lemma 12.** *Given an* ITERATION*-program* $\mathbf{Q}$ *such that whenever* $\mathbf{Q}$ *terminates, all its registers are zero-valued, i.e.* $\chi_{\mathbf{Body_Q}}^{\mathrm{count}_{\mathbf{Q}}(i)} = (0, 0, \ldots)$ *for terminating input* $i \in \mathbb{N}$*, there is a non-mutually recursive unary system defining a stream function* w *such that* $\overline{w} = \mathrm{count}_{\mathbf{Q}}$*.*

Proof. In anticipation of applying Lemma 11, we extend this system with a new equation

$$q(s) = \mathrm{head}(s) :: \mathrm{head}(s) :: \mathrm{tail}^2(v(q(\mathrm{tail}(s)))).$$

Given an input $i \in \mathbb{N}$ and corresponding initial register state $R \in \mathbb{N}^{(\mathbb{N})}$, the termination condition in Lemma 11 can equivalently be expressed as follows:

$$\overline{v}^{c(\widehat{R})}(\widehat{R}) < 2 \iff \widehat{\chi}_{\mathbf{Body_Q}}^{c(\widehat{R})}(\widehat{R}) = 1 \iff \chi_{\mathbf{Body_Q}}^{c(\widehat{R})}(R) = (0, 0, \ldots).$$

Now, by our assumption on the behaviour of $\mathbf{Q}$, the first point in time all registers are zero equals the first point in time the loop register attains zero. But by our definition of the iteration count, this just means that $c(\widehat{R}) = \mathrm{count}_{\mathbf{Q}}(i)$, and Lemma 11 shows that $\overline{q}(\widehat{R}) = \overline{\mathrm{tail}}^{c(\widehat{R})}(0) = c(\widehat{R}) = \mathrm{count}_{\mathbf{Q}}(i)$.

All that remains is to produce the initial register state $R_{(i)}$ with only $R_{(i)}(\mathsf{input_Q}) = i$ and $R_{(i)}(\mathsf{loop_Q}) = 1$ non-zero. For this, we define

$$r(s) = \mathrm{head}(\mathrm{tail}^{p_{\mathsf{loop_Q}}}(s)) :: r(\mathrm{proj}_{p_{\mathsf{input_Q}}}(s))$$

and utilise Lemma 1 to prove that $\overline{r}(i) = \overline{\mathrm{proj}_{p_{\mathsf{input}_{\mathbf{Q}}}}}^{i}(p_{\mathsf{loop}_{\mathbf{Q}}}) = p^{i}_{\mathsf{input}_{\mathbf{Q}}} \cdot p_{\mathsf{loop}_{\mathbf{Q}}} = \widehat{R_{(i)}}$. Defining $w(s) = r(q(s))$, we verify that $\overline{w}(i) = \overline{q}(\overline{r}(i)) = \overline{q}(\widehat{R_{(i)}}) = \mathrm{count}_{\mathbf{Q}}(R_{(i)})$. ◀

Unfortunately, the set of possible iteration count functions constitutes only a small part of the set of all computable functions. Intuitively, this is because even very small values can be the result of prohibitively expensive operations. However, this range can still be seen as containing Turing-complete fragments under certain encodings. This is what we exploit in the next step by shifting the role of the output register to the iteration count under a particular such encoding. The trick is to have each possible output value correspond to infinitely many iteration counts in a controlled way such that after having computed the result, by being self-aware of the current iteration count, we can consciously terminate the loop at one of these infinitely many counts, no matter how long the computation took.

▶ **Lemma 13.** *Given an* ITERATION*-program* $\mathbf{P}$*, there is an* ITERATION*-program* $\mathbf{Q}$ *such that for every input i natural,* $\mathbf{Q}$ *terminates if and only if* $\mathbf{P}$ *terminates, and furthermore if* $\mathbf{P}$ *terminates with output $o \in \mathbb{N}$, then* $\mathbf{Q}$ *terminates after exactly* $(3m+1) \cdot 3^{o+1}$ *iterations with all registers zero-valued where $m \in \mathbb{N}$ depends on i.*

Proof. Let $r_0, \ldots, r_{k-1} \in \mathbb{N}$ denote all the registers occurring in $\mathbf{Body_P}$ except for $\mathsf{output}_{\mathbf{P}}$ and $\mathsf{loop}_{\mathbf{P}}$ (but including $\mathsf{input}_{\mathbf{P}}$). We choose $\mathsf{loop}_{\mathbf{Q}}$ as a fresh natural number distinct from all previously mentioned registers. Both programs will have the same input register, i.e. $\mathsf{input}_{\mathbf{Q}} := \mathsf{input}_{\mathbf{P}}$. The output register of $\mathbf{Q}$ is irrelevant since we aim to have all registers reset at termination.

The body of $\mathbf{Q}$ is listed in Exhibit A. Note that the body of $\mathbf{P}$ is textually inserted at line 14. Register names main-phase, run-time, mod-three, swap-phase, copy also designate fresh natural numbers. To enhance readability, we used some lyrical freedom with the syntax: for example, **if** $R(\mathsf{output}_{\mathbf{P}}) \neq 0$ **then A else B end if** translates to $\mathrm{ifz}(\mathsf{output}_{\mathbf{P}}, \mathbf{B}, \mathbf{A})$. Since the program is somewhat complex, we will describe its function in great detail.

Execution of $\mathbf{Q}$, i.e. iterated execution of $\mathbf{Body_Q}$ until decrement of $\mathsf{loop}_{\mathbf{Q}}$, is split into two main phases, as signalled by the flag register main-phase. The first phase, when main-phase has value 0 (lines 2–22), is dedicated to simulating the original program $\mathbf{P}$ while keeping track of the total iteration count in a dedicated register run-time. At the end of this phase, after $\mathbf{P}$ has exited with result $o \in \mathbb{N}$ in register $\mathsf{output}_{\mathbf{P}}$, we want the total iteration count to equal $3m+1$ for some arbitrary $m \in \mathbb{N}$. The second phase, when main-phase has value 1 (lines 23–44) is dedicated to tripling the total iteration count o times, plus an additional tripling to reset run-time, so that the total iteration count becomes $(3m+1) \cdot 3^{o+1}$.

In detail, the first phase (lines 2–22) contains three separate components:

- Lines 3–4 are executed only at the beginning of the first iteration and initialise the loop register of $\mathbf{P}$ (note that the input register of $\mathbf{P}$ does not need to be initialised as $\mathsf{input}_{\mathbf{P}} = \mathsf{input}_{\mathbf{Q}}$) and mod-three (see below).
- Lines 6-12 keep track not only of the current iteration count by incrementing run-time once per iteration, but also of how many iterations modulo 3 we are afar from meeting the $(3m+1)$-condition in a dedicated register mod-three.
- Lines 13–22 execute the body of $\mathbf{P}$ once per iteration until termination is signalled by $\mathsf{loop}_{\mathbf{P}}$ being set to zero (lines 13–14). In subsequent iterations, the registers used in $\mathbf{P}$ are incrementally reset (lines 15–19). Finally, we wait up to two iterations for the iteration count (including the current iteration) to have the proper remainder modulo 3 (line 20), and proceed to the second phase (lines 21).

Exhibit A The body of **Q** from Lemma 13

```
 1. if $R(\mathsf{main\text{-}phase}) = 0$ then
 2.     if $R(\mathsf{run\text{-}time}) = 0$ then
 3.         inc($\mathsf{loop_P}$)
 4.         inc($\mathsf{mod\text{-}three}$)
 5.     end if
 6.     inc($\mathsf{run\text{-}time}$)
 7.     if $R(\mathsf{mod\text{-}three}) = 0$ then
 8.         inc($\mathsf{mod\text{-}three}$)
 9.         inc($\mathsf{mod\text{-}three}$)
10.         inc($\mathsf{mod\text{-}three}$)
11.     end if
12.     dec($\mathsf{mod\text{-}three}$)
13.     if $R(\mathsf{loop_P}) \neq 0$ then
14.         Body$_\mathbf{P}$
15.     else if $R(r_0) \neq 0$ then
16.         dec($r_0$)
17.     [...]
18.     else if $R(r_{n-1}) \neq 0$ then
19.         dec($r_{n-1}$)
20.     else if $R(\mathsf{mod\text{-}three}) = 0$ then
21.         inc($\mathsf{main\text{-}phase}$)
22.     end if
23. else if $R(\mathsf{swap\text{-}phase}) = 0$ then
24.     dec($\mathsf{run\text{-}time}$)
25.     inc($\mathsf{copy}$)
26.     if $R(\mathsf{run\text{-}time}) = 0$ then
27.         inc($\mathsf{swap\text{-}phase}$)
28.     end if
29. else
30.     dec($\mathsf{copy}$)
31.     if $R(\mathsf{output_P}) \neq 0$ then
32.         inc($\mathsf{run\text{-}time}$)
33.         inc($\mathsf{run\text{-}time}$)
34.         inc($\mathsf{run\text{-}time}$)
35.         if $R(\mathsf{copy}) = 0$ then
36.             dec($\mathsf{swap\text{-}phase}$)
37.             dec($\mathsf{output_P}$)
38.         end if
39.     else if $R(\mathsf{copy}) = 0$ then
40.         dec($\mathsf{swap\text{-}phase}$)
41.         dec($\mathsf{main\text{-}phase}$)
42.         dec($\mathsf{loop_Q}$)
43.     end if
44. end if
```

After the final iteration of this phase, the iteration count is $3m + 1$ for some $m \in \mathbb{N}$ and the only possibly non-zero registers are main-phase and swap-phase of value 1 and $\mathsf{output_P}$ of value o. We duly note that if **P** does not terminate, then neither does **Q**.

In similar detail, the second phase (lines 23–44) contains two alternately executed subphases (lines 24–28 and 30–43) responsible for shifting the iteration count back and forth between the registers run-time and copy. The current subphase is indicated by the flag register swap-phase:

- The first subphase in lines 24–28 started with register values $[\mathsf{swap\text{-}phase} : 1, \mathsf{run\text{-}time} : x \geq 1, \mathsf{copy} : 0]$ will end, after x iterations, with values $[\mathsf{swap\text{-}phase} : 0, \mathsf{run\text{-}time} : 0, \mathsf{copy} : x]$.
- The second subphase in lines 30–43 started with register values $[\mathsf{swap\text{-}phase} : 0, \mathsf{run\text{-}time} : 0, \mathsf{copy} : x \geq 1]$ will end, after x iterations, depending on the value of register $\mathsf{output_P}$,
  - if non-zero, with $[\mathsf{swap\text{-}phase} : 1, \mathsf{run\text{-}time} : 3x, \mathsf{copy} : 0]$ and $\mathsf{output_P}$ decremented,
  - if zero, with all registers zero and $\mathsf{loop_Q}$ decremented to zero in the last iteration.

Taken together, we deduce that starting (the first subphase) with $[\mathsf{swap\text{-}phase} : 1, \mathsf{run\text{-}time} : x \geq 1, \mathsf{copy} : 0]$ and $\mathsf{output_P}$ non-zero, after $2x$ iterations, the effective changes will be tripling of run-time and decrement of $\mathsf{output_P}$. In particular, if x and hence run-time denoted the iteration count before these iterations, run-time will again denote the iteration count after these iterations. After following this reasoning o times, the iteration count and value of run-time will be $(3m + 1) \cdot 3^o$ while $\mathsf{output_P}$ attains zero. One last instance of each phase, costing $2 \cdot (3m + 1) \cdot 3^o$ iterations, yield a total iteration count of $(3m + 1) \cdot 3^{o+1}$ with all registers having been cleared. ◀

4.3 Proof of the Main Result

▶ **Theorem 14.** *A unary polymorphic stream function is definable by a non-mutually recursive unary system if and only if its indexing function is computable.*

Proof. We need only consider the reverse implication. Let a computable function $\phi : \mathbb{N} \to \mathbb{N}_\perp$ be represented as an ITERATION-program $\mathbf{P}$, i.e. $\phi = \phi_P$, and let $\mathbf{Q}$ be the modified ITERATION-program as defined in Lemma 13. By Lemma 12, there is a non-mutually recursive unary system defining a stream function w such that $\overline{w}(i) = \mathrm{count}_{\mathbf{Q}}(i) = (3m+1) \cdot 3^{\phi_{\mathbf{P}}(i)+1}$ for all $i \in \mathbb{N}$ and some $m \in \mathbb{N}$ depending on i. As stated at the beginning, the latter expression is taken to mean $\perp$ if $\phi_{\mathbf{P}}(i) = \perp$.

Our strategy for extracting the final output value $\phi_{\mathbf{P}}(i)$ from this expression is by iterating a second program, adding a tail for each time the stream index is divisible by 3. In particular, from Lemma 6, it is clear how to give a non-mutually recursive unary system defining u such that

$$\overline{u}(k) = \begin{cases} k/3 & \text{if } 3 \mid k, \\ 0 & \text{else} \end{cases}$$

since this is a Collatz function. But note that we can alternatively directly define

$$\begin{aligned} u(s) = {} & \mathrm{head}(s) :: \mathrm{head}(s) :: \mathrm{head}(s) :: \\ & \quad \mathrm{head}(\mathrm{tail}(s)) :: \mathrm{head}(s) :: \mathrm{head}(s) :: \mathrm{tail}^3(u(\mathrm{head}(s) :: \mathrm{tail}^2(s))) \end{aligned}$$

using only a single equation. For either choice, we define $v(s) = \mathrm{head}(s) :: \mathrm{tail}(u(v(\mathrm{tail}(s))))$. A second application of Lemma 11 shows that $\overline{v}((3m+1) \cdot 3^{i+1}) = i + 1$ for $i, m \in \mathbb{N}$. This function is of almost as critical importance as w as it repeats each natural number output infinitely many times in a controlled way and reverses the iteration count result encoding of program $\mathbf{Q}$.

We now have all the parts necessary for concluding our venture. Defining $f(s) = w(v(\mathrm{head}(s) :: s))$, we see that

$$\begin{aligned} \overline{f}(i) &= \max(\overline{v}(\overline{w}(i)) - 1, 0) \\ &= \max(\overline{v}((3m+1) \cdot 3^{\phi_{\mathbf{P}}(i)+1}) - 1, 0) \\ &= \max((\phi_{\mathbf{P}}(i) + 1) - 1, 0) = \phi_{\mathbf{P}}(i) \end{aligned}$$

with our usual convention regarding $\perp$, proving $\overline{f} = \phi_{\mathbf{P}}$. ◀

Further Work

A thorough inquisition of the proofs in the previous section shows that, in total, ten equations were defined for proving the main result, two of which can be inlined. It is natural to ask: what is the minimum number of unary equations required to define an arbitrary computable indexing function? The answer is *four* (and still being free of mutual recursion), but space constrains prevent us from presenting the rather involved proof.

Furthermore, we can show that recognising productivity is undecidable with complexity Π^0_2 even for unary systems with only *two* non-mutually recursive equations. Interestingly, we can prove that productivity is decidable for a *singleton* unary system. Again, the proof of this positive result is quite complex and in need of more space to be presented.

Altogether, this amounts to an exhaustive classification of definability and complexity of recognising productivity based on unary system size. However, it would be euphemistic to say that the proofs involved are not very abstract, making them even more unsuitable for exposition in a conference article.

Acknowledgements

The authors would like to thank Venanzio Capretta for introducing us to the definability problem for polymorphic stream equations and spending much time with proof checking and helpful discussions, Nicolai Kraus for proofreading, the Functional Programming Laboratory at the University of Nottingham for the encouraging research environment, and the anonymous referees for providing helpful suggestions regarding the exposition of our results.

References

1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theor. Comp. Sci.*, 342:3–27, 2005.
2 Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.
3 Wilfried Buchholz. A term calculus for (co-)recursive definitions on streamlike data structures. *Ann. Pure Appl. Logic*, 136(1-2):75–90, 2005.
4 John H. Conway. Fractran: A simple universal programming language for arithmetic. In T. M. Cover and B. Gopinath, editors, *Open Problems in Communication and Computation*, chapter 2, pages 4–26. Springer, 1987.
5 Edgar W. Dijkstra. On the producitivity of recursive definitions. EWD749, 1980.
6 Jörg Endrullis, Herman Geuvers, Jacob G. Simonses, and Hans Zantema. Levels of undecidability in rewriting. *Inf. Comput.*, 209(2):227–245, 2011.
7 Jörg Endrullis, Clemens Grabmayer, and Dimitri Hendriks. Data-oblivious stream productivity. In *Proc. 15th Int. Conf. on LPAR*, LPAR '08, pages 79–96. Springer, 2008.
8 Jörg Endrullis, Clemens Grabmayer, and Dimitri Hendriks. Complexity of Fractran and productivity. In *CADE*, pages 371–387, 2009.
9 Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. In *Proc. FCT 2007*, volume 4639 of *LNCS*, pages 274–287. Springer, 2007.
10 Gerhard Gierz, Karl Heinrich Hofmann, Klaus Keimel, Jimmie D. Lawson, Michael Mislove, and Dana S. Scott. *Continuous Lattices and Domains*, volume 93 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2003.
11 David Harel. On folk theorems. *SIGACT News*, 12:68–80, 1980.
12 Andreas M. Hinz. The Tower of Hanoi. *Enseign. Math.*, 35(2):289–321, 1989.
13 Stephen C. Kleene. Recursive predicates and quantifiers. *Trans. AMS*, 53(1):41–73, 1943.
14 Stuart A. Kurtz and Janos Simon. The undecidability of the generalized Collatz problem. In *TAMS*, volume 4484 of *LNCS*, pages 542–553. Springer, 2007.
15 Jeffery C. Lagarias. *The Ultimate Challenge: The* $3x+1$ *Problem*. AMS, 2010.
16 Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.
17 Eleonora Perkowska. Theorem on the normal form of a program. *Bull. Acad. Pol. Sci., Ser. Sci. Math. Astr. Phys.*, 22(4):439–442, 1974.
18 Grigore Roşu. Equality of streams is a Π_2^0-complete problem. In *ICFP*. ACM, 2006.
19 Jan M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theor. Comp. Sci.*, 308(1-3):1–53, 2003.
20 Jakob Grue Simonsen. The Π_2^0-completeness of most of the properties of rewriting systems you care about (and productivity). In *Proc. 20th Int. Conf. on RTA*, RTA '09, pages 335–349. Springer, 2009.
21 Robert I. Soare. *Recursively Enumerable Sets and Degrees*. Perspectives in Mathematical Logic. Springer, 1987.
22 Hans Zantema. Well-definedness of streams by transformation and termination. *LMCS*, 6(3), 2010. paper 21.