Linearization of Automatic Arrays, Weave Specifications, and Variadic Sequences

David Sprunger
Indiana University

May 18, 2013

Abstract

Grabmayer, Endrullis, Hendriks, Klop, and Moss [5] developed a method for defining automatic sequences in terms of 'zip specifications', and proved that a sequence is automatic [2] iff it has a zip specification where all zip terms have the same arity. An open question regards what kinds of sequences are given by zip specifications where the zip terms do *not* have a constant arity, and whether it can be decided if two such specifications have the same solution.

This paper begins by investigating a similar definitional scheme for the higher-dimensional counterpart of automatic sequences, automatic arrays. In the course of establishing the results required for this machinery, we find an isomorphism between a final coalgebra for arrays and the standard final coalgebra for sequences. This isomorphism preserves automaticity properties: an array is k, l-automatic iff its corresponding sequence is kl-automatic. The former notion of automaticity (k, l-automatic, note the comma) is defined for arrays as in [2], and the latter notion is the standard notion of automaticity for sequences where kl is just the product of k and l. It also provides a convenient way to translate between stream zip specifications and array zip specifications.

Further investigation of the properties of this isomorphism, which is closely related to the z-order curve [7], yields a natural type of automaton related to the automata used to generate automatic arrays. This variadic automaton can be used to provide a partial answer (in the affirmative) to the question of decidability of mixed zip specifications. If a zip specification, now allowed to have zip terms of different arities, satisfies a particular condition, we will use variadic automaton to generate the kth entry in the solution effectively. Given two such zip specifications we can run an algorithm for bisimulation on the automata generating the solutions to determine whether the solutions to the specifications match.

1 Introduction

The automatic sequences are a class of sequences arising naturally in both computer science and many different fields of mathematics [2]. A well-known automatic sequence, the Prouhet-Thue-Morse sequence, has been rediscovered over and over again by its appearances in combinatorics, algebra, number theory, differential geometry, and combinatorial game theory. This has prompted Allouche and Shallit to regularly prepend the honorific "the ubiquitous" to the sequence's name. (See [1] for more details on the appearances of the PTM sequence.)

As an example, sequences are interesting to algebraists as the expansions of numbers in particular bases. A number is rational, for example, iff its representation in base k is an eventually periodic sequence of digits. Sequences which are eventually periodic are highly regular; at the other end of the spectrum are the totally random sequences, the investigation of which relies primarily on probabilistic methods and/or information-theoretic methods. Automatic sequences form a class of sequences "more random" than the eventually periodic sequences and yet

with enough structure that their properties can be investigated without needing probabilistic methods. Numbers whose base k expansion are automatic sequences are useful concrete examples in the study of transcendental numbers. Automatic sequences also play a crucial role in determining transcendence of power series over function fields—this is the content of Christol's theorem [3].

While the identities and properties of particular automatic sequences are often encountered and used in mathematics, their definition and the method by which they are generated lies more naturally in the realm of theoretical computer science. Automatic sequences list the outputs of a simple class of automata on representations of the natural numbers. This class of automata is a slight generalization of the better known deterministic finite automata (DFA), which is the preferred model in formal language theory for recognizing the regular languages. As a consequence of this similarity, the methodologies used for regular languages and automatic sequences have strong resemblances to each other, but also have some important differences.

Regular languages enjoy a well developed notational system for defining languages and language operations, namely regular expressions. Commonly juxtaposition or $\cdot$ indicate language concatenation, L^* denotes the Kleene closure of L, and $\cup$ or $+$ commonly indicates the union of languages. Since the words in regular languages are finite, this definitional scheme is able to compactly describe languages while simultaneously hinting at the structure of their elements.

On the other hand, the class of automatic sequences has necessarily infinite objects, so schemes giving a finite definition of even one element of this space require some thought. Given two finite presentations of infinite objects like sequences, it can be quite difficult to determine whether the two generated objects are even equal. To reason about infinite objects, principles of coinduction and corecursion arising from coalgebra are often employed, as introduced by Rutten [9]. Though the coalgebraic approach is not the original automata theory approach to regular languages, the theory of regular languages and DFAs has been recast in those terms by Rutten [8].

The standard finite scheme to define an automatic sequence is to give a finite automaton which generates that sequence. This scheme is useful when computing entries in the sequence, but suffers from some ambiguities involving input representation. Grabmayer et al. proposed an alternate definitional scheme which gives a sequence as the result interleaving the entries of other sequences. They showed the collection of automatic sequences exactly coincides with the sequences that can be defined with their style of zip specifications [5].

In this paper we first by summarize the main techniques used by Grabmayer et al. to define zip-specifications for automatic sequences. Then we begin to investigate a similar scheme for automatic arrays, which are a two-dimensional generalization of automatic sequences. This investigation will uncover an interesting isomorphism between automatic arrays and sequences, which we will go on to use to give a partial answer to an open question regarding zip specifications.

2 A brief introduction to automatic sequences, coalgebras, and zip specifications

We begin by outlining the primary techniques used in the study of automatic sequences and coalgebras. For readers seeking more details on the former topic there is a standard book by Allouche and Shallit [2]. For the latter, there are many papers by Rutten, Kupke, and many others expositing coalgebraic methods most notably [9], see also [6] particularly for coalgebraic methods applied to sequences of symbols. With some understanding of these techniques, we

then turn to how these techniques are used to construct zip-specifications and check for equality of solutions.

2.1 Automata with output

We now describe in greater detail the process of generating a sequence from an automaton [2]. A **DFAO** (**D**eterministic **F**inite **A**utomaton with **O**utput) is an automaton with the following components: an input alphabet (A), a finite set of states (Q), a designated start state ($q_0 \in Q$), a transition map taking the current state and an input symbol and returning the next state ($\delta : Q \times A \to Q$), an output alphabet ($\Delta$) and a output map $f : Q \to \Delta$. These generalize their better known cousin, the DFA, in that a DFA's output alphabet is necessarily $\Delta = \{\text{accept}, \text{reject}\}$.

We can extend a DFAO's transition function from single letters in A to all words in A^* with the following recursive scheme: $\delta(q, w) = \begin{cases} q & \text{if } w = \epsilon \\ \delta(\delta(q, a), w') & \text{if } w = aw' \end{cases}$ for $w, w' \in A^*$, $a \in A$ and ϵ being the empty word in A^*. As we are about to note, this is not the only possible nor the only common way to extend the transition function to all of A^*. We will indicate that a DFAO uses *this* convention for its extended transition function by saying the DFAO "reads its input in left-to-right order" or "uses the frontwards order".

We could alternatively extend a DFAO's transition function to all of A^* with a slightly different recursive scheme: $\delta(q, w) = \begin{cases} q & \text{if } w = \epsilon \\ \delta(\delta(q, a), w') & \text{if } w = w'a \end{cases}$. In this case, we say the DFAO reads its input in the right-to-left order or the backwards order.

In either case, we say the DFAO outputs $d \in \Delta$ on input $w \in A^*$ if $d = f(\delta(q_0, w))$. We say a DFAO is a k-DFAO if $|A| = k \in \omega$ and in this case assume $A = \{0, 1, 2, \ldots, k-1\} = \mathbb{N}_{<k}$. Note that each integer is naturally associated to a string in A^*: its standard base-k representation. We denote the standard base-k representation of n by $[n]_k \in \mathbb{N}^*_{<k}$. We say a k-DFAO generates the sequence $\sigma = \sigma_0\sigma_1\sigma_2 \ldots \sigma_n \ldots$ if $\sigma_n = f(\delta(q_0, [n]_k))$. That is, the DFAO outputs σ_n on the input $[n]_k$.[1] If σ is a sequence generated by a k-DFAO, we say it is k-automatic.

At this point, there are a few good and well-answered questions to point out. How is the collection of sequences generated by k-DFAOs using the frontwards order related to the collection of sequences generated by k-DFAOs using the backwards order? Is it always possible to modify a k-DFAO to allow the representation $[n]_k$ to have leading 0's without affecting the stream it generates? It turns out the collection of automatic sequences is exactly the same under each of these input conventions, and indeed the functions taking a DFAO with one input convention to any of the others are computable [2, p. 159]. As a result, we will assume for the rest of the paper that our DFAOs read their input in the backwards order.

2.2 Coalgebras, finality and bisimulation

The next critical component in this machinery is the notion of coalgebras for a functor [9]. In this paper we will be considering the Set endofunctor $F : X \mapsto \Delta \times X^k$ where the functor acts on arrows by sending the Set morphism $f : X \to Y$ to $Ff : \Delta \times X^k \to \Delta \times Y^k$ defined by $Ff = \langle id_\Delta, f, f, \ldots, f \rangle$. A **coalgebra for this functor** is a set C together with a function $c : C \to FC = \Delta \times C^k$.

[1] Note the DFAO is not a Mealy/Moore machine or other kind of simple transducer, so it is not generating this sequence in a single run. Each run of the DFAO produces a single output and by listing these individual results of each of the runs for these particular inputs we get a sequence.

The map c is usually called the structure map for the coalgebra, while C is its carrier. We will often need to talk about the different components of the structure map, $c = \langle o, c_0, c_1, \ldots, c_{k-1} \rangle$, where $o : C \to \Delta$ and $c_i : C \to C$. o is often called the **observation component** of the structure, while the c_i are called the **transitions** for the structure.

We define the category of F-coalgebras as follows: objects in the category are F-coalgebras like (C, c) and a morphism of coalgebras $\varphi : (C, c) \to (D, d)$ is a function $\varphi : C \to D$ such that the following diagram commutes:

$$\begin{array}{ccc} C & \xrightarrow{c} & \Delta \times C^k \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle F\varphi} \\ D & \xrightarrow[d]{} & \Delta \times D^k \end{array}$$

We say an F-coalgebra is **final** if it is a final object in the category of F-coalgebras. Final coalgebras are particularly important in the theory of coalgebras because there is a powerful technique for establishing the equality of two elements of a final coalgebra: bisimulation.

A **bisimulation** on an F-coalgebra (C, c) is a relation $R \subseteq C \times C$ such that for all $(x, y) \in R$ we have $o(x) = o(y)$ and for all $0 \leq i < k$ we have $(c_i(x), c_i(y)) \in R$. A standard theorem of coalgebra is that every bisimulation on a final coalgebra is a subset of the identity relation [9][10]. That is, if R is a bisimulation on a final coalgebra and $(x, y) \in R$, then $x = y$.

2.3 The k-kernel and the $\mathbf{zip}_k$ function

An important object in the study of automatic sequences is the so-called kernel of a sequence. To define this, we first define projection maps on the set of sequences in Δ: $\pi_{i,k} : \Delta^\omega \to \Delta^\omega$ is given by $\pi_{i,k}(\sigma) = \sigma_i \sigma_{i+k} \sigma_{i+2k} \ldots \sigma_{i+nk} \ldots = \{\sigma_{i+nk}\}_n$. The k-kernel of the sequence σ is the set of all sequences which can be reached by repeatedly applying the maps $\pi_{0,k}, \pi_{1,k}, \ldots, \pi_{k-1,k}$ to σ. A well-known theorem states that a sequence is k-automatic iff its k-kernel is finite [2, p. 185].

Coalgebras and bisimulations are commonly used in the analysis of state transition systems and other process calculi. DFAs and DFAOs are simple examples of state transition systems and so they form a natural example of coalgebras and thereby a place to do bisimulations. Less readily apparent is that there is a coalgebra structure on the set of sequences themselves.[2]

A 2-DFAO generating an automatic sequence naturally gives a coalgebra for the functor $F_2 X = \Delta \times X^2$, via the map $Q \to \Delta \times Q \times Q$ given by $\langle f, \delta(\cdot, 0), \delta(\cdot, 1) \rangle$. As a result of the so-called input robustness results mentioned in section 2.1, we can also assume without loss of generality that our DFAOs (which read in the backwards order) ignore trailing zeroes. That is, we have $f \circ \delta(\cdot, 0) = f$. This property, as will be discussed later, is called zero-consistency. In the case of general coalgebras it is quite restrictive, but not so for automatic sequences [6].

Now, it is a fact due to [6] and [5] independently that Δ^ω is the carrier for a coalgebra $(\Delta^\omega, \langle \mathtt{hd}, \pi \ldots_{0,2}, \pi_{1,2} \rangle)$ which is final for the subcategory of F_2-coalgebras including DFAOs ignoring leading zeroes.[3] Therefore there is a unique map, *seq*, such that the following diagram commutes:

[2] For more details on the coalgebraic structure of DFAs and their final coalgebra, see [8].

[3] Indeed, Δ^ω is the carrier for a coalgebra for the functor $F_k X = \Delta \times X^k$ *for all* k, not just 2, where the structure map for the coalgebra has first component `hd` and all other components are the k-kernel maps $\pi_{i,k}$ [6].

$$\begin{array}{ccc} Q & \xrightarrow{\langle f,\delta(\cdot,0),\delta(\cdot,1)\rangle} & \Delta \times Q \times Q \\ {\scriptstyle seq}\downarrow & & \downarrow{\scriptstyle Fseq} \\ \Delta^\omega & \xrightarrow[\langle \mathtt{hd},\pi_{0,2},\pi_{1,2}\rangle]{} & \Delta \times \Delta^\omega \times \Delta^\omega \end{array}$$

The *seq* map takes a state q to the automatic sequence generated by the DFAO when using q as the start state.

The $\mathbf{zip}_k$ function takes k sequences and merges them into a single sequence by alternating through their terms. For example, $\mathbf{zip}_2(\sigma,\tau) = \mathbf{zip}_2(\sigma_0\sigma_1\sigma_2\ldots,\tau_0\tau_1\tau_2\ldots) = \sigma_0\tau_0\sigma_1\tau_1\sigma_2\tau_2\ldots$. The $\mathbf{zip}_k$ function in some sense serves to invert the action of the k-kernel maps. Note $\pi_{i,k}(\mathbf{zip}_k(\sigma_0,\sigma_1,\ldots,\sigma_{k-1})) = \sigma_i$ and conversely $\mathbf{zip}_k(\pi_{0,k}(\sigma),\pi_{1,k}(\sigma),\ldots,\pi_{k-1,k}(\sigma)) = \sigma$.

Using the **zip** function, we can set up systems of equations to define streams. For example, the system

$$\begin{cases} m = 0 : x \\ x = 1 : \mathbf{zip}(x,y) \\ y = 0 : \mathbf{zip}(y,x) \end{cases}$$

has the Thue-Morse sequence as the solution for m.

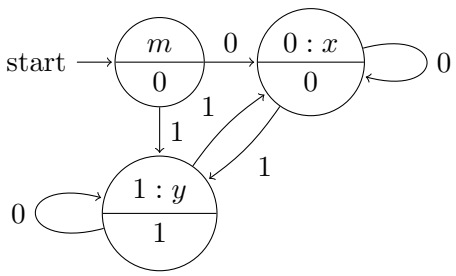
To be particular, a zip specification in an alphabet A is a set of variables, S, along with a zip term for each variable, where a zip term is generated by the BNF grammar

$$T ::= s \mid a : T \mid \mathbf{zip}_k(T,T,\ldots,T)$$

where $s \in S$, $a \in A$ and k zip terms are provided as the argument for a term using the $\mathbf{zip}_k$ rule. For example, $\mathbf{zip}_2(1 : x, \mathbf{zip}_3(0 : y, x, 1 : 0 : z))$ is a zip term in the alphabet $\{0,1\}$ with variables $\{x,y,z\}$. A zip specification then gives a zip term for each variable in the set.

Grabmayer et al. gave conditions for the existence and uniqueness of solutions to these zip specifications. Roughly stated, there is a solution to the specification if each variable has exactly one definition, and there is a unique solution if the first symbol can be determined for each variable. They then proved that solutions to zip specifications where all terms using the **zip** function have arity k are k-automatic. This claim tells us that all solutions for the variables in the above specification are 2-automatic, for example.

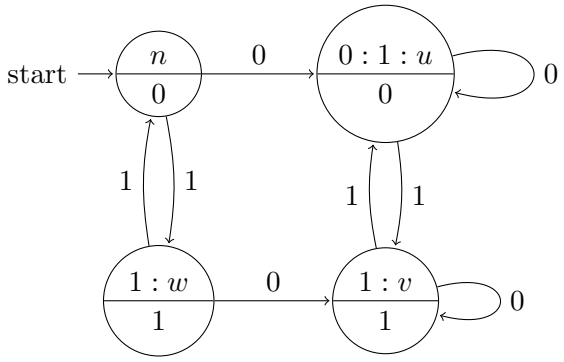
Generating an automaton for this is fairly straightforward. We start with the root (first) variable and apply the $\pi_{i,2}$ maps just as if we're generating the 2-kernel of a sequence, but we get state names instead. For example, $\pi_{0,2}(m) = \pi_{0,2}(0 : x) = \pi_{0,2}(0 : 1 : \mathbf{zip}(x,y)) = 0 : \pi_{0,2}(\mathbf{zip}(x,y)) = 0 : x$, so after reading 0, our automaton will transition from a state labelled m to a state labelled $0 : x$. Similarly, $\pi_{1,2}(0 : x) = \pi_{1,2}(0 : 1 : \mathbf{zip}(x,y)) = 1 : y$ so the 1 transition out of $0 : x$ goes to $1 : y$ and so on. The full 2-DFAO for this specification is given below:



Now given a second zip specification, such as

$$\begin{cases} n = 0 : \mathbf{zip}(1 : w, 1 : u) \\ u = 1 : \mathbf{zip}(v, u) \\ v = 0 : \mathbf{zip}(v, 1 : u) \\ w = \mathbf{zip}(n, v) \end{cases}$$

it is fairly straightforward to check that the first few entries in the solution for n match the first few entries for the solution of m, but is a nearly impossible game of index bookkeeping to check that they have the same values at all entries. To aid our understanding and to simplify matters we use the machinery of coalgebras. First we generate a 2-DFAO for this specification in the same manner as the one before it:



Next we find attempt to find a bisimulation from the first DFAO to the second relating m to n. $R = \{(m, n), (0 : x, 01 : u), (0 : x, n), (1 : y, 1 : w), (1 : y, 1 : v)\}$ is such a bisimulation. Then if we apply the *seq* map to this bisimulation it remains a bisimulation in Δ^ω. In particular, $(seq(m), seq(n)) \in seq(R)$, which is a bisimulation in a final coalgebra, so $seq(m) = seq(n)$. This allows us to conclude the sequence generated in the first DFAO starting from m is the same as the sequence generated from the second DFAO starting from state n.

2.4 Automatic arrays

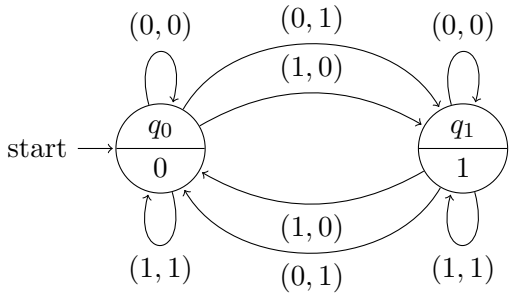
Automatic arrays are the generalization of automatic sequences to two dimensions. [2, Ch. 14]

Let's imagine we have a DFAO with input alphabet $\mathbb{N}_{<k} \times \mathbb{N}_{<l}$. (We will sometimes abbreviate this situation with the term k, l-DFAO.) We can encode a pair $(m, n) \in \mathbb{N} \times \mathbb{N}$ as a word in this alphabet in a standard way: we find the base-k representation of m, $[m]_k$, and the base-l representation of n, $[n]_l$, and pad whichever has fewer digits with leading zeroes until they are the same length. Then the standard k, l-encoding of (m, n) is the sequence of pairs of these digits starting with the most significant digits.

For example, the standard 2,5-encoding for $(13, 82)$ is formed by taking $[13]_2 = 1101$ and $[82]_5 = 312 = 0312$ and then forming pairs: $(1, 0)(1, 3)(0, 1)(1, 2)$.

Now imagine an infinite grid with positions indexed by $\mathbb{N} \times \mathbb{N}$ with the symbol at position (m, n) being the output of the DFAO when given the standard encoding for (m, n) in $\mathbb{N}_{<k} \times \mathbb{N}_{<l}$. Such an infinite array is called a k, l**-automatic array** in analogy to the definition of automatic sequences. When the bases k and l can be inferred from context, we may abbreviate and use the term "automatic array".

As an example consider the following automaton and the automatic array it generates below. Note that this automaton has input alphabet $\mathbb{N}_{<2} \times \mathbb{N}_{<2}$.



0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1
1	0	0	1	0	1	1	0	**0**	1	1	0	1	0	0	1
0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1
0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1
1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1
0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1
0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0
1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1
1	0	0	1	0	1	1	0	0	1	1	0	1	0	0	1
0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0

The bold zero on this line is at (13, 8). This gets encoded as (1,1)(1,0)(0,0)(1,0). You can check to see that this is the right output using the automaton.

2.5 Previously known results on automatic arrays

At this point the same kinds of questions that are natural for automatic sequences can be asked for automatic arrays. Does the collection of automatic arrays change if we make our k, l-DFAOs read in backwards order vs. frontwards order? Can we allow leading $(0, 0)$ characters for frontwards reading DFAOs without changing the notion of which arrays are k, l-automatic? These standard input robustness results work out the same way for automatic arrays as they do for sequences: changing the direction of the input and adding leading/trailing zeroes does not change the collection of automatic arrays [2, p. 408].

Indeed, very many of the results for automatic sequences have a corresponding result for automatic arrays. One important example of this phenomenon regards the kernel of an array. We define $\pi_{\frac{i}{k},\frac{j}{l}}$ to take an array a with entries $a_{m,n}$ for all $n, m \geq 0$ to the array with entries $a_{km+i,ln+j}$ for all $n, m \geq 0$. Then the k, l-kernel of the array a is the set of all arrays which can be reached from a by repeatedly applying the functions $\pi_{\frac{i}{k},\frac{j}{l}}$ for $i \in \mathbb{N}_{<k}$ and $j \in \mathbb{N}_{<l}$. It turns out the k, l-kernel of an array is finite iff the array is k, l-automatic [2, p. 409].

3 Weave specifications for automatic arrays

To establish results for zip specifications for sequences, several steps were needed. First, one must give conditions for existence and uniqueness of solutions to zip specifications. Then, for zip specifications fulfilling these criteria, one shows there is a natural way to produce a DFAO which generates the sequence which solves the zip specification. Lastly, using a final coalgebra

one has a basis for determining equality of solutions to different zip specifications. We begin by developing this last step for automatic arrays.

3.1 Final coalgebras for automatic arrays

Denote by $\mathrm{Gr}_\Delta = \Delta^{\omega^2}$ the set of all quarter-infinite grids with symbols in Δ. Let $c : \mathrm{Gr}_\Delta \to \Delta$ be the function taking a grid to its corner entry. That is, $c(g) = g_{0,0}$. We will be showing that Gr_Δ with the transition structure given by function c and the projections π defined above make it into a final coalgebra.

Suppose we have a functor $FX = \Delta \times X^{kl}$ and $(G, s : G \to \Delta \times G^{kl})$ is a coalgebra for that functor. Then we name the $kl + 1$ components of the structure:

$$s = \langle o, s_{(0,0)}, s_{(0,1)}, \ldots, s_{(0,l-1)}, s_{(1,0)}, \ldots, s_{(k-1,l-1)} \rangle$$

where $o : G \to \Delta$ and $s_{(i,j)} : G \to G$. As before, o is the observation component of the structure, while the other components are the transition components of the structure. We say a coalgebra for this structure is **zero − consistent** when $o = o \circ s_{(0,0)}$, which is to say applying the zero-transition doesn't change the observation. For general coalgebras, the zero-consistency condition is a restrictive requirement, but for automatic sequences, where we know we can always modify our DFAOs to accept input with leading zeroes, zero-consistency is not a strong requirement [6].

We will also employ a notation for writing out the composition of many transitions in these coalgebras. Rather than writing $s_3 \circ s_2 \circ s_3 \circ s_1$, we will write $s_{(3)(2)(3)(1)}$, where the subscript is a word in the available transition subscripts. So, for example, we might write $(\pi_{\frac{0}{2},\frac{0}{2}} \circ \pi_{\frac{1}{2},\frac{0}{2}} \circ \pi_{\frac{0}{2},\frac{1}{2}})(x) = \pi_{(\frac{0}{2},\frac{0}{2})(\frac{1}{2},\frac{0}{2})(\frac{0}{2},\frac{1}{2})}(x)$.

Before proving that Gr_Δ is a final coalgebra, we prove two simple facts.

Lemma 1: Let $(m, n)_{k,l}$ be the standard k, l-encoding of the pair (m, n). Then $(m, n)_{k,l}(i, j)$, the encoding of (m, n) followed by the symbol (i, j) where $i \in \mathbb{N}_{<k}$, $j \in \mathbb{N}_{<l}$, is the k, l-encoding of $(mk + i, nl + j)$.

Proof: Let's consider $(mk + i, nl + j)_{k,l}$. It is clear $[mk + i]_k = [m]_k \cdot i$ and $[nl + j]_l = [n]_l \cdot j$. Then the last pair in the standard k, l-encoding will be (i, j), and the preceding symbols will just be the k, l-encoding of (m, n). That is, $(mk + i, nl + j)_{k,l} = (m, n)_{k,l}(i, j)$, as desired.

Lemma 2: For all $a \in \mathrm{Gr}_\Delta$, $(c \circ \pi_{(m,n)_{k,l}})(a) = a_{m,n}$.

Proof: We show this by induction on the length of the k, l-coding for (m, n). If the encoding is empty, then $m = n = 0$ so we have $c(a) = a_{0,0}$, which is true by the definition of c.

Now suppose this statement is true for all encodings of length $\leq d$ and suppose the length of $(m, n)_{k,l}$ is $d + 1$. Further let $m = q_1 k + i$ and $n = q_2 l + j$ where $i \in \mathbb{N}_{<k}$ and $j \in \mathbb{N}_{<l}$ by the division algorithm. There is at least one symbol in our encoding, so we write $(m, n)_{k,l} = w(i, j)$ where (i, j) is a single symbol in $\mathbb{N}_{<l} \times \mathbb{N}_{<k}$ and $w = (q_1, q_2)_{k,l}$ is the remainder of the coding, which has length d. Then we are considering $(c \circ \pi_{(m,n)_{k,l}})(a) = (c \circ \pi_w \circ \pi_{\frac{i}{k},\frac{j}{l}})(a)$ by definition of the structure subscripts. Now the induction hypothesis kicks in and tells us $c \circ \pi_w$ finds the (q_1, q_2) element of the array it's applied to, so the above reduces to $(\pi_{\frac{i}{k},\frac{j}{l}}(a))_{q_1,q_2}$. Now the definition of $\pi_{\frac{i}{k},\frac{j}{l}}(a)$ above tells us the q_1, q_2 entry in this array is the $(kq_1 + i, lq_2 + j) = (m, n)$ entry of a, as desired.

We are now ready to prove the promised result regarding the finality of the coalgebra of Gr_Δ with c and the $\pi_{\frac{i}{k},\frac{j}{l}}$.

Propsition 3: The coalgebra $\langle c, \pi_{\frac{i}{k},\frac{j}{l}} \rangle : \mathrm{Gr}_\Delta \to \Delta \times (\mathrm{Gr}_\Delta)^{kl}$ where $i \in \mathbb{N}_{<k}$ and $j \in \mathbb{N}_{<l}$ is final for the zero-consistent coalgebras of the functor $FX = \Delta \times X^{kl}$.

Proof: Suppose $\langle o, p_{(i,j)} \rangle : A \to \Delta \times A^{kl}$ is a zero-consistent F-coalgebra. We define a map $\varphi : A \to \mathrm{Gr}_\Delta$ by the following: $a \in A$ gets mapped to the Δ-array whose (m,n) entry is given by $\varphi(a)_{m,n} = o(p_{(m,n)_{k,l}}(a))$.

We must show that this is an F-coalgebra morphism. For this we verify the observation and transition parts separately.

$$\begin{array}{ccc} A & \xrightarrow{\langle o, p_{(i,j)} \rangle} & \Delta \times A^{kl} \\ \varphi \downarrow & & \downarrow id_\Delta \times \varphi^{kl} \\ \mathrm{Gr}_\Delta & \xrightarrow[\langle c, \pi_{\frac{i}{k},\frac{j}{l}} \rangle]{} & \Delta \times (\mathrm{Gr}_\Delta)^{kl} \end{array}$$

(Observation) We must show $o(a) = c(\varphi(a))$. We know $c(\varphi(a)) = \varphi(a)_{0,0} = o(p_{(0,0)}(a))$ by our definition of φ. Then $o(p_{(0,0)}(a)) = o(a)$ since $(A, \langle o, p_{(i,j)} \rangle)$ is a zero-consistent F-coalgebra. Hence we have $c(\varphi(a)) = o(a)$, as desired.

(Transitions) We must show $\varphi \circ p_{(i,j)} = \pi_{\frac{i}{k},\frac{j}{l}} \circ \varphi$ for all $i \in \mathbb{N}_{<k}$ and $j \in \mathbb{N}_{<l}$.

We first claim that $c \circ \varphi \circ p_{(m,n)_{k,l}} = c \circ \pi_{(m,n)_{k,l}} \circ \varphi$ for all $m, n \geq 0$. Note that $\varphi(a)_{m,n} = (o \circ p_{(m,n)_{k,l}})(a)$ by our definition of φ. We also know from Lemma 2 that $c \circ \pi_{(m,n)_{k,l}}$ finds the (m,n) element of an array, so $\varphi(a)_{m,n} = (c \circ \pi_{(m,n)_{k,l}} \circ \varphi)(a)$. Therefore $o \circ p_{(m,n)_{k,l}} = c \circ \pi_{(m,n)_{k,l}} \circ \varphi$. Now from the observation result above, we know $c \circ \varphi = o$, so we get $c \circ \varphi \circ p_{(m,n)_{k,l}} = c \circ \pi_{(m,n)_{k,l}} \circ \varphi$ as desired.

Now we return to proving $\varphi \circ p_{(i,j)} = \pi_{\frac{i}{k},\frac{j}{l}} \circ \varphi$. Let $a \in A$ and $m, n \in \mathbb{N}$. Then we must show $(\varphi \circ p_{(i,j)})(a)_{m,n} = (\pi_{\frac{i}{k},\frac{j}{l}}) \circ \varphi)(a)_{m,n}$ for all such a, m, n. Since $c \circ \pi_{(m,n)_{k,l}}$ selects the m, n entry of a grid by Lemma 2, we rewrite this as

$$(c \circ \pi_{(m,n)_{k,l}} \circ \varphi \circ p_{(i,j)})(a) = (c \circ \pi_{(m,n)_{k,l}} \circ \pi_{\frac{i}{k},\frac{j}{l}} \circ \varphi)(a)$$
$$(c \circ \pi_{(m,n)_{k,l}} \circ \varphi \circ p_{(i,j)})(a) = (c \circ \pi_{(mk+i,nl+j)_{k,l}} \circ \varphi)(a)$$

where the equality is by Lemma 1. Now using the claim we just proved we can rewrite the left hand side as

$$(c \circ \pi_{(m,n)_{k,l}} \circ \varphi \circ p_{(i,j)})(a) = (c \circ \varphi \circ p_{(m,n)_{k,l}(i,j)})(a) = (c \circ \varphi \circ p_{(mk+i,nl+j)_{k,l}})(a)$$

which is again by Lemma 1. Therefore, we have shown φ is a coalgebra map.

To show φ is the final coalgebra map, we must show it is the unique map with these properties. This is a straightforward induction argument where most of the hard work is done by our previous lemmas. Since we do not need the details later, we omit them.

3.2 Array linearization with final stream coalgebras

In the last section we showed that the set of arrays in Δ, Gr_Δ, carries a final coalgebra structure for the zero-consistent coalgebras of the functor $FX = \Delta \times X^{kl}$. We also know the stream coalgebra $(\Delta^\omega, \langle \mathtt{hd}, \mathcal{N}_{kl} \rangle) = (\Delta^\omega, \langle \mathtt{hd}, \pi_{0,kl}, \ldots \pi_{kl-1,kl} \rangle)$ is a final coalgebra for the zero-consistent coalgebras of F. [5] Consequently, these two coalgebras must be isomorphic. We shall next make this isomorphism explicit.

The coalgebra $(\Delta^\omega, \langle \mathtt{hd}, \mathcal{N}_{kl} \rangle)$ has the structure maps $\mathtt{hd}(\sigma) = \sigma_0$ and $\pi_{j,kl}(\sigma_n) = \{\sigma_{nkl+j}\}_n$ being the stream kernel maps with $0 \leq j < kl$. Now fortunately, our result above gives the

explicit construction of the final coalgebra map from this coalgebra into the Gr_Δ coalgebra. Let $\varphi : \Delta^\omega \to \mathrm{Gr}_\Delta$ be this final map.

For simplicity, let's consider the case where $k = l = 2$ and $\sigma = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots)$, and figure out what $\varphi(\sigma)$ looks like. To find the entry at position (m, n) we find the standard 2,2-encoding of the pair and the follow the corresponding stream projections to find the correct entry. Working through this calculation many times shows that $\varphi(\sigma)$ begins like

42	43	46	47	58	59	62	63
40	41	44	45	56	57	60	61
34	35	38	39	50	51	54	55
32	33	36	37	48	49	52	53
10	11	14	15	26	27	30	31
8	9	12	**13**	24	25	28	29
2	3	6	7	18	19	22	23
0	1	4	5	16	17	20	21

Now since Δ^ω with the `hd` and $\pi_{i,4}$ stream projections is also a final coalgebra for the functor $FX = \Delta \times X^4$, we know the coalgebra morphism φ must be an isomorphism. In particular, φ^{-1} takes an array and transforms it into a stream. Having figured out what φ does to a stream, we can easily invert it to see how φ^{-1} linearizes an array.

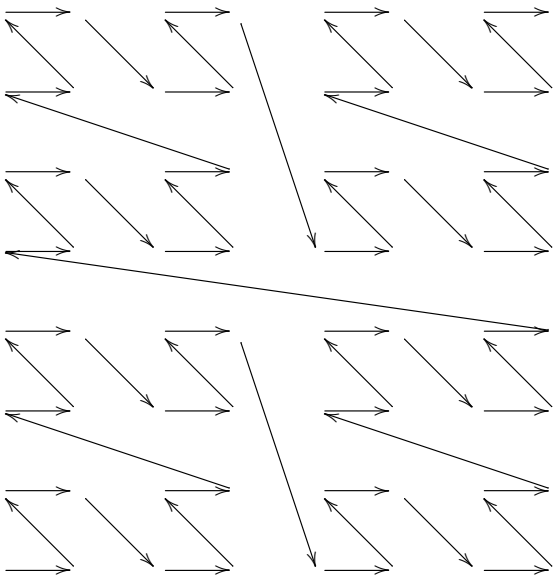


Figure 1: The start of $\mathtt{lin}_{2,2}$

This is already known to computer scientists as the "z-order curve" as introduced by G. Morton [7]. It has an important property which makes it easy for machines working with binary representations to use: the entry at (m, n) in the array gets mapped to the element in the stream at the position which is formed by zipping the digits of $[m]_2$ and $[n]_2$. We see this on the diagram below, which is $\varphi(\sigma)$ as above but with everything translated into binary and with row and column labels.[4]

[4]We have colored all the column-related digits red, so grayscale copies of this document may appear to have a lighter shade for these digits.

111	101010	101011	101110	101111	111010	111011	111110	111111
110	101000	101001	101100	101101	111000	111001	111100	111101
101	100010	100011	100110	100111	110010	110011	110110	110111
100	100000	100001	100100	100101	110000	110001	110100	010101
011	001010	001011	001110	001111	011010	011011	011110	011111
010	001000	001001	001100	**001101**	011000	011001	011100	011101
001	000010	000011	000110	000111	010010	010011	010110	010111
000	000000	000001	000100	000101	010000	010001	010100	010101
	000	001	010	011	100	101	110	111

Indeed we might have guessed this from our definition of the automata operating on pairs of digit representations. If we look at the pairs of red and black digits, they give the pairs in the encoding of that position in the array. At the position $(2,3)_{2,2} = (1,1)(0,1) = (0,0)(1,1)(0,1)$, highlighted in bold, we indeed get the thirteenth element of the stream and $001101 = [13]_2$.

With this operation in mind, it is easy to describe what φ does as well. Given an element in a sequence, we take its position and write that position in binary, padding with leading zeroes to make the representation have an even number of digits. Then every other digit starting with the first forms the representation for the row in the array, and every other digit starting with the second forms the representation of the column in the array. For example, the 28th element of the stream gets mapped by φ to the $(2,6)$ element of the array since $[28]_2 = 11100 = 011100 = 011100$ and $010 = [2]_2$ and $110 = [6]_2$.

3.3 Connections between sequences and arrays using linearization

To avoid confusion between φ and φ^{-1}, we'll call $\varphi = \texttt{ord}$ since it takes a sequence and z-orders it into a grid and we'll call $\varphi^{-1} = \texttt{lin}$ since it linearizes an array. Each of these should be subscripted with k and l when the grid coalgebra they refer to is unclear from context.

As an example application of this isomorphism, consider the following proposition.

Proposition 4: Suppose $\sigma \in \Delta^\omega$ is a sequence and $g \in \mathrm{Gr}_\Delta$ is a grid. g is k,l-automatic iff $\texttt{lin}_{k,l}(g)$ is kl-automatic, and σ is kl-automatic iff $\texttt{ord}_{k,l}(\sigma)$ is k,l-automatic.

Proof: We prove the first statement, regarding g and $\texttt{lin}(g)$. This will then immediately give the second fact by taking $g = \texttt{ord}(\sigma)$ and noting that $\texttt{lin}(\texttt{ord}(\sigma)) = \sigma$.

The fact that $\texttt{lin} : \Delta^\omega \to \mathrm{Gr}_\Delta$ is a coalgebra morphism gives us $\pi_{(i.j)} \circ \texttt{lin} = \texttt{lin} \circ \pi_{\frac{i}{k},\frac{j}{l}}$ for all $i \in \mathbb{N}_{<k}$ and $j \in \mathbb{N}_{<l}$. Then every unique image of g under repeated application of the $\pi_{\frac{i}{k},\frac{j}{l}}$ corresponds to a unique image of $\texttt{lin}(g)$ under repeated application of the $\pi_{m,kl}$. Therefore, the k,l-kernel of g is in 1-1 correspondence with the kl-kernel of $\texttt{lin}(g)$. Since an array is k,l-automatic iff its k,l-kernel is finite, and a sequence is kl-automatic iff its kl-kernel is finite, we have g is k,l-automatic iff $\texttt{lin}(g)$ is kl-automatic.

A further illustration of the usefulness of this isomorphism is to extend the work of Grabmayer et al. in automatic sequences and zip specifications to automatic arrays. First we must define the analog of the **zip** function for automatic arrays.

Definition: Let the function $\mathbf{wv}_{k,l} : \mathrm{Gr}_\Delta^{kl} \to \mathrm{Gr}_\Delta$ (read "weave") be defined by

$$\mathbf{wv}_{k,l}(g_0, g_1, \ldots, g_{kl-1}) = \texttt{ord}_{k,l}(\mathbf{zip}(\texttt{lin}_{k,l}(g_0), \texttt{lin}_{k,l}(g_1), \ldots, \texttt{lin}_{k,l}(g_{kl-1}))).$$

Let's take the 2x2 weave as an example. Suppose

$$A = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ a_{20} & a_{21} & a_{22} & \dots \\ a_{10} & a_{11} & a_{12} & \dots \\ a_{00} & a_{01} & a_{02} & \dots \end{bmatrix},$$

and B, C and D are similar. Then

$$\mathbf{wv}_{2,2}(A,B,C,D) = \mathbf{wv}_{2,2}\begin{pmatrix} C & D \\ A & B \end{pmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ c_{10} & d_{10} & c_{11} & d_{11} & \dots \\ a_{10} & b_{10} & a_{11} & b_{11} & \dots \\ c_{00} & d_{00} & c_{01} & d_{01} & \dots \\ a_{00} & b_{00} & a_{01} & b_{01} & \dots \end{bmatrix}$$

Note this is the block-wise weave of these grids in the pattern suggested by the second argument style. **wv** is similar in spirit to **zip** since it inverts the action of the k,l-kernel maps. That is, $\pi_{\frac{i}{k},\frac{j}{l}}\left(\mathbf{wv}\begin{pmatrix} g_{k-1,0} & \cdots & g_{k-1,l-1} \\ \vdots & & \vdots \\ g_{0,0} & \cdots & g_{0,l-1} \end{pmatrix}\right) = g_{i,j}$ and conversely

$$\mathbf{wv}\begin{pmatrix} \pi_{\frac{k-1}{k},\frac{0}{l}}(g) & \dots & \pi_{\frac{k-1}{k},\frac{l-1}{l}}(g) \\ \vdots & & \vdots \\ \pi_{\frac{0}{k},\frac{0}{l}}(g) & \dots & \pi_{\frac{0}{k},\frac{l-1}{l}}(g) \end{pmatrix} = g.$$

Now we are ready to develop a notion of a weave specification for arrays. Let S be a finite set of variables. We define a **weave term** in S by the BNF grammar

$$W ::= s \mid a : W \mid \mathbf{wv}_{k,l}(W,\dots,W)$$

where s is a variable from S, $a \in \Delta$, $k,l \in \mathbb{N}_{\geq 2}$ and there are kl weave terms provided as the argument to $\mathbf{wv}_{k,l}$. A **weave specification** is a pairing of each variable in S to a weave term in S.

We have described how to interpret $\mathbf{wv}_{k,l}(W,\dots,W)$. $a : W$ is to be interpreted as the grid $\mathtt{ord}(a : \mathtt{lin}(W))$. As a result of these definitions and the fact that they have isomorphic generating grammars, every weave term is the z-ordering of a zip term. This means weave specifications will have the same existence and uniqueness conditions on their solutions as their related zip specifications.

For example, the weave specification:

$$\begin{cases} m = 0 : x \\ x = 1 : \mathbf{wv}\begin{pmatrix} x & y \\ 1 : y & 0 : x \end{pmatrix} \\ y = 0 : \mathbf{wv}\begin{pmatrix} y & x \\ 0 : x & 1 : y \end{pmatrix} \end{cases}$$

has the unique solution given by the 2,2-automatic array from section 2.4. As a result of the fact that all weave terms are the z-ordering of a zip term, we can translate between the two formats easily. This gives the following theorem immediately.

Proposition 5: For grids $g \in \mathrm{Gr}_\Delta$ the following are equivalent:
(i) g is k,l-automatic.

(ii) g can be defined by a $\mathbf{wv}_{k,l}$ specification.

(iii) g has a finite k,l-kernel

Proof: We have already noted the equivalence of (i) and (iii) is known in the literature [2]. Every $\mathbf{wv}_{k,l}$ specification is the z-ordering of a $\mathbf{zip}_{kl}$ specification, and similarly every $\mathbf{zip}_{kl}$ specification can be written as the linearization $\mathbf{wv}_{k,l}$ specification. Our earlier proposition shows that a grid is k,l-automatic iff its linearization is kl-automatic. Therefore, the equivalence of (i) and (ii) is the same as the equivalence of a sequence being kl-automatic and having a $\mathbf{zip}_{kl}$ specification. The latter equivalence is proven in [5], so our result follows.

4 Variadic sequences

The results from the previous section suggest that a large portion of the theory of automatic sequences can be lifted directly to statements about automatic arrays by means of the z-ordering isomorphism. This, in turn, may suggest that the theory of automatic arrays may be only a reflection of the theory of automatic sequences, unremarkable in its own right. In this section we attempt to dispel this notion by using automatic arrays to solve a problem for which automatic sequences do not immediately suffice.

4.1 A subclass of zip-mix specifications

In this section we will briefly study a very restricted subclass of zip-mix specifications, which we call **zip-mix specifications of alternating arity**. The question of whether there was a general algorithm for deciding whether two sequences defined by zip-mix specifications remained open at the end of [5] but was recently resolved in [4]. We pursue this example primarily, therefore, to illustrate how automatic arrays can be used in a place where automatic sequences are limited.

To be exact, we consider zip-mix specifications with three properties: 1) the set of variables for the specification is partitioned into two pieces $V = V_k + V_l$, 2) if $x \in V_k$ then the term for x in the specification (i.e. T_x so that $x = T_x$ is in the specification) is a zip-k term and the variables used in that term all come from V_l, and 3) similarly the term for $y \in V_l$ must be a zip-l term mentioning only variables from V_k. As an example, consider the following:

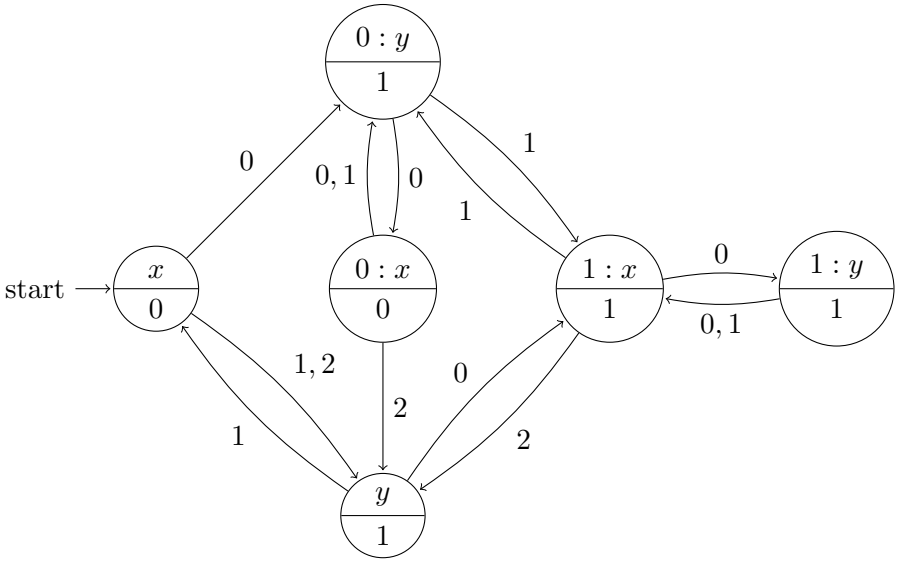
$$\begin{cases} x = 0 : \mathbf{zip}_3(y, y, y) \\ y = 1 : \mathbf{zip}_2(x, x) \end{cases}$$

In this case, $\{x\} = V_3$ and $\{y\} = V_2$, which partitions the set of variables. The term defining x is a zip-3 term which only mentions variables from V_2 and similarly the term defining y is a zip-2 term which only mentions variables from V_3, so this specification has all three properties required. We call such a zip-specification a **zip-mix specification of alternating arity**, but so as not to have to repeat this name too often, we assume all zip-specifications in this section have these properties unless specified otherwise.

If we were to apply the procedure from section 2.3 to create a DFAO for this specification, we would want to use the 3-kernel maps when looking at the state x and the 2-kernel maps when looking at the state y. That would mean we would need to have three different transitions out of the state labelled x and two different transitions out of the state labelled y. Ordinary DFAOs do not allow this, so we must alter our definition of DFAO slightly to accommodate. Therefore we define a **k,l-alternating DFAO** to be a DFAO with the following modifications: $A = \mathbb{N}_{<k} + \mathbb{N}_{<l}$, $Q = Q_k + Q_l$, $q_0 \in Q_l$ and $\delta : (Q_k \times \mathbb{N}_{<k}) + (Q_l \times \mathbb{N}_{<l}) \to Q$ must have the property that $\delta[Q_k \times \mathbb{N}_{<k}] \subseteq Q_l$ and conversely $\delta[Q_l \times \mathbb{N}_{<l}] \subseteq Q_k$. The intuition here is that we've partitioned the set of states into two disjoint pieces, Q_k and Q_l, analogous to requirement (1) on our zip-specifications. While in Q_k we read a digit from base k (i.e. from $\mathbb{N}_{<k}$) and then

transition to a state in Q_l and then vice versa, analogous to requirements (2) and (3) on our zip-specifications.

Now we can create a 2,3-alternating DFAO from the zip-mix specification given above using the process described in section 2.3:



Now one way to build up the machinery needed to give an algorithm for deciding whether the solutions to two zip-mix specifications of alternating arity are equal would be to follow the general outline from [5] which we used for weave specifications earlier. First we would have to figure out how to represent this type of machine as the coalgebra of a functor, and then we would have to find a final coalgebra for that functor. This is already enough of a task, but a further demerit to this approach is that at the end we will only be able to tell whether two k, l-alternating DFAOs generate the same sequence. We would not be able to use the result, for example, to decide whether a k, l-alternating DFAO and an l, k-alternating DFAO generate the same sequence. Instead, we will leverage the input flexibility of k, l-DFAOs (the automata used to generate automatic arrays) along with our previous results about k, l-DFAOs.

4.2 k, l-alternating DFAOs and k, l-DFAOs

For every finite input string $\sigma = \sigma_1 \dots \sigma_{n-1}\sigma_n$ to a k, l-alternating DFAO, we can form an input to a k, l-DFAO by making the length of σ even by possibly adding a leading 0 and then pairing digits. We say a k, l-alternating DFAO and a k, l-DFAO "have the same behavior on input σ" if the two machines have the same output when the k, l-alternating DFAO is given σ and the k, l-DFAO is given the paired version of σ. We say these two machines of these types "have the same behavior" if they have the same behavior on every valid input string to the k, l-alternating DFAO.

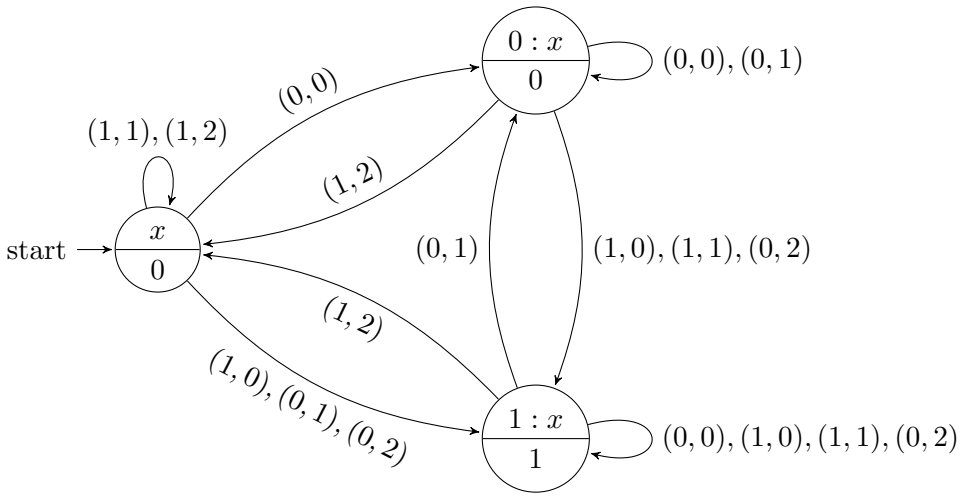
Proposition: For every zero-consistent k, l-alternating DFAO there is a k, l-DFAO which has the same behavior.

Proof: Suppose $(\mathbb{N}_{<k} + \mathbb{N}_{<l},\ Q_k + Q_l,\ q_0,\ \delta,\ \Delta,\ f)$ is a k, l-alternating DFAO. We take the alphabet for our k, l-DFAO to be $\mathbb{N}_{<k} \times \mathbb{N}_{<l}$, the set of states to be Q_l with the same start state, the output alphabet remains Δ and the final output map is just the restriction of the original output map to our set of states, $f|_{Q_l}$. The interesting part is the transition map for our k, l-DFAO:

$$\delta'(q, (i, j)) = \delta(\delta(q, j), i)$$

With this definition it is easy to check that the extended transition functions of these DFAOs coincide for all even-length input. We must use the zero-consistency property if the input to the k, l-DFAO was padded with an extra leading 0. As noted before, insisting on zero-consistency for DFAOs is not a strong requirement.

If we apply this process to the k, l-alternating DFAO given above, we get the following 2,3-DFAO

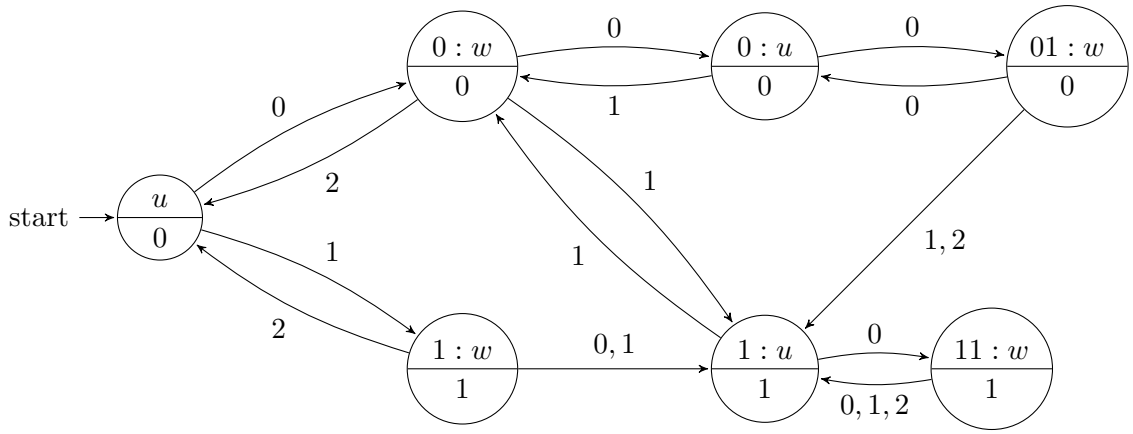


Now we may use our earlier results for automatic arrays. Since the array generated by the k, l-DFAO is by definition an automatic array, its linearization is kl-automatic by Proposition 4. Hence to check whether two zip-mix specifications of alternating arity have the same stream as the solution to their root, it suffices to find a bisimulation relating these kl-DFAOs. (TODO?)

As an example, consider the following zip-mix specification:

$$\begin{cases} u = 01 : \mathbf{zip}_2(w, w) \\ w = 1 : \mathbf{zip}_3(u, u, u) \end{cases}$$

We claim this specification has the same solution (for u) as the specification listed earlier in this section (for x). We first find the $3, 2$-alternating DFAO for this specification:



5 Summary

We have presented a natural final coalgebra for arrays, for which automatic arrays are the rational part. This coalgebra is final for a functor also commonly used in the study of automatic sequences and hence gives rise to an isomorphism between the array coalgebra and sequence coalgebra preserving many important automaticity properties. We then lifted several results about automatic sequences to facts about automatic arrays, including the zip specification scheme of [5] to the case of automatic arrays. Finally, using the additional flexibility of automatic arrays we gave a partial answer to an open problem regarding zip specifications of mixed arity. There are a few natural avenues for investigation to proceed from here, which we summarize briefly.

m-partitions and other variadic zip specifications We assumed in section 4.2 that we had a bipartition of the terms in the specification. If we have a tripartition or, more generally, a partition of the variables into m sets with the property that everything in the same class has the same zip-arity and all variables mentioned in the terms in that set come from another class of the partition, a similar theory using an automatic complex of dimension m should yield decidability. We would like some exploration on what automata naturally emerge from relaxing this restriction on the variables and perhaps answering the decidability question for zip specifications where the base determination is made by a DFA. (See [5] for more details on this set up.)

Other space filling curves We found the z-order/Morton curve as an isomorphism between array coalgebras and sequence coalgebras. There are many other space filling curves (such as the Hilbert curve or the Peano curve), so we wonder whether there are coalgebras which naturally give rise to these curves as isomorphism between arrays and sequences.

Acknowledgement

I owe many thanks to Larry Moss for his invaluable advice, comments, and patience reading much rougher drafts.

References

[1] J.-P. Allouche and J. Shallit. The ubiquitous Prouhet-Thue-Morse sequence. In T. Helleseth C. Ding and H. Niederreiter, editors, *Sequences and their applications, Proceedings of SETA'98*, pages 1–16. Springer Verlag, 1999.

[2] J.-P. Allouche and J. Shallit. *Automatic Sequences*. Cambridge University Press, 2003.

[3] G. Christol. Ensembles presque periodiques k-reconnaissables. *Theoret. Comput. Sci.*, 9(1):141–145, 1979.

[4] J. Endrullis, C. Grabmayer, and D. Hendriks. Mix-automatic sequences. In *Language and Automata Theory and Applications (LATA 2013)*, pages 262–274. Springer, 2013.

[5] C. Grabmayer, J. Endrullis, D. Hendriks, J.W. Klop, and L.S. Moss. Automatic sequences and zip-specifications. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, pages 335–344. IEEE Computer Society, 2012.

[6] C. Kupke and J.J.M.M. Rutten. On the final coalgebra of automatic sequences. In *Logic and Program Semantics*, pages 149–164. Springer, 2012.

[7] G.M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[8] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In *CONCUR'98: concurrency theory (Nice)*, volume 1466 of *Lecture Notes in Comput. Sci.*, pages 194–218. Springer, Berlin, 1998.

[9] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1):3–80, 2000. Modern algebra and its applications (Nashville, TN, 1996).

[10] J. J. M. M. Rutten and D. Turi. On the foundations of final semantics: nonstandard sets, metric spaces, partial orders. In *Semantics: foundations and applications (Beekbergen, 1992)*, volume 666 of *Lecture Notes in Comput. Sci.*, pages 477–530. Springer, Berlin, 1993.

6 Appendix A: The 2,3-ordering

The standard $2,3$-ordering of $(0,1,2,3,4,\ldots)$ is as follows:

21	22	23	27	28	29	33	34	35
18	19	20	24	25	26	30	31	32
3	4	5	9	10	11	15	16	17
0	1	2	6	7	8	12	13	14

and the $2,3$-linearization proceeds as follows: