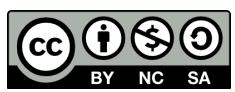
CDM

Mealy Machines

Klaus Sutner

Carnegie Mellon University

Spring 2025



Upcoming Attractions

Burnside/Pólya/Redfield is an application of group theory to combinatorial counting, a sort of computational problem.

Next we will go in the opposite direction and show how computability, and in particular the theory of finite state machines can be applied to group theory.

As it turns out, FSMs can provide very concise and elegant descriptions of groups with amazing properties (added benefit: the machines are often tiny).

> The author (along with many other people) has come recently to the conclusion that the functions computed by the various machines are more important–or at least more basic–than the sets accepted by these devices.

Dana Scott
Some Definitional Suggestions for Automata Theory
1967

One of the key ideas of Bourbaki is to organize mathematics around the study of structures, and in particular first-order structures of the form

$$\mathcal{C} = \langle A; f_1, f_2, \ldots, f_k, R_1, R_2, \ldots, R_\ell \rangle$$

Here A is the carrier set, the f_i are functions on A, and the R_i are relations on A. The signature of the structure describes the arity of these functions and relations.

We can use first-order logic over the language $\mathcal{L}(f_1, \ldots, f_k, R_1, \ldots, R_\ell)$ to describe properties of such structures.

For our purposes it is convenient to think of functions as just being special kinds of relations and simply deal exclusively with the latter.

A relational structure is a FO structure of the form

$$\mathcal{C} = \langle A; R_1, R_2, \ldots, R_k \rangle$$

Again, this is no serious restriction, we can always fake functions as relations:

$$x \twoheadrightarrow y \iff f(x) = y$$

Here $\twoheadrightarrow$ is just a binary relation with certain special properties (total and single-valued).

Note that this switch to relations changes our formulae a bit.

For example, consider the simple atomic formula $f(f(a)) = b$.
Utterly standard and useful notation, but it actually hides a quantifier:

$$\exists\, z \left(f(a) = z \wedge f(z) = b \right)$$

To be clear, the notation is perfectly good, but any algorithm dealing with the formula has to cope with this invisible quantifier, one way or another.

In a purely relational structure everything is clearly visible, we have to write something like

$$\exists\, z \left(a \rightarrowtriangle z \wedge z \rightarrowtriangle b \right)$$

This can make life very slightly easier for algorithms.

But Why?

Following Scott's suggestion, here is a moon shot.

Wild Idea:

Can one describe mathematical structures using only FSM?

From a certain perspective, these structures would be fairly simple, though nowhere near as simple as one might suspect. They are by no means trivial.

The hope is that we can use the perfectly algorithmic theory of FSMs to analyze these structures and understand them in great detail.

The structures we are interested in have the restricted form

$$\mathcal{C} = \langle A; R_1, R_2, \ldots, R_k \rangle$$

where everything is represented by finite state machines.:

- $A \subseteq \Sigma^\star$ is a recognizable language, and
- $R_i \subseteq A^{\ell_i}$ is a recognizable relation on words.

We already know how to handle the carrier set, but we do not have any concept "recognizable relation" at this point.

There are two basic options to generalize our machines from languages to relations:

- Modify the machines so that they by recognize relations (i.e., k-tuples of words) rather than just words.
- Generalize Kleene's algebraic characterization of regular languages in terms of regular expressions.

The first option is critical for algorithms; we can lift some (but emphatically not all) algorithms from regular languages to relations.

The algebraic approach serves as a sanity check and is in many ways less ad hoc; the generalization is fairly canonical.

So the next project is to generalize recognizable languages to some reasonable class of recognizable relations, which are often called rational relations.

As already mentioned, we have two basic options to tackle this problem:

- Invent some kind of memoryless machine that takes k-tuples of words as input, rather than just single words.
- Exploit Kleene's algebraic characterization in terms of regular expressions and modify them from languages to relations.

We'll start with the machine model and then develop the corresponding algebraic approach. As it turns out, they agree entirely.

A transduction is a relation of the form

$$\rho \subseteq \Sigma^\star \times \Gamma^\star$$

In other words, ρ is a binary relation on words (or, alternatively, a language of 2-track words). Usually we have $\Sigma = \Gamma$, but occasionally the general form is more useful.

It often helps to think of such a relation as a map

$$\rho : \Sigma^\star \longrightarrow \mathfrak{P}(\Gamma^\star)$$

where $\rho(x) = \{\, y \mid \rho(x, y) \,\}$. We are given x as input, and want to compute y as output (but note that transductions are not single-valued in general).

Our definition nicely generalizes to k-ary relations for $k > 2$. Instead of single words over an alphabet we have k-tuples of words, possibly over different alphabets:

$$u \in \Sigma_1^\star \times \Sigma_2^\star \times \ldots \times \Sigma_k^\star$$

To emphasize that we still have word-specific operations such as concatenation on these objects we will refer to them as k-track words or multi-words.

To display the component words we usually write

$$u = u_1{:}u_2{:}\ldots{:}u_k$$

The most important case is when $k = 2$ and we will write

$$x{:}y \qquad \text{or} \qquad x/y$$

to indicate that x is replaced by y.

So What?

Wurzelbrunft immediately concludes the following: We are living in StringWorld™, so we can easily code up a relation $R \subseteq \Sigma^\star \times \Sigma^\star$ as

$$R^{\#} = \{\, x \# y \mid x, y \in \Sigma^\star, R(x, y) \,\} \subseteq \Sigma^\star \# \Sigma^\star$$

where $\# \notin \Sigma$.

Then we use a FSM to work with $R^{\#}$, done.

Sadly, no. Unless x is tiny, the machine will have forgotten almost all of it when it gets to y.

We could not even handle "x is a prefix of y" this way.

The simple solution is to not read x and y sequentially, but in parallel.

This requires to interleave the symbols in some way, either by alternating between x and y as in strict shuffle.

Alternatively, we could switch to a product alphabet $\Sigma \times \Sigma$ and read one symbol each at every step. To handle strings of different lengths we can use padding.

Before we go into the weeds, a particularly simple example that is already very important in group theory.

The most simple-minded transductions are alphabetic: each input symbol is replaced by an output symbol in a completely deterministic manner.

In a Mealy machine, the transitions are described by a function

$$\delta : Q \times \Sigma \longrightarrow \Sigma \times Q$$

So transitions are labeled by pairs of letters

$$p \xrightarrow{a/b} q$$

Mealy machines are length-preserving: the output string has the same length as the input string.

Alternative Definition

Instead of using a single transition function $\tau : Q \times \Sigma \to \Sigma \times Q$ it is sometimes more convenient to split things into two functions

$$\delta : Q \times \Sigma \to Q \qquad \text{state transitions}$$
$$\rho : Q \times \Sigma \to \Sigma \qquad \text{output function}$$

In particular the curried versions of these functions

$$\delta_a : Q \to Q \qquad a \in \Sigma$$
$$\rho_p : \Sigma \to \Sigma \qquad p \in Q$$

are often more convenient to use than the original definition.

Usually Mealy machines have initial and final states, but they don't matter for us at this point. Sam Eilenberg referred to these machines as output modules.

For any state p in a Mealy machine we get a transduction

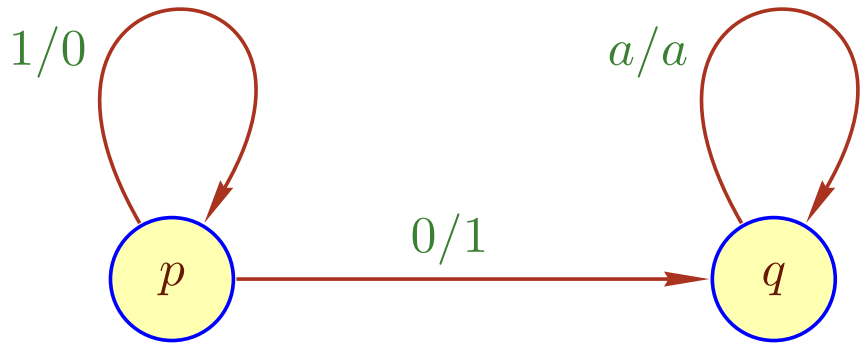
$$\underline{p} : \Sigma^\star \to \Sigma^\star$$

by considering runs starting at state p.

Since Mealy machines are deterministic, $\underline{p}$ really is a function.

Exercise

Give a formal inductive definition of $\underline{p}$.



We can describe the transductions defined by this machine as follows:

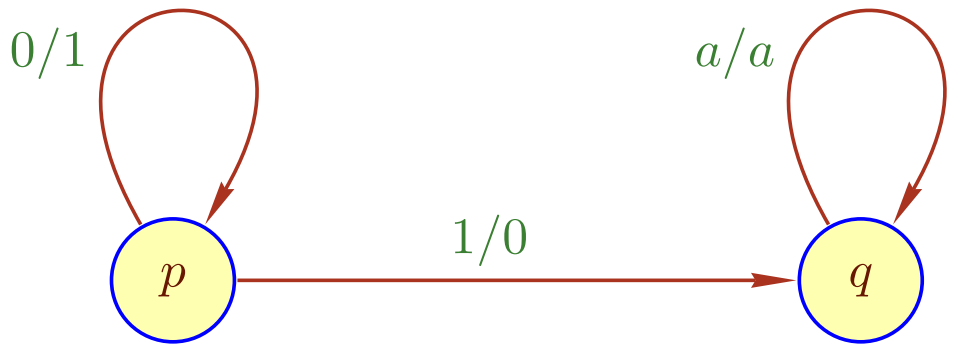
$$\underline{p}\,(0x) = 1\,\underline{q}\,(x)$$
$$\underline{p}\,(1x) = 0\,\underline{p}\,(x)$$
$$\underline{q}\,(x) = x$$

On k-bit strings[†], $\underline{p}$ is the successor modulo 2^k.

In the literature, this is called the "adding machine."

[†]On infinite strings we get the true successor function, but on 2-adic numbers.

The last machine has a special property: its transduction is a bijection on $\mathbf{2}^\star$. In fact, we get a permutation of Σ^n for each n. The permutation is just a cycle of length 2^n.

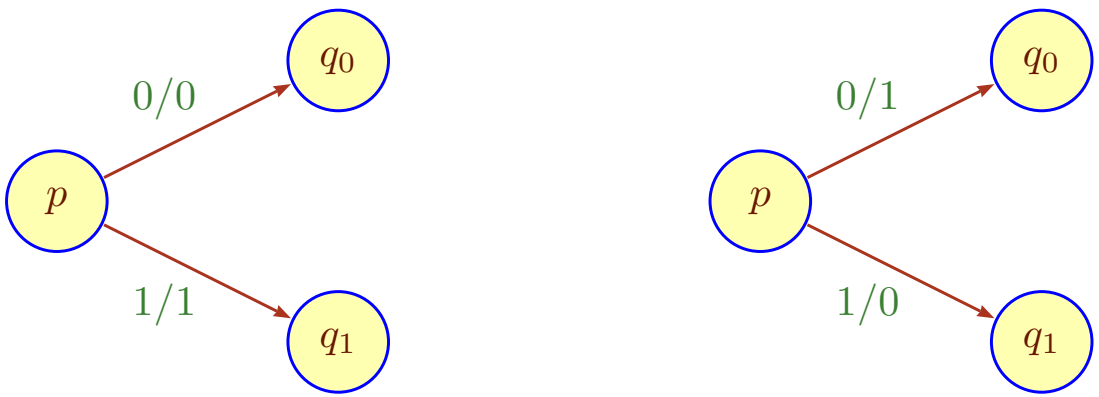


The transducer that computes the inverse function: simply swap all transition labels in the successor machine.

Definition

A Mealy transducer is invertible if, for every state p, the output maps ρ_p are permutations of Σ.

In particular for $\Sigma = \mathbf{2}$ there are only two types of states: copy and toggle.



For the moment, we will focus on invertible transducers.

In an invertible Mealy machine $\mathcal{A}$ we get a collection of transductions $\underline{p}$, $p \in Q$ that are all permutations of $\Sigma^\star$ (and in fact of Σ^n). Define

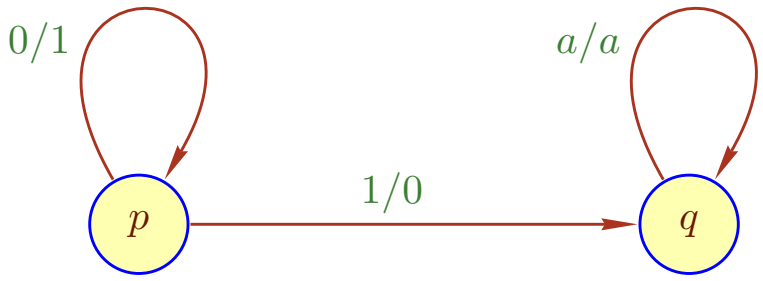
$$\begin{aligned} \mathrm{Sgrp}(\mathcal{A}) &= \text{ the semigroup generated by the } \underline{p} \\ \mathrm{Grp}(\mathcal{A}) &= \text{ the group generated by the } \underline{p} \end{aligned}$$

For the semigroup we take all possible compositions of the basic transductions, producing an automaton semigroup.

For the group we add the inverse transductions $\underline{p}^{-1}$, producing an automaton group.

Sometimes $\mathrm{Sgrp}(\mathcal{A})$ is already a group, but typically it is not.

Example: Successor



The semigroup is (isomorphic to) $\langle \mathbb{N}; + \rangle$:
$\underline{p}^{\,k}$ corresponds to $k \geq 1$ and $\underline{q}$ corresponds to 0.

The group is (isomorphic to) $\langle \mathbb{Z}; + \rangle$:
we have to add the inverse $\underline{p}^{\,-k}$ for $k \geq 1$.

. . . is suboptimal, as usual.

The notion of an automaton semi/group is the officially sanctioned one for our Mealy machines.

But there are also automatic semi/groups, that also involve finite state machine but in a different way. Essentially, there are finite state machines that compute the group operations.

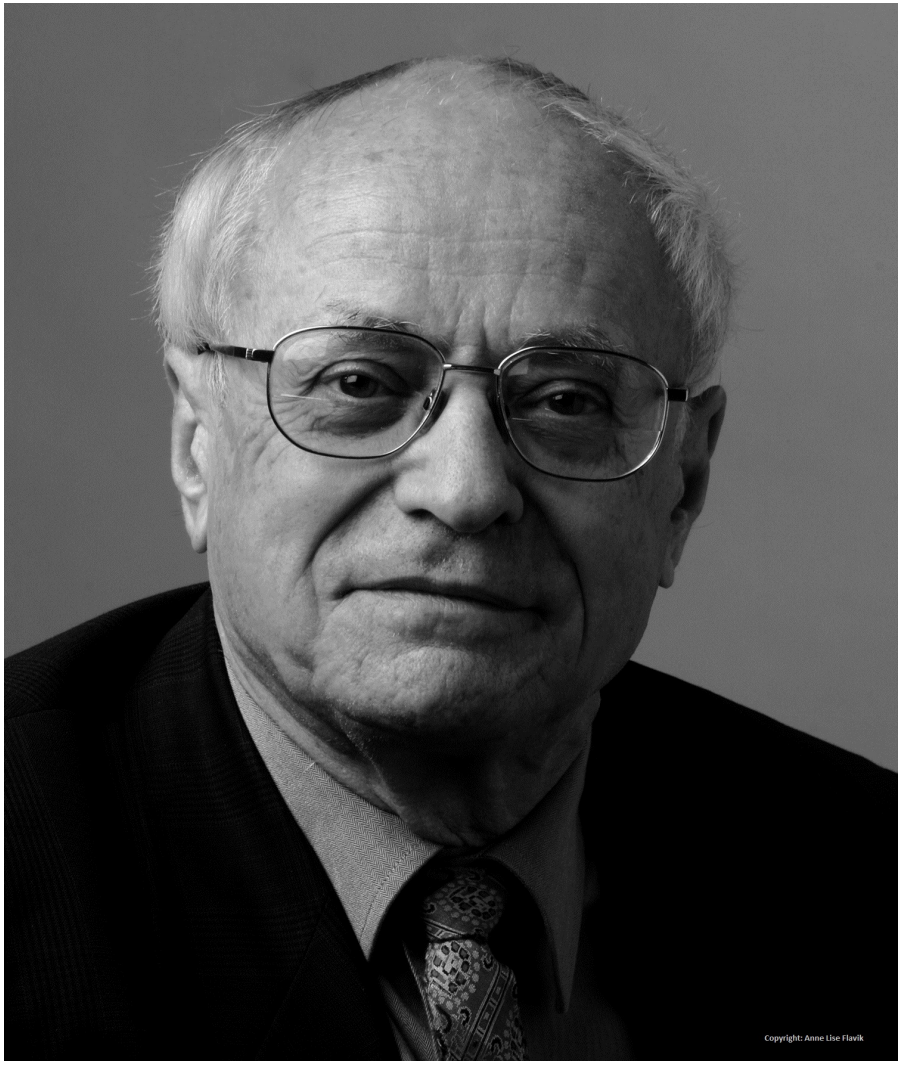
And there are transition semi/groups that are defined as the semigroups generated by the maps δ_a of a DFA. In fact, these are historically the first attempt to exploit algebra to understand finite state machines (algebraic automata theory).

Right now, automaton semi/groups are all that matters to us.

Why should anyone care about these semigroups/groups?

Two reasons:

- Automaton groups have become the goto source for examples and counterexamples in group theory. The key is that some enormously complicated groups have descriptions in terms of automata with just a handful of states.
- Automatic groups are important in studying low dimensional manifolds (Thurston et al., solvable word problem).
- Transducers operating on binary and 2-adic numbers are quite useful in understanding digital circuits. Take a look at Vuillemin.



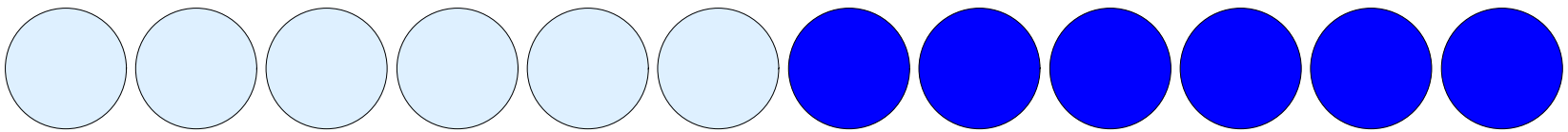
Fields Medal, Abel Prize, Steele Prize, Wolf Prize

Member Bourbaki

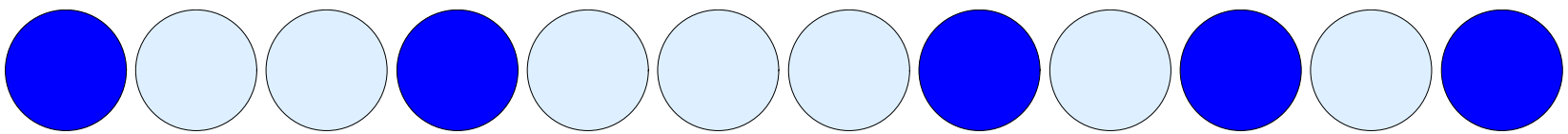
Arbres, Amalgames, SL_2 (1977)

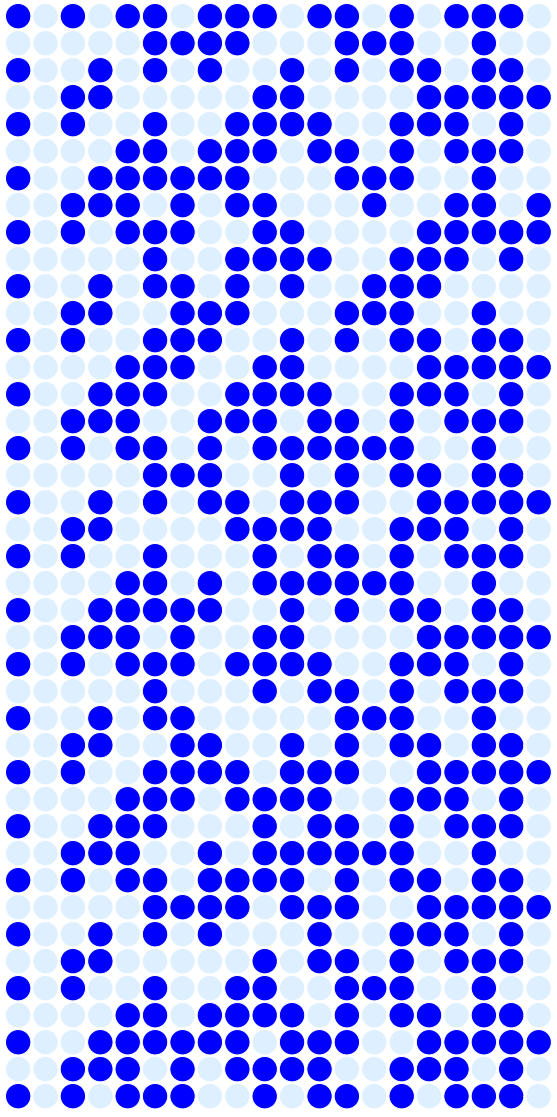
A Pebble Game

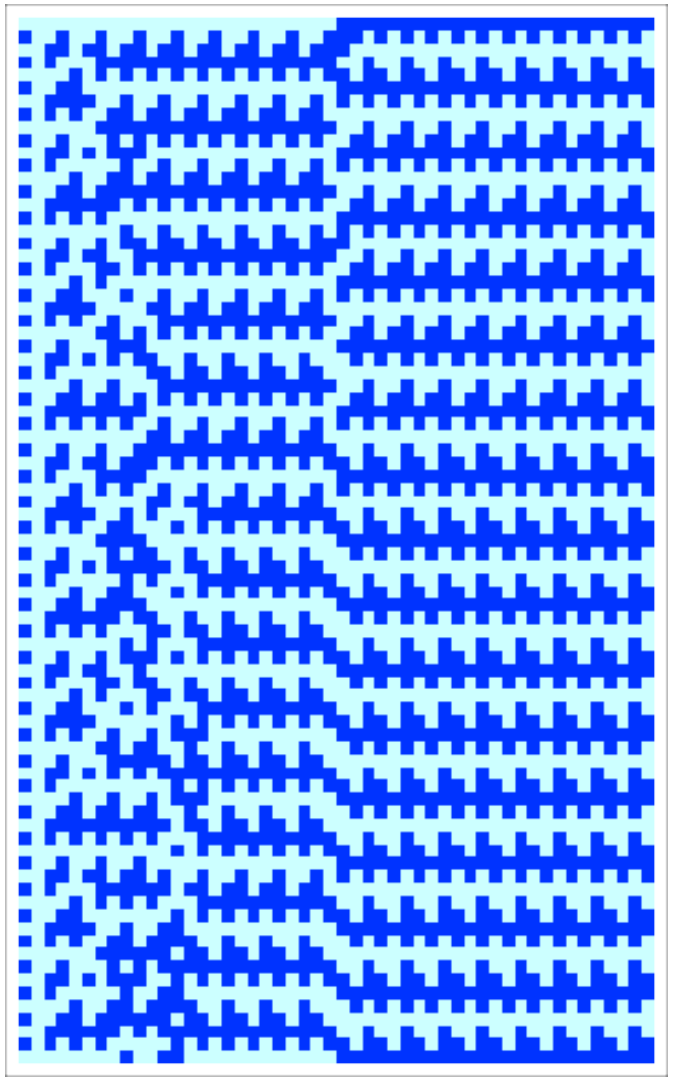
Take a row of tokens, white on one side, blue on the other.

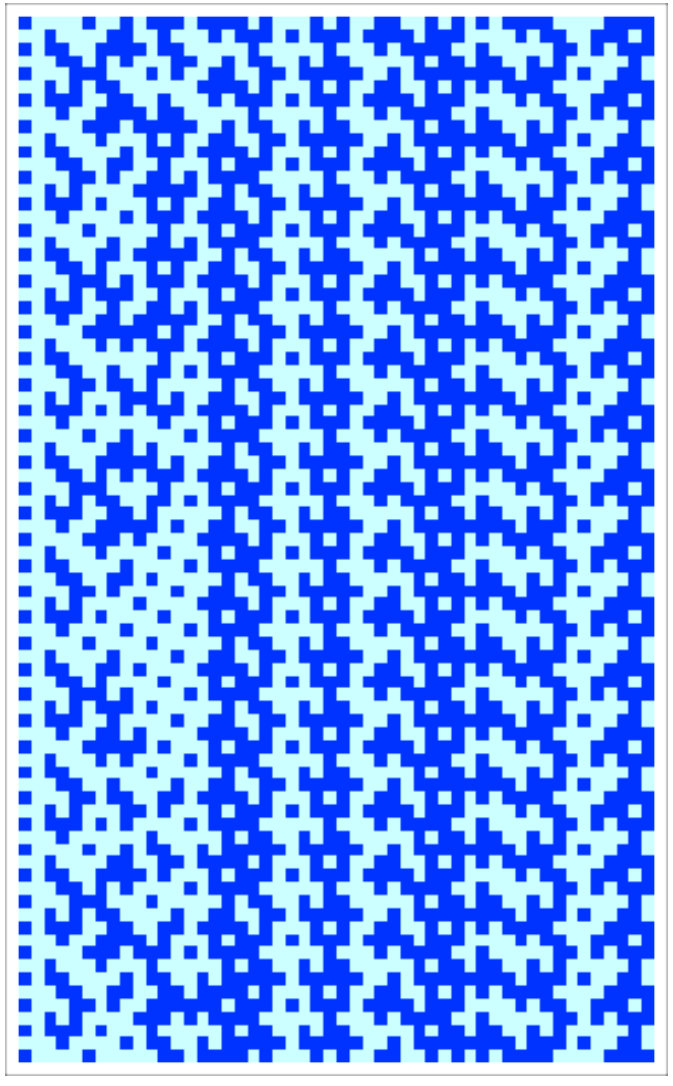


Starting at the left, flip the current token. If it is now white, skip the next token. Otherwise, skip the next two tokens. Repeat till you fall off the end.







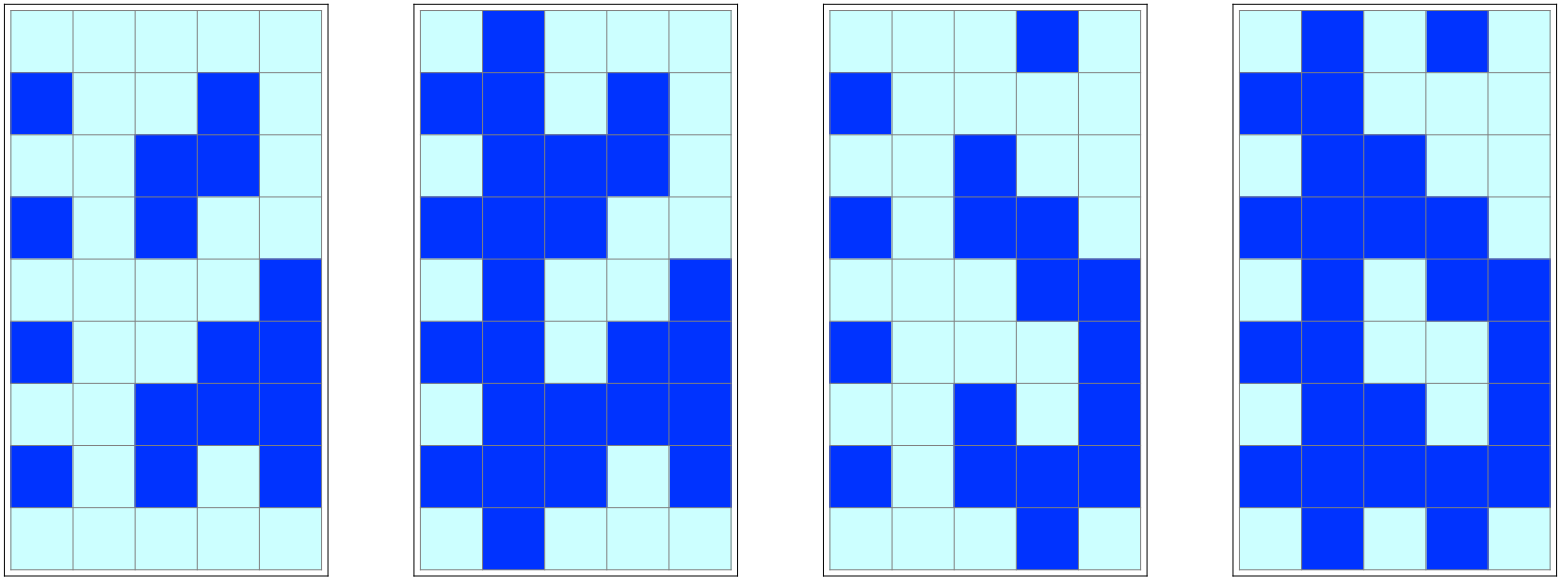


We are dealing with an operation $\phi : \mathbf{2}^\star \to \mathbf{2}^\star$.

Easy Observations:

- ϕ is trivially length-preserving.
- ϕ is injective.
- ϕ is a permutation of $\mathbf{2}^n$ for each n.

Hence the orbits of ϕ are all plain cycles;
it seems like a reasonable project to determine the lengths of these cycles.



There are 4 cycles of length 8.

Cycle Count/Length Table

	1	2	4	8	16	32
0	1					
1		1				
2		2				
3			2			
4			4			
5				4		
6				8		
7					8	
8					16	
9						16
10						32

$\rightarrow$: cycle length
$\downarrow$: word length
missing entries are 0

Consider a cycle

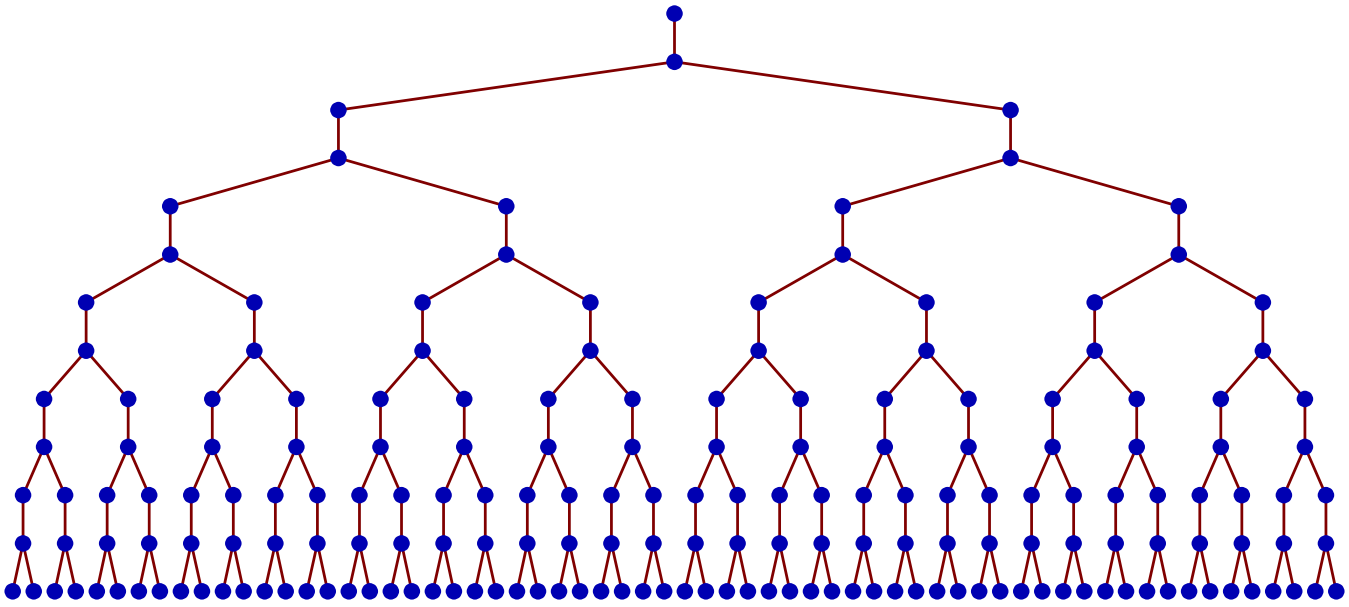
$$C = u_0, u_1, \ldots, u_{n-1}$$

Then either there are two cycles

$$u_0 0, u_1 b_1, \ldots, u_{n-1} b_{n-1} \qquad u_0 1, u_1 \overline{b_1}, \ldots, u_{n-1} \overline{b_{n-1}}$$

or there is a single cycle

$$u_0 0, u_1 b_1, \ldots, u_{n-1} b_{n-1}, u_0 1, u_1 \overline{b_1}, \ldots, u_{n-1} \overline{b_{n-1}}.$$



Conjecture

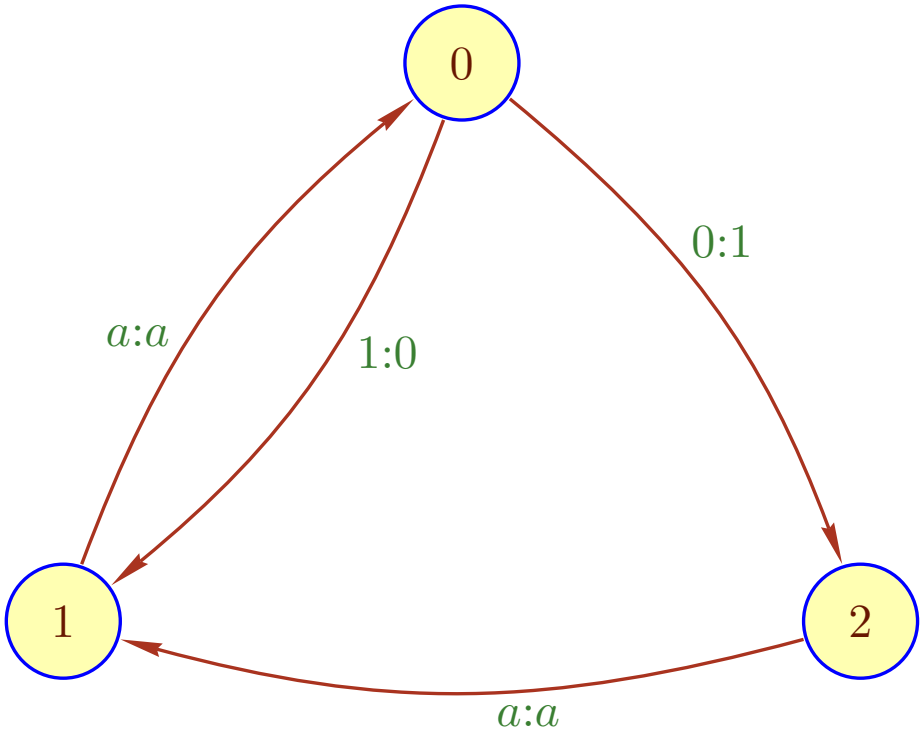
There are $2^{\lfloor k/2 \rfloor}$ cycles on words of length k.
The length of each cycle is $2^{\lceil k/2 \rceil}$.

Wild Idea:

Can we think of pebble flipping as some semigroup action?

We don't mean the obvious action on $\mathbb{N}$, $k \mapsto \phi^k(x)$, that works for all maps on $\Sigma^\star$ and is too general.

Instead, we would like some framework that is custom designed for ϕ and helps to understand its internal structure.



An invertible Mealy machine $\mathcal{A}$ that implements the pebble flipping operation.

$$\underline{0}\,(0x) = 1\,\underline{2}\,(x)$$
$$\underline{0}\,(1x) = 0\,\underline{1}\,(x)$$
$$\underline{1}\,(ax) = a\,\underline{0}\,(x)$$
$$\underline{2}\,(ax) = a\,\underline{1}\,(x)$$

So $\underline{0}$ is our flipping operation ϕ, but $\underline{1}$ and $\underline{2}$ explain how it works.

We might as well try to understand how the semigroup Sgrp$(\mathcal{A})$ acts on $\mathbf{2}^\star$.

Is it really easier to $S = \mathsf{Sgrp}(\mathcal{A})$ than just a narrow focus on iterated flipping?

Pierre Deligne commented on Alexandre Grothendieck's work:

> He would aim at finding and creating the home which was the problem's natural habitat.

The natural habitat for the flipping operation is the semigroup, basta.

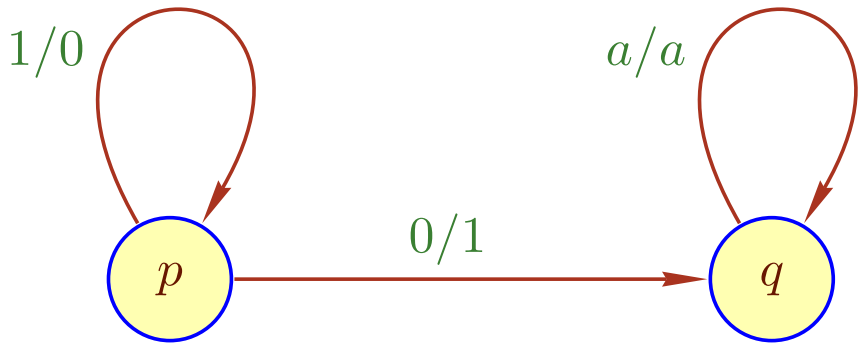
How hard is it to

- test membership in a cycle?
- compute $\phi^t(x)$?
- compute t such that $\phi^t(x) = y$?
- compute the least element of a cycle?

These are trivially primitive recursive, but we are looking for fast methods.

In particular, for words of length k, the algorithms should be polynomial in k.

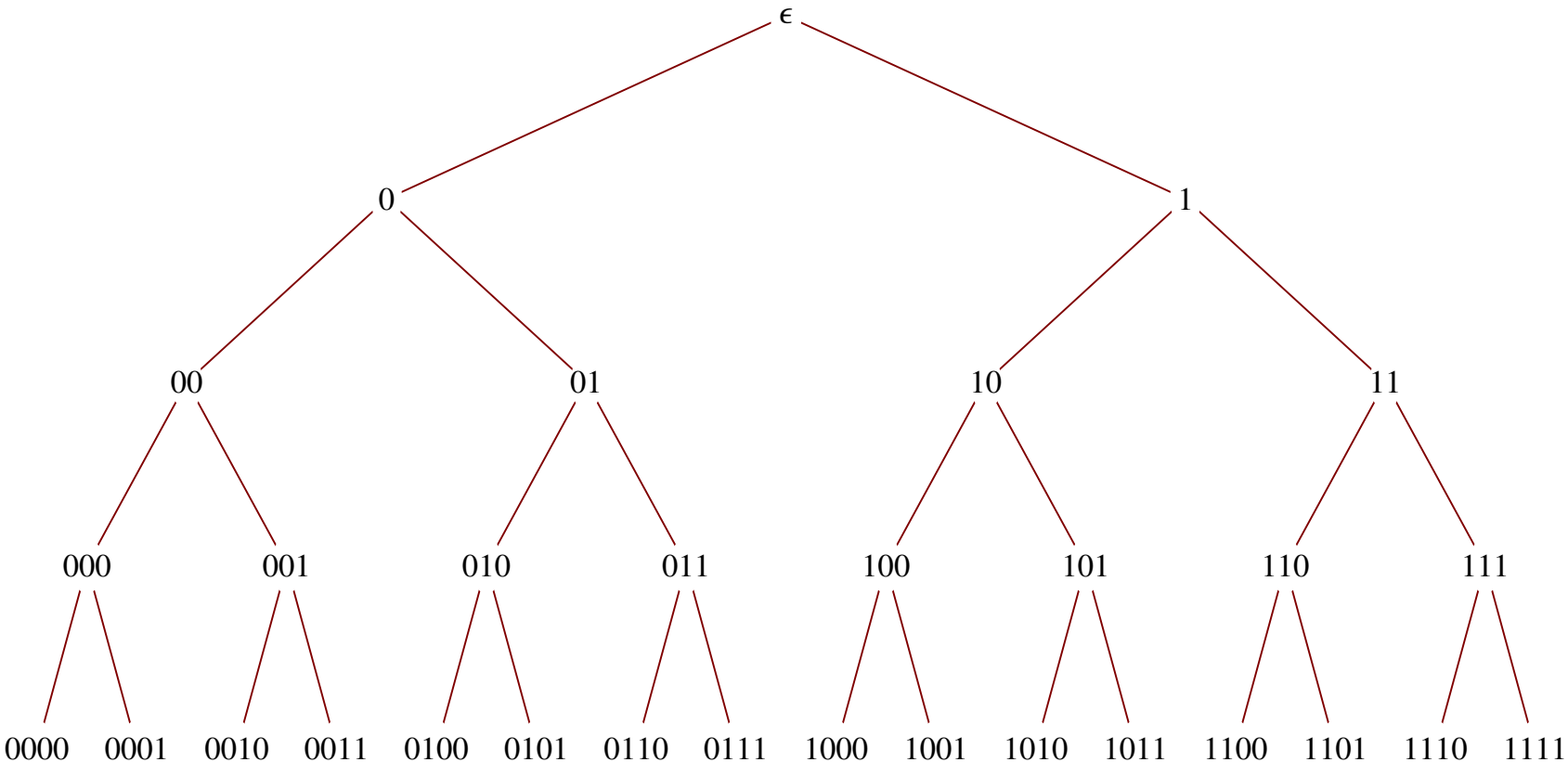
We already talked about the "adding machine."



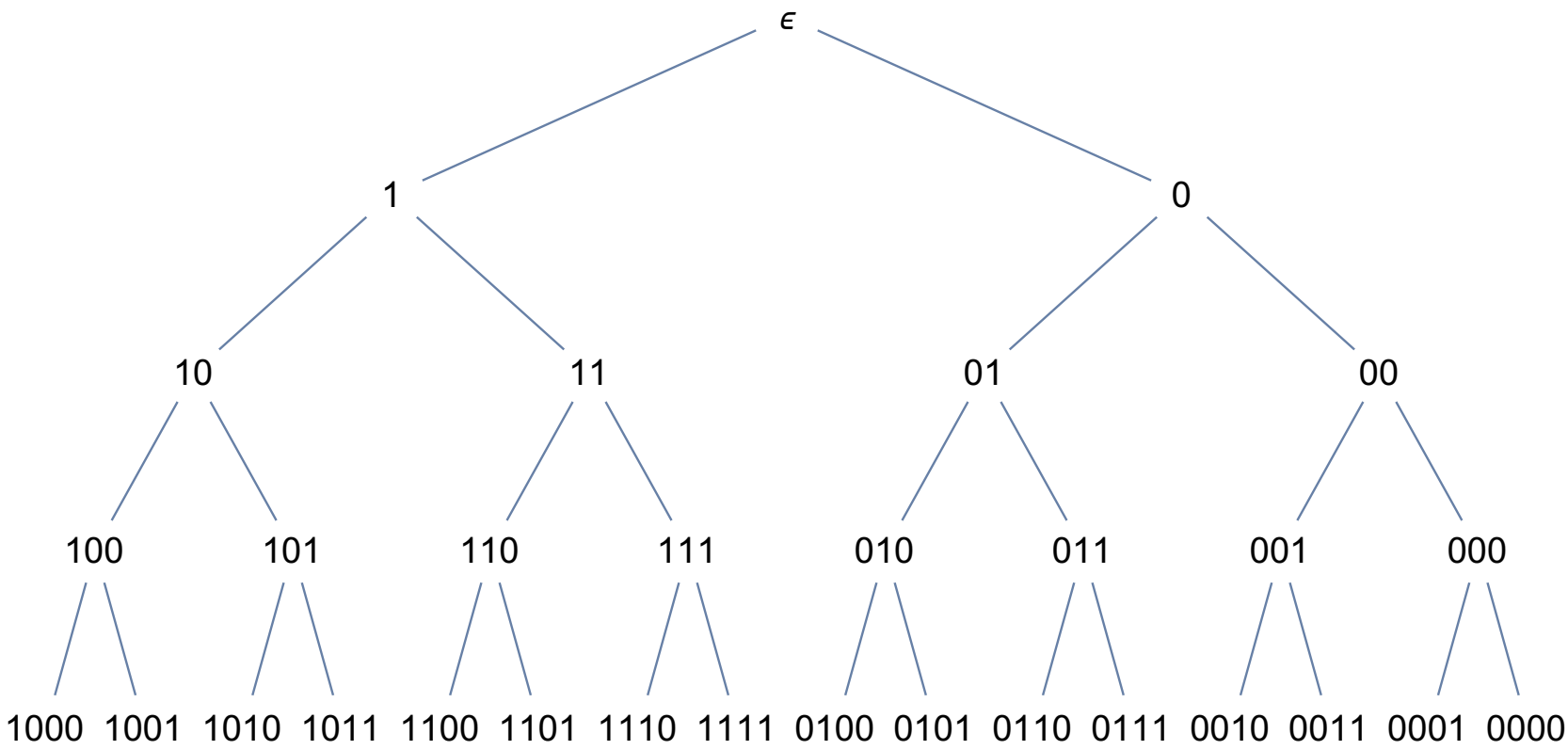
The semigroup generated by this machine is $\mathbb{N}$.

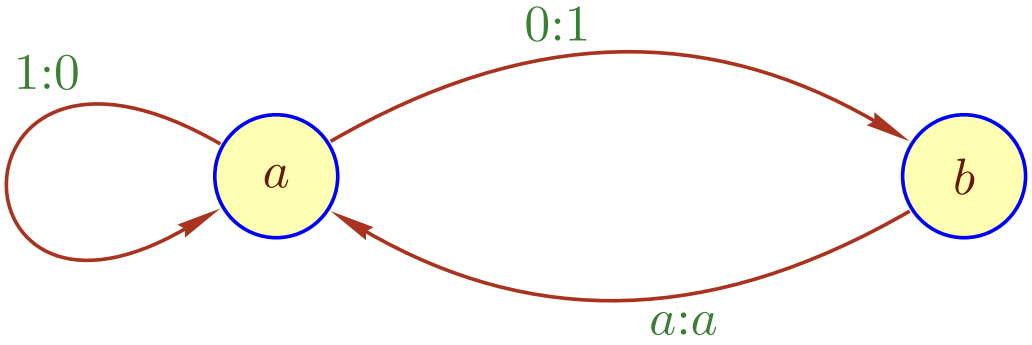
The group generated by this machine is $\mathbb{Z}$.

To get a group, we have to add the inverse of p, the predecessor function.


ϵ
0
1
00
01
10
11
000
001
010
011
100
101
110
111
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

After the Transduction





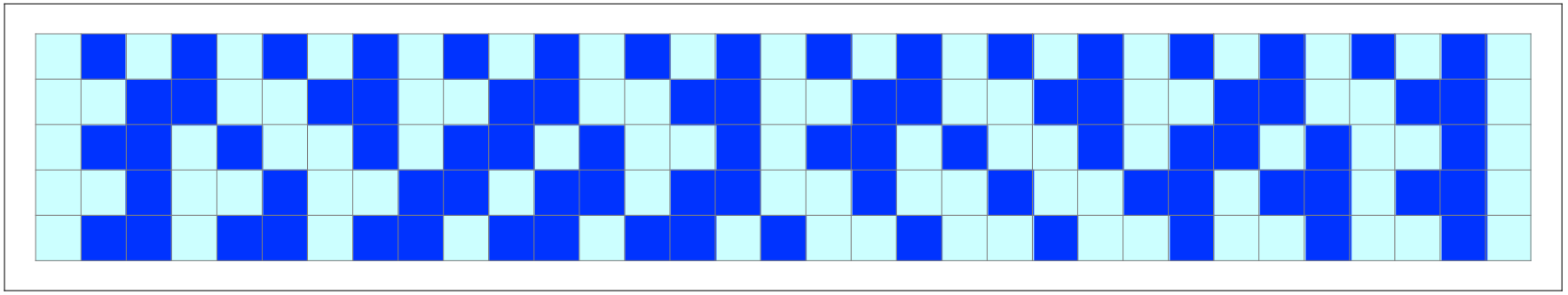
In recursive form:

$$\underline{a}\,(0x) = 1\,\underline{b}\,(x)$$
$$\underline{a}\,(1x) = 0\,\underline{a}\,(x)$$
$$\underline{b}\,(sx) = s\,\underline{a}\,(x)$$

This is a broken successor machine, instead of copying the rest, it copies only one bit and then goes back into increment mode.

What will the orbits look like?

Example



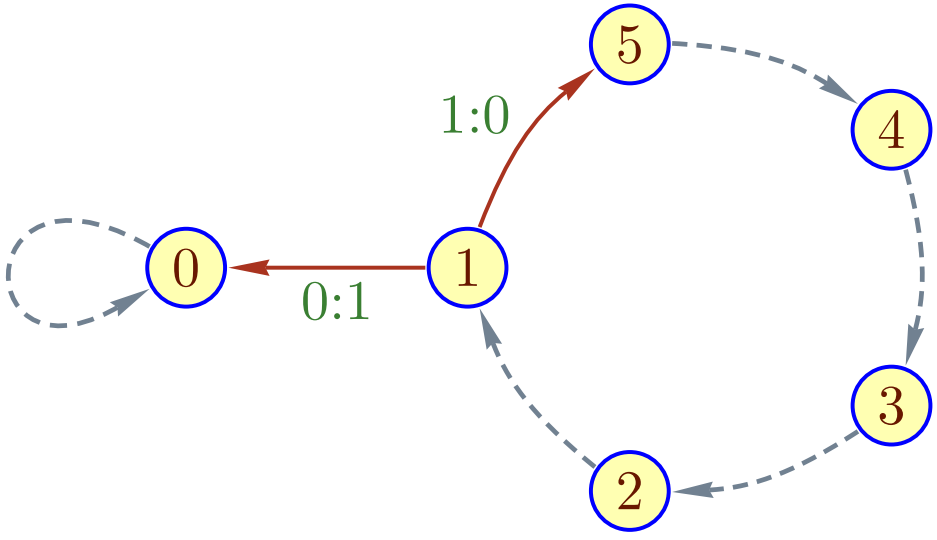
The full orbit of 00000. Length is 32.

This is yet another counter, albeit in weird number system.

Exercise

Show that all orbits of length k strings have length 2^k.

Sausage Automaton



This is the so-called sausage automaton $\mathcal{S}_5$.
Gray edges are copy.

Question: What semigroup/group is associated with this machine?

For simplicity, consider a word $x = z_1 z_2 \dots z_k$ where $z_i \in \mathbf{2}^5$.

Then $\underline{1}$ affects only the first bits of each block: $z_{11}, z_{21}, \dots, z_{k1}$. On these bits, it acts like the successor.

Similarly, $\underline{2}$ affects only the second bits of each block: $z_{12}, z_{22}, \dots, z_{k2}$, again acting like the successor.

And so on for the others.

Claim:

The sausage automaton generates the semigroup $\mathbb{N}^5$ and the group $\mathbb{Z}^5$.

Implement the Collatz function as efficiently as possible.

E.g., for $x = 2^k - 1$, $k = 10000, \ldots, 10010$ my code takes 4.25 seconds to produce

k	stop	up	down	width
10000	86278	48126	38152	15850
10001	86279	48126	38153	15853
10002	86280	48126	38154	15853
10003	86281	48126	38155	15856
10004	86282	48126	38156	15856
10005	86283	48126	38157	15858
10006	86284	48126	38158	15860
10007	86285	48126	38159	15864
10008	86286	48126	38160	15864
10009	86287	48126	38161	15864
10010	86288	48126	38162	15866

It is a bad idea to use some arbitrary precision integer arithmetic library to do this: the computation requires some specialized bit manipulations but not really arithmetic.

Write numbers in reverse binary, work from left to right.
Ponder deeply:

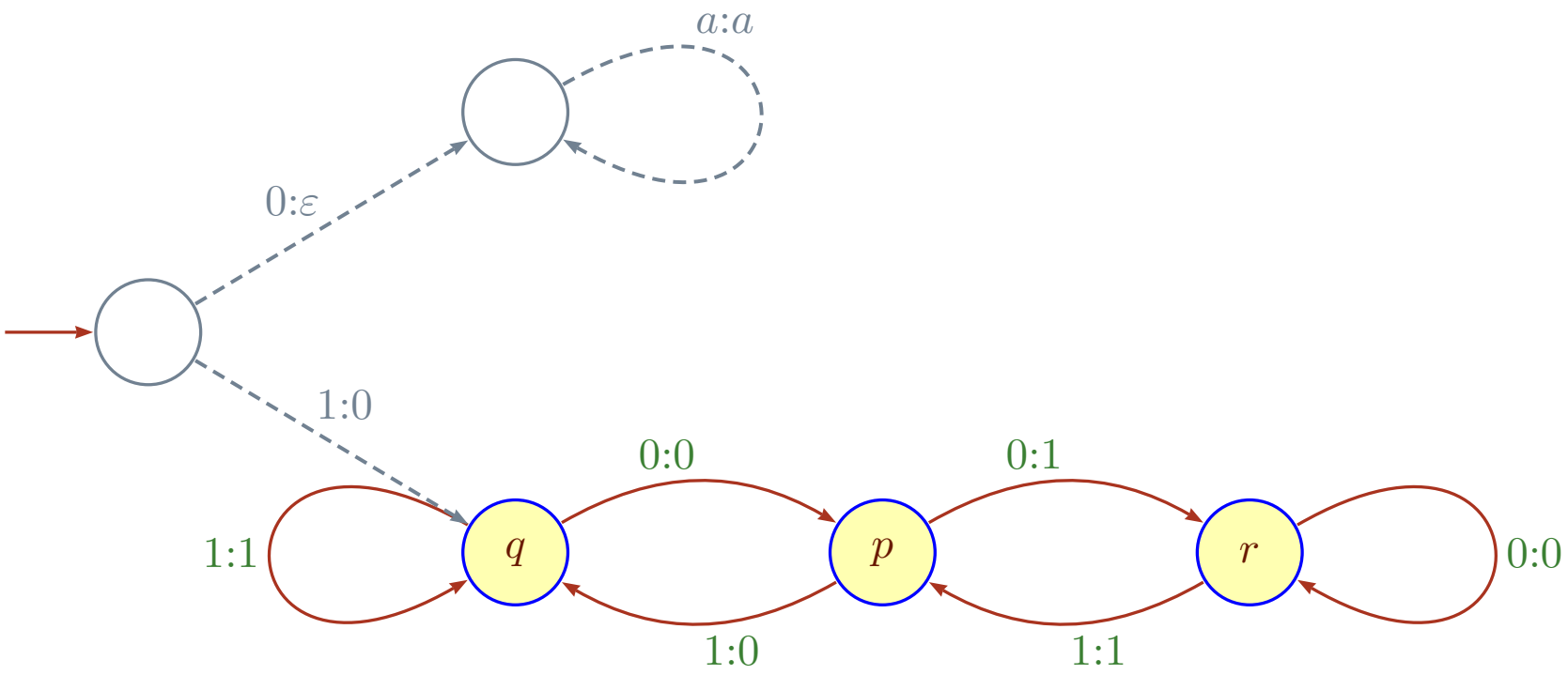
x	1	x_1	x_2	x_3	x_4	x_5	x_6	0	0	0	0	$\ldots$
$2x{+}1$	1	1	x_1	x_2	x_3	x_4	x_5	x_6	0	0	0	$\ldots$
$3x{+}1$	0	x_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	0	0	$\ldots$

The bits y_i involve the input bits as well as carries.

To compute with n-bit numbers:

- Allocate a bit array of length $5n$, write input on the left.
- Update by one sweep, left to right.
- Bits slowly migrate to the right; move back to the left if necessary.

5 is a magic number that works for 10000 bits: no left shifts are needed at all.



The upper part is not a Mealy machine.

But the 3 nodes at the bottom form an invertible Mealy automaton.

This transducer computes the value of the infamous Collatz function, assuming inputs x and are written in reverse binary and are padded to $x00$[†]

The upper part clearly divides by 2.

The lower part computes the maps

$$\underline{q} : n \mapsto 3n + 2$$
$$\underline{p} : n \mapsto 3n + 1$$
$$\underline{r} : n \mapsto 3n$$

Another indication that iteration of Mealy machines can produce very complicated results, even though the machines are basically trivial.

[†]Again, infinite strings are cleaner.