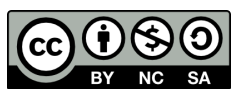
CDM

Automatic Structures

Klaus Sutner

Carnegie Mellon University

Spring 2025



Recall: Word Structures

We were interested in structures that can be completely described in terms of finite state machines.

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

- $A \subseteq \Sigma^\star$ 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 have only one feeble example of a "recognizable relation."

We showed that the following structure based on the invertible Mealy automaton $\mathcal{A}_2^3$

$$\mathcal{C} = \langle \mathbf{2}^\star; \underline{0}, \underline{1}, \underline{2} \rangle$$

describes a group, isomorphic to $\mathbb{Z} \times \mathbb{Z}$.

Issues to attend to:

- Mealy automata only produce relations (actually, functions) that are length-preserving. We need to generalize.
- An example unrelated to group theory would be reassuring. We will use logic and give a decision procedure for Presburger arithmetic (arithmetic without multiplication).

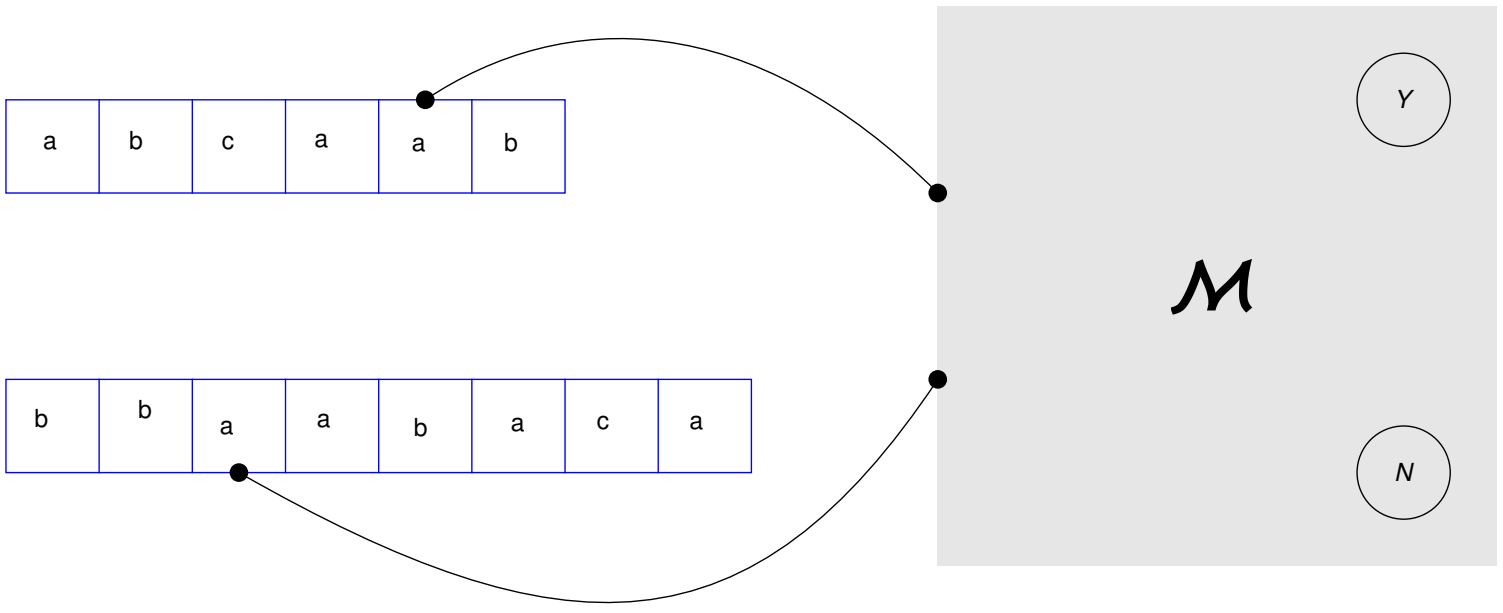
A fairly good intuitive way to think about handling relations in general is to modify our old finite state machines:

- Keep the finite state control.
- Allow 2 separate input tapes with separate read-only heads.

The read heads are still one-way, but they can move independently from each other; in particular, one head can get arbitrarily far ahead of another.

When both heads have consumed all of their input, acceptance depends on whether the machine is in a final state.

The relations obtained this way are called rational.

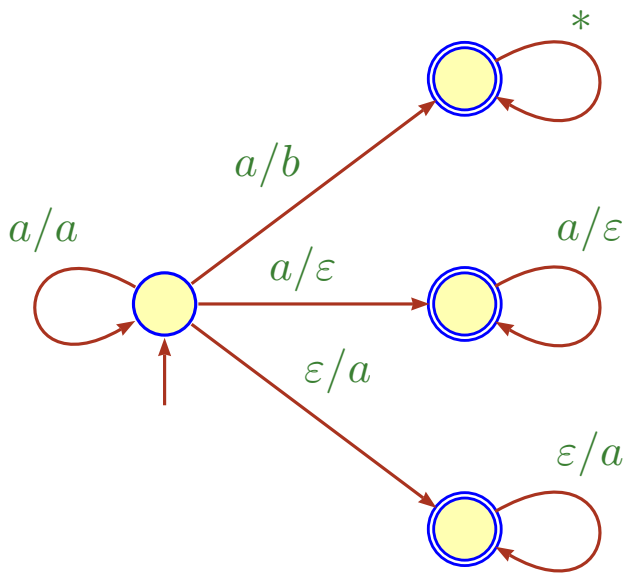

a
b
c
a
a
b
b
b
a
a
b
a
c
a
Y
M
N

Rational Examples

- The identity relation on $\Sigma^\star$.
- The unequal relation $x \neq y$.
- The prefix/factor/suffix relations.
  Note that the last two require nondeterministic guessing.
- Lexicographic order, length-lex order.
- Concatenation as a ternary relation $\{\, (x, y, z) \mid xy = z \,\}$.
- The addition relation $\{\, (x, y, z) \mid x + y = z,\ x, y, z \in \mathbf{2}^\star \,\}$.
  Here the arguments are written in reverse binary.

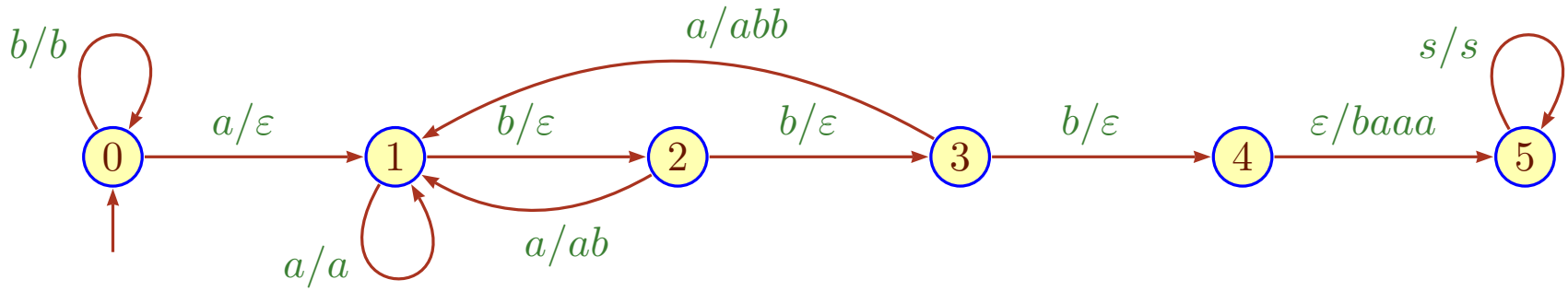
Un-Equal

Here is a machine whose behavior is the relation $x \neq y$.



In the diagram, a and b are supposed to range over Σ, and $a \neq b$.
$*$ means eternal bliss.

A transducer that (essentially) replaces the first occurrence of $abbb$ by $baaa$.



Exercise

Why does this transducer not quite work? Fix the problem.
Change the machine so that all occurrences are replaced.

Non-Rational Relations

- The reversal relation $x = y^{\mathsf{op}}$.
- The copy relation $\{\, (x, xx) \mid x \in \Sigma^\star \,\}$.
- The permutation relation $\{\, (x, \pi(x)) \mid x \in \Sigma^\star, \pi \text{ permutation} \,\}$
- The multiplication relation $\{\, (x, y, z) \mid x \cdot y = z, x, y, z \in \mathbf{2}^\star \,\}$.
  Here the arguments are written in reverse binary.

Less interesting: any non-recognizable language L is non-rational as a unary relation.

Rational relations are very interesting and obviously useful, but they are a bit too complicated for our purposes.

Here are some of the problems:

- Rational relations are not closed under complementation or intersection.
- Nondeterministic machines cannot be avoided in general.

The lack of Boolean closure means that we could not handle negations and conjunctions, a total disaster for a decision algorithm.

Workaround

We will home in on a small subclass of rational relations, so-called synchronous relations or automatic relations.

These are defined in a way that essentially preserves all the strong results we have wrto recognizable languages.

And, they are still expressive enough to produce interesting applications.

Inspired by Mealy machines one might try to make do with length-preserving relations only: $R(x, y)$ implies $|x| = |y|$.

x_1	x_2	$\ldots$	x_n
y_1	y_2	$\ldots$	y_n

In this case we could think of $R \subseteq (\Sigma \times \Sigma)^\star$, so our 2-track words are actually words over the product alphabet $\Sigma \times \Sigma$. The hope is that one can then use an ordinary 1-tape finite state machine to recognize them (see below for a theorem the justifies this hope).

Alas, strictly length-preserving relations are bit too restricted for our purposes. We can relax things a little by using a padding symbol #: $\Sigma_\# = \Sigma \cup \{\#\}$ where $\# \notin \Sigma$.

The alphabet for 2-track words is $\Delta_\# = \Sigma_\# \times \Sigma_\#$:

$x{:}y =$

x_1	x_2	$\ldots$	x_n	$\#$	$\ldots$	$\#$
y_1	y_2	$\ldots$	y_n	y_{n+1}	$\ldots$	y_m

This is called the convolution[†] of x and y and is written $x{:}y$.

[†] Horrible terminology, this has nothing to do with the integral transform by the same name.

Note that we are not using all of $\Delta_{\#}^{\star}$ but only the recognizable subset coming from convolutions. In other words, $\#$ can only appear as a suffix, and in at most one track. For example,

a	$\#$	b	$\#$
a	b	a	a

a	b	b	$\#$	$\#$
a	b	a	b	$\#$

are not allowed.

It is easy to see that the collection of all convolutions forms a recognizable language over $\Delta_{\#}$.

As always, a similar approach clearly works for $k$ary relations.

Here is an idea going back to Büchi and Elgot in 1965.

Definition

A relation $R \subseteq \Sigma^\star \times \Sigma^\star$ is synchronous or automatic if there is a finite state machine $\mathcal{A}$ over $\Delta_\#$ such that

$$\mathcal{L}(\mathcal{A}) = \{\, x{:}y \mid R(x,y) \,\} \subseteq \Delta_\#^\star$$

k-ary relations are treated similarly.

Note that this machine $\mathcal{A}$ is just a language recognizer, not a transducer: since we pad, we can read one symbol in each track at each step.

In a sense, synchronous relations are the most basic examples of transductions that are not entirely trivial.

By contrast, one sometimes refers to arbitrary rational relations as asynchronous.

- Equality and inequality are synchronous.
- Lexicographic order is synchronous.
- The prefix-relation is synchronous.
- The ternary addition relation is synchronous.

- The suffix-relation is not synchronous.
- The relations "x is a factor of y" and "x is a (scattered) subword of y" are not synchronous.

For any accepting computation π on $x{:}y \in L$, define its lag to be the maximum distance between the two heads. The lag of $x{:}y$ is the minimum lag over all computations π.

If there is a universal bound B on the lag of all $x{:}y \in L$ one can construct a synchronous transducer that accepts the same language. Essentially, keep track of a word $d \in \Sigma^{\leq B}$ representing the part of the input that the leading head has already read. If the heads are in the same position, $d = \varepsilon$.

E.g., if the leading head moves forward and reads an a we update d to da if $|d| < B$, and crash otherwise. If the trailing head reads a we replace d by $\mathrm{tail}(d)$ if $d \neq \varepsilon$, otherwise we set $d = a$.

If the lag is unbounded, then the transduction cannot be length-preserving.

Theorem (Elgot, Mezei 1965)

Any length-preserving rational relation is already synchronous.

It is a good exercise to produce a constructive proof that builds the synchronous machine as efficiently as possible (in general there will be an exponential blow-up in size).

Claim

Given two k-ary synchronous relations ρ and σ on $\Sigma^\star$, the following relations are also synchronous:

$$\rho \sqcup \sigma \qquad \rho \sqcap \sigma \qquad \rho - \sigma$$

The proof is very similar to the argument for recognizable languages: one can effectively construct the corresponding automata using the standard product machine idea.

This is a hugely important difference between general rational relations and synchronous relations: the latter do form an effective Boolean algebra, but we have already seen that the former are not closed under intersection (nor complement).

On the upside, synchronous relations are closed under composition.

Suppose we have two binary relations $\rho \subseteq \Sigma^\star \times \Gamma^\star$ and $\sigma \subseteq \Gamma^\star \times \Delta^\star$.

Theorem

If both ρ and σ are synchronous relations, then so is their composition $\rho \circ \sigma$.

Exercise

Prove the theorem.

Here is another important closure property. Suppose ρ is a k-ary relation on words. We define the projection of ρ to be

$$\rho'(x_2, \ldots, x_k) \iff \exists\, z\, \rho(z, x_2, \ldots, x_k)$$

Lemma

Whenever ρ is synchronous, so is its projection ρ'.

Proof.

The proof is nearly trivial: simply erase the first track in the k-track alphabet:

$$p \xrightarrow{a_1:a_2:\ldots:a_k} q \quad \rightsquigarrow \quad p \xrightarrow{a_2:\ldots:a_k} q$$

Done! □

The same result holds for rational relations in general, with the same proof.

From the machine perspective, projections are easily linear time.

Alas, erasing a track will usually turn a deterministic machine into a nondeterministic one. If we need to determinize later (e.g., to handle negation) this may be a source of exponential blow-up.

Recall the tree automorphisms defined by reversible Mealy machines. Their study involves composition, iteration and group theory; overall, it is fairly difficult to get good results.

Here is a dream: could we build a decision algorithm for simple assertions that exploits our algorithmic machinery for finite state machines?

Since relational structures are easier to handle, we think of a function $\Sigma^\star \to \Sigma^\star$ as a binary relation $\rightarrowtriangle$. We are interested in statements about

$$\mathfrak{C} = \langle \mathbf{2}^\star; \rightarrowtriangle \rangle$$

Examples

$$\forall\, x, y, z\, (x \rightarrowtriangle y \wedge x \rightarrowtriangle z \Rightarrow y = z)$$

$$\forall\, x, y, z\, (x \rightarrowtriangle y \wedge z \rightarrowtriangle y \Rightarrow x = z)$$

$$\forall\, x\, \exists\, y\, (y \rightarrowtriangle x)$$

$$\exists\, x, y, z\, (x \rightarrowtriangle y \wedge y \rightarrowtriangle z \wedge z \rightarrowtriangle x \wedge x \neq y)$$

$$\forall\, x\, \exists\, y, z\, \Big((y \rightarrowtriangle x \wedge z \rightarrowtriangle x \wedge y \neq z) \wedge \forall\, u\, (u \rightarrowtriangle x \Rightarrow u = y \vee u = z)\Big)$$

What is the meaning of these formulae?

Suppose we have a synchronous relation $\rightarrowtriangle$ and some FO sentence Φ in the language $\mathcal{L}(\rightarrowtriangle)$.

We want an algorithm to test whether Φ holds over $\mathfrak{C} = \langle \mathbf{2}^\star; \rightarrowtriangle \rangle$.

For simplicity, we may assume that quantifiers use distinct variables and that the formula is in prenex-normal-form[†], say:

$$\Phi = \exists\, x_1 \,\forall\, x_2 \,\forall\, x_3 \,\ldots \exists\, x_k \;\varphi(x_1, \ldots, x_k)$$

The matrix $\varphi(x_1, \ldots, x_k)$ is quantifier-free, so all we have there is Boolean combinations of atomic formulae.

[†]This is actually a bad idea for efficiency reasons, but it simplifies the discussion of the basic algorithm.

In our case, there are only two possible atomic cases:

- $x_i = x_j$
- $x_i \rightarrowtriangle x_j$

Given an assignment for x_i and x_j (i.e., actual strings) we can easily test these atomic formulae using two synchronous transducers $\mathcal{A}_{=}$ and $\mathcal{A}_{\rightarrowtriangle}$.

So $\varphi(x_1, \ldots, x_k)$ defines a k-ary relation over $\mathbf{2}^\star$, constructed from $\rightarrowtriangle$ and $=$ using Boolean operators.

The next step is to build a k-track synchronous machine that recognizes the k-ary relation on $\mathbf{2}^\star$ defined by the quantifier-free formula

$$\varphi(x_1, x_2, \ldots, x_k)$$

We can do this by induction on the subformulae of φ.

The atomic pieces read from two appropriate tracks and check $\rightarrowtriangle$ or $=$.

Note that there is a bureaucratic problem: the atomic machines are 2-track, but the machine for the matrix is usually k-track for some $k > 2$.

More precisely, use superscripts to indicate the number of tracks of a machine as in $\mathcal{A}^{(2)}_{\rightarrow\triangleright}$ and $\mathcal{A}^{(2)}_{=}$.

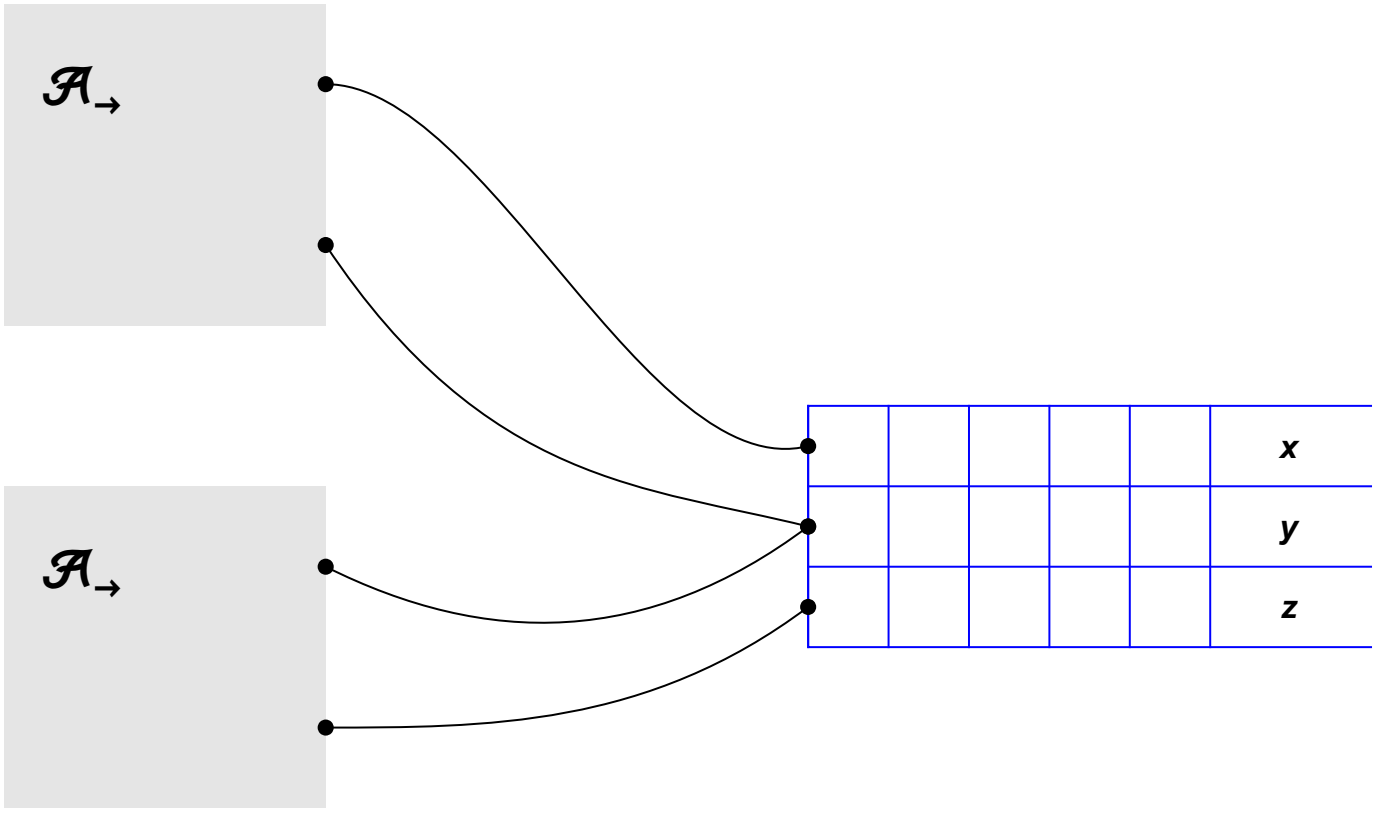
Let $m \leq n$. We need an embedding operation

$$\mathsf{emb}^{(n)}_{\boldsymbol{t}} : m\text{-track} \longrightarrow n\text{-track}$$

where $\boldsymbol{t} = t_1, \ldots, t_m$, $t_i \in [n]$, all distinct.

So $\mathsf{emb}^{(n)}_{\boldsymbol{t}}(\mathcal{A}^{(m)}) = \mathcal{B}^{(n)}$ means that track i of $\mathcal{A}^{(m)}$ is identified with track t_i in $\mathcal{B}^{(n)}$. The other tracks are free (all possible transitions). This does not affect the state set, but it can cause potentially very large alphabets and, correspondingly, large numbers of transitions in the embedded automaton[†].

[†]One of the reasons why state complexity alone is not really a good measure of the size of an automaton, one needs to add the number of transitions.



A product machine to check $x \twoheadrightarrow y \wedge y \twoheadrightarrow z$.

Slightly more generally, we can combine any two 2-track machines $\mathcal{A}_i^{(2)}$ and construct the product machine

$$\mathcal{B} = \mathsf{emb}_{1,2}^{(3)}\left(\mathcal{A}_1^{(2)}\right) \times \mathsf{emb}_{2,3}^{(3)}\left(\mathcal{A}_2^{(2)}\right)$$

$\mathcal{B}$ checks for strings $x{:}y{:}z$ such that $\mathcal{A}_1$ recognizes $x{:}y$ and $\mathcal{A}_2$ recognizes $y{:}z$.

An so on for any number of embedded automata. Note that the product machine construction can produce uncomfortably large state sets.

Suppose $\varphi = \psi_1 \wedge \psi_2$ with corresponding machines $\mathcal{A}_{\psi_1}$ and $\mathcal{A}_{\psi_2}$. We can use a product machine construction to get $\mathcal{A}_\varphi$.

Disjunctions are even easier: just take the disjoint union, there is really no way to get around nondeterminism here.

But negations are potentially expensive: we have to determinize first.

At any rate, we wind up with a composite automaton $\mathcal{A}_\varphi$ that recognizes the relation defined by the matrix:

$$\mathcal{L}(\mathcal{A}_\varphi) = \{\, u_1{:}u_2{:}\ldots{:}u_k \mid \mathcal{C} \models \varphi(u_1, u_2, \ldots, u_k) \,\}$$

There is a natural dual to embeddings: projections.

Let $m \leq n$. We have a projection operation

$$\mathsf{prj}_{\boldsymbol{t}}^{(n)} : n\text{-track} \longrightarrow m\text{-track}$$

where $\boldsymbol{t} = t_1, \ldots, t_{n'}$, $n' \leq n$, $t_i \in [n]$, all distinct, $m = n - n'$.

So $\mathsf{prj}_{\boldsymbol{t}}^{(n)}(\mathcal{A}^{(n)}) = \mathcal{B}^{(m)}$ means that, for all transitions in $\mathcal{A}^{(n)}$, the tracks t_i of the transition labels have been erased, producing $\mathcal{B}^m$. The state set is unaffected.

It is fine to have $n = n'$, in which case it is understood that we are left with an unlabeled digraph (with special initial and final nodes).

It remains to deal with all the quantifiers in the prefix of Φ. First consider a single existential quantifier, say

$$\exists\, x\, \psi(x)$$

We have a machine $\mathcal{A}_{\psi}^{(n)}$ that has a track t for variable x.

Simply erase the x-track from all the transition labels.

In other words, $\mathrm{prj}_t^{(n)}(\mathcal{A}^{(n)})$ corresponds exactly to existential quantification over variable x.

Alas, for universal quantifiers we have to use the old equivalence $\forall\ \equiv \neg\exists\neg$.

This is all permissible, since projections and negations do not disturb automaticity–though they may increase the machine size substantially.

Recall the machine checking $x \rightarrowtriangle y \wedge y \rightarrowtriangle z$.

$$\mathcal{B} = \mathsf{emb}^{(3)}_{1,2}\left(\mathcal{A}_{\rightarrowtriangle}\right) \times \mathsf{emb}^{(3)}_{2,3}\left(\mathcal{A}_{\rightarrowtriangle}\right)$$

Projecting away the y-track

$$\mathcal{B}' = \mathsf{prj}^{(n)}_{2}(\mathcal{B})$$

produces a machine that recognizes $x{:}z$ such that $\exists\, y\, (x \rightarrowtriangle y \wedge y \rightarrowtriangle z)$.

Similarly we can handle $f^k(x) = z$ for any fixed value of k. However, the size of the machine is only bounded by m^k.

In the process of removing quantifiers, we lose one track at each step and get intermediate machines $\mathcal{B}_{\varphi,\ell}$

$$\mathcal{L}(\mathcal{B}_{\varphi,\ell}) = \{\, u_1{:}u_2{:}\ldots{:}u_\ell \mid \mathcal{C} \models \varphi_\ell(u_1, u_2, \ldots, u_\ell) \,\}$$

for $\ell \leq k$. In the end $\ell = 0$, and we are left with an unlabeled transition system $\mathcal{B}_{\varphi,0}$. This transition system has a path from I to F iff the original sentence Φ is valid.

So the final test is nearly trivial (DFS anyone?), but it does take a bit of work to construct the right machine.

Why does this all work, fundamentally? It is all a direct consequence of various closure properties:

$\vee$	union
$\wedge$	intersection
$\neg$	complement
$\exists$	homomorphism
emb	inverse homomorphism

Needless to say, all the closures are effective: we have algorithms to construct all the corresponding machines.

- $\vee$ and $\exists$ are linear if we allow nondeterminism.
- $\wedge$ is at most quadratic via a product machine construction.
- $\neg$ is potentially exponential since we need to determinize first.
- $\forall$ well ...

So this is a bit disappointing: we may run out of computational steam even when the formula is not terribly large. Universal quantifiers, in particular, can be a major problem.

A huge amount of work has gone into streamlining this and similar algorithms to deal with instances that are of practical relevance.

Example: 3-Cycles

Let's figure out the details on how to determine the existence of a 3-cycle in $\mathcal{C}$. The obvious formula to use is this:

$$\Phi \equiv \exists\, x, y, z\, (x \rightarrowtriangle y \wedge y \rightarrowtriangle z \wedge z \rightarrowtriangle x \wedge x \neq y \wedge x \neq z \wedge y \neq z)$$

The first part ensures that there is a cycle, and the second part prevents the cycle from being shorter than 3.

Perfectly correct, but note the following. Suppose the basic machine $\mathcal{A}_{\rightarrowtriangle}$ that checks $\rightarrowtriangle$ has m states. Then the first part of the formula produces a machine of possibly m^3 states. The non-equal part blows things up further to at least $8\, m^3$ states.

We could replace Φ by any equivalent formula, which would be usefully if we could find a smaller formula. It seems hard to get around the m^3 part, checking for each inequality doubles the size of the machine, so we get something 8 times larger than the machine for the raw 3-cycle. It is better to realize that since $\rightarrowtriangle$ is functional, the last formula is equivalent to

$$\exists\, x, y, z\, (x \rightarrowtriangle y \wedge y \rightarrowtriangle z \wedge z \rightarrowtriangle x \wedge x \neq y)$$

Exercise

Figure out how to deal with k-cycles for arbitrary k.

So, based on the better formula, we use the 3-track alphabet $\mathbf{2}^3 = \mathbf{2} \times \mathbf{2} \times \mathbf{2}$ plus padding to recognize

$$\{\, u{:}v{:}w \mid u \rightarrowtriangle v \rightarrowtriangle w \rightarrowtriangle u \wedge u \neq v \,\}$$

Let $\mathcal{A}_{i,j} = \mathsf{emb}^{(3)}_{i,j}\left(\mathcal{A}^{(2)}_{\rightarrowtriangle}\right)$. Also, let $\mathcal{D}^{(2)}_{\neq}$ be the machine that checks for inequality and $\mathcal{D} = \mathsf{emb}^{(3)}_{1,2}\left(\mathcal{D}^{(2)}_{\neq}\right)$.

We can now concoct a 3-track product machine for the conjunctions:

$$\mathcal{B} = \mathcal{A}_{1,2} \times \mathcal{A}_{2,3} \times \mathcal{A}_{3,1} \times \mathcal{D}$$

where $\mathcal{A}_{\rightarrowtriangle,i,j}$ tests if the word in track i evolves to the word in track j.

So we get a machine $\mathcal{B}$ that is roughly cubic in the size of $\mathcal{A}_{\rightarrow\triangleright}$ (disregarding possible savings for accessibility).

Once $\mathcal{B}_3$ is built, we erase all the labels and are left with a digraph (since φ has no universal quantifiers there is no problem with negation).

This digraph has a path from an initial state to a final state if, and only if, there is a 3-cycle under $\rightarrow\triangleright$.

Note, though, how the machines grow if we want to test for longer cycles: the size of $\mathcal{B}_k$ is bounded only by m^k, where m is the size of $\mathcal{A}_{\rightarrow\triangleright}$, so this will not work for long cycles. And, we need several products with $\mathcal{D}_{i,j}$, each at least doubling the size of the product.

Example: Elementary Cellular Automata

A elementary cellular automaton (ECA) is a Boolean function $\rho : \mathbf{2}^3 \to \mathbf{2}$.

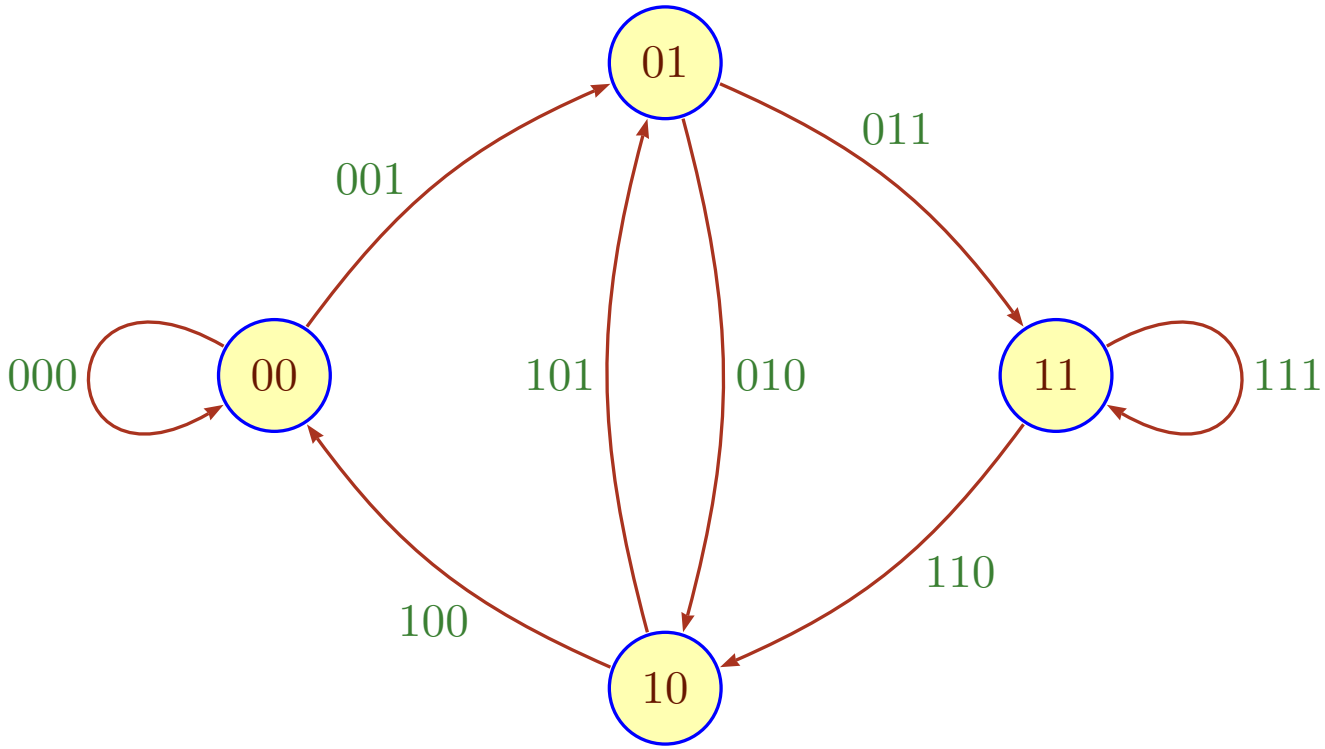
We can extend ρ to a map $\widehat{\rho} : \mathbf{2}^{\mathbb{Z}} \to \mathbf{2}^{\mathbb{Z}}$ by chopping $X \in \mathbf{2}^{\mathbb{Z}}$ into overlapping blocks of 3 bits, and applying ρ pointwise.

$$\widehat{\rho}(X)(i) = \rho(X_{i-1}, X_i, X_{i+1})$$

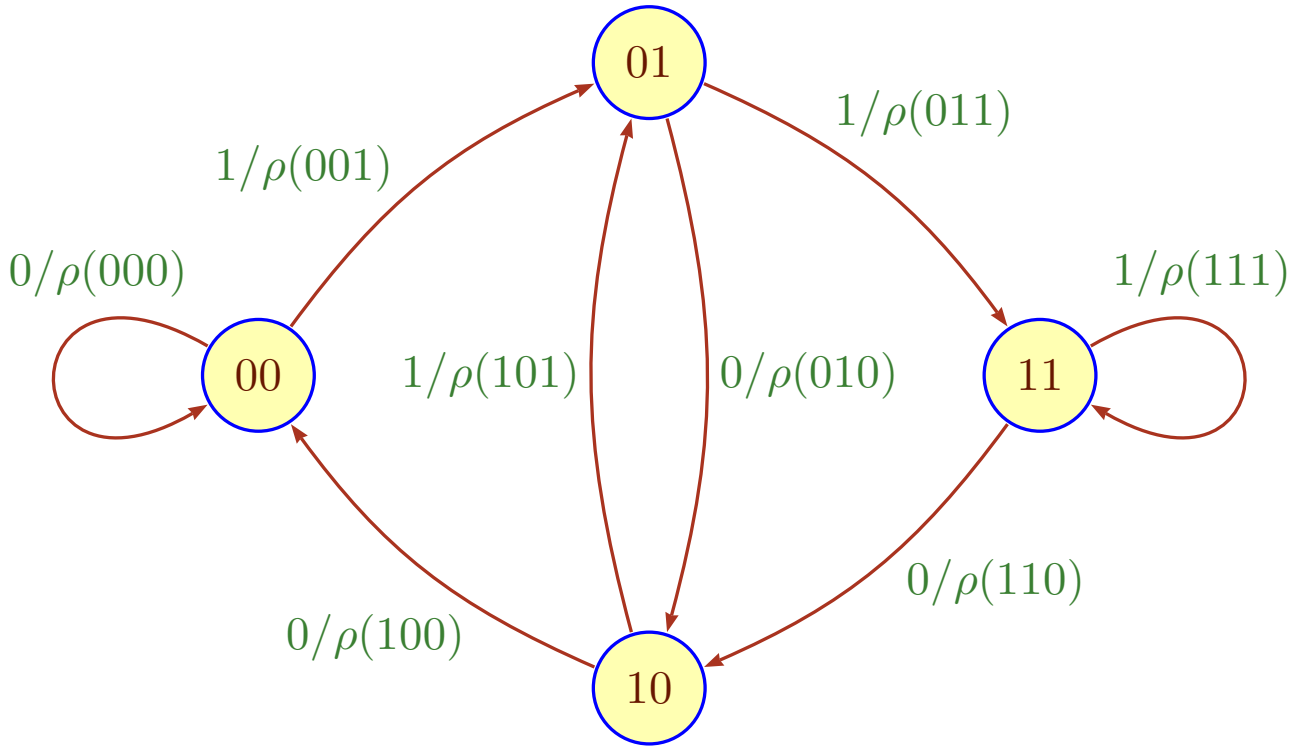
The two maps are called the local map and global map, respectively. Don't worry about the biinfinite words, we'll get back to finite in a moment.

What would the basic one-step automaton $\mathcal{A}_{\to}$ for an ECA look like?

First, an automaton that corresponds to sliding a window of length 2 across the configuration. The states will naturally be $\mathbf{2}^2$, and the edges corresponds to just having seen 3 bits in a row.



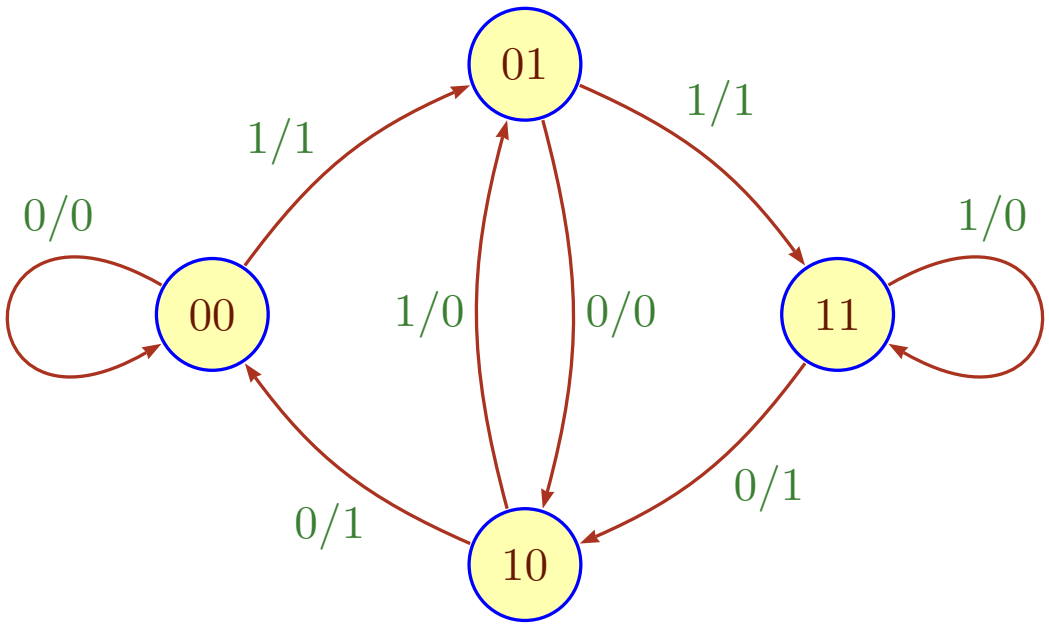
Each configuration in $\mathbf{2}^{\mathbb{Z}}$ corresponds to exactly one biinfinite path in this automaton. And every biinfinite path corresponds to a configuration (at least if we are a bit relaxed about where the origin is).



If we replace the edge labels xyz by $xyz/\rho(xyz)$, where ρ is the local map, we get a transducer that corresponds to the global map. All states are initial and final, we are interested in biinfinite runs.

Example: Rule 90

The ECA $\rho(x, y, z) = x \oplus z$ (the 8 values are 90 written in binary).



This happens to be an invertible Mealy automaton, but that is just coincidence; arbitrary ECA produce more complicated machines.

Boundary Conditions

Computationally, we need to deal with finite configurations rather than biinfinite ones. To extend ρ to finite binary words we need to deal with the endpoints: a priori they have no left/right neighbors.

- Cyclic boundary conditions: assume the configuration wraps around.
- Fixed boundary conditions: assume there are two phantom bits 0 pre/appended.

So for $\boldsymbol{x} = x_1 x_2 \dots x_n$ we apply the local map to n many 3-blocks:

$$\begin{array}{ll} \text{CBC} & x_n x_1 x_2 \quad x_1 x_2 x_3 \quad \dots \quad x_{n-1} x_n x_1 \\ \text{FBC} & 0\, x_1 x_2 \quad x_1 x_2 x_3 \quad \dots \quad x_{n-1} x_n 0 \end{array}$$

We need to modify the transducer for $\mathbf{2}^{\mathbb{Z}}$ to work for plain $\mathbf{2}^{n}$. Say, we use fixed boundary conditions. The central problem is this: we are scanning two words

$$u{:}v = \begin{array}{|c|c|c|c|} \hline u_1 & u_2 & \ldots & u_n \\ \hline v_1 & v_2 & \ldots & v_n \\ \hline \end{array}$$

But a synchronous transducer must read the letters in pairs, both read heads move in lockstep.

We need to check whether $v_1 = \rho(0, u_1, u_2)$, and we do not know u_2 after scanning just the first bit pair.

It seems that some kind of look-ahead is required (memory versus anticipation), but synchronous automata don't do look-ahead, they live in the here-and-now. Looks like we are sunk.

If we drop the synchronicity condition, there is no problem: it easy to see that $\multimap$ is rational. And $\multimap$ is clearly length-preserving.

But remember the theorem by Elgot and Mezei:

> Rational and length-preserving implies synchronous.

So our relation must be synchronous. Of course, that's not enough: we need to be able to construct the right transducer, not just wax poetically about its existence.

Exercise

Show that $\multimap$ is rational.

Nondeterminism saves the day: we can guess what x_2 is and then verify in the next step.

Automaton $\mathcal{A}_{\rightarrow}$ uses state set $Q = \{\bot, \top\} \cup \mathbf{2}^3$.

$\bot$ is the initial state, $\top$ the final state and the transitions are given by

$$\bot \xrightarrow{a/e} 0ab \qquad e = \rho(0, a, b)$$

$$abc \xrightarrow{c/e} bcd \qquad e = \rho(b, c, d)$$

$$abc \xrightarrow{c/e} \top \qquad e = \rho(b, c, 0)$$

So, this is more complicated than the plain de Bruijn transducer for $\mathbf{2}^{\mathbb{Z}}$.

input	state	condition
$-$	$\bot$	$-$
$u_1{:}v_1$	$0\,u_1u_2$	$v_1 = \rho(0u_1u_2)$
$u_2{:}v_2$	$u_1u_2u_3$	$v_2 = \rho(u_1u_2u_3)$
$u_3{:}v_3$	$u_2u_3u_4$	$v_3 = \rho(u_3u_3u_4)$
	$\vdots$	
$u_{n-1}{:}v_{n-1}$	$u_{n-2}u_{n-1}u_n$	$v_n = \rho(u_{n-1}u_n0)$
$u_n{:}v_n$	$\top$	$-$

A successful computation on input $u_1u_2\ldots u_n{:}v_1v_2\ldots v_n$.

Define a 3-track machine that checks whether x and y both evolve to z; then project away the z-track.

$$\mathcal{A} = \mathsf{prj}_3^{(3)}\left(\mathsf{emb}_{1,3}^{(3)}\left(\mathcal{A}_{\twoheadrightarrow}\right) \times \mathsf{emb}_{2,3}^{(3)}\left(\mathcal{A}_{\twoheadrightarrow}\right)\right)$$

Then

$$\mathcal{L}(\mathcal{A}) = \{\, x{:}y \mid \widehat{\rho}(x) = \widehat{\rho}(y) \,\}$$

So we only need to check

$$\mathcal{L}(\mathcal{A} \times \mathcal{A}_{\neq}) = \emptyset$$

to verify that the global map $\widehat{\rho}$ is injective.

State-explosion is a major issues with our approach, it may well happen that some of the (intermediate) machines are so large that they cannot be handled.

One way of keeping the machines small is to rewrite the formula under consideration into an equivalent one that produces smaller machines. Typical example: checking for 3-cycles. One also should avoid prenex-normal-form like the plague and try to handle projections early.

If the outermost block of quantifiers is universal, the last check can be more naturally phrased in terms of Universality rather than Emptiness. In this case one should try to use Universality testing algorithms without complementation (e.g., the antichain method that avoids direct determinization).