# Algebraic models of computation

Sarah Rees,
University of Newcastle, UK

Braunschweig, 23rd May 2013

# 0. Introduction

**Qn:** Why would we want to study algebraic models of computation?

**Ans:** To interpret computability and complexity of computation in a mathematical framework.

**Qn:** What in particular do I want to model?

**Ans:** Really I'm searching for algebraic models of quantum computation, particularly at a low level (corresponding to the classical models of finite state automata, pushdown automata etc.) But there's a lot we need to understand first about modelling classical computation. So I'll say very little about quantum computation today.

**My collaborators?** Andrew Duncan, Nick Loughlin, colleagues Dritschel, Vdovina, White + helpful input from Kambites, Lawson, Gilman.

## Classical vs quantum computation

The states of a **classical computation** can be viewed as binary strings. So we can view a classical computation as a sequence of such strings, and semigroups and monoids, in their actions on these sets of states, provide a natural algebraic framework to study classical computation.

But the states of a **quantum computer** are complex linear combinations of binary strings, and so belong to a Hilbert space. Transition between these states is through unitary transformations, is always reversible. $C^*$-algebras are the natural objects to act on Hilbert spaces, and so are excellent candidates to model quantum computation algebraically.

# Polycyclic monoids and Cuntz-Krieger algebras

We are particularly interested in the **Cuntz-Krieger algebras**, which can be constructed out of **polycylic monoids**; those monoids are fundamental to the study of pushdown automata (rather basic machines for classical computation).

Maybe these algebras describe a type of quantum computation occupying a similar position in the hierarchy of quantum computation to that of pushdown automata in the classical hierarchy?

Maybe they do. But before we can answer that question we need to understand better than we currently do how monoids such as the polycylic monoids can be used to model classical computation. That is what this presentation is about. It describes an ongoing exploration.

## Plan for today

1. Using machines: I'll set up some notation for Turing's model for computation, and the simplest machines in the Chomsky hierarchy, finite state automata FSA and pushdown automata PDA.
2. Standard semigroups: I'll describe the construction of Post's semigroup and the syntactic monoid for a Turing machine, and the transition monoid for a FSA.
3. Monoid automata: I'll describe a model of computation defined by adding a monoid register to a FSA, studied by Kambites and others, and show that this seems to provide answers to many of my questions.

# 1. Using machines to model computability and complexity

Alan Turing, who introduced the Turing machine in 1936 and (relative to this) defined notions of computability and computable numbers, proved the equivalence of his notion of computability and Alonzo Church's notion of 'effective calculability'. According to the Church-Turing thesis, this is the **right notion** of computability.

From Turing's definition of a Turing machine it is straightforward to deduce

- a proof that some sets of numbers, some functions are not computable,
- a proof that there are procedures that do not terminate on all input.

Different types of algorithms (of varying complexity) are easily defined by imposing various types of restrictions on the definition of a Turing machine; so finite state machines are defined as Turing machines with bounded memory, pushdown automata as machines with a simple push-down stack as memory.

Turing machines are frequently associated with semigroups and monoids; the existence of a Turing machine with insoluble halting problem is equivalent to the existence of a semigroup with insoluble word problem.

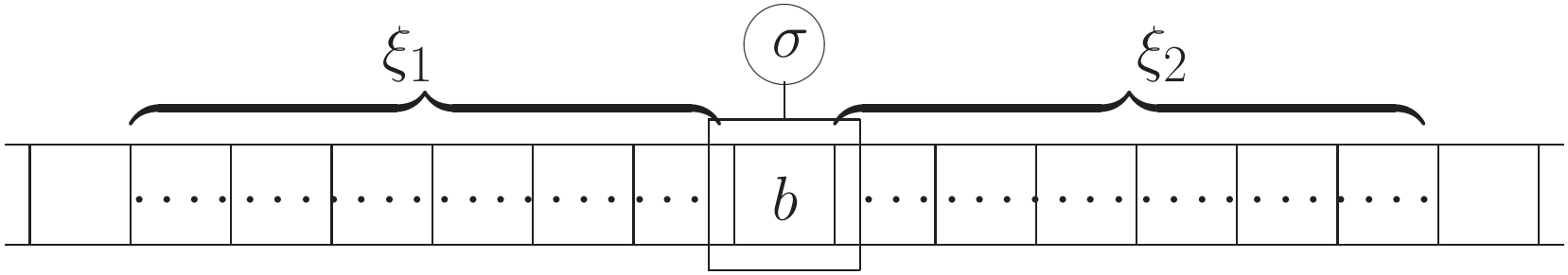
Many other questions of computability can be seen to be equivalent to questions about semigroups or monoids.

# What is a Turing machine?

The simplest of many equivalent models is probably a device consisting of

- a single bi-infinite tape divided up into discrete cells, each of which may contain a letter of a finite alphabet $B$ or be blank,
- a read-write (rw) head that can move left and right along the tape,
- that can exist in any one of a finite number of states $\in \Sigma$, an initial state $\sigma_0$, some halting states, and some of those accepting states.

At any one time the head points to a single cell of the tape. In a single move, the tape head reads from that cell, writes to the cell, changes state, and moves right or left one cell, or stays still, according to a rule $(\sigma, b) \mapsto (\sigma', b', \eta)$, where $\sigma, \sigma' \in \Sigma$, $b, b' \in B$, $\eta \in \{\mathrm{R}, \mathrm{L}, \mathrm{C}\}$.




A string (over an alphabet $A \subseteq B$) is input to the machine by writing it on the tape, and is accepted if the machine moves from an initial configuration with that input into an accepting state; the final contents of the tape are the corresponding output.

The configuration of $\mathcal{M}$ at any point is determined by its state, the position of the head, and the contents of the tape. Hence it can be described by a string of the form $\xi_1 \sigma b \xi_2$, where $\sigma \in \Sigma$ is the current state, $b$ is the symbol under the tape head, and $\xi_1, \xi_2 \in B^*$ are the strings to the left and right of the tape head.

We can consider a Turing machine as a device

- to compute a set of strings that it enumerates sequentially,
- to compute the function that maps each input string to the corresponding output string, or
- to recognise the set of inputs from which it reaches an accepting state.

Other models, with any finite number of bi-infinite or semi-infinite tapes, and a rw head for each, are equivalent.

A set of strings is

**recursively enumerable** if it's the language recognised by a Turing machine, or, equivalently, can be enumerated by a TM,

**recursive** if it's the language of a TM that halts on all input.

## Finite state automata and pushdown automata

The most basic computational machines, namely

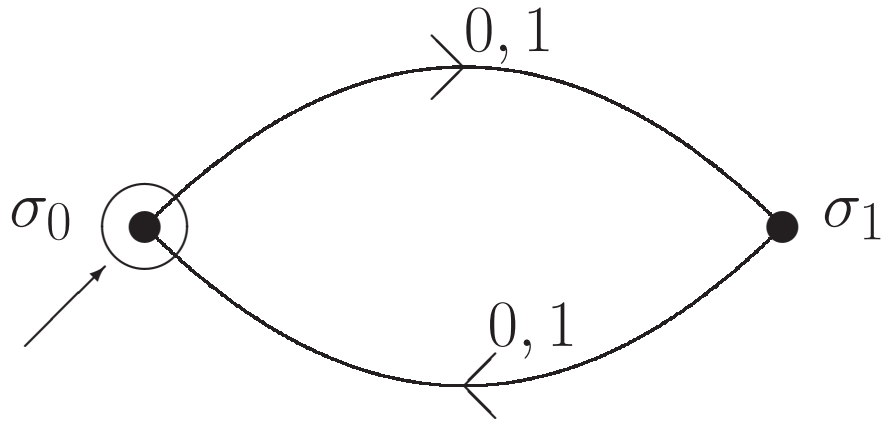
**finite state automata**, which have bounded memory, and

**pushdown automata**, with pushdown stacks as memory,

can easily be described within this framework of Turing machines, although other descriptions may be more natural.

## E.g. FSA recognising binary strings of even length

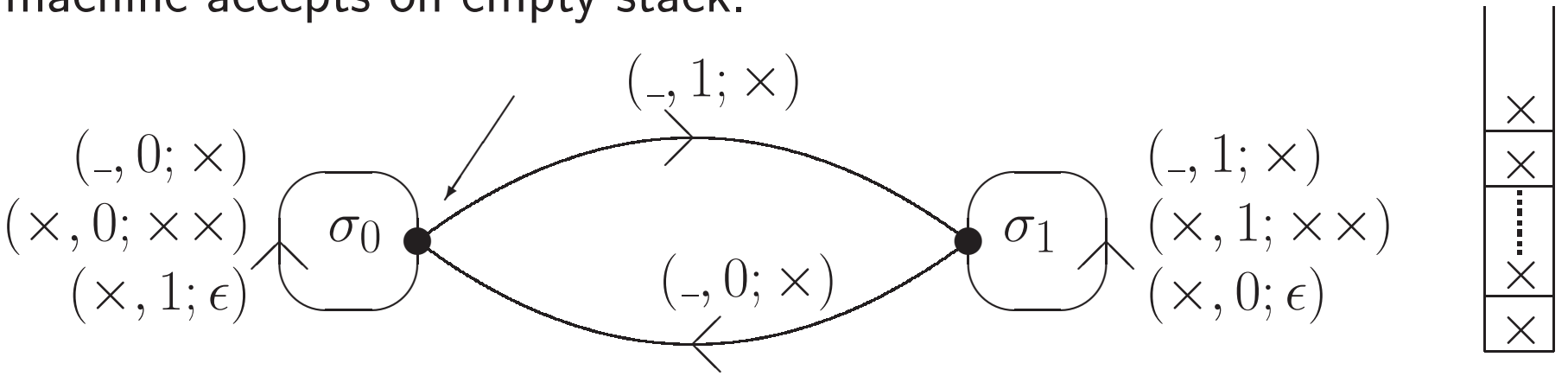
The state records the parity of the string length.




**Any** FSA can be modelled as a Turing machine with a single semi-infinite tape used only for input. The rw head starts at the left hand end, moves right on each move, halts at the right hand end.

## E.g. PDA recognising strings with as many 0's as 1's.

The state records whether more 0's or more 1's have been read, and the stack the (positive) difference between the numbers of 0's and 1's. The machine accepts on empty stack.




The natural TM **model for a** PDA uses two semi-infinite tapes, the 1st for the stack, the 2nd for input. The 1st tape head moves so as to remain at the right hand end of its tape, the 2nd moves rightward, at most one cell in a move (head stays still for an $\epsilon$-move, where no input is processed).

# Determinism for a Turing machine

A TM $\mathcal{M}$ is **deterministic** if from any a given input string $w$ there is no choice in the operation of $\mathcal{M}$, **non-deterministic** otherwise.

For an FSA, determinism corresponds to (at most) one transition from each state $\sigma$ on any one input symbol $a \in A$. ('No transition' would be interpreted as transition to an invisible **failure state**.)

For a PDA, determinism corresponds to (at most) one transition defined by any one combination $(\sigma, y, a)$ of state $\sigma \in \Sigma$, top stack symbol $y \in Y \cup \{\_\}$ and input symbol $a \in A$.

A deterministic PDA could well include $\epsilon$-moves, but corresponding to a given pair $(\sigma, y)$ there may not be than one $\epsilon$-move, nor both an $\epsilon$-move and a move reading an input symbol.

Working simply from the definition of a Turing machine, we can prove

- that there are functions that cannot be computed,
- that there are algorithms that do not halt on all input.

The **first result** follows from the fact that

each Turing machine can be labelled by a unique binary string, its 'code', and so there are countably many Turing machines, but

there are uncountably many sets of strings over any finite alphabet,

The **second result** can be deduced from the fact that the set of Turing machine codes not accepted by the machines they label cannot be recognised, but its complement can.

These results are standard. We can use more algebraic arguments once we associate a semigroup to a Turing machine.

## 2. Standard semigroups for a Turing machine $\mathcal{M}$

- Post constructed a semigroup (called $\gamma(\mathcal{M})$ in Rotman's book) on the set of products over a set containing $\Sigma \cup B$, equating any two products corresponding to configurations related by a move of the TM $\mathcal{M}$.
  If $L(\mathcal{M})$ is non-recursive then $\Gamma(\mathcal{M})$ has insoluble word problem.
- The syntactic monoid $\mathrm{SYN}(L)$ of the language $L = L(\mathcal{M})$ is the quotient of $A^*$ by the equivalence relation $\approx$ defined by

$$u \approx u' \iff (\forall v_1, v_2 \in A^*, (v_1 u v_2 \in L \iff v_1 u' v_2 \in L))$$

- To any fsa $\mathcal{M}$ we can associate the transition monoid $\mathrm{TR}(\mathcal{M})$, that describes the partial action of $\mathcal{M}$ on its set of states. Of course $\mathcal{M}$ is completely specified by that representation of $\mathrm{TR}(\mathcal{M})$, together with an assignment of types to its state set. **Our goal:** a meaningful definition of $\mathrm{TR}(\mathcal{M})$ for a general TM $\mathcal{M}$.

What use are these semigroups?

## Post's semigroup

expresses the computing power of the machine in terms of an equation in the semigroup. He used it to construct a semigroup with insoluble word problem out of a Turing machine that doesn't halt on all input (proving the 'Markov-Post theorem').

# The syntactic monoid

$\textsc{Syn}(L)$ 'recognises' $L$, (By definition, a monoid $M$ recognises a subset $L$ of $A^*$ if

$$\exists \alpha : A^* \Rightarrow M, \quad L = \alpha^{-1}(\alpha(L)).$$

And in fact $\textsc{Syn}(\mathcal{M})$ is a quotient of any monoid that recognises $L$.

It's clear that, for any $L$, $\textsc{Syn}(L) = \textsc{Syn}(A^* \setminus L)$.

$\textsc{Syn}(L)$ is finite $\iff$ $L$ is regular (i.e. the language of an $\textsc{fsa}$).

This result is the basis of the proof that the equivalence of two regular expressions (defining regular languages) is decidable.

## Classifying regular languages according to $\textsc{Syn}(L)$

Eilenberg (in his 1976 book) finds a correspondence between varieties of regular languages and (pseudo-)varieties of monoids.

We can express various properties of a regular language (star-freeness, local testability, etc) in terms of rules holding in its syntactic monoid.

| Property of $L$ | Definition | Property of $M = \text{Syn}(L)$ |
|---|---|---|
| Star-free | Built from finite sets using concatenation, $\cap$, $\cup$, $^c$; correspond to Boolean circuits with limiting fanning. | Aperiodic, i.e. containing no non-trivial subgroups (Schützenberger) |
| $k$-testable | $w \in L$? depends on nature of substrings of length $\leq k$. | For all $x, y, z \in A^*$ with $\|x\| = k-1$, $xyxzx =_M xzxyx$, and $xy = zx \Rightarrow xy =_M xy^2$. (Brzozowski&Simon) |
| Locally testable | $k$-testable for some $k$ | $\text{Syn}(L)$ is locally (idempotent and commutative), (McNaughton; Brzozowski&Simon) |
| Piecewise testable | For some fixed $k$, $w \in L$? depends on nature of subsequences of length $\leq k$, | J-trivial, i.e. admits a partial order $\preceq$ s.t. $\forall x, y,\ xy \preceq x, y$ |

## Transition monoid of an FSA

If $\mathcal{M}$ is the minimal FSA accepting a language $L$, $\mathrm{Tr}(\mathcal{M})$ and $\mathrm{Syn}(L)$ are isomorphic. We can use the relationship between $\mathrm{Tr}(\mathcal{M})$ and $\mathrm{Syn}(L)$ together with the finiteness of $\mathrm{Syn}(\mathcal{M})$ to prove the pumping lemma for regular languages:

> For any regular language $L$, $\exists N$ s.t. any $w \in L$ with $|w| > N$ has the form $uxv$, where $ux^n v \in L$ for all $n \geq 0$.

We can define $\mathrm{Tr}(\mathcal{M})$ for any deterministic FSA $\mathcal{M}$, but we need $\mathcal{M}$ to be minimal for $L$ to deduce $\mathrm{Tr}(\mathcal{M}) \cong \mathrm{Syn}(L)$.

For $\mathcal{M}$ non-deterministic without $\epsilon$-moves we can define a monoid of transition relations on $\Sigma(\mathcal{M})$; that's isomorphic to $\mathrm{Tr}(\mathcal{M}')$, where $\mathcal{M}'$ is the deterministic FSA constructed from $\mathcal{M}$ using the standard algorithm.

# The syntactic monoid: Beyond regular languages

Once we leave regular languages, $\mathrm{Syn}(L)$ is infinite and its structure is less constrained.

Languages in different parts of the Chomsky hierarchy can have the same infinite syntactic monoid.

**Example:**

$$L := \{w \in \{a, b\}^* : w =_{\mathbb{Z}^2} 1\}$$

$L$ and its complement have the same syntactic monoid; any group is the syntactic monoid of its word and co-word problem. But while $L^c$ is context-free, $L$ itself is not (HRRT,2005 + Muller&Schupp,1983).

On the whole, $\mathrm{Syn}(L)$ is not easy to compute. But we can look at some examples (computed by Nick Loughlin).

# Syntactic monoids of some context-free languages

The two context-free languages, $\{a^i b^i : i \geq 0\}$ and $\{a^i b^j : 0 \leq i \leq j\}$, have the same syntactic monoid

$$\langle a, b, 1, 0 \mid ba = 0, a^2 b^2 = ab \rangle.$$

For the one-sided Dyck language $D_1$ of 'balanced parentheses' over $\{(, )\}$, the syntactic monoid is the bicyclic monoid

$$\textsc{Syn}(D_1) \cong P_1 = \langle a, b \mid ab = 1 \rangle,$$

For the one-sided Dyck language $D_n$ of 'balanced parentheses' over $n$ distinct pairs of brackets we have the polycylic monoid,

$$\textsc{Syn}(D_n) \cong P_k = \langle p_1, \ldots p_k, q_1, \ldots, q_k \mid p_i q_j = \delta_{ij} \rangle$$

For the two-sided Dyck language $\tilde{D}_n$ (where we drop the requirement that one symbol in a pair 'opens', the other 'closes') we have $\textsc{Syn}(\tilde{D}_n) \cong \mathbb{F}_n$.

In his thesis (1979) Sakarovitch proves some general results about the syntactic monoids of context-free languages, such as:

**Theorem (Sakarovitch 1979)** A context-free language whose syntactic monoid is an abelian group is deterministic $\iff$ the rank of the group is at most 1.

In general it seems that the syntactic monoid of a non-regular language is hard to compute, somewhat unconstrained, and not tremendously useful. However it turns out that the polycyclic monoids, as syntactic monoids of Dyck languages are important.

## Context-free languages: the big results

**Theorem (Chomsky-Schützenberger, 1963)** Any context-free language is the homomorphic image of the intersection of a one-sided Dyck language with a regular language (over the same alphabet of $2n$ symbols). The same is true with respect to a two-sided Dyck language.

Hence one might hope to relate every context-free language to a monoid associated with a polycylic monoid. We'll see soon that in fact we can.

We might also hope for an algebraic explanation of the pumping lemma for context-free languages (see Amarillo& Jeanmougin 2012):

> For any context-free language $L$, $\exists N$ s.t. any $w \in L$ with $|w| > N$ has the form $uvxyz$, where
>
> $$|vxy| \leq N, \quad |vy| \geq 1, \quad \forall n \geq 0,\, uv^n xy^n z \in L.$$

## Defining $\textsc{Tr}(\mathcal{M})$ for a fully deterministic PDA

We call a PDA $\mathcal{M}$ **fully deterministic** if it is deterministic with no $\epsilon$-transitions. In that case, we can define a monoid

$$\textsc{Tr}(\mathcal{M}) = \langle \tau_a : a \in A \rangle$$

of partial maps on $\Sigma \times Y^*$.

Where $\tau : \Sigma \times (Y \cup \{\_\}) \times A \to \Sigma \times (Y^* \cup \{\_\})$ is the transition function of the pda, we define partial maps $\tau_a : a \in A$ on $\Sigma \times Y^*$ as follows.

If $u \in Y^*$ is a non-empty string, let $\mathrm{l}[u]$ be the last letter of $u$, and $\mathrm{pre}[u]$ the prefix of $u$ obtained by deleting $\mathrm{l}[u]$. Suppose that $a \in A, \sigma \in \Sigma$.

If $u \in Y^*$ is non-empty and $\tau(\sigma, \mathrm{l}[u], a) = (\sigma', u')$,
then $\tau_a(\sigma, u) := (\sigma', \mathrm{pre}[u]u')$.
If $\tau(\sigma, \_, a) = (\sigma'', u'')$, then $\tau_a(\sigma, \epsilon) := (\sigma'', u'')$.

**Example:** We compute the transition monoid of the PDA $\mathcal{M}$ that recognises the identity problem of the polycylic monoid $P_n$.

$\mathcal{M}$ has a single state $\sigma_0$, stack alphabet $y_1, \ldots y_n$, and the following transition function:

$$\begin{aligned} \tau(\sigma_0, \_, p_i) &= (\sigma_0, y_i) \\ \tau(\sigma_0, y_j, p_i) &= (\sigma_0, y_j y_i) \\ \tau(\sigma_0, y_i, q_i) &= (\sigma_0, \epsilon) \end{aligned}$$

The description above defines partial maps $\tau_{p_i}, \tau_{q_i}$ via

$$\tau_{p_i}(\sigma_0, v) = (\sigma_0, v y_i), \quad \tau_{q_i}(\sigma_0, v y_i) = (\sigma_0, v),$$

for any string $v \in Y^*$. Clearly it's natural to consider this as a partial action on $Y^*$ alone. $\tau_{p_i}$ appends $y_i$ to the stack, while $\tau_{q_i}$ deletes $y_i$ from the top of the stack, provided it is there. Of course $\tau_{p_i}$ and $\tau_{q_i}$ generate precisely the polycyclic monoid.

So far this definition of transition monoid seems to be quite restricted. However it turns out that we can apply it much more generally, once we start to model computation using monoid automata.

Monoid automata give us models of computation in which basically the complexity of the computation is encoded in the monoid, $M$, so e.g. we can set the monoid to be a group with insoluble word problem and hence describe a machine on which the word problem for the group can be solved. That example might seem a little absurd ... Nonetheless we see that different choices of monoids will define distinct families of monoid automata with differing computational capabilities.

My background comes from Mark Kambites' 2006 article 'Formal languages and groups as memory'.

## 3. Monoid automata

An $M$-automaton $\mathcal{M}$ over a given monoid $M$, alphabet $A$ is defined by adjoining to an FSA over $A$ a register to hold an element $m \in M$.

- An element of $M$ is attached to each transition of the FSA.
- Initially the register a selected element $m_0$ of $M$.
- As a string is read through the FSA
  - the state $\sigma$ is updated according to the transition rules of the FSA,
  - $m$ is updated by successive right multiplications by the monoid elements labelling the transitions.
- $w$ is accepted by the monoid automaton if
  - $w$ leads to an accepting state of the FSA,
  - the final element in the monoid register is equal to $1_M$.

## Capturing the entire Chomsky hierarchy

- Any FSA can be seen as a monoid automaton based on the monoid with one element.
- A PDA with a stack alphabet of $n$ symbols can be modelled by a $P_n$-automaton based on the same underlying FSA. We'll describe the construction later.
- Any PDA can be modelled by a $P_2$-automaton.
- Any PDA can be modelled by an $\mathbb{F}_2$-automaton.
- Any Turing machine can be modelled using states and two independent stacks, so as a $P_2 \times P_2$-automaton or an $\mathbb{F}_2 \times \mathbb{F}_2$-automaton.

It's no coincidence that we see the syntactic monoids of both the Dyck languages here.

# Representing a PDA as a $P_n$-automaton

A PDA $\mathcal{M}$ with stack alphabet $Y = \{y_1, \ldots, y_n\}$ is modelled by a $P_n$-automaton $\mathcal{M}'$ based on the same underlying FSA, as follows, where $p : Y^* \to P_n$ is the homomorphism defined by $\forall i, y_i \mapsto p_i$.

- Where $\mathcal{M}$ is initialised with $u_0$ on the stack, initialise the monoid register of $\mathcal{M}'$ to $p(u_0)$.
- Corresponding to each transition $\tau(\sigma, \_, x) = (\sigma', u)$ of $\mathcal{M}$, define a transition of $\mathcal{M}'$ from $\sigma$ to $\sigma'$ labelled by $(x, p(u))$.
- Corresponding to each transition $\tau(\sigma, y_i, x) = (\sigma', u)$ of $\mathcal{M}$, define a transition of $\mathcal{M}'$ from $\sigma$ to $\sigma'$ labelled by the pair $(x, q_i p(u))$.

NB: If an element $m \in P_n$ is not a product ending in $p_i$, then no product $mq_ip(u)m'$ can ever equal the identity.

# Determinism for a monoid automaton

According to my definition of determinism for a TM, we would expect to define an $M$-automaton to be deterministic if for a given input word $w$, there's only one way to read $w$ through the machine.

But such a definition allows the underlying FSA to have multiple transitions from a given state $\sigma$ on the same input symbol $a$, as well as $\epsilon$-transitions. Indeed that situation may well occur in the natural translation of a deterministic PDA.

To deal with this unnatural situation, we define a monoid automaton to be **fully deterministic** if the underlying FSA admits at most one transition from a given state $\sigma$ on a given input symbol $a$, and no $\epsilon$-transitions.

## Elementary results

Kambites proves a number of elementary results about monoid automata, such as

**Proposition (Kambites,2006)** For any monoid $M$, if $L$ is the language of an $M$-automaton then $L$ is the language of an $N$-automaton, for some fg submonoid $N$ of $M$.

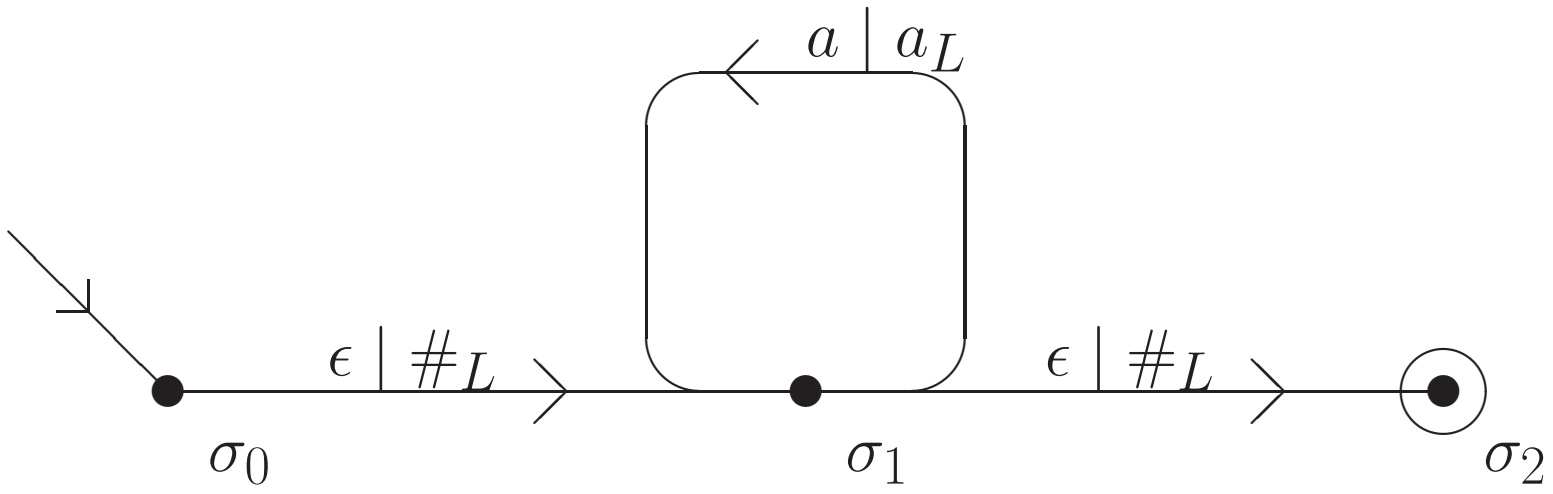
We just take the fg submonoid generated by the monoid elements that label edges of the underlying automaton.

# A tailor-made monoid $M_L$ and $N_L$-automaton

For any language $L$, over an alphabet $A_L$, Loughlin (2013) describes a monoid

$$M_L = \langle A_L, \#_L \mid \#_L l \#_L = 1, \quad \forall l \in L \rangle$$

and then an $M_L$-automaton accepting $L$ as follows.



## Rational transductions and Chomsky-Schützenberger

A **rational transducer** $\mathcal{M}$ from $B$ to $A$ is defined to be an fsa over $B \times A$. A language $L \subseteq A^*$ is the rational transduction (via $\mathcal{M}$) of a language $K \subseteq B^*$ if it has the form

$$L = \{w \in A^* : \exists u \in K, (u, w) \in L(\mathcal{M})\}$$

**Proposition (Kambites, 2006)** For any f.g. monoid $M$, $L \subseteq A^*$ is the language of an $M$-automaton iff $L$ is a rational transduction of the identity language of $M$ wrt some (equiv. wrt every) finite generating set.

From this we can deduce a form of Chomsky-Schützenberger, that

> every context-free language is the image under a rational transduction of the one-sided Dyck language,

from which we can (apparently) deduce the C-S result as already stated.

## Defining $\textsc{Tr}(\mathcal{M})$ for a fully deterministic $M$-automaton

We can easily define a transition monoid for a monoid automaton $\mathcal{M}$ provided that the underlying fsa $\mathcal{M}_F$is deterministic with no $\epsilon$-moves. In that case we say that $\mathcal{M}$ is **fully deterministic**.

Let $A$ be the input alphabet, $\Sigma$ the state set, and $M$ the associated monoid for $\mathcal{M}$,

We can define a partial action of $A^*$ on $\Sigma \times M$ as follows..

Suppose there's a transition from $\sigma \in \Sigma$ to $\sigma'$ labelled by $a$, with associated monoid element $m_{\sigma,a}$. Then, for all $m \in M$, we define

$$\tau_a(\sigma, m) = (\sigma', mm_{\sigma,a})$$

This bears a strong resemblance to the monoid we already defined from a fully deterministic PDA.

## Defining $\textsc{Tr}(\mathcal{M})$ for a general $M$-automaton $\mathcal{M}$.

This is a little tricky.

If we could define a procedure that took as input an $M$-automaton $\mathcal{M}$ as input and output a fully deterministic $\tilde{M}$-automaton $\tilde{\mathcal{M}}$ (for a monoid $\tilde{M}$, related to $M$), with $L(\tilde{\mathcal{M}}) = L(\mathcal{M})$, we could use the construction above to define a transition monoid for $c\tilde{M}$ and hence for $\mathcal{M}$. However we have been unable to find such a procedure.

It's proved by Zetzsche (2011) that we certainly cannot hope to succeed with $\tilde{M}$ equal to $M$.

However the natural construction, derived from the standard construction for an $\textsc{fsa}$, although it does not seem to give a monoid automaton per se, does give us a reasonable definition of a transition monoid.

## The idea

We can make a device recognising $L(\mathcal{M})$ that is based on

- a deterministic fsa $\tilde{\mathcal{M}}_F$ recognising the same language as $\mathcal{M}_F$, together with
- a monoid $\tilde{M}$ related to $M$,

but which is not quite a $\tilde{M}$-automaton, because we cannot relate acceptance to the identity element of $\tilde{M}$ in the way we need to.

We define a monoid of partial maps in terms of this device.

## The construction of the deterministic FSA $\tilde{\mathcal{M}}_F$

The construction of a deterministic fsa $\tilde{\mathcal{M}}_F$ from the underlying fsa $\mathcal{M}_F$ of the $M$-automaton $\mathcal{M}$ is standard.

The FSA $\tilde{\mathcal{M}}_F$ has state set $\tilde{\Sigma}$ corresponding to the set of subsets of $\Sigma$.

For $S \subseteq \Sigma$, $a \in A$,
transition in $\tilde{\mathcal{M}}_F$ from $S$ on $a$ is to the state corresponding to $S' \subseteq \Sigma$, the set of all states that are the target in $\mathcal{M}_F$ of some $\sigma \in S$ via a path whose label contains a single symbol $a$ and any number of $\epsilon$s.

## The construction of the monoid $\tilde{M}$

The monoid $\tilde{M}$ is defined on the set $\mathbb{M}_{N,N}(2^M)$ of $N \times N$ matrices over the set $2^M$ of subsets of $M$, where $N = |\Sigma|$ as follows:

- on $2^M$ we define multiplication via the rule
$$AB = \{ab : a \in A, b \in B\}$$
- for $P, Q \in \mathbb{M}_{N,N}(2^M)$ we define the entry $(PQ)_{ij}$ of the product $PQ$ as
$$\bigcup_{k=1}^{N} P_{ik}Q_{kj}$$
where the products $P_{ik}Q_{kj}$ of subsets of $M$ are as defined above.

## The construction of $\mathrm{Tr}(\mathcal{M})$

The transition from $S$ to $S'$ labelled by $a \in A$ is labelled by the matrix whose $i, j$-th entry is

the set of all monoid elements that label paths labelled by a single $a$ together with any number of $\epsilon$s from the state $\sigma_i$ to the state $\sigma_j$ , provided that $\sigma_i \in S, \sigma_j \in S'$,

and otherwise is the empty set.

Hence for each $a \in A$ we can define a partial map

$$\tau_a : \Sigma \times \mathbb{M}_{N,N}(2^M) \to \Sigma \times \mathbb{M}_{N,N}(2^M).$$

We define

$$\mathrm{Tr}(\mathcal{M}) := \langle \tau_a : a \in A \rangle$$

## Remarks

- $\mathrm{Tr}(\mathcal{M})$ has an action on the states of the FSA $\tilde{\mathcal{M}}_F$ that is equivalent to the action of the transition monoid of the FSA (which we recall is isomorphic to $\mathrm{Tr}(\mathcal{M}_F)$.
- The monoid we already described for a fully deterministic $P_n$-automaton bears a strong resemblance to the transition monoid we also described for a fully deterministic PDA.
- Nick Loughlin hopes he may be able to define a 'Rees co-congruence' on $\tilde{M}$, and hence a monoid $\overline{M}$ and a fully deterministic $\overline{M}$-automaton, based on the (images of the) partial maps described above, that accepts the same language as $\mathcal{M}$, and has $\mathrm{Tr}(\mathcal{M})$ (or something closely related) as its transition monoid.