EMBEDDING A MODULAR MACHINE INTO A GROUP

PATRICK STEVENS

https://www.patrickstevens.co.uk/misc/ModularMachines/EmbedMMIntoTuringMachine.pdf

1. Introduction

This document was born from a set of lectures I attended, given by Dr Maurice Chiodo in his half of the Part III Maths course *Infinite Groups and Decision Problems*, which he lectured jointly with Dr Jack Button in 2016. The treatment of modular machines was a bit heavy in text, and they are actually fairly intuitive objects, so I decided to expand on them here.

A warning for those who are following the official course notes: I accidentally interchanged L and R in my definition of a modular machine below, relative to the official notes. It doesn't change any of the ideas, and is just a different labelling convention, but by the time it was pointed out to me, most of the document was written, and I can't be bothered to fix it now. Use at your own risk.

Mistakes in this article are due entirely to me; thanks to Joshua Hunt and Daniel Zheng for catching some before this went to press.

2. What is a modular machine?

A modular machine is a Turing-equivalent form of computation. They operate on two tapes, which are each bounded at one end. Each cell of a tape may be filled with an integer from 0 to $m-1$ inclusive; this is where the name "modular" comes from.

a_1	
a_2	b_1
a_3	b_2
$\vdots$	$\vdots$
a_{n-1}	b_{k-1}
a_n	b_k

The machine is considered to have a "head" which is looking at the bottom two cells (which are on the edge of the tape); in this instance, the head is looking at a_n and b_k. The machine also has a list of instructions, of the form

$$(\alpha, \beta, (x, y), L)$$

or

$$(\alpha, \beta, (x, y), R)$$

These instructions are constantly active, and the machine just does whichever it can at each stage. To ensure that the machine only has one thing it can do at any time, we insist any α, β may only have one instruction in which they appear in the first two

places. For example, if $(1, 2, (3, 3), L)$ is an instruction, then $(1, 2, (5, 2), R)$ cannot also be an instruction.

Since α, β will appear on the tape during execution, we require that $0 \leq \alpha, \beta < m$; since x, y will be written onto the tape during execution, we require that $0 \leq x, y < m$.

When the head sees entries $(\alpha = a_n, \beta = b_k)$ and it has the instruction

$$(a_n, b_k, (x, y), L)$$

it executes the following procedure:

(1) Shift the left-hand tape up one:

a_1	
a_2	
a_3	b_1
$\vdots$	b_2
a_{n-1}	$\vdots$
a_n	b_{k-1}
	b_k

(2) Write x, y into the left-hand tape's bottom cells:

a_1	
a_2	
a_3	b_1
$\vdots$	b_2
a_{n-1}	$\vdots$
x	b_{k-1}
y	b_k

(3) Shift the right-hand tape down one:

a_1	
a_2	
a_3	
$\vdots$	b_1
a_{n-1}	b_2
x	$\vdots$
y	b_{k-1}

If, instead, the instruction was

$$(\alpha, \beta, (x, y), R)$$

then the same procedure would be carried out, but with right and left interchanged. This would result in the final state

	b_1
a_1	b_2
a_2	$\vdots$
a_3	b_{k-1}
$\vdots$	x
a_{n-1}	y

Remark. The very convenient thing about modular machines is that they can be easily coded into numbers. All the state can be tracked with just two numbers:

$$(A, B) := (\sum_{i=0}^{n} a_i m^i, \sum_{i=0}^{k} b_i m^i)$$

and, for instance, the operation $(\alpha, \beta, (x, y), L)$ produces

$$([A - (A \mod m)]m + xm + y, [B - (B \mod m)]/m)$$

We could even collapse (x, y) into a single integer $xm + y$ which is less than m^2.

3. Turing equivalence

We will take our Turing machines to be in "quintuple form": they consist of a list of instructions of the form

$$(q, a, a', q', L/R)$$

where:

- q is the current state
- a is the symbol under the head
- a' is the symbol to write to the tape
- q' is the state to move to
- L/R is the direction the head moves after this instruction executes.

Note that the machine always writes and always moves on every execution step.
We may implement a Turing machine as a modular machine as follows.

Definition 1 (Instantaneous description)**.** *An* instantaneous description *of a Turing machine is a string of the form*

$$s_1 s_2 \dots s_k q a s_{k+2} \dots s_r$$

where s_i are the symbols written on the alphabet, q is the state the machine is currently in, and a is the symbol under the head. It captures completely the state of execution at a given instant in time.

We will implement the instantaneous description $s_1 s_2 \dots s_k q a s_{k+2} \dots s_r$ as both of two possible modular machine states:

s_1	
s_2	s_r
$\dots$	s_{r-1}
s_{k-1}	$\dots$
s_k	s_{k+2}
q	a

or

s_1	
s_2	s_r
$\dots$	s_{r-1}
s_{k-1}	$\dots$
s_k	s_{k+2}
a	q

It will soon become clear why we want to be able to use both these states. While the modular machine can only ever occupy one of the states, the Turing machine it's emulating will be in the same state whichever of the two the modular machine happens to be in.

Define the modulus m to be the number of Turing-machine states, plus the number of Turing-machine symbols, plus 1; this is just to make sure we have plenty of symbols to work with, and can store all the information we need in any given cell of the tape.

How can we express a Turing-machine instruction? Remember, they are one of the two forms

- (q, a, a', q', L), which would convert

  $$s_1 s_2 \dots s_k q a s_{k+2} \dots s_r$$

  to

  $$s_1 s_2 \dots s_{k-1} q' s_k a' s_{k+2} \dots s_r$$

- (q, a, a', q', R), which would convert

  $$s_1 s_2 \dots s_k q a s_{k+2} \dots s_r$$

  to

  $$s_1 s_2 \dots s_k a' q s_{k+2} \dots s_r$$

Therefore, taking our correspondence between Turing-machine instantaneous descriptions and internal states of a modular machine, we need the instruction (q, a, a', q', L) to take

s_1					s_r
s_2	s_r		s_1	s_{r-1}	
$\dots$	s_{r-1}	$\mapsto$	s_2	$\dots$	
s_{k-1}	$\dots$		$\dots$	s_{k+2}	
s_k	s_{k+2}		s_{k-1}	a'	
q	a		q'	s_k	

or to

	s_r
s_1	s_{r-1}
s_2	$\dots$
$\dots$	s_{k+2}
s_{k-1}	a'
s_k	q'

Now it is clear why we needed the two possible representations of a single Turing-machine instantaneous description: only the second of the above transitions is easy to implement in the modular machine, but it has swapped q from the left-hand register to the right-hand register. This is perfectly kosher, but only because we were careful to state that the current-state and current-symbol letters were interchangeable in the modular machine. It can be performed using the modular machine instruction $(q, a, (a', q'), R)$.

Similarly, since we might have started this whole affair with the other representation of the TM instantaneous description, we need to do the same with

s_1	
s_2	s_r
$\dots$	s_{r-1}
s_{k-1}	$\dots$
s_k	s_{k+2}
a	q

which, by modular machine instruction $(a, q, (a', q'), R)$ is taken to

	s_r
s_1	s_{r-1}
s_2	$\dots$
$\dots$	s_{k+2}
s_{k-1}	a'
s_k	q'

Notice, as an aside, that the Turing-machine head was moving left, and the modular-machine "head" symbol q' has ended up on the right-hand tape whichever of the two representations of the instantaneous description we used.

We can do the same for the Turing machine instructions which involve moving rightwards.

4. Summary

If we take our Turing machine states $(q, a, a', q', L/R)$ and, for each one, create a pair of modular machine instructions $(a, q, (a', q'), L/R)$ and $(q, a, (a', q'), L/R)$, we end up with a modular machine that precisely emulates the Turing machine. We could represent the pair (a', q') as a single integer which is between 0 and m^2: namely, by taking $a'm+q'$. This makes the strings a bit shorter, but not as comprehensible.

Definition 2 (Halting set)**.** *We define the "halting set" of a modular machine to be the collection of states of the tape from which, when the machine runs, we eventually end up with both tapes zeroed out. For this to be a sensible definition, we want the machine not to have an instruction corresponding to head-states* $(0,0)$*, so that the machine really does stop eventually if it started in a halting state.*

5. Embedding a modular machine into a group

What we seek now is a way to embed a MM into a group. A MM has two pieces of state: the left-hand tape and the right-hand tape. These can be easily coded as integers.

5.1. **How could we apply a machine instruction?** Let's imagine we have a way of representing the state (a,b) as a group word in some group: $t(a,b)$. What we now want is a way of applying a transformation to obtain the different word $t(a',b')$ which corresponds to executing the MM instruction $(\alpha,\beta,(x,y),L/R)$. A very good way of applying reversible transformations is to conjugate, so let's add lots of letters to the group: one r_i for each instruction $(\alpha,\beta,(x,y),L)$, and one s_i for each $(\alpha,\beta,(x,y),R)$. Conjugating by the letter r_i will apply the ith L-instruction.

The required effect is

$$r_i t(\alpha+mu,\beta+mv)r_i^{-1} = t(xm+y+m^2u,v)$$

5.2. **How do we store the states?** The next idea is that since our states are merely integers, we might store them as exponents of a group generator: $x^a y^b$ where a is the left-hand tape and b the right-hand. In this scheme, it'll be easier if we allow x and y to commute, since all we care about is their exponents.

Now, it will be convenient to introduce a third letter, t, which will let us store separately the "head" and the "body" of the tapes. Our storing scheme will be

$$y^{mv}x^{mu}(x^\alpha y^\beta t y^{-\beta}x^{-\alpha})x^{-mu}y^{-mv}$$

which we will denote $t(\alpha+mu,\beta+mv)$, corresponding to the tape which has α,β as the two heads, and then u,v as the data on the rest of the tape.

So we define

$$K := \langle x,y,t \mid xy=yx\rangle$$

Remark. Notice that the only words in $K := \langle x,y,t \mid xy=yx\rangle$ which can be expressed in this form are the words $y^a x^b t x^{-b} y^{-a}$. Therefore it makes sense to define a subgroup

$$T := \langle t(r,s) : r,s \in \mathbb{Z}\rangle$$

which consists of "all possible machine states".

5.3. **How do the machine instructions work?** How does r_i act, then? $(\alpha,\beta,(p,q),L)$ needs to take

$$t(\alpha+mu,\beta+mv) = y^{mv}x^{mu}(x^\alpha y^\beta t y^{-\beta}x^{-\alpha})x^{-mu}y^{-mv}$$

to

$$t(pm+q+m^2u,v'm+\nu) = y^{v'm}x^{(um+p)m}(x^q y^\nu t y^{-\nu}x^{-q})x^{-(um+p)m}y^{-v'm}$$

where we are writing $v = v'm + \nu$. More concretely,

$$y^{v'm^2+\nu m}x^{mu}(x^\alpha y^\beta t y^{-\beta}x^{-\alpha})x^{-mu}y^{-v'm^2-\nu m} \mapsto y^{v'm}x^{um^2+pm}(x^q y^\nu t y^{-\nu}x^{-q})x^{-um^2-pm}y^{-v'm}$$

Notice that this is exactly performed by the map $x^m \mapsto x^{m^2}, y^m \mapsto y, t(\alpha,\beta) \mapsto t(q+pm,0)$. How can we make that well-defined (since we clearly can't send y^m to y without messing up the map of $t(\alpha,\beta)$)? The trick is to forget that we're working with x, y, t, and start working with $t(\alpha,\beta)$, x^m, y^n as atomic blocks.

Define a new subgroup

$$K^{M,N}_{\alpha,\beta} := \langle \overline{t(\alpha,\beta)}, \overline{x}^M, \overline{y}^N \rangle \leq K$$

Then the map

$$\phi_i : t(\alpha,\beta) \mapsto t(q+pm,0), x^m \mapsto x^{m^2}, y^m \mapsto y$$

would do what we want. What is the domain and range of that map? If we view it as being $K^{m,m}_{\alpha,\beta} \to K^{m^2,0}_{q+pm,1}$, then it's actually an isomorphism: it's just matching up the generators of the respective groups.

OK, we have a map which we want to apply whenever we see $r_i w r_i^{-1}$. The way we can do that is to create an HNN extension: take $K*_{\phi_i}$ with stable letter r_i.

We can do the whole lot again with ψ_i corresponding to $s_i w s_i^{-1}$, which performs an R instruction (where ϕ performed an L instruction).

5.4. **How do we turn this all into a group?** We've got all these groups floating around; to specialise to a group in which we can only be dealing with machine states, consider

$$T'_{\mathcal{M}} := \langle \overline{t(\alpha,\beta)} : (\alpha,\beta) \in H_0(\mathcal{M}); \overline{r_i} : i \in I; \overline{s_j} : j \in J \rangle \leq K*_{\phi_i;\psi_j}$$

where H_0 refers to the halting set of the modular machine $\mathcal{M}$.

If we take an element $\overline{t(\alpha,\beta)}$ of $T'_{\mathcal{M}}$ which contains no r_i or s_j, it reduces to the word t in this group if and only if, when applying r_i's and s_j's, we end up in the halting set of the modular machine: the HNN extension has quotiented out by the relation "conjugating by r_i applies effect ϕ_i, which moves the modular machine according to the ith L-instruction". The word t is precisely symbolising the empty tape. Of course, we could have lots of unused instructions or parts of instructions floating around: the group word $r_i t$ still has empty tape, so is still a halting state.

5.5. **Subgroup membership to word problem.** We have constructed a subgroup $\langle t \rangle' := \langle t, r_i, s_j \rangle$ such that (α,β) is a halting state of the modular machine if and only if $t(\alpha,\beta)$ lies in that subgroup. Using an HNN extension, we can make a group where we just need to check equality of words: create the group

$$G_{\mathcal{M}} := \langle K*_{\phi_i,\psi_j}; k \mid khk^{-1} = h \ \forall h \in \langle t \rangle' \rangle$$

Then conjugating an element by k does nothing precisely when that element was in $\langle t \rangle'$: that is, precisely when the modular machine halts from that starting state.

5.6. **What have we missed out?** There are many checks to be done along the way. Our HNN extensions need to be well-defined. The ϕ_i, ψ_j need to be iso. $G_{\mathcal{M}}$ is in fact finitely presented, but we need to show that.

However, once those checks are done, we have produced an explicit recipe for constructing a finitely presented group which can simulate a given arbitrary modular machine.

6. Higman's construction

It turns out that, somewhat shockingly, there is a beautiful way to use the above to embed a recursively-presented group $C = \langle X \mid R \rangle$ into a finitely-presented group. Think of the earlier construction as telling us how to execute a Turing machine inside a group. Then, morally, we do the following.

(1) Take the machine that halts precisely on members of R;
(2) Embed it into a group G;
(3) Glue G onto $C = \langle X \mid R \rangle$ in such a way that we can use the machine-part, G, to decide which reductions to make (rather than querying the infinite set R).

Of course, the construction is rather complicated, but the upshot is that the finite presentation which defines G can be used to capture all the information that lies in the infinite relator-set R.

6.1. **General approach.** We will make a (large!) group whose elements include what I will call "collections", which have the following structure:

(1) machine state (for those following the course notes, this is $K_{\mathcal{M}}$ in Step 13 of Construction 11.2)
(2) word under consideration (this is $\langle b_1, \dots, b_n \mid \cdot \rangle$)
(3) group element that word corresponds to (this is $\overline{C}$)
(4) marker (this is d)

The group $\langle X \mid R \rangle$ embeds in the third component ("group element that word corresponds to"). The "machine state" section will be implemented as $K_{\phi_i;\psi_j}$ from earlier (which, recall, was finitely presented). The "word under consideration" will be how we extract information from the "machine state". The "marker" serves no purpose for interpretation, but it turns out to be an important fiddly detail in the construction.

6.2. **Construction.** We are going to use a whole lot of HNN extensions to manipulate collections.

6.2.1. *Which modular machine to use?*

c_{i_n}	0
$\dots$	$\dots$
c_{i_2}	0
c_{i_1}	0

Take a modular machine such that, if we start the tape with the above state, we halt with an empty tape if and only if $c_{i_1} \dots c_{i_n}$ is in R. (Recall that our recursive presentation is $C = \langle X \mid R \rangle$, and this is the group we want to embed.) It is possible to do this (for those following the course notes, this is steps 1 through 5): symmetrise the generators

if necessary so that each generator $c \in X$ has an inverse c^{-1} and a relator $cc^{-1} = e$. Then make a Turing machine that enumerates the trivial words in this new presentation (which is, in spirit, the same as the old presentation; it certainly defines the same group). Convert that Turing machine into a modular machine, where each c or c^{-1} of the group corresponds to a cell-state a_c or $a_{c^{-1}}$.

Once we've got that modular machine, we can embed it into a group using what's happened already, although we're actually only going to need

$$K_{\mathcal{M}} := K*_{\phi_i,\psi_j}$$

which, recall, is the group which holds an MM state as the word $x^\alpha y^\beta t y^{-\beta} x^{-\alpha}$ which can also have the MM instructions directly applied to it, by conjugating with the stable letters r_i, s_j respectively to apply the ith Left-instruction or jth Right-instruction.

6.2.2. *Creating a collection.* We're only interested in certain modular machine states: namely, those corresponding to $t(\alpha, 0)$ for certain α. (Recall that $t(\alpha, \beta)$ is our notation for how the group $K_{\mathcal{M}}$ stores the current state (α, β) of the MM.) That is, we're only interested in how the modular machine behaves when we start it off with input $\alpha = \sum_{i=0}^{r} c_{k_i} m^i$, say: equivalently, when we ask it the question "Is $c_{k_0} \dots c_{k_r}$ in the relating set of C?".

So, while other MM-states appear encoded in our $K_{\mathcal{M}}$—for example, the state corresponding to "starting with 5 on the left-hand tape and 2 on the right-hand, perform the fourth left-instruction in the list of possible instructions"—our remaining manipulations to the group will all refer to $t(\alpha, 0)$ directly. That is, our remaining manipulations will ignore non-interesting MM states.

Given an MM state $t(\alpha, 0)$, we unpack it into a collection by conjugating with a new stable letter, p, and taking an HNN extension. (In the course notes, this is steps 17 and 18.) The extension will take $t(\alpha, 0)$ and unpack it into the word

$$[t(\alpha, 0), w_\alpha(b)]$$

where $w_\alpha(b)$ simply means "take the word which is currently loaded into the MM's memory, as its left-hand tape where the right-hand tape is 0, and write it down with b's in the abstract".

Example. If $\alpha = c_3 + c_7 m + c_1 m^2$, we would have $w_\alpha(b) = b_3 b_7 b_1$, a word in the abstract group $\langle b_1, \dots, b_n \mid \cdot \rangle$. I'm playing fast and loose with the ordering here; I may mean $b_1 b_7 b_3$, but I can't be bothered to check which is right.

To recap, the unpacked word $[t(\alpha, 0), w_\alpha(b)]$ has two components so far, then: the modular machine state $t(\alpha, 0)$, and an abstract statement $w_\alpha(b)$ of the word we're asking about. I'll add a third component: the "marker" letter d, so our unpacked word actually has three components and looks like

$$[t(\alpha, 0), w_\alpha(b), d]$$

We obtained the unpacked word by conjugating $t(\alpha, 0)$ by a new stable letter p, and taking an HNN extension which adds the relators

$$p t(\alpha, 0) p^{-1} = t(\alpha, 0) w_\alpha(b) d$$

for each α.

For reasons which will be important later, I added the d at this point: an end-marker, sitting after the $w_\alpha(b)$. I can't motivate its presence in this section, but eventually we will start appending things to $w_\alpha(b)$, and it will become vital to know where the word ends. We use the marker d for that.

Remark. This is an awful lot of relators. Infinitely many, in fact! Don't panic; we will eventually show that we can actually replace most of them with a finite number of relators. We haven't actually tied the behaviour of the machine to the manipulation of collections yet; only expanded a machine into a collection.

Later on, we will add one more component (our last one) to the collection. It will be a bona fide word on C's generators, and it will be the word which α represents. We can do this by insisting that $b_i c_j = c_j b_i$ in general, and by doing another HNN extension, which will have the effect of taking b_j to $b_j c_j$. That is, $b_3 b_2 \mapsto b_3 c_3 b_2 c_2 = b_3 b_2 c_3 c_2$; then we view the b chunk and the c chunk as being distinct, forming the second and third components of our four-component collection respectively.

Formally, in a little while we will add a stable letter V and add the relators that $V b_j V^{-1} = b_j c_j$ (and that $V a V^{-1} = a$ for most of the other possible a – for example, for $a = t$ or $a = r_i$). In the course notes, ψ_+ adds this component to the collection.

So our final collection will be

$$[t(\alpha, 0), w_\alpha(b), w_\alpha(c), d]$$

However, we don't bother adding this yet. In essence, the original three components of the collection are where the magic happens; but because $w_\alpha(b)$ is a "purely syntactic" readout of the machine $t(\alpha, 0)$, we won't actually have an embedded copy of C in the group unless we add one. So as the last step of our entire construction, we will put in this fourth component that "evaluates the word $w_\alpha(b)$ as an element of C".

6.2.3. *Tying the $w_\alpha(b)$-component to the machine component.* The real magic happens at this point. Everything we've done so far has added only finitely many relators, *except*

$$p t(\alpha, 0) p^{-1} = t(\alpha, 0) w_\alpha(b) d$$

Recall that this was unpacking a machine (with a word loaded onto its tape) into a pair of (the machine, the word).

How can we specify this with only finite amounts of information? Well, the word is taken over a finite alphabet $b_1, \ldots, b_n$, so we'd be done if, starting from an empty-tape machine, we had a way of loading a word onto the tape of the "machine" component of the collection one letter at a time, and at the same time appending the word to the "word" component of the collection one letter at a time. This will involve reaching down into the implementation of the machine, which (recall) is currently being held as one component of a collection; the collection itself is just a word over a particular alphabet which we haven't yet specified (we can extract it at the end).

Define a bunch of HNN extensions, one for each b_i, taking $t(\alpha, 0)$ to $t(\alpha m + c_i, 0)$ and $w_\alpha(b)$ to $w_\alpha(b) \cdot b_i$. Here is where we need d: we need to know where the end of $w_\alpha(b)$ is, so that we can append something to it.

Formally, define stable letters U_i such that:

- $U_i b_r U_i^{-1} = b_r$ (dealing with the $w_\alpha(b)$-chunk)

- $U_i d U_i^{-1} = b_i d$ (appending b_i to the abstract word)
- $U_i x U_i^{-1} = x^m$ (shifting the left-hand tape of the machine up by one, to make room for the new symbol)
- $U_i t U_i^{-1} = x^i t x^{-i}$ (filling that new empty slot with the required state)

The miracle of what we have just done is that we can implement all the infinitely-many $pt(\alpha, 0)p^{-1} = t(\alpha, 0)w_\alpha(b)d$ in terms of these finitely-many new relators! To obtain the effect of $pt(\alpha, 0)p^{-1}$ where α represents the word c_3c_6, just conjugate the collection $ptp^{-1} = [t(0, 0), \text{empty word}, d]$ in turn by U_3 and U_6.

6.2.4. *Adding an embedded copy of C.* So far, we have been dealing syntactically with symbols b_i which represent the generators of C. But we haven't actually got an embedded copy of C yet, or at least we haven't obviously got one. What we need is a way of evaluating $w_\alpha(b)$ to obtain $w_\alpha(c) \in C$.

That's easy, though: do one HNN extension that will have the effect of taking b_i to b_ic_i, and insist that the b_i, c_j all commute.

Formally, add a single stable letter V such that:

- $Vb_jV^{-1} = b_jc_j$
- $VdV^{-1} = d$ (so the marker is unchanged)
- $VtV^{-1} = t$, $Vr_iV^{-1} = r_i$, $Vs_jV^{-1} = s_j$ (so the $K_{\mathcal{M}}$ component is unchanged)
- $VpV^{-1} = p$ (so that V's unpacking won't do anything until we have explicitly performed p's unpacking)

and add (finitely many) relators $b_ic_j = c_jb_i$. (This HNN extension is ψ^+ in the course notes.)

Remark. One might wonder why we don't need V to commute with the U_j (which, recall, load letters onto the tape of the machine). The answer is that we only want to do the unpacking right at the end of the procedure: at the "load letters onto tape" stage, we don't do any unpacking. Therefore, in our final uber-group which encodes all of Higman's construction (whose elements are words which are collections), there may be strange fragments of instructions floating around, like in $Vt(\alpha, 0)w_\alpha(b)d$ where V is "half of an instruction". We just let these hang around, and ignore them: Britton's lemma on HNN extensions tells us that none of these are equal to words in which the instructions have all been fully executed (that is, in which no V's appear). While all this junk is still sitting around in the group, it doesn't interfere with the interesting part of the group.

The situation is analogous to the embedding of a modular machine into a group, where we ignore any chunks of half-instruction like $r_ix^ay^bty^{-b}x^{-a}$ (where r_i is an un-completed instruction) because they don't interfere with what we're really doing.

Consider that a word w_α evaluates in C to something trivial if and only if the modular machine halts in the zero state from $t(\alpha, 0)$. (That's how we defined the modular machine.) The modular machine halts in the zero state from $t(\alpha, 0)$ if and only if conjugating by r_i, s_j in some order causes $t(\alpha, 0)$ to be taken to the single letter t. That is, if and only if the machine, starting from state $t(\alpha, 0)$, eventually reaches the state that is the single letter t.

But because V commutes with the r_i and s_j (that is, the "instructions" in $K_{\mathcal{M}}$ telling it to execute the appropiate machine instructions), we have

$$r^i s^j V t(\alpha, 0) V^{-1} s^{-j} r^{-i} = V r^i s^j t(\alpha, 0) s^{-j} r^{-i} V^{-1} = V t(0,0) V^{-1} = t$$

if the machine halts on the zero state through applying instructions r_i, s_j.

That is,

$$V t(\alpha, 0) V^{-1} = r^{-i} s^{-j} t r^i s^j$$

which, since the machine is deterministic (and we're undoing the instructions that took us from $t(\alpha, 0)$ to $t(0,0)$), is $t(\alpha, 0)$.

Otherwise, if the machine doesn't halt through r_i, s_j, there are some interfering x's left at the end (we end up with $V t(a,b) V^{-1} = V x^i y^j t y^{-j} x^{-i} V^{-1}$) so V can't cancel.

The upshot is that conjugating the three-element collection $[K_{\mathcal{M}}, w_\alpha(b), d]$ by V unpacks into a collection

$$[K_{\mathcal{M}}, w_\alpha(b), w_\alpha(c), d]$$

without any V's if and only if $w_\alpha(c)$ is trivial as a member of C.

Example. Suppose

$$C = \mathbb{Z}_2 = \langle c_1 \mid c_1^2 = e \rangle$$

Expand the presentation to

$$\langle c_1, c_2 \mid c_1 c_2 = e, c_1^2 = e, c_1^2 c_2^2 = e, \dots \rangle$$

where the ellipsis indicates every trivial word.

Let us examine c_1^2, which is trivial; it is coded into the machine-component of the group as $t(1+m, 0)$.

There is a sequence of r_i, s_j such that conjugating $t(1+m,0)$ by $r_i s_j$ results in $t(0,0) = t$. To make this more readable, let's say $r_i s_j$ are precisely the two instructions we need to use to do this.

Now, $V t(1+m,0) V^{-1}$ is equal to

$$V r_i s_j t(0,0) s_j^{-1} r_u^{-1} V^{-1}$$

But V commutes with all those terms, so it is just $t(1+m,0)$.

Hence $V t(1+m,0) b_1^2 d V^{-1}$ is

$$t(1+m,0) b_1^2 c_1^2 d$$

because of the effect of V on the b_i terms.

On the other hand, $V t(1+m,0) b_1^2 d V^{-1}$ is

$$V p t(1+m,0) p^{-1} V^{-1}$$

and V commutes with p, so it comes to

$$p t(1+m,0) p^{-1} = t(1+m,0) b_1^2 d$$

Therefore

$$t(1+m,0) b_1^2 c_1^2 d = t(1+m,0) b_1^2 d$$

so c_1^2 is trivial.

6.3. **List of generators.** As a recap, here is a list of all the relators (HNN or otherwise) of the uber-group we have made.

- $r_i t(\alpha, \beta) r_i^{-1} = t$(new machine state) for each L-machine-instruction
- $s_j t(\alpha, \beta) s_j^{-1} = t$(new machine state) for each R-machine instruction
- $pt(\alpha, 0)p^{-1} = t(\alpha, 0)w_\alpha(b)d$, the instructions for unpacking a machine into a machine with its word (all these can be made by other instructions)
- $ptp^{-1} = td$ (a single special case of the above)
- $U_i b_r U_i^{-1} = b_r$ (when loading i onto the tape, deals with the $w_\alpha(b)$-chunk)
- $U_i d U_i^{-1} = b_i d$ (when loading i onto the tape, appends b_i to the abstract word)
- $U_i x U_i^{-1} = x^m$ (when loading i onto the tape, shift the left-hand tape of the machine up by one, to make room for the new symbol)
- $U_i t U_i^{-1} = x^i t x^{-i}$ (filling that new empty slot with the required state)
- $V b_j V^{-1} = b_j c_j$ (unpacking a syntactic word into its represented element)
- $V d V^{-1} = d$
- $V t V^{-1} = t$, $V r_i V^{-1} = r_i$, $V s_j V^{-1} = s_j$ (so the $K_{\mathcal{M}}$ component is unchanged)
- $V p V^{-1} = p$
- $b_i c_j = c_j b_i$

There are finitely many of all of these, except the $pt(\alpha, 0)p^{-1} = t(\alpha, 0)w_\alpha(b)d$ which can be made by other instructions.