Logarithmic Space and Permutations

Clément Aubert, Thomas Seiller

Logarithmic Space and Permutations[☆]

Clément Aubert[a], Thomas Seiller[b]

[a] *LIPN – UMR CNRS 7030 Institut Galilée - Université Paris-Nord 99, avenue J.-B. Clément 93430 Villetaneuse – France*
[b] *Institut des Hautes Études Scientifiques, Le Bois-Marie, 35 Route de Chartres, 91440 Bures-sur-Yvette, France*

Abstract

In a recent work, Girard [1] proposed a new and innovative approach to computational complexity based on the proofs-as-programs correspondence. In a previous paper [2], the authors showed how Girard's proposal succeeds in obtaining a new characterization of co-NL languages as a set of operators acting on a Hilbert Space. In this paper, we extend this work by showing that it is also possible to define a set of operators characterizing the class L of logarithmic space languages.

Keywords: Linear Logic, Complexity, Geometry of Interaction, Pointer Machine, Finite Automata, Logarithmic Space

1. Introduction

1.1. Linear Logic and Implicit Computational Complexity

Logic, and more precisely proof theory – the domain whose purpose is the formalization and study of mathematical proofs – recently yielded numerous developments in theoretical computer science. These developments are founded on a correspondence, often called Curry-Howard correspondence, between mathematical proofs and computer programs (usually formalized in lambda-calculus). The main interest of this correspondence lies in its

[☆] Work partially supported by the ANR projects ANR-2010-BLAN-021301 LOGOI and ANR-08-BLAN-0211-01 COMPLICE.

Email addresses: aubert@lipn.fr (Clément Aubert), seiller@ihes.fr (Thomas Seiller)

dynamic nature: program execution corresponds to a procedure on mathematical proofs known as the cut-elimination procedure.

In the eighties, Jean-Yves Girard discovered linear logic through a study of mathematical models of the lambda-calculus. This logical system, as a direct consequence of this correspondence between proofs and programs, is particularly interesting from the point of view of the mathematical foundations of computer science for its resource-awareness. In particular, it gave birth to a number of developments in the field of implicit computational complexity, for instance through the definition and study of restricted logical systems (sub-systems of linear logic) in which the set of representable functions captures a complexity class. For instance, elementary linear logic (ELL) restricts the rules governing the use of exponential connectives – the connectives dealing with the duplication of the arguments of a function – and the set of representable functions in ELL is exactly the set of elementary time functions [3]. It was also shown [4] that a characterization of logarithmic space computation can be obtained if one restricts both the rules of exponential connectives and the use of universal quantifiers. Finally, a variation on the notion of linear logic *proof nets* succeeded in characterizing the classes **NC** of problems that can be efficiently parallelized [5].

1.2. Geometry of Interaction

A deep study of the formalization of proofs in linear logic, in particular their formalization as proof nets, led Jean-Yves Girard to initiate a program entitled *geometry of interaction* (GoI) [6]. This program, in a first approximation, intends to define a semantics of proofs that accounts for the dynamics of the cut-elimination procedure. Through the correspondence between proofs and programs, this would define a semantics of programs that accounts for the dynamics of their execution. However, the geometry of interaction program is more ambitious: beyond the mere interpretation of proofs, its purpose is to completely reconstruct logic around the dynamics of the cut-elimination procedure. This means reconstructing the logic of programs, where the notion of formula – or type – accounts for the behavior of algorithms.

Informally, a geometry of interaction is defined by a set of *paraproofs* together with a notion of interaction, in the same way one defines strategies and their composition in game semantics. An important tool in the construction is a binary function that measures the interaction between two paraproofs. With this function one defines a notion of orthogonality that corresponds to the negation of logic and reconstructs the formulas as sets of paraproofs equal

to the orthogonal of a given set of paraproofs: a formula is therefore a set of "programs" that interact in a similar way to a given set of *tests*.

Since the introduction of this program Jean-Yves Girard proposed different constructions to realize it. These constructions share the notion of paraproofs: operators in a von Neumann algebra. They however differ on the notion of orthogonality they use: in the first constructions, this notion was founded on the nilpotency of the product of two operators, while the more recent construction [7] uses Fuglede-Kadison determinant – a generalization of the usual determinant of matrices that can be defined in type II_1 factors.

Since the reconstruction of logic is based on the notion of execution, geometry of interaction constructions are particularly interesting for the study of computational complexity. It is worth noting that the first construction of GoI [8] allowed Abadi, Gonthier, and Lévy [9] to explain the optimal reduction of λ-calculus defined by Lamping [10]. This first GoI construction was also used to obtain results in the field of implicit computational complexity [11].

1.3. A new approach to complexity

Recently Jean-Yves Girard proposed a new approach for the study of complexity classes that was inspired by his latest construction of a geometry of interaction. Using the crossed product construction between a von Neumann algebra and a group acting on it, he proposed to characterize complexity classes as sets of operators obtained through the internalization of outer automorphisms of the type II_1 hyperfinite factor. The authors showed in a recent paper [2] that this approach succeeds in defining a characterization of the set of **co-NL** languages as a set of operators in the type II_1 hyperfinite factor. The proof of this result was obtained through the introduction of *non-deterministic pointer machines*, which are abstract machines designed to mimic the computational behavior of operators. The result was obtained by showing that a **co-NL** complete problem could be solved by these machines.

In this paper, we extend these results in two ways. The first important contribution is that we give an alternative proof of the fact that **co-NL** is indeed characterized by non-deterministic pointer machines. This new proof consists in showing that pointer machines can simulate the well-known and studied two-way multi-head finite automata [12, 13]. The second contribution of this paper consists in obtaining a characterization of the class **L** as a set of operators in the hyperfinite factor of type II_1. By studying the set of operators that characterize the class **co-NL** and in particular the encoding of non-deterministic pointer machines as operators, we are able to show that

the operators encoding a deterministic machine satisfy a condition expressed in terms of norm. We then manage to show that the language decided by an operator satisfying this norm condition is in the class **L**, showing that the set of all such operators characterizes **L**.

2. The Basic Picture

The construction uses an operator-theoretic construction known as the crossed product of an algebra by a group acting on it. The interested reader can find a quick overview of the theory of von Neumann algebras in the appendix of the second author's work on geometry of interaction [14], and a brief presentation of the crossed product construction in the authors' previous work [2] on the characterization of **co-NL**. For a more complete presentation of the theory of operators and the crossed product construction, we refer to the well-known series of Takesaki [15, 16, 17].

In a nutshell, the crossed product construction $\mathfrak{A} \rtimes_\alpha G$ of a von Neumann algebra $\mathfrak{A}$ and a group action $\alpha : G \to \mathrm{Aut}(\mathfrak{A})$ defines a von Neumann algebra containing $\mathfrak{A}$ and unitaries that internalize the automorphisms $\alpha(g)$ for $g \in G$. For this, one considers the Hilbert space[1] $\mathbb{K} = L^2(G, \mathbb{H})$ where $\mathbb{H}$ is the Hilbert space $\mathfrak{A}$ is acting on, and one defines two families of unitary operators[2] in $\mathscr{L}(\mathbb{K})$:

- the family $\{\pi(u) \mid u \in \mathfrak{A} \text{ unitary}\}$ which is a representation of $\mathfrak{A}$ on $\mathbb{K}$;
- the family $\{\lambda(g) \mid g \in G\}$ which contains unitaries that internalize the automorphisms $\alpha(g)$.

Then the crossed product $\mathfrak{A} \rtimes_\alpha G$ is the von Neumann algebra generated by these two families.

---

[1] The construction $L^2(G, \mathbb{H})$ is a generalization of the well-known construction of the Hilbert space of square-summable functions: in case G is considered with the discrete topology, the elements are functions $f : G \to \mathbb{H}$ such that $\sum_{g \in G} \|f(g)\|^2 < \infty$.

[2] Recall that in the algebra $\mathscr{L}(\mathbb{H})$ of bounded linear operators on the Hilbert space $\mathbb{H}$ (we denote by $\langle \cdot, \cdot \rangle$ its inner product), there exists an anti-linear involution $(\cdot)^*$ such that for any $\xi, \eta \in \mathbb{H}$ and $A \in \mathscr{L}(\mathbb{H})$, $\langle A\xi, \eta \rangle = \langle \xi, A^* \eta \rangle$. This *adjoint operator* coincides with the conjugate-transpose in the algebras of square matrices. A *unitary operator* u is an operator such that $uu^* = u^*u = 1$.

As a by-product of Girard's work on geometry of interaction [7], we know[3] how to represent integers as operators in the type II_1 hyperfinite factor $\mathfrak{R}$.

This representation comes from logical considerations, and it has some specificities, among which the fact that the integer is represented as a *circular* binary string, and the fact that every bit is considered as having an input and an output[4]. This representation arises from the interpretation of proofs of the type $\forall X!(X \multimap X) \multimap (!(X \multimap X) \multimap !(X \multimap X))$ (the type of binary lists in Elementary Linear Logic) in the setting of geometry of interaction. The obtained interpretation is the image of a matrix through an embedding of the matrix algebra in which it is contained in the hyperfinite factor. Since there are no natural choice of a particular embedding, a given integer does not have a unique representation in $\mathfrak{R}$, but a family of representations. Luckily, any two representations of a given integer are unitarily equivalent (we denote by $\mathfrak{M}_k(\mathbf{C})$ the algebra of $k \times k$ matrices over $\mathbf{C}$):

Proposition 1 ([2, Proposition 10])**.** *Let $N, N' \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{R}$ be two representations of a given integer. There exists a unitary $u \in \mathfrak{R}$ such that:*

$$N = (1 \otimes u)^* N' (1 \otimes u)$$

The next step is then to define a representation of machines, or algorithms, as operators that do not discriminate two distinct representations of a given integer, i.e. operators that act uniformly on the set of representations of a given integer. As the authors have shown in their previous work, the crossed product construction allows one to characterize an algebra of operators acting in such a uniform way. We recall the construction in the particular case of the group of finite permutations, which will be the only setting that will be studied in this work.

The algebra $\mathfrak{R}$ can be embedded in the infinite tensor product algebra $\mathfrak{T} = \bigotimes_{n \in \mathbf{N}} \mathfrak{R}$ through the morphism $\iota_0 : x \mapsto x \otimes 1 \otimes 1 \otimes \dots$. We will denote by $\mathfrak{N}_0$ the image of $\mathfrak{R}$ in $\mathfrak{T}$ through ι_0. Notice that this tensor product algebra is isomorphic to $\mathfrak{R}$. The idea of Girard is then to use the action of the group $\mathscr{S}$ of finite permutations of $\mathbf{N}$ onto $\mathfrak{T}$, defined by:

$$\sigma.(x_0 \otimes x_1 \otimes \dots \otimes x_k \otimes \dots) = x_{\sigma^{-1}(0)} \otimes x_{\sigma^{-1}(1)} \otimes \dots \otimes x_{\sigma^{-1}(k)} \otimes \dots$$

[3] A detailed explanation of this representation can be found in Appendix A, in the authors' previous work [2] or in the second author's PhD thesis [18].

[4] Something that will be proven useful in Section 5, for it will help us to determine in which direction the integer is read.

This group action defines a sub-algebra $\mathfrak{G}$ of the crossed product algebra $\mathfrak{K} = (\bigotimes_{n\in\mathbf{N}} \mathfrak{R}) \rtimes \mathscr{S}$. This algebra $\mathfrak{G}$ is the algebra generated by the family of unitaries $\lambda(\sigma)$ for $\sigma \in \mathscr{S}$, and we think of it as the algebra of machines, or algorithms. As it turns out, the elements of this algebra act uniformly on the set of representations of a given integer:

Proposition 2 ([2, Proposition 11])**.** *Let $N, N' \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{N}_0$ be two representations of a given integer, and $\phi \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G}$. Then:*

$$\phi N \text{ is nilpotent iff } \phi N' \text{ is nilpotent}$$

To be a bit more precise, we will be interested in the elements of the algebras of operators $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{M}_k(\mathbf{C})$. By tensoring with a matrix algebra, which we call the *algebra of states*, we will be able to develop more computational power. An element of one of these algebras will be called an *observation*, and it still acts uniformly on distinct representations N_n, N'_n of a given integer: if ϕ is an observation, $\phi(N_n \otimes 1_{\mathfrak{M}_k(\mathbf{C})})$ is nilpotent if and only if $\phi(N'_n \otimes 1_{\mathfrak{M}_k(\mathbf{C})})$ is nilpotent[5]. From this proposition, one can justify that the following definition makes sense:

Definition 3. Let $\phi \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{M}_k(\mathbf{C})$ be an observation. We define the language accepted by ϕ by (N_n denotes any representation of the integer n):

$$[\phi] = \{n \in \mathbf{N} \mid \phi(N_n \otimes 1) \text{ nilpotent}\}$$

By extension, if P is a set of observations, we denote by $[P]$ the set $\{[\phi] \mid \phi \in P\}$.

To sum up the construction, one has the following objects:

- an algebra containing representations of integers: the hyperfinite factor $\mathfrak{R}$ of type II_1, embedded in $\mathfrak{K}$ through the morphism $\pi \circ \iota_0$;
- an algebra containing the representations of machines, or algorithms: the von Neumann sub-algebra $\mathfrak{G}$ of $\mathfrak{K}$ generated by the set of unitaries $\{\lambda(\sigma) \mid \sigma \in \mathscr{S}\}$;
- a notion of acceptance: if N is a representation of a integer and ϕ is a representation of a machine, we say that ϕ accepts N when ϕN is nilpotent.

[5] We will in the following simply write 1 for the identity element of $\mathfrak{M}_k(\mathbf{C})$.

3. Pointer Machines

We previously introduced [2] non-deterministic pointers machines that were designed to mimic the computational behavior of operators: they do not have the ability to write, their input tape is cyclic, and they are "universally non-deterministic", i.e. rejection is meaningful whereas acceptation is the default behavior. We exhibited a non-deterministic pointer machine that decides a **co-NL**-complete language and showed how a reduction could be performed with pointers. Here, we complete this result by simulating multi-head finite automata [19], a method that allows us to get a deeper understanding of pointer machines, in both their deterministic and non-deterministic versions.

A pointer machine is given by a set of pointers that can move back and forth on the input tape and read (but not write) the values it contains, together with a set of states. For $1 \leqslant i \leqslant p$, given a pointer p_i, only one of three different *instructions* can be performed at each step: p_i+, i.e. "move one step forward", p_i-, i.e. "move one step backward" and ϵ_i, i.e. "do not move". In the following definition, we let $I^k = \{p_k+, p_k-, \epsilon_k\}$ be the set of instructions for the k-th pointer and $\Sigma = \{0, 1, \star\}$ be the alphabet. We will denote by $\#p_i$ the address of the pointer.

Definition 4. A non-deterministic pointer machine with $p \geqslant 1$ pointers is a couple $M = \{Q, \rightarrow\}$ where Q is the set of *states* and $\rightarrow \subseteq (\Sigma^p \times Q) \times \prod_{i=1}^{p} I^i \times Q) \cup \{\textbf{accept, reject}\})$ is the *transition relation* and it is total. We write NDPM(p) the set of non-deterministic pointer machines with p pointers.

A configuration of $M \in \text{NDPM}(p)$ is as usual a "snapshot" of M at a given time, and we define a *pseudo-configuration* c of $M \in \text{NDPM}(p)$ as a "partial snapshot": $c \in \Sigma^p \times Q$ contains the last values read by the p pointers and the current state, *but does not contain the addresses of the p pointers*. It is therefore impossible to resume the computation from a pseudo-configuration provided the description of M and the input, but one can know what would be the set of instructions and the new state resulting from the application of $\rightarrow$. The set of pseudo-configurations of a machine M is written C_M and it is the domain of the transition relation. If $\rightarrow$ is functional, M is a *deterministic* pointer machine. We write DPM(p) the set of deterministic pointer machines with p pointers.

Let $M \in \text{NDPM}(p)$, $s \in C_M$ and $n \in \mathbf{N}$ an input. We define $M_s(n)$ as M with

n encoded as a string[6] on its circular input tape (as $\star a_1 \dots a_k$ for $a_1 \dots a_k$ the binary encoding of n and $a_{k+1} = a_0 = \star$) starting in the pseudo-configuration s with $\#p_i = 0$ for all $1 \leqslant i \leqslant p$ (that is, the pointers are initialized with the address of the symbol $\star$). The pointers may be considered as variables that have been declared but not initialized yet. They are associated with *memory slots* that store the values and are updated only when the pointer moves, so as the pointers did not move yet, those memory slots haven't been initialized. The *initial pseudo-configuration* s initializes those p registers, not necessarily in a faithful way (it may not reflect the values contained at $\#p_i$). The entry n is *accepted* (resp. *rejected*) by M with *initial pseudo-configuration* $s \in C_M$ if after a finite number of transitions every branch of $M_s(n)$ reaches **accept** (resp. at least a branch of M reaches **reject**). We say that $M_s(n)$ halts if it accepts or rejects n and that M decides a set S if there exists a pseudo-configuration $s \in C_M$ such that $M_s(n)$ accepts if and only if $n \in S$.

As we will see, this notion of non-deterministic pointer machines is similar to the classical notion of multi-head finite automata:

Definition 5. [13, Definition 1] For $k \geqslant 1$, a non-deterministic two-way k-head finite automaton is a tuple $M = \{S, A, k, \triangleright, \triangleleft, s_0, F, \sigma\}$ where:

- S is the finite set of *states*;
- A is the *alphabet*[7];
- k is the number of heads;
- $\triangleright$ and $\triangleleft$ are the *left and right endmarkers*;
- $s_0 \in S$ is the *initial state*;
- $F \subseteq S$ is the set of *accepting states*;
- $\sigma \subseteq (S \times (A \cup \{\triangleright, \triangleleft\})^k) \times (S \times \{-1, 0, 1\}^k)$ is the *transition relation*, where -1 means to move the head one square to the left, 0 means to keep the

[6] Of course, one could do the exact same work taking binary words instead of integers. This choice of presentation was originally driven by Girard's will to enlighten that, among the infinitely many isomorphic representations of an integer, "none of them [was] more 'standard' than the others." [1, p. 244]

[7] We take it to be always equal to $\{0, 1\}$, as any alphabet is equivalent to this one modulo a reasonable translation.

head on the current square and 1 means to move it one square to the right.

Moreover, whenever[8] $(s',(d_1,\ldots,d_k)) \in \sigma(s,(a_1,\ldots,a_k))$, then $a_i = \triangleright$ $(1 \leqslant i \leqslant k)$ implies $d_i \in \{0,1\}$, and $a_i = \triangleleft$ $(1 \leqslant i \leqslant k)$ implies $d_i \in \{-1,0\}$.

In the following, we will denote by 2NDFA(k) the set of non-deterministic two-ways k-head finite automata.

One defines configurations and transitions in a classical way. One should however notice that the heads cannot move beyond the endmarkers because of the definition of the transition relation.

Let $M \in$ 2NDFA(k) and $n \in \mathbf{N}$ an input whose binary writing is w. We say that M *accepts* n if M starts in state s_0, with $\triangleright\, w \,\triangleleft$ written on its input tape and all of its heads on $\triangleright$, and if after a finite number of transitions at least one branch of M reaches a state belonging to F. We say that M always halt if for all input all branches of M reach after a finite number of transitions a configuration such that no transition may be applied. If σ is functional, M is said to be *deterministic*. We write 2DFA(k) the set of deterministic two-way k heads automata.

4. Simulation and first results

Multi-head finite automata were introduced in the early seventies [12] and provided many promising perspectives[9]. Their nice characterization of **L** and **NL** and the similarities they share with non-deterministic pointer machines will be the key to prove (anew) that non-deterministic pointer machines characterize **co-NL** and that their deterministic restriction characterizes **L**.

We will denote by **DPM**, **NDPM**, 2**DFA** and 2**NDFA** the sets of languages decided by respectively the sets $\cup_{k\geqslant 1}$DPM(k), $\cup_{k\geqslant 1}$NDPM(k), $\cup_{k\geqslant 1}$2DFA(k) and $\cup_{k\geqslant 1}$2NDFA(k).

Theorem 6 ([12], p. 338)**.** ***L*** = 2***DFA*** *and* ***NL*** = 2***NDFA***.

We will denote in the following by $\mathscr{L}(X)$ (resp. $\overline{\mathscr{L}(X)}$) the language accepted (resp. rejected) by X.

[8] Of course, one may see σ as a partial function from $S \times (A \cup \{\triangleright,\triangleleft\})^k$ to $\mathscr{P}(S \times \{-1,0,1\}^k)$, and that justifies this notation that we will use from now on.

[9] An overview of the main results and open problems of the theory was recently published [13].

Proposition 7. *For all $M \in 2NDFA(k)$ (resp. $M \in 2DFA(k)$), there exists k' and $M' \in 2NDFA(k')$ (resp. $M' \in 2DFA(k')$) such that M' always halt and $\mathscr{L}(M) = \mathscr{L}(M')$.*

Proof. Given an input $n \in \mathbf{N}$, we know that the number of configurations of M is bounded by $\mathrm{Card}(S) \times (\log_2(n)+2)^k$, that is to say by $\log_2(n)^d$ for d a constant fixed with M. We will construct M' so that it halts rejecting after performing more than this number of transitions.

We set $k' = k + d + 1$, and we construct M' so that its k first heads act as the heads of M, and the $d+1$ heads act as the hands of a clock, going back and forth between the two endmarkers. For all $s \in S$ of M, S' contains the set of states $s \times \{\rightarrow, \leftarrow\}^{d+1}$ that give the current direction of the $d+1$ heads. At every transition the $k+1$-th head moves according to its direction. For $k < i < k'$, when the i-th head has made a round-trip on n, the $i+1$-th head moves one square according to its direction. If the k'-th head has made a round-trip on n – that is, is back on $\triangleright$ – M' halts rejecting[10].

Now remark that if a branch of M accepts, there exists a branch of M that accepts without ever looping, i.e. without going through the same configuration twice. Moreover, branches without loops are of length at most $\mathrm{Card}(S) \times (\log_2(n)+2)^k$. Thus, if M has no branches of length less than $\log_2(n)^d$ for the suitable d, then it has no accepting branch. By construction, M' always halts after $\log_2(n)^{d+1}$ transitions and accepts exactly as M does, so we proved that $\mathscr{L}(M) = \mathscr{L}(M')$ and that M' always halts. Notice that if M was deterministic, so is M'. □

Proposition 8. *For all $k \geqslant 1$, for all $M \in 2NDFA(k)$ (resp. $M \in 2DFA(k)$) that always halts, there exists $M' \in NDPM(k)$ (resp. $M' \in 2DFA(k)$) such that $\overline{\mathscr{L}(M)} = \mathscr{L}(M')$.*

Proof. There is only little work to do, since NDPMs are essentially "re-arrangements" of NDFAs. Given $M = \{S, A, k, \triangleright, \triangleleft, s_0, F, \sigma\}$, we design $M' = \{Q, \rightarrow\} \in \mathrm{NDPM}(k)$ that rejects an input iff M accepts it. We set $Q = S \setminus F$. The transition relation $\rightarrow$ is obtained from σ as follows:

- If the state of the resulting configuration of a transition belongs to F, the same transition belongs to $\rightarrow$ with its right-hand side replaced by **reject**;

[10] That can be simply done by halting the computation in a state not belonging to F.

- The role of both endmarkers $\triangleright$ and $\triangleleft$ is played by the symbol $\star$;
- The instructions $-1, 0$ and 1 in position $i \in \mathbf{N}$ are translated by p_i-, ϵ_i and p_i+.

There is one point that needs some care. If a transition t_1 in σ have for premise $(q,(\ldots,\triangleleft,\ldots))$ and another one t_2 have for premise $(q,(\ldots,\triangleright,\ldots))$, we cannot translate in $\rightarrow$ both with $(q,(\ldots,\star,\ldots))$: that would introduce non-determinism in deterministic machines, and could lead to false positive as well as false negative. Remark that this "critical pair" may involve more than one pointer.

If such a situation occurs in σ, one may patch it by creating duplicates of each states, to label them with the last directions of each pointer. This modification provokes a global rewriting of $\rightarrow$ to take into account new states, but do not raise any particular difficulty. Thanks to a slight overhead in the number of state, it is possible to encode if $\star$ was read from the right (and hence encode $\triangleright$) or from the left (and hence encode $\triangleleft$).

At last, to make $\rightarrow$ total, one just have to complement it by adding, from any pseudo-configuration not in the domain of definition of σ, transitions leading to **accept**. Finally, we chose the initial pseudo-configuration c to be $\{\star,\ldots,\star,s_0\}$.

As M always halts, rejection is exactly "not accepting", and M' always halts. One can check that $M'_c(n)$ accepts iff $M(n)$ rejects. Moreover, if M is deterministic, so is M'. □

This proposition has as an immediate corollary the inclusions **co-2DFA** $\subseteq$ **DPM** and **co-2NDFA** $\subseteq$ **NDPM**. By combining this result with Theorem 6 and the fact that **L** is closed under complementation[11], we obtain the expected result:

Corollary 9. ***L*** $\subseteq$ ***DPM*** *and* ***co-NL*** $\subseteq$ ***NDPM***.

The converse inclusion will be a by-product of the encoding of NDPMs into operators and won't be proved directly, but it is reasonable to think of NDPMs

[11] Using the fact that **NL** = **co-NL** [20], we could show that **NL** $\subset$ **NDPM**. However, we choose not to use the well-known equality between **NL** and **co-NL** in the hope that this new approach to complexity may provide a new proof of this result. Moreover, it is easier to grasp the computation of observations as "universally non-deterministic".

as "modified" 2NFAs. Those modifications are introduced to ease the simulation by operators: we already mentioned that the transition relation was total, the input circular, the importance of the initial pseudo-configuration to initialize the computation, but we should also mention that any NDPM can be modified so that it does not move more than one pointer at each transition.

5. Pointer Machines and Operators

In this section, we will briefly explain the encoding of pointer machines as operators. The non-deterministic case, which was already defined in our previous work [2], will be given in Section 5.1. The specifics of the deterministic case, which is a novelty, concludes in Section 5.2 this section. An example, detailed in Appendix A, will help the reader to grasp the main steps of that encoding.

5.1. Encoding NDPMs

We will begin by a definition of two families of observations.

Definition 10. Let $\phi \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{M}_k(\mathbf{C})$ be an observation that we will write as a $6k \times 6k$ matrix $(a_{i,j})_{1 \leqslant i,j \leqslant 6k}$ with coefficients in $\mathfrak{G}$. We say that ϕ is:

- *A positive observation* if for all $1 \leqslant i, j \leqslant 6k$, we have $a_{i,j} = \sum_{m=0}^{l} \alpha_m^{i,j} \lambda(g_m^{i,j})$ where l is an integer (possibly null), and $\alpha_m^{i,j}$ are positive real numbers.
- *A boolean observation* if for all $1 \leqslant i, j \leqslant 6k$, we have $a_{i,j} = \sum_{m=0}^{l} \lambda(g_m^{i,j})$ where l is an integer (possibly null).

Definition 11. We define the following sets:

$$P_{\geqslant 0} = \{\phi \mid \phi \text{ is a positive observation}\}$$
$$P_{+} = \{\phi \mid \phi \text{ is a boolean observation}\}$$

We showed [2] that any non-deterministic pointer machine M can be encoded as an operator in P_+, implying that **co-NL** $\subseteq [P_+]$.

The encoding of non-deterministic pointer machines was defined as follows: we encode each couple $(c,t) \in \rightarrow$ by an operator $\phi_{c,t}$, and then the encoding of the transition relation corresponds to the sum:

$$\rightarrow^{\bullet} = \sum_{c \in C_M} \left(\sum_{t \text{ s.t. } c \rightarrow t} \phi_{c,t} \right)$$

To define the encoding correctly, we need to introduce a number of additional states. We will denote by $Q^{\uparrow}$ the extended set of states obtained from Q. According to the notation introduced in Figs. 1 and 2, this set has cardinality $\mathrm{Card}(Q) + p \times (\mathrm{Card}(Q)^2 + 2)$ and is (a subset of):

$$Q^{\uparrow} = Q \cup \bigcup_{j=1}^{p} \Big(\{\mathbf{mov\text{-}back}_j, \mathbf{back}_j\} \cup \{\mathbf{mov}_j^{\mathbf{q},\mathbf{q'}} \mid \mathbf{q}, \mathbf{q'} \in Q\} \Big)$$

The machine will then be encoded as an operator in $\mathfrak{M}_6(\mathfrak{G}) \otimes \mathfrak{P}$, where $\mathfrak{P}$ is the algebra of *pseudo-states*, i.e. an algebra that will encode the memory slots that contain the last value read by the pointers, and the extended set of states $Q^{\uparrow}$. That is:

$$\mathfrak{P} = \underbrace{\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{M}_6(\mathbf{C}) \otimes \cdots \otimes \mathfrak{M}_6(\mathbf{C})}_{p \text{ times}} \otimes \mathfrak{M}_{\mathrm{Card}(Q^{\uparrow})}(\mathbf{C})$$

Summing up, the encoding of a non-deterministic pointer machine with a set of states Q will be an observation in the algebra $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{M}_k(\mathbf{C})$ with $k = 6^p \times \mathrm{Card}(Q^{\uparrow}) = 6^p \times (\mathrm{Card}(Q) + p \times (\mathrm{Card}(Q)^2 + 2)$.

The following encoding differs from the one used in our earlier work [2]. The reason for this (small) change is that the encoding of the "move forward" and "move backward" instructions were defined as operators whose norm was strictly greater than 1. Since we want the encoding of a deterministic machine to be of norm at most 1 (see the next subsection), it is necessary to define a more convenient encoding. The encoding of acceptance and rejection are not modified, but we recall them anyway. The reader should also keep in mind that the encoding of the integer is somehow redundant, for every bit will have an input and an output: this will help us to discriminate between the movements of a pointer from right to left and the movements from left to right.

Before describing the encoding in details, we need to introduce several notations that will be used for the definition of operators. We recall that the integer is encoded as a circular string, whose start and end is $\star$, and that every bit has an output that is connected to the input of the following bit. So for instance "reading" $0i$ amounts to asking (the integer) what is the bit before the bit 0 under treatment, "reading" $\star o$ amounts to asking what is the first bit of the integer. The way the "pointer" will parse the entry is highly interactive, or dynamic, for it is always on the edge of reading the next bit. So we define the projections π_* of $\mathfrak{M}_6(\mathbf{C})$ ($* \in \{0i, 0o, 1i, 1o, e, s\}$) as the projections on the

subspace induced by the basis element[12]. We will sometimes denote e (resp. s) by $\star i$ (resp. $\star o$), and we also define the projections $\pi_{\text{in}} = \pi_{0\text{i}} + \pi_{1\text{i}} + \pi_e$ and $\pi_{\text{out}} = \pi_{0\text{o}} + \pi_{1\text{o}} + \pi_s$. Those projections correspond to the following matrices.

$$\pi_{\text{in}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \pi_{\text{out}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We will moreover use matrices named (in $\to$ out) and (out $\to$ in), shown below. Those matrices will be used for continuing a movement in the same direction as the last movement, i.e. if the integer answered that the last read value was a 0, it will answer on the basis element $0i$ – for "0 in" – if the pointer is moving forward. Then, to ask the next value, the pointer needs to ask the next value on the basis element $0o$ – for "0 out". To go from $0i$ to $0o$, one can apply the matrix (in $\to$ out). The matrix (out $\to$ in) plays the same role when moving in the opposite direction. Moreover, changes of directions are dealt with the projections π_{in} and π_{out}.

$$(\text{in} \to \text{out}) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad (\text{out} \to \text{in}) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

For the sake of simplicity, we also define the following operators in $\mathfrak{P}$: if $c = (a_1, \ldots, a_p, \mathbf{q})$ and $c' = (a'_1, \ldots, a'_p, \mathbf{q}')$, we define the partial isometry:

$$(c \to c') = (a_1 \to a'_1) \otimes \cdots \otimes (a_p \to a'_p) \otimes (\mathbf{q} \to \mathbf{q}')$$

[12]To be coherent with the notations of our earlier paper, we will always denote by $\{0i, 0o, 1i, 1o, e, s\}$ the basis of $\mathfrak{M}_6(\mathbf{C})$ we are working with.

where $(p \to p')$ (for $p, p' \in Q^{\uparrow}$ or $p, p' \in \{0i, 0o, 1i, 1o, e, s\}$) is defined as:

$$\begin{array}{cc} & p \\ p' & \begin{pmatrix} 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ 0 & \dots & 1 & \dots & 0 \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 \end{pmatrix} \end{array}$$

For $\mathbf{S}$ a set of states, we will use the notation $(\mathbf{S} \to a'_i)$ (denoted $(\to a'_i)$ when $\mathbf{S}$ contains all possible states) for the sum of partial isometries $\sum_{s \in \mathbf{S}} (s \to a'_i)$ that encodes a transition from any state in $\mathbf{S}$ to a'_i. The notation $(\mathbf{S} \to)$ will denote the projection on the subspace generated by $\mathbf{S}$.

A transition that impacts only on the values stored in the subset of pointers[13] $p_{i_1}, \dots, p_{i_l}$ and the state $\mathbf{q}$ will be denoted by

$$((a_{i_1} \to a'_{i_1})_{i_1}; \dots; (a_{i_l} \to a'_{i_l})_{i_l}; \mathbf{q} \to \mathbf{q'})$$

This operator is defined as

$$u_1 \otimes u_2 \otimes \cdots \otimes u_p \otimes (\mathbf{q} \to \mathbf{q'})$$

where $u_i = (a_{i_j} \to a'_{i_j})$ if $\exists j, i = i_j$, $u_i = 1$ elsewhere.

Finally, we will denote by $\tau_{i,j}$ the operator in $\mathfrak{G}$ induced by the transposition exchanging the integers i and j.

Now, we can explain how the different transitions are encoded. Figures 1 and 2 give the definitions of the operators used in the following description.

Move a pointer, read a value and change state. We first explain how to encode the action "move forward the pointer j". When we are in the pseudo-configuration $c = (a_1, \dots, a_p; \mathbf{q})$, the machine, during this transition, moves the pointer j one step to the right, reads the value a'_j stored at $\#p_j$, updates the memory slot in consequence and changes the state from $\mathbf{q}$ to $\mathbf{q}'$. There are two cases: either the last movement of the pointer j was a forward move,

[13] In fact this subset will always be a singleton, for our encoding corresponds to a NDPM that moves at most one pointer at each transition and any NDPM can be replaced by another NDPM that moves its pointers one at a time.

$$
\begin{aligned}
\mathrm{bf}^{c}_{j,\mathbf{q'}} &= \pi_{\mathrm{out}} \otimes \tau_{0,j} \otimes ((\{0o, 1o, \star o\} \rightarrow)_j ; \mathbf{q} \rightarrow \mathbf{mov}^{\mathbf{q},\mathbf{q'}}_j) \\
\mathrm{ff}^{c}_{j,\mathbf{q'}} &= (\mathrm{in} \rightarrow \mathrm{out}) \otimes \tau_{0,j} \otimes ((\{0i, 1i, \star i\} \rightarrow)_j ; \mathbf{q} \rightarrow \mathbf{mov}^{\mathbf{q},\mathbf{q'}}_j) \\
\mathrm{fb}^{c}_{j,\mathbf{q'}} &= \pi_{\mathrm{in}} \otimes \tau_{0,j} \otimes ((\{0i, 1i, \star i\} \rightarrow)_j ; \mathbf{q} \rightarrow \mathbf{mov}^{\mathbf{q},\mathbf{q'}}_j) \\
\mathrm{bb}^{c}_{j,\mathbf{q'}} &= (\mathrm{out} \rightarrow \mathrm{in}) \otimes \tau_{0,j} \otimes ((\{0o, 1o, \star o\} \rightarrow)_j ; \mathbf{q} \rightarrow \mathbf{mov}^{\mathbf{q},\mathbf{q'}}_j) \\
\mathrm{rec}^{c}_{j,\mathbf{q'}} &= \sum_{\bullet \in \{0,1,\star\}} \left(\pi_{\bullet i} \otimes \tau_{0,j} \otimes ((\rightarrow \bullet i)_j ; \mathbf{mov}^{\mathbf{q},\mathbf{q'}}_j \rightarrow \mathbf{q'}) \right)
\end{aligned}
$$

Figure 1: The operators in $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{P}$ encoding forward and backward move instructions

$$
\begin{aligned}
\mathrm{rm}_j &= 1 \otimes \tau_{0,j} \otimes (\mathbf{back}_j \rightarrow \mathbf{mov\text{-}back}_j) \\
\mathrm{rr}^0_j &= \pi_{0\mathrm{o}} \otimes \tau_{0,j} \otimes ((\rightarrow 0o)_j ; \mathbf{mov\text{-}back}_j \rightarrow \mathbf{back}_j) \\
\mathrm{rr}^1_j &= \pi_{1\mathrm{o}} \otimes \tau_{0,j} \otimes ((\rightarrow 1o)_j ; \mathbf{mov\text{-}back}_j \rightarrow \mathbf{back}_j) \\
\mathrm{rc}_j &= \begin{cases} \pi_{\star\mathrm{o}} \otimes \tau_{0,j} \otimes ((\rightarrow a_j)_j ; \mathbf{mov\text{-}back}_j \rightarrow \mathbf{back}_{j+1}) & (1 \leqslant j < p) \\ \pi_{\star\mathrm{o}} \otimes \tau_{0,p} \otimes ((\rightarrow a_p)_p ; \mathbf{mov\text{-}back}_p \rightarrow \mathbf{q}_0) & (j = p) \end{cases}
\end{aligned}
$$

Figure 2: The operators in $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{P}$ encoding rejection.
The operator rc_j is parametric on an initial pseudo-configuration $s = (a_1, \ldots, a_p ; \mathbf{q}_0)$.

or it was a backward move. The case we are dealing with is obtained from the value stored in memory slot of the j-th pointer: if the value is $0i$, $1i$ or $\star i$, then the last move was a forward move, if the value is $0o$, $1o$ or $\star o$, the last move was a backward move. In the first case, the operator $\mathrm{ff}^c_{j,\mathbf{q}'}$ will be applied (notice the projection π_{in} on the j-th memory slot), and in the second the $\mathrm{bf}^c_{j,\mathbf{q}'}$ operator will be applied. Both these operators somehow *activate* the j-th pointer by using the transposition $\tau_{0,j}$ and prepare for reading the representation of the integer. This representation will then give the value of the next (when moving forward) digit of the input. The $\mathrm{rec}^c_{j,\mathbf{q}'}$ operator is then applied in order to simultaneously update the value of the j-th memory slot and *deactivate* the j-th pointer.

The operator that will encode the moving forward instruction is then defined as $\mathrm{forward}^c_{j,\mathbf{q}'} = \mathrm{bf}^c_{j,\mathbf{q}'} + \mathrm{ff}^c_{j,\mathbf{q}'} + \mathrm{rec}^c_{j,\mathbf{q}'}$ which is an element of the algebra $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{P}$.

In case of a "move backwards pointer j" instruction, the operator encoding the transition is $\mathrm{backward}^c_{j,\mathbf{q}'} = \mathrm{fb}^c_{j,\mathbf{q}'} + \mathrm{bb}^c_{j,\mathbf{q}'} + \mathrm{rec}^c_{j,\mathbf{q}'}$, one again an element of the algebra $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{P}$.

In the following, we will forget about the subscripts and superscripts of these operators[14] and write ff, fb, bf, bb and rec in order to simplify notations.

Accept. The case of acceptance is especially easy: we want to stop the computation[15], so every transition $(a_1, \ldots, a_n; \mathbf{q}) \to \mathbf{accept}$ will be encoded by 0.

Reject. The transitions of the form $(a_1, \ldots, a_n; \mathbf{q}) \to \mathbf{reject}$ are encoded by the operator that represents the "transition" $(a_1, \ldots, a_n; \mathbf{q}) \to (a_1, \ldots, a_n; \mathbf{back}_1)$.

This defines the operator $\to^\bullet$ encoding the transition relation $\to$. But we need a bit more to interpret a machine. Indeed, as we already explained, the representation of an integer is accepted by an observation ϕ if the product of the two operators is nilpotent. Hence, a rejection should lead to the creation of a loop. We therefore add two new states – $\mathbf{back}_j$ and $\mathbf{mov\text{-}back}_j$ – for each $j = 1, \ldots, p$. We can then define $\mathrm{reject}_s = \sum_{j=1}^p \mathrm{rm}_j + \mathrm{rr}^0_j + \mathrm{rr}^1_j + \mathrm{rc}_j$. This

[14] Remark by the way that all those indices were redundant, for the pseudo-configuration c entails a choice of j and $\mathbf{q}'$.

[15] This is one of the reasons we did not considered **accept** to be a state: we want the computation to stop immediately, there is no need to provide "one last set of instructions".

operator actually encodes a reinitialization[16] of the machine that starts when the computation reaches the state $\mathbf{back}_1$: the pointers are moved back one after the other until they point on the symbol $\star$, and the memory slots and the state are initialized according to the given initial pseudo-configuration $s = (a_1, \ldots, a_p; \mathbf{q}_0)$. The reason for this complicated way of encoding rejection comes from the fact that the computation simulated by the operators is massively parallel: the operator $M_s^\bullet$ is a big sum that recollect every single evolution from a pseudo-configuration to any other "reachable" with $\to$, and it starts moreover in all pseudo-configurations at the same time.

Another complication due to this parallel computation and this particular way of defining rejection is that the operators simulate correctly only *acyclic machines*, i.e. machines that never enter a computational loop whichever configuration – including intermediary ones – is chosen as a starting point for computation[17]. This is necessary because we want the entering of a loop to be the result of rejection. Fortunately, it can be shown that:

Proposition 12 ([2, Lemma 25])**.** *For all $M \in NDPM(p)$ there exists $M' \in NDPM(p')$ such that M' is acyclic and $\mathscr{L}(M) = \mathscr{L}(M')$.*

In the particular case of acyclic machines, one can show that the encoding is sound:

Proposition 13 ([2, Lemma 29])**.** *Let $M \in NDPM(p)$ be acyclic, $c \in C_M$ and $M_s^\bullet = \to^\bullet + \mathrm{reject}_s$. For all $n \in \mathbf{N}$ and every binary representation $N_n \in vnM_6(\mathbf{C}) \otimes \mathfrak{N}_0$ of the integer n:*

$$M_s(n) \text{ accepts} \Leftrightarrow M_s^\bullet(N_n \otimes 1) \text{ is nilpotent.}$$

These two results can then be combined with the results of the previous section to get the following proposition:

[16] And this is why the encoding of a NDPM is done provided an initial pseudo-configuration s. In practice, one can always take s to be the "faithful" pseudo-configuration, i.e. $s = \star, \ldots, \star; \mathbf{q}_0)$. We choose to dwell on the parametricity of the encoding to highlight the particular mechanism of rejection, and its flexibility.

[17] For instance, let us consider a machine M that contains a state **loop** and, for a chosen pseudo-configuration c, the transition $(c, \mathbf{loop}) \to (\epsilon_1, \ldots, \epsilon_p, \mathbf{loop}\}$. Then this machine is not acyclic, even if the state **loop** cannot be accessed during the computation, i.e. even if M do not contain any transition leading to the state **loop**.

Proposition 14.

$$\textbf{\textit{co-NL}} \subseteq [P_+]$$

We will now study the encoding of deterministic pointer machines in more detail.

5.2. The Encoding of DPMs

We will show that the encoding of deterministic pointer machines satisfies a particular property: their 1-norm is less or equal to 1. We first define what is the 1-norm on the algebras $\mathfrak{M}_k(\mathfrak{M}_6(\mathfrak{R}))$; this defines in particular the 1-norm of observations since $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{P}$ is a subalgebra of $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{R} \otimes \mathfrak{M}_k(\mathbf{C})$ (for $k = 6^p \times \mathrm{Card}(Q^\uparrow)$), which is isomorphic to $\mathfrak{M}_k(\mathfrak{M}_6(\mathfrak{R}))$.

Definition 15. We define the family of norms $\|\cdot\|_1^k$ on $\mathfrak{M}_k(\mathfrak{M}_6(\mathfrak{R}))$ by:

$$\|(a_{i,j})_{1\leqslant i,j\leqslant k}\|_1^k = \max_{1\leqslant j\leqslant k} \sum_{i=1}^{k} \|a_{i,j}\|$$

where $\|a_{i,j}\|$ is the usual norm (the C*-algebra norm) on $\mathfrak{M}_6(\mathfrak{R})$.
We will simply write $\|\cdot\|_1$ when the superscript is clear from the context.

Remark. For each k, the map $\|\cdot\|_1^k : \mathfrak{M}_k(\mathfrak{M}_6(\mathfrak{R})) \to \mathbf{R}_{\geqslant 0}$ just defined is a norm and, as such, does not depend on the chosen basis. To understand this, notice that this norm can be defined – considering the isomorphism between $\mathfrak{M}_k(\mathfrak{M}_6(\mathfrak{R}))$ and $\mathfrak{M}_k(\mathbf{C}) \otimes \mathfrak{M}_6(\mathfrak{R})$ – as the tensor product of the 1-norm on $\mathfrak{M}_k(\mathbf{C})$ and the usual norm on $\mathfrak{M}_6(\mathfrak{R})$. The 1-norm on $\mathfrak{M}_k(\mathbf{C})$ is formally defined from the usual 1-norm on vectors as $\|A\|_1 = \sup \frac{\|Ax\|_1}{\|x\|_1}$. It can then be shown equal, for any (orthonormal) basis in which one may write A as a matrix $(a_{i,j})_{1\leqslant i,j\leqslant k}$, to the expression $\max_{1\leqslant j\leqslant k} \sum_{i=1}^{k} |a_{i,j}|$.

A corollary of the following proposition will be used extensively in the proofs.

Proposition 16. *Let A, B be operators such that $AB^* = B^*A = 0$. Then:*

$$\|A + B\| = \max\{\|A\|, \|B\|\}$$

Proof. Actually, this proposition appears as Theorem 1.7 (b) in an article by P. J. Maher [21]. This last theorem is stated differently (with conditions on ranges) in the given reference. To obtain the result as we stated it, simply notice that $AB^* = 0$ implies $\mathrm{Ran}(B^*) \subseteq \ker(A) = (\mathrm{Ran}(A^*))^\perp$ so $\mathrm{Ran}(B^*) \perp$

$\mathrm{Ran}(A^*)$. Similarly, $B^*A = 0$ implies that $\mathrm{Ran}(A) \subseteq \ker(B^*) = (\mathrm{Ran}(B))^\perp$ so $\mathrm{Ran}(A) \perp \mathrm{Ran}(B)$. Thus, if $AB^* = B^*A = 0$, we have $\mathrm{Ran}(A) \perp \mathrm{Ran}(B)$ and $\mathrm{Ran}(A^*) \perp \mathrm{Ran}(B^*)$ and one can then apply the theorem of Maher. □

Corollary 17. *The operators ff+bf, rec, bb+fb, and $rr_j^0 + rr_j^1 + rc_j$ (j fixed) are of norm* 1.

Proof. It is clear that $\mathrm{bf} \times \mathrm{ff}^* = 0$ since $\pi_{\mathrm{out}}\pi_{\mathrm{in}} = 0$. Similarly, $\mathrm{ff}^* \times \mathrm{bf} = 0$. Thus, using the preceding proposition, we have $\|\mathrm{ff} + \mathrm{bf}\| = \max\{\|\mathrm{bf}\|, \|\mathrm{ff}\|\} = 1$.

A similar argument shows that $\|\mathrm{fb} + \mathrm{bb}\| = 1$.

Clearly, as a consequence of $\pi_{ai}\pi_{bi} = 0$ for $a \neq b$, we deduce $\|\mathrm{rec}\| = 1$ by Proposition 16.

We are now proving the result for $\mathrm{rr}_j^0 + \mathrm{rr}_j^1 + \mathrm{rc}_j$, so we fix $1 \leqslant j \leqslant p$. First, notice that $\mathrm{rc}_j \times (\mathrm{rr}_j^0)^* = 0$ as a consequence of $\pi_{\mathbf{mov\text{-}back}_j} \times \pi_{\mathbf{back}_j} = 0$. Conversely, $(\mathrm{rr}_j^0)^* \times \mathrm{rc}_j = 0$ as a consequence of[18] $\pi_{\mathbf{back}_{j+1}} \times \pi_{\mathbf{mov\text{-}back}_j} = 0$. A similar argument shows that $\mathrm{rc}_j \times (\mathrm{rr}_j^1)^* = 0$ and $(\mathrm{rr}_j^1)^* \times \mathrm{rc}_j = 0$. Moreover, we know that $(\mathrm{rr}_j^0)^* \times \mathrm{rr}_j^1 = 0$ and $\mathrm{rr}_j^1 \times (\mathrm{rr}_j^0)^* = 0$ by just looking at the first term of the tensor in their definition. By an iterated use of Proposition 16, we get that $\|\mathrm{rr}_j^0 + \mathrm{rr}_j^1 + \mathrm{rc}_j\| = \max\{\|\mathrm{rr}_j^0\|, \|\mathrm{rr}_j^1\|, \|\mathrm{rc}_j\|\} = 1$. □

We are now in possession of all the material needed to characterize the operators coming from the encoding of a deterministic pointer machine.

Proposition 18. *Let $M \in DPM(p)$ be an acyclic machine, and $M_s^\bullet = \rightarrow^\bullet + reject_s$ its encoding as an operator, $\|M_s^\bullet\|_1 \leqslant 1$.*

Proof. Since we are working with deterministic machines, the transition relation is functional: for each $\rho \in C_M$ there is at most one t, say t_ρ, such that $\rho \to t_\rho$. Thus:

$$M_s^\bullet = \sum_{\rho \in C_M} \phi_{\rho, t_\rho} + \mathrm{reject}_s$$

Since $M_s^\bullet$ is an element of $\mathfrak{M}_k(\mathfrak{M}_6(\mathfrak{R}))$, we will now compute $\|M_s^\bullet\|_1^k$. We first show that this matrix has at most one non-null coefficient in each column. Then we will use the fact that these coefficients are of norm at most 1.

[18] When $j = p$, we consider that $\mathbf{back}_{j+1} = \mathbf{q}_0$ to ease the argument.

To prove this we introduce the set $P = \{0i, 0o, 1i, 1o, \star i, \star o\}^p \times Q^\uparrow$ of *extended pseudo-configurations*. This set is a basis of the algebra of pseudo-states, and we write $M_s^\bullet = (a_{\rho,\rho'})_{\rho,\rho' \in P}$. This way, the 1-norm of $M_s^\bullet$ is

$$\|M_s^\bullet\|_1 = \max_{\rho' \in P}\{\sum_{\rho \in P} \|a_{\rho,\rho'}\|\}$$

Let $\rho = (a_1, \ldots, a_p; q)$ be a (non-extended) pseudo-configuration. If ρ is a pseudo-configuration of the machine (i.e. q is an element of Q), then there is at most one pseudo-configuration t_ρ such that $\rho \to t_\rho$. This *atomic transition* can be:

- Accept: in this case the operator ϕ_{ρ,t_ρ} is equal to 0 and the column is empty.
- Reject: in this case the column corresponding to ρ contains only the operator $(\mathbf{q} \to \mathbf{back}_1)$ that encodes the transition $\rho \to (a_1, \ldots, a_p; \mathbf{back}_1)$, and the norm of this operator is equal to 1.
- Move forward a pointer, read a value and change state to **q'**: then the only extended pseudo-configurations introduced by the encoding is $\tilde{\rho} = (a_1, \ldots, a_n; \mathbf{mov}_j^{\mathbf{q,q'}})$. The column corresponding to ρ contains the operator ff + bf, which is of norm 1 by Corollary 17. The column corresponding to $\tilde{\rho}$ contains only the operator rec whose norm is equal to 1 by Corollary 17.
- Move backwards: this case is similar to the previous one.

Now let us take a look at the operator reject_s. The extended pseudo-configurations introduced for the encoding of the rejection are, for $j = 1, \ldots, p$, $\bar{\rho}_j^m = (a_1, \ldots, a_n; \mathbf{mov\text{-}back}_j)$ and $\bar{\rho}_j^b = (a_1, \ldots, a_n; \mathbf{back}_j)$. The column corresponding to ρ contains only the operator encoding the transition from ρ to $\bar{\rho}_1^b$, which is a norm 1 operator. Let us now fix $1 \leqslant j \leqslant p$. The column corresponding to the extended pseudo-state $\bar{\rho}_j^b$ contains only the operator rm_j, which is of norm 1. The column corresponding to $\bar{\rho}_j^m$ contains the operator $\text{rr}_j + \text{rc}_j$, which is of norm 1 by Corollary 17.

We just showed that each column of the matrix contains at most one operator different from 0, and that this operator is always of norm 1. Hence, for any fixed extended pseudo-configuration ρ:

$$\sum_{\rho \in P} a_{\rho,\rho'} \leqslant 1$$

As a consequence of the definition of the 1-norm, we have:

$$\|M^\bullet\|_1 = \max_{\rho' \in P}\{\sum_{\rho \in P} a_{\rho,\rho'}\} \leqslant 1$$

□

The last proposition motivates the following definition:

Definition 19.

$$P_{+,1} = \{\phi \mid \phi \in P_+ \text{ and } \|\phi\|_1 \leqslant 1\}$$

Now, by looking carefully at the proof of Proposition 12, one can show that a similar result holds in the particular case of deterministic pointer machines. Indeed, if M is deterministic, the machine M' constructed from M in the proof is also deterministic:

Proposition 20. *For all $M \in DPM(p)$, there exists $M' \in DPM(p)$ such that M' is acyclic and $\mathscr{L}(M) = \mathscr{L}(M')$.*

As a consequence, one gets the following result:

Proposition 21.

$$\boldsymbol{L} \subseteq [P_{+,1}]$$

The question is then: is this inclusion strict or did we find a characterization of the class **L**? To answer this question, we need to look at the proof of the inclusion $[P_{\geqslant 0}] \subseteq$ **co-NL** and see what happens when we restrict to the operators in $P_{+,1}$.

6. Operators and Logarithmic Space

In this section, we will show that $[P_{+,1}] \subseteq \mathbf{L}$, concluding the proof that $[P_{+,1}] = \mathbf{L}$. To prove this inclusion, we need a technical lemma, which we proved in our earlier paper.

Lemma 22 ([2, Lemma 31])**.** *Let N_n be a binary representation of an integer n in $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{N}_0$ and $\Phi \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{S} \otimes \mathfrak{M}_k(\mathbf{C})$ be an observation in $P_{\geqslant 0}$. There exist an integer f, an injective morphism $\psi : \mathfrak{M}_f(\mathbf{C}) \to \mathfrak{R}$ and two matrices $M \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{M}_f(\mathbf{C})$ and $\bar{\Phi} \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{M}_f(\mathbf{C}) \otimes \mathfrak{M}_k(\mathbf{C})$ such that $Id \otimes \psi(M) = N_n$ and $Id \otimes \psi \otimes Id_{\mathfrak{C}}(\bar{\Phi}) = \Phi$.*

This lemma is of the utmost importance. Even though the hyperfinite factor $\mathfrak{R}$ is necessary to have a uniform representation of integers, it is an algebra of operators acting on an infinite-dimensional Hilbert space. It is therefore not obvious that one can check the nilpotency of such an operator with finite resources, and it is of course not possible in general.

However, the technical lemma above has as a direct consequence that checking the nilpotency of a product $\Phi(N_n \otimes 1)$ where Φ is an observation in $P_{\geqslant 0}$ is equivalent to checking the nilpotency of a product of matrices.

Corollary 23. *Let $\Phi \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{M}_k(\mathbf{C})$ be an observation and $N_n \in \mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{N}_0$ a representation of the integer n. There exists* matrices *(i.e. operators acting on a finite-dimensional Hilbert space) $\bar{\Phi}$ and M such that:*

$$\Phi(N_n \otimes 1_{\mathfrak{M}_k(\mathbf{C})}) \text{ is nilpotent if and only if } \bar{\Phi}(M \otimes 1_{\mathfrak{M}_k(\mathbf{C})}) \text{ is nilpotent}$$

The matrices $\bar{\Phi}$ and $M \otimes 1_{\mathfrak{M}_k(\mathbf{C})}$ are acting on a Hilbert space of dimension $6 \times (\log_2(n)+1)^p p! \times k$, i.e. the integer f in Lemma 22 is equal to $(\log_2(n)+1)^p p!$. Moreover the degree of nilpotency of the two products is equal. We can then show that there exists a Turing machine that decides the nilpotency of the product ΦN_n using only logarithmic space (i.e. in $\log_2(n)$).

Proposition 24 ([2, Proposition 32])**.** *If $\Phi \in P_{\geqslant 0}$ and N_n is a representation of $n \in \mathbf{N}$, there is a **co-NL** machine that checks if $\bar{\Phi}(M \otimes 1_{\mathfrak{M}_k(\mathbf{C})})$ is nilpotent.*

This proposition, together with Proposition 14 and the fact that $[P_+] \subseteq [P_{\geqslant 0}]$ gives us a proof of the following equality:

Theorem 25 ([2, Theorem 33])**.**

$$\textbf{\textit{co-NL}} = [P_+] = [P_{\geqslant 0}]$$

We recall how the proof of Proposition 24 works. The algebra we are working with is:

$$\mathfrak{F} = \mathfrak{M}_6(\mathbf{C}) \otimes ((\underbrace{\mathfrak{M}_{\log_2(n)+1}(\mathbf{C}) \otimes \cdots \otimes \mathfrak{M}_{\log_2(n)+1}(\mathbf{C})}_{p \text{ copies}}) \rtimes \mathfrak{S}_p) \otimes \mathfrak{M}_k(\mathbf{C})$$

We let in the following $K = 6k(\log_2(n)+1)^p p!$ be the dimension of the Hilbert space the algebra $\mathfrak{F}$ is acting on.

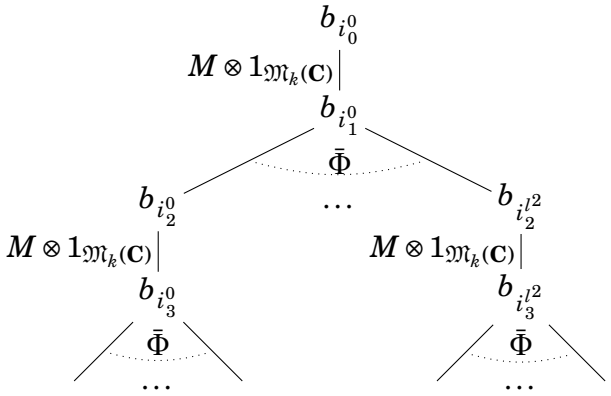
We chose the basis of this algebra defined as the set of elements of the form:

$$(\pi, a_0, a_1, \ldots, a_p; \sigma; e)$$

where π is an element of the basis $(0o, 0i, 1o, 1i, s, e)$ of $\mathfrak{M}_6(\mathbf{C})$, a_i ($i \in \{1, \ldots, p\}$) are the elements of the basis chosen to represent the integer n, $\sigma \in \mathfrak{S}_p$ and e is an element of a basis of $\mathfrak{M}_k(\mathbf{C})$. When we apply $M \otimes 1_{\mathfrak{M}_k(\mathbf{C})}$ representing the integer to an element of this basis, we obtain one and only one vector of the basis $(\pi, a_0, a_1, \ldots, a_p; \sigma; e)$. When we apply to this element the matrix $\bar{\Phi}$ we obtain a (positive) linear combination of $l \in \mathbf{N}$ elements of the basis:

$$\bar{\Phi}(\pi, a_0, a_1, \ldots, a_p; \sigma; e) = \sum_{i=0}^{l} \alpha_i (\rho, a_{\tau_i(0)}, \ldots, a_{\tau_i(p)}; \tau_i \sigma; e_i) \tag{1}$$

So we obtain the following picture:



To decide if the product $\bar{\Phi}(M \otimes 1_{\mathfrak{M}_k(\mathbf{C})})$ is nilpotent, it is then sufficient to check, for each possible value of $b_{i_0^0}$, that the branches of this tree are finite. Since they are either infinite or of length at most K, this can be checked by a non-deterministic (to deal with the branchings that appears in the figure above) Turing machine. This machine only uses logarithmic space since it needs only to store at each step the value of $b_{i_0^0}$, the current basis element $b_{i_{2j}^h}$, and the number of iterations j of $\bar{\Phi}(M \otimes 1_{\mathfrak{M}_k(\mathbf{C})})$ that were computed to get from $b_{i_0^0}$ to $b_{i_{2j}^h}$.

Now, let us look at the case when $\Phi \in P_{+,1}$. In this particular case, Eq. (1) becomes:

$$\bar{\Phi}(\pi, a_0, a_1, \ldots, a_p; \sigma; e) = (\rho, a_{\tau_{b_0}(0)}, \ldots, a_{\tau_{b_0}(p)}; \tau_{b_0} \sigma; e_{b_0}) \tag{2}$$

Let us write $\bar{\Phi}$ as a $\mathrm{Card}(Q) \times \mathrm{Card}(Q)$ matrix $(\bar{\Phi}_{q,q'})_{q,q' \in Q}$ (here Q denotes a basis of $\mathfrak{M}_k(\mathbf{C})$, which is a matrix algebra from the definition of observations). We can deduce from $\|\bar{\Phi}\|_1 \leqslant 1$ that for all $q \in Q$, $\sum_{q' \in Q} \|\bar{\Phi}_{q,q'}\| \leqslant 1$.

As Φ is an element of P_+, the matrix $\bar{\Phi}$ satisfies the following equation:

$$\bar{\Phi}(\pi, a_0, a_1, \ldots, a_p; \sigma; e) = \sum_{i=0}^{l} (\rho, a_{\tau_i(0)}, \ldots, a_{\tau_i(p)}; \tau_i \sigma; e_i) \tag{3}$$

We deduce that $\|\bar{\Phi}_{e,e_i}\| \geqslant 1$ $(i = 1, \ldots, l)$, and therefore $\sum_{e' \in Q} \|\bar{\Phi}_{e,e'}\| \geqslant l$. Since $\Phi \in P_{+,1}$, we know that $\|\bar{\Phi}_{q,q'}\| \leqslant 1$ from which we deduce that $l \leqslant 1$.

That is, the previous picture becomes:

$$\begin{array}{r} b_{i_0^0} \\ M \otimes 1_{\mathfrak{M}_k(\mathbf{C})} \big| \\ b_{i_1^0} \\ \bar{\Phi} \big| \\ b_{i_2^0} \\ M \otimes 1_{\mathfrak{M}_k(\mathbf{C})} \big| \\ b_{i_3^0} \\ \bar{\Phi} \big| \\ \vdots \end{array}$$

Notice that the nilpotency degree of $\bar{\Phi}(M \otimes 1_{\mathfrak{M}_k(\mathbf{C})})$ is again at most K. One can thus easily define a deterministic Turing machine that takes the basis elements one after the other and compute the sequence $b_{i_0^0}$, $b_{i_1^0}$, ...: if the sequence stops before the K-th step, the machine starts with the next basis element as $b_{i_0^0}$, and if the sequence did not stop at step K it means the matrix was not nilpotent. This Turing machine needs no more than logarithmic space since it stores only the current starting basis element (i.e. $b_{i_0^0}$), the last computed term of the sequence $b_{i_{2j}^0}$ and the number of iterations computed so far j. We thus obtain the following proposition.

Proposition 26. *If $\Phi \in P_{+,1}$ and N_n is a representation of $n \in \mathbf{N}$, there is a $\boldsymbol{L}$ machine that checks if $\bar{\Phi}M$ is nilpotent.*

Putting together the results of this section and the last, we can finally state the main theorem of this paper:

Theorem 27.

$$\boldsymbol{L} = [P_{+,1}]$$

Together with Theorem 25, we thus obtained two sets of observations P_+ and $P_{+,1}$, satisfying $P_{+,1} \subsetneq P_+$ and such that $\mathbf{L} = [P_{+,1}] \subseteq [P_+] = \mathbf{co\text{-}NL}$.

7. Conclusion

This work both completes and extends the results obtained in our earlier paper [2] where we showed that the approach recently proposed by Girard [1] for studying complexity classes succeeds in characterizing the complexity class **co-NL**. On the one hand, we showed that the non-deterministic pointer machines, which were designed to mimic the computational behavior of operators, are really close to a well-known abstract machine, two-way multi-head finite automata. This result gives a much better insight on the pointer machines, and we hope this will help us in extending the model of pointer machines to study others complexity classes. It moreover strengthens the proof of the fact that the set of operators P_+ characterizes the class **co-NL**. On the other hand, we extended the result by finding a subset of P_+, defined through a condition on the norm, that corresponds to the encoding of deterministic pointer machines. We showed first that deterministic pointer machines were equivalent to the notion of deterministic two-way multi-head finite automata, hence that **L** is a subset of the language associated to $P_{+,1}$. We then showed that the converse inclusion holds, showing that the set of operators $P_{+,1}$ is indeed a characterization of the complexity class **L**.

The notion of acceptance used in this work (nilpotency) is closely related [22] to the notion of interaction in Girard's geometry of interaction in the hyperfinite factor (GoI5). This should leads to the possibility of defining types in GoI5 corresponding to complexity classes. For this reason, and the fact that the GoI5 construction allows more classical implicit computational complexity approaches such as defining constrained exponential connectives, the GoI5 framework seems a perfect candidate for a general mathematical framework for the study of complexity. The combined tools offered by geometry of interaction and the numerous tools and invariants of the theory of operators that could be used in this setting offer new perspectives for the obtention of separation results.

We also believe that the approach explored in this paper can be used to obtain characterizations of other complexity classes, and in particular the

classes **P** and **co-NP**. These characterizations may be obtained through the use of a more complex group action in the crossed product construction, or by defining a suitable superset of P_+. An approach for obtaining such a result would be to generalize the notion of pointer machines to get a characterization of **P** or **co-NP**. Since the pointer machines are closely related to the operators and the way the latter interact with the representation of integers, such a result would be a great step towards a characterization of these classes.

References

[1] J.-Y. Girard, Normativity in logic, in: P. Dybjer, S. Lindström, E. Palmgren, G. Sundholm (Eds.), Epistemology versus Ontology, Vol. 27 of Logic, Epistemology, and the Unity of Science, Springer, 2012, pp. 243–263. doi:10.1007/978-94-007-4435-6_12.

[2] C. Aubert, T. Seiller, Characterizing co-nl by a group action, Mathematical Structures in Computer Science (FirstView) (2014) 1–33. doi:10.1017/S0960129514000267.

[3] V. Danos, J.-B. Joinet, Linear logic & elementary time, Information and Computation 183 (1) (2003) 123–137. doi:10.1016/S0890-5401(03)00010-5.

[4] U. Schöpp, Stratified bounded affine logic for logarithmic space, in: LICS, IEEE Computer Society, 2007, pp. 411–420. doi:10.1109/LICS.2007.45.

[5] K. Terui, Proof nets and boolean circuits, in: LICS, IEEE Computer Society, 2004, pp. 182–191. doi:10.1109/LICS.2004.1319612.

[6] J.-Y. Girard, Towards a geometry of interaction, in: J. W. Gray, A. Ščedrov (Eds.), Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference held June 14-20, 1987, Vol. 92 of Categories in Computer Science and Logic, American Mathematical Society, 1989, pp. 69–108. doi:10.1090/conm/092/1003197.

[7] J.-Y. Girard, Geometry of interaction V: logic in the hyperfinite factor, Theoretical Computer Science 412 (20) (2011) 1860–1883. doi:10.1016/j.tcs.2010.12.016.

[8] J.-Y. Girard, Geometry of interaction 1: Interpretation of system F, Studies in Logic and the Foundations of Mathematics 127 (1989) 221–260. doi:10.1016/S0049-237X(08)70271-4.

[9] G. Gonthier, M. Abadi, J.-J. Lévy, The geometry of optimal lambda reduction, in: R. Sethi (Ed.), POPL, ACM Press, 1992, pp. 15–26. doi:10.1145/143165.143172.

[10] J. Lamping, An algorithm for optimal lambda calculus reduction, in: F. E. Allen (Ed.), POPL, Association for Computing Machinery, ACM Press, 1990, pp. 16–30. doi:10.1145/96709.96711.

[11] P. Baillot, M. Pedicini, Elementary complexity and geometry of interaction, Fundamenta Informaticae 45 (1–2) (2001) 1–31.

[12] J. Hartmanis, On non-determinancy in simple computing devices, Acta Informatica 1 (4) (1972) 336–344. doi:10.1007/BF00289513.

[13] M. Holzer, M. Kutrib, A. Malcher, Multi-head finite automata: Characterizations, concepts and open problems, in: T. Neary, D. Woods, A. K. Seda, N. Murphy (Eds.), CSP, Vol. 1 of Electronic Proceedings in Theoretical Computer Science, 2008, pp. 93–107. doi:10.4204/EPTCS.1.9.

[14] T. Seiller, Interaction graphs: Multiplicatives, Annals of Pure and Applied Logic 163 (2012) 1808–1837. doi:10.1016/j.apal.2012.04.005.

[15] M. Takesaki, Theory of Operator Algebras 1, Vol. 124 of Encyclopedia of Mathematical Sciences, Springer, 2001.

[16] M. Takesaki, Theory of Operator Algebras 2, Vol. 125 of Encyclopedia of Mathematical Sciences, Springer, 2003.

[17] M. Takesaki, Theory of Operator Algebras 3, Vol. 127 of Encyclopedia of Mathematical Sciences, Springer, 2003.

[18] T. Seiller, Logique dans le facteur hyperfini : géometrie de l'interaction et complexité, Ph.D. thesis, Université de la Méditerranée (2012). URL http://tel.archives-ouvertes.fr/tel-00768403/

[19] A. L. Rosenberg, On multi-head finite automata, IBM Journal of Research and Development 10 (5) (1966) 388–394. doi:10.1147/rd.105.0388.

[20] N. Immerman, Nondeterministic space is closed under complementation, in: CoCo, IEEE Computer Society, 1988, pp. 112–115. doi:10.1109/SCT.1988.5270.

[21] P. J. Maher, Some operator inequalities concerning generalized inverses, Illinois Journal of Mathematics 34 (3) (1990) 503–514.

[22] T. Seiller, Interaction graphs: Additives, Arxiv preprint abs/1205.6557, accepted for publication in Annals of Pure and Applied Logic. arXiv:1205.6557.

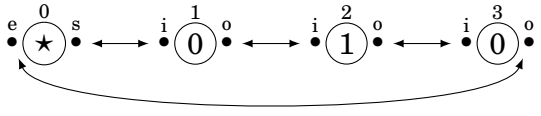
A. Examples

We develop below some examples that should enlighten the encoding of integers, the notion of pointer machine, and the encoding of the latter as operators. We begin by recalling the encoding of the input adopted, with two examples. Then, we briefly sketch a pointer machine that accepts palindromes and encode it as an observation. This allows us to make general remarks on the "tricks" one can adopt to lighten the encoding, which is in all generality unnecessarily heavy. Although this makes this example somehow strange to illustrate the paper as it is not encoded using the method employed for the proofs, we believe it gives a number of insights on how the observations interact with integers and how the computation actually takes place.

A.1. Integers

As explained in details in our earlier paper [2], integers are represented as matrices which are then embedded into the hyperfinite factor $\mathfrak{R}$. The matrices are themselves adjacency matrices of a graph representing the links between two adjacent symbols in the binary representation of the integer. Let us work out two examples.

First, the integer $\star 010$ will be represented by the adjacency matrix of the following graph. Notice the integers on top of the circled symbols which represent a sort of "state" – or sorts of "locations" for symbols – , with a unique such state for each symbol in the list (counting the symbol $\star$).



This adjacency matrix is a 6×6 matrix with 4×4 matrices as coefficients, where the number 4 stands for the length of the list (these are the "states" mentioned above) and the number 6 stand for the 6 different types of vertices: $0i$, $0o$, $1i$, $1o$, e (i.e. $\star i$) and s (i.e. $\star o$). Here is the corresponding matrix:

$$M_{\star 010} = \begin{array}{c} \overbrace{}^{0}\ \overbrace{}^{1}\ \overbrace{}^{*} \\ \left(\begin{array}{cccccc} 0 & 0 & 0 & l_{10} & s_0 & 0 \\ 0 & 0 & l_{01}^* & 0 & 0 & e_0^* \\ 0 & l_{01} & 0 & 0 & 0 & 0 \\ l_{10}^* & 0 & 0 & 0 & 0 & 0 \\ s_0^* & 0 & 0 & 0 & 0 & 0 \\ 0 & e_0 & 0 & 0 & 0 & 0 \end{array}\right) \end{array} \begin{array}{l} \}0 \\ \}0 \\ \}1 \\ \}1 \\ \}* \\ \}* \end{array}$$

where 0 stands for the zero 4×4 matrix, the adjoint $(\cdot)^*$ corresponds to the conjugate-transpose, and the matrices l_{10}, l_{01}, s_0 and e_0 are defined as follows:

$$l_{01} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} l_{10} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$s_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} e_0 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

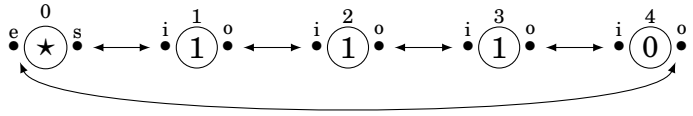
The matrix l_{01} express the edges of the above graph that are going from left to right, whose source is a 0 and whose target is a 1. This matrix is therefore located – in the 6×6 matrix – on the column corresponding to "$0o$" (for "0 out") – because its source is a vertex "$0o$" – and on the row "$1i$" – because its target is a "$1i$" vertex. This matrix then contains a single non-zero element because the list contains exactly one such edge going left-to-right from a 0 to a 1, and shows how this edge connects the locations; in this case, the only such edge goes from the location named 1 to the location named 2, which explains why the non-zero element is located in the second column (locations are numbered starting from 0) and the third row.

The definition and placement of the matrix l_{10} is explained in the same fashion, while the matrix s_0 (resp. e_0) represent those edges going from left

to right whose source is a "s" vertex (resp. a "0" vertex) and target is a "0" vertex (resp. a "e" vertex). Similar matrices named l_{00} and l_{11} could appear in the encoding of an integer if its binary writing contained two symbols 0 or two symbols 1 following each other; in our example, however, these matrices are empty.

Lastly, the adjoint matrices such as l_{01}^* actually correspond to the right-to-left edge in the above graph.

As a second example, we consider the integer $\star 1100$. It is represented as a 6×6 matrix with 5×5 matrices as coefficients. The corresponding graph is as follows.



The corresponding matrix is then:

$$M_{\star 1110} = \begin{pmatrix} 0 & 0 & 0 & l_{10} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & e_0^* \\ 0 & 0 & 0 & l_{11} & s_1 & 0 \\ l_{10}^* & 0 & l_{11}^* & 0 & 0 & 0 \\ 0 & 0 & s_1^* & 0 & 0 & 0 \\ 0 & e_0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

where the matrices l_{10}, l_{11}, s_1 and e_0 are defined as:

$$l_{10} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad l_{11} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$s_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad e_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

A.2. The Palindrome DPM

We will now define a deterministic pointer machine $P = \{Q, \to\}$ that decides the language of palindromes:

$$\mathrm{Pal} = \{\star a_0 a_1 \dots a_n \mid \forall i \in \{0, \dots, n\}, a_i = a_{n-i}\}$$

To decide Pal, the machine P will make use of two pointers. It will start with both pointers on $\star$ and move one pointer from left to right over the input and move the second pointer from right to left – or from the end to the beginning. It will move the pointers alternatively and check if the i-th symbol is indeed equal to the $n-i$-th one. It will be defined with five states: an initial state **init**, and states $\mathbf{pass}_1$, $\mathbf{pass}_2$, $\mathbf{fail}_1$ and $\mathbf{fail}_2$. The subscript will be used to remember which pointer will be moving next. We now define formally the machine P.

Definition 28 (The Palindrome Pointer Machine)**.** Let $P = \{Q, \to\}$ be the DPM(2) defined as follows: $Q = \{\mathbf{init}, \mathbf{pass}_1, \mathbf{pass}_2, \mathbf{fail}_1, \mathbf{fail}_2\}$, and the following transitions, where "$\cdot$" (resp. "0/1") is a notation standing for any value in $\{0, 1, \star\}$ (resp. in $\{0, 1\}$):

$$(\star, \star, \mathbf{init}) \to (p_1+, \epsilon_2, \mathbf{pass}_2) \tag{A.1}$$
$$(\cdot, \cdot, \mathbf{pass}_2) \to (\epsilon_1, p_2-, \mathbf{pass}_1) \tag{A.2}$$
$$(0, 0, \mathbf{pass}_1) \to (p_1+, \epsilon_2, \mathbf{pass}_2) \tag{A.3}$$
$$(1, 1, \mathbf{pass}_1) \to (p_1+, \epsilon_2, \mathbf{pass}_2) \tag{A.4}$$
$$(0, 1, \mathbf{pass}_1) \to (p_1+, \epsilon_2, \mathbf{fail}_2) \tag{A.5}$$
$$(1, 0, \mathbf{pass}_1) \to (p_1+, \epsilon_2, \mathbf{fail}_2) \tag{A.6}$$
$$(0/1, 0/1, \mathbf{fail}_1) \to (p_1+, \epsilon_2, \mathbf{fail}_2) \tag{A.7}$$
$$(\cdot, \cdot, \mathbf{fail}_2) \to (\epsilon_1, p_2-, \mathbf{fail}_1) \tag{A.8}$$
$$(\star, \star, \mathbf{pass}_1) \to \mathbf{accept} \tag{A.9}$$
$$(\star, \star, \mathbf{fail}_1) \to \mathbf{reject} \tag{A.10}$$

This transition relation is then completed into a total relation by considering that each configuration not appearing above leads to an acceptation. The machine P is then a *deterministic* pointer machine.

To represent rejection when encoding this machine as an observation, we will have to create a loop by returning to the initial state (the "re-initialisation

trick"). The formal encoding defined in the paper works for any machine, but it is a bit heavy. In the case of P, we use a simple trick to simplify the representation: we consider the machine defined as above but:

- without the state **init** and with initial configuration $(\star, \star, \mathbf{fail}_1)$ – thus the transition A.1 is replaced by $(\star, \star, \mathbf{fail}_1) \to (p_1+, \epsilon_2, \mathbf{pass}_2)$;
- in which the transition A.10 is replaced by $(\star, \star, \mathbf{fail}_1) \to (p_1+, \mathbf{pass}_2)$.

This modified machine then either accepts – if the input is a palindrome – or loops by going through the initial state over and over – this loop replaces rejection. By encoding this modified machine as is, we obtain an observation that will accept the same language as P; this observation is however simpler than the one we would obtain from applying mindlessly the translation presented in Section 5.1.

A.3. The Palindrome Observation

As explained above, we will actually encode the following transitions, and no other – none of the transitions steps ending with acceptation are encoded since acceptance corresponds to putting a stop to the computation.

$$(\star, \star, \mathbf{fail}_1) \to (p_1+, \epsilon_2, \mathbf{pass}_2) \tag{A.11}$$
$$(\cdot, \cdot, \mathbf{pass}_2) \to (\epsilon_1, p_2-, \mathbf{pass}_1) \tag{A.12}$$
$$(\cdot, \cdot, \mathbf{fail}_2) \to (\epsilon_1, p_2-, \mathbf{fail}_1) \tag{A.13}$$
$$(0, 0, \mathbf{pass}_1) \to (p_1+, \epsilon_2, \mathbf{pass}_2) \tag{A.14}$$
$$(1, 1, \mathbf{pass}_1) \to (p_1+, \epsilon_2, \mathbf{pass}_2) \tag{A.15}$$
$$(0, 1, \mathbf{pass}_1) \to (p_1+, \epsilon_2, \mathbf{fail}_2) \tag{A.16}$$
$$(1, 0, \mathbf{pass}_1) \to (p_1+, \epsilon_2, \mathbf{fail}_2) \tag{A.17}$$
$$(0/1, 0/1, \mathbf{fail}_1) \to (p_1+, \epsilon_2, \mathbf{fail}_2) \tag{A.18}$$

We will moreover use a number of small "hacks" that will decrease the size of the resulting observation. Indeed, the encoding described in this paper is general and therefore applies blindly to every pointer machines; it may however be optimized in specific cases. The first optimization concerns the set of additional states. The second concerns the way the representation deals with pointers. The third "hack" will be used to decrease the size of the matrices that represent the pointers "memory cells".

Here is the first "hack". The set of states we are starting from is $Q = \{\mathbf{pass}_1, \mathbf{pass}_2, \mathbf{fail}_1, \mathbf{fail}_2\}$. As it contains 4 elements, the set $Q^{\uparrow}$ should have $4 + 2(4^2 + 2) = 40$ elements. Of course, not all these elements are needed here. First, we won't need the specific states introduced to deal with rejection since – as we already explained – rejection is already represented by the creation of a loop through the initial state. We are thus left with $4 + 2 \times 4^2 = 36$ states. It turns out that only $4 + 6 = 10$ states is enough in this case, as all we need are states to transition:

- from $\mathbf{pass}_1$ to $\mathbf{pass}_2$ (transitions A.14 and A.15); we write it as $\mathbf{pass}_{1\to 2}$;
- from $\mathbf{pass}_1$ to $\mathbf{fail}_2$ (transitions A.16 and A.17); we write it as **error**;
- from $\mathbf{pass}_2$ to $\mathbf{pass}_1$ (transition A.12); we write it as $\mathbf{pass}_{2\to 1}$;
- from $\mathbf{fail}_2$ to $\mathbf{fail}_1$ (transition A.13); we write it as $\mathbf{fail}_{2\to 1}$;
- from $\mathbf{fail}_1$ to $\mathbf{fail}_2$ (transition A.18); we write it as $\mathbf{fail}_{1\to 2}$;
- from $\mathbf{fail}_1$ to $\mathbf{pass}_2$ (transition A.11); we write it as **reinit**.

Let us now explain the second "hack". We notice that this machine never uses any "ϵ transition" – i.e. the machine moves a pointer at each transition step – and always alternates the movements of the first and second pointer. The encoding presented in Section 5.1 makes use of a kind of "dummy pointer", and then activates/deactivates pointers in order to move them. We will here get rid of this "dummy pointer" by considering that it represents the first pointer. As a consequence, the use of the operator $\tau_{0,1}$ will represent the simultaneous activation of a pointer and the deactivation of the other pointer. Moreover, it has an effect on the set of additional states needed to encode the operators: since we simultaneously activate a pointer and deactivate the other, we can get rid of the six additional states explained above and work with the 4 states of the original pointer machine. This will however complicate the understanding of the encoding in that an operator encoding a given transition will also encode the "recording" of the value read during the previous transition into memory cells. i.e. we will consider operators that are a mixing of the operators $\mathrm{bf}^{c}_{j,\mathbf{q}'} + \mathrm{ff}^{c}_{j,\mathbf{q}'}$ for the currently encoded transition and the operator $\mathrm{rec}^{c'}_{k,\mathbf{q}}$ encoding the recording of the value read during the previous transition. This complication is however worth considering since it will greatly reduce the size of the resulting matrices.

Lastly, let us recall the value last read by a pointer is stored as a state in the encoding of machines. In the general encoding presented in this paper, we store those as 6×6 matrices, which is coherent with the fact that the integer answers belong to a six-elements set $\{0i, 0o, 1i, 1o, s, e\}$. However, all we will need is the information about the symbol read, and we won't care whether we read this symbol by moving left or moving right. This allows us to use 3×3 matrices as memory cells, decreasing once again the size of the resulting observation.

Finally, each transition will be represented as an element of $\mathfrak{M}_6(\mathbf{C}) \otimes \mathfrak{G} \otimes \mathfrak{M}_3(\mathbf{C}) \otimes \mathfrak{M}_3(\mathbf{C}) \otimes \mathfrak{M}_4(\mathbf{C})$. The algebra $\mathfrak{M}_6(\mathbf{C})$ will allow the observation to interact with the integer, i.e. ask what is the next or previous symbol on the input tape. The two copies of $\mathfrak{M}_3(\mathbf{C})$ represent the "memory cells" corresponding to the pointers, i.e. they will be used to record the last values read by the pointers. The algebra $\mathfrak{M}_4(\mathbf{C})$ is used to deal with states. We represent the 6×6, 3×3 matrices and 4×4 matrices using the respective bases $\{0i, 0o, 1i, 1o, s, e\}$, $\{0, 1, \star\}$ and $\{\mathbf{pass}_1, \mathbf{pass}_2, \mathbf{fail}_1, \mathbf{fail}_2\}$ with basis elements considered in this order exactly.

We first represent the transitions A.12 and A.13. We represent them simultaneously to gain some space; this is possible because they are the same transition but for the involved states. These transitions do not depend on the values read by the pointers, but will write down the new values in the two copies of $\mathfrak{M}_3(\mathbf{C})$ that represent the "memory cells". It will therefore be represented as a sum of the following nine operators:

$$\begin{pmatrix} 0&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 1&1&1\\ 0&0&0\\ 0&0&0 \end{pmatrix} \otimes \begin{pmatrix} 1&0&0\\ 0&0&0\\ 0&0&0 \end{pmatrix} \otimes \begin{pmatrix} 0&1&0&0\\ 0&0&0&0\\ 0&0&0&1\\ 0&0&0&0 \end{pmatrix}$$

$$\begin{pmatrix} 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&1&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 1&1&1\\ 0&0&0\\ 0&0&0 \end{pmatrix} \otimes \begin{pmatrix} 0&0&0\\ 0&0&0\\ 0&0&1 \end{pmatrix} \otimes \begin{pmatrix} 0&1&0&0\\ 0&0&0&0\\ 0&0&0&1\\ 0&0&0&0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Notice that these operators are all morally the same. One should consider them by groups of three. The first three correspond to the case when the value read by the first pointer during the last transition was a 0. In this case, the integer answered on the second column of the $\mathfrak{M}_6(\mathbf{C})$ matrix since this column corresponds to the basis element $0o$ (for "0 out") and the first pointer moves forward. Then the first operator corresponds to the case of the last value read by the second pointer is also a 0. Thus, to ask what is the next value read by this second pointer one needs to activate it – this is the role of $\tau_{0,1}$ – and the machine needs to output on the second row of the 6×6 matrix: this is because we want to know what symbol *preceded* the last 0 read by the second pointer (remember that the second pointer moves backwards). The second operator in this first group corresponds to the case when the last value read by the second pointer was 1. Then, the 6×6 matrix used moves from the basis element $0o$ – which corresponds to the previous answer given by the integer - to the basis element $1o$ – to ask what symbol preceded the last 1 read by the second pointer. The third operator corresponds to the case when the last symbol read by the second pointer was a $\star$. Finally, the two following groups of three matrices play the same role as this first group in the eventuality that the last value read by the first pointer was a 1 or a $\star$.

Notice also the right-hand matrix in each of these transitions. This matrix takes care of the states. This is why there are two non-zero coefficients in those matrices here: one represents the transition A.12 and the second represent the transition A.13.

Let us now deal with the representation of the transition A.14. It records the last value read by the second pointer, which should be 0, and asks the value following the last value read by the first pointer, which is 0 also. Thus, the first matrix goes from $0i$ (since the second pointer goes backwards) to $0i$ (since the first pointer moves forward). Then, the first 3×3 matrix ensures that the last value read by the first pointer was indeed 0, and the second 3×3 matrix records the new value read by the second pointer: whichever value was stored beforehand, it is now replaced by a 0. Lastly, the 4×4

matrix deals with the change of states. As before, we use the fact that this transition is almost similar to the transition A.18 when both values are equal to 0 and represent both by a single operator: this is shown by the fact that the right-hand matrix not only contains a transition from $\mathbf{pass}_1$ to $\mathbf{pass}_2$ (for transition A.14) but also a transition from $\mathbf{fail}_1$ to $\mathbf{fail}_2$.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The operator encoding the transitions A.15, A.16 and A.17 and the corresponding cases of transition A.18 are quite similar and are shown below in that order.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Finally, we represent the transition A.11. It is represented as the following

operator:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \otimes \tau_{0,1} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$