Unification and Logarithmic Space

Aubert Clément, Marc Bagnol

UNIFICATION AND LOGARITHMIC SPACE

CLÉMENT AUBERT AND MARC BAGNOL

Aix Marseille Université, CNRS, Centrale Marseille, I2M UMR 7373, 13453, Marseille, France
e-mail address: clement.aubert@math.cnrs.fr
URL: https://lacl.fr/~caubert/

Department of Mathematics and Statistics, University of Ottawa
e-mail address: mbagnol@uottawa.ca
URL: http://www.normalesup.org/~bagnol/

Abstract. We present an algebraic characterization of the complexity classes Logspace and NLogspace, using an algebra with a composition law based on unification. This new bridge between unification and complexity classes is rooted in proof theory and more specifically linear logic and geometry of interaction.

We show how to build a model of computation in the unification algebra and then, by means of a syntactic representation of finite permutations in the algebra, we prove that whether an observation (the algebraic counterpart of a program) accepts a word can be decided within logarithmic space. Finally, we show that the construction naturally corresponds to pointer machines, an convenient way of understanding logarithmic space computation.

Introduction

Proof Theory and Implicit Complexity Theory. Complexity theory classifies the difficulty of problems by studying the asymptotic bounds on the resources (time, memory, processors, etc.) needed by a model of computation to run a program that solves them. It was originally dependent on models of computation (Turing machines, random access machines,

1998 ACM Subject Classification: F.1.3 Complexity Measures and Classes, F.4.1 Mathematical Logic.
2012 ACM Subject Classification: [**Theory of computation**] Models of computation — Abstract machines; Computational complexity and cryptography — Complexity theory and logic; Logic — Proof theory; Semantics and reasoning — Program semantics — Algebraic semantics.
Key words and phrases: Implicit Complexity, Unification, Logarithmic Space, Proof Theory, Pointer Machines, Geometry of Interaction.
This is an extended version of a proceeding work [AB14].
This work was partly supported by the ANR-11-INSE-0007 REVER, the ANR-11-BS02-0010 Récré and the ANR-10-BLAN-0213 Logoi.
This work was partly supported by the ANR-10-BLAN-0213 Logoi and the ANR-11-BS02-0010 Récré.

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

Boolean circuits, etc.), associated to a reasonable cost-model, that ran the implementation of an algorithm, a program. The aim of implicit computational complexity (ICC) theory is to abstract away the specificities of hardware by focusing on the way programs are written. For instance, weaker recursion schemata [BC92], stratified recurrence [Lei93], or quasi-interpretation [BMM11] restrict expressivity via internal limitations on programming languages or function algebras rather than on available resources.

There is a longstanding tradition of relating proof theory (more specifically linear logic [Gir87]) and implicit complexity theory, thanks to the Curry-Howard—or *proofs as programs*—correspondence. Indeed, mathematical proofs and typed programs, both endowed with an evaluation mechanism (respectively cut-elimination and execution), are viewed as isomorphic, so that restrictions on the former translate seamlessly into limitations on the latter. Fragments of linear logic—bounded [GSS92, DLH10], elementary [DJ03], light [Gir95b] or stratified [Sch07, BM10], to name a few—were proven to characterize complexity classes. By removing or restricting rules of derivation, one excludes proofs and henceforth algorithms: the class of programs accepted can then be proven to (extensionally) correspond to functions of a certain complexity class. In these restricted logics, the cut-elimination procedure—which represents execution of programs as rewriting of proofs—is simpler, and problems that are undecidable in general (such as termination of computation) can become of a manageable complexity.

Geometry of Interaction. The study of cut-elimination has grown to a central topic in proof theory and as a consequence its mathematical modelling became of great interest. The geometry of interaction [Gir89b] research program led to mathematical models of cut-elimination in terms of paths [ADLR94], token machines [Lau01], operators algebras [Gir89a, Gir11, Sei14] or graphs [Dan90, Sei12a, Sei12b]. The general perspective is to consider untyped objects modelling untyped programs and to represent algebraically cut-elimination.

This approach was already used with complexity concerns [BP01, Gir12, AS15, AS14].[1] It differs from usual ICC via proof theory because the restrictions on the expressivity of programs is not obtained through restrictions on type systems. Instead, limitations imposed on the objects representing proofs rule out computational principles on the semantics side and allow to capture complexity classes. This enables the use of methods coming from all areas of mathematics: for instance, an action of the group of permutations on an unbounded tensor product will provide us with our basic computational principle.

Unification. Unification is one of the key-concepts of theoretical computer science, for it is used in logic programming and is a classical subject of study for complexity theory. Its different names and variants—unification, matching, resolution rule, etc.—always comes down to the same question: is there a substitution to make two first-order terms equal? It is an interesting mechanism of computation that can be seen as more primitive than other evaluation procedures such as the β-reduction of λ-calculus.

The resolution rule of logic programming [Rob65] serves as a basis for a more syntactical version of geometry of interaction [Gir95a, Gir13], where cut-elimination is represented as iterated matching in a *unification algebra* [Bag14]. In this setting, proofs are represented as

[1]For a more advanced discussion on the "sister approaches" relying on the theory of von Neumann algebras [Gir12, AS14, AS15] one should refer to the "related work" section, page 17.

clauses (or "flows"), which have a natural notion of size, height, etc. and relate closely to the study of complexity of logic programming [DEGV01]. This is an intuitive framework, yet expressive enough for our purposes.

Contribution and Outline of the Article. We carry on the methodology of bridging geometry of interaction and complexity theory with this renewed approach. It relies on an simple representation of execution in a unification-based algebra, defined in Section 1, that is shown to represent some algebraic structures syntactically.

In Section 2, we present the framework where computation takes place and show how to represent data and programs. Inputs are considered to be words over a finite alphabet, encoded thanks to the classical Church representation of lists (Section 2.1). This raises a question about invariance up to different representations of the same input, addressed in Section 2.2.

This construction is finally specialized in Section 3 to a subalgebra relying on a representation of permutations in the unification algebra. The soundness of the construction with respect to logarithmic space computation (both deterministic and non-deterministic) is proven thanks to a procedure deciding the outcome of the interaction of the representation of a program with the representation of a data. Observations are the algebraic counterpart of programs, and they are shown in Section 3.2 to correspond to a natural notion of read-only Turing machines: pointer machines. In that perspective the algebraic notion of isometricity will correspond to reversibility of computation.

1. The Unification Algebra

1.1. Unification.

Unification can be thought of as the study of formal solving of equations between terms. This topic was introduced by Herbrand [Her30], but became really widespread after the work of J. A. Robinson [Rob65] on automated theorem proving. The unification technique is also at the core of the logic programming language Prolog and type inference for functional programming languages such as CaML and Haskell.

Notation 1.1. *We consider first-order terms, written $t, u, v, \dots$, built from variables, noted in italics font (e.g. x, y), and function symbols with an assigned finite arity, written in typewriter font (e.g. $\mathtt{c}$, $\mathtt{f}(\cdot)$, $\mathtt{g}(\cdot,\cdot)$). Symbols of arity 0 will be called* constants.

Sets of variables and of function symbols of any arity are supposed infinite. We distinguish a binary function symbol $\bullet$ (in infix notation) and a constant symbol $\star$. We will omit the parentheses for $\bullet$ and write $t \bullet u \bullet v$ for $t \bullet (u \bullet v)$.

We write $\mathrm{Var}(t)$ the set of variables occurring in the term t and say that t is closed *if $\mathrm{Var}(t) = \varnothing$. We will write θt the result of applying the substitution θ, written $\{t_1 \mapsto u_1; t_2 \mapsto u_2; \dots\}$, to the term t.*

Definition 1.2 (renaming and instance)**.** A *renaming* is a substitution α that bijectively maps variables to variables. A term t' is a *renaming* of t if $t' = \alpha t$ for some renaming α. Two substitutions θ, ψ are equal *up to renaming* if there is a renaming α such that $\psi = \alpha\theta$.

A substitution ψ is an *instance* of θ if there is a substitution σ such that $\psi = \sigma\theta$.

Examples 1.3. Let

$$\alpha = \{x \mapsto y; y \mapsto x\} \quad \theta = \{x \mapsto \mathtt{c}; z \mapsto \mathtt{g}(\star)\} \quad \psi = \{y \mapsto \mathtt{c}; z \mapsto \mathtt{g}(\star)\}$$

be three substitutions and

$$t = \mathtt{f}(x) \bullet z \quad t' = \mathtt{f}(y) \bullet z$$

be two terms. Then α is a renaming, t' is a renaming of t, and θ and ψ are equal up to renaming. As $\mathrm{Var}(t) = \{x, z\}$, $\theta t = \mathtt{f}(\mathtt{c}) \bullet \mathtt{g}(\star)$ is a closed term.

Definition 1.4 (unification)**.** Two terms t, u are *unifiable* if there is a substitution θ such that $\theta t = \theta u$. We say that θ is a *most general unifier (MGU) of* t, u if any other unifier of t, u is an instance of θ.

Remark 1.5. It is easy to check that any two MGU of a pair of terms are equal up to renaming.

We will be interested mostly in the weaker variant of unification where one can first perform renamings on terms to make their variables distinct. We therefore introduce a specific vocabulary for it.

Definition 1.6 (disjointness and matching)**.** Two terms t, u are *matchable* if t', u' are unifiable, where t', u' are renamings (Definition 1.2) of t, u such that $\mathrm{Var}(t') \cap \mathrm{Var}(u') = \varnothing$.

If two terms are not matchable, they are said to be *disjoint*.

Example 1.7. The terms x and $\mathtt{c} \bullet x$ are not unifiable, but they are matchable, as a renaming of x, for instance $\alpha x = \{x \mapsto y; y \mapsto x\}x = y$, is unifiable with $\mathtt{c} \bullet x$.

A fundamental result on first-order unification is the (decidable) existence of most general unifiers in cases where the unification problem has a solution.

Proposition 1.8 (MGU)**.** *If t and u are unifiable, they have a MGU. Whether two terms are unifiable and, in case they are, finding a MGU is a decidable problem.*

As unification grew in importance, the study of its complexity gained in attention. A complete survey [Kni89] tells the story of the bounds getting sharpened: general first-order unification was finally proved [DKM84] to be a Ptime-complete problem.

In this article, we will consider a much simpler case of the problem: matching a term against a closed term, which has been shown to be tractable within deterministic logarithmic space.

Theorem 1.9 (Matching is in Logspace [DKM84, p. 49])**.** *Deciding if two terms t, u with u closed are unifiable and, if so, producing their MGU, is in* Logspace.

This result will be of crucial interest later on, in the proof of Theorem 3.2, where we will unify what will be called a flow against a closed terms: the only point is to notice that unification involving a closed term is always matching.

1.2. **Flows and Wirings.** We now design an algebra with a product based on unification. Let us start by setting up a monoid with a partially defined product, which will be the basis of the construction.

Definition 1.10 (flows)**.** A *flow* is an oriented pair of first-order terms $t \leftharpoonup u$ such that $\mathrm{Var}(t) = \mathrm{Var}(u)$.

Flows are considered up to renaming: for any renaming α, $t \leftharpoonup u = \alpha t \leftharpoonup \alpha u$.

We set $I := x \leftharpoonup x$ and $(t \leftharpoonup u)^\dagger := u \leftharpoonup t$, so that $(.)^\dagger$ is an involution.

A flow $t \leftharpoonup u$ can be thought of as a `match...with u -> t` in a ML-style language or as a specific kind of Horn clause.[2] The composition of flows follows this intuition: it is an instance of the resolution rule of logic programming.

Definition 1.11 (product of flows)**.** Let $u \leftharpoonup v$ and $t \leftharpoonup w$ be two flows. Suppose we have chosen two representatives of the renaming classes such that their sets of variables are disjoint. The *product* of $u \leftharpoonup v$ and $t \leftharpoonup w$ is defined if v, t are unifiable with MGU θ (the choice of a MGU does not matter because, see Remark 1.5) and in that case:

$$(u \leftharpoonup v)(t \leftharpoonup w) := \theta u \leftharpoonup \theta w$$

Definition 1.12 (action on closed terms)**.** If t is a closed term, $(u \leftharpoonup v)(t)$ is defined whenever v and t are unifiable, with MGU θ, in that case $(u \leftharpoonup v)(t) := \theta u$

Examples 1.13. Composition of flows: $(x \bullet \mathtt{c} \leftharpoonup (\mathtt{c} \bullet \mathtt{c}) \bullet x)(y \bullet z \leftharpoonup z \bullet y) = x \bullet \mathtt{c} \leftharpoonup x \bullet \mathtt{c} \bullet \mathtt{c}$.

Action on a closed term: $(x \bullet \mathtt{c} \leftharpoonup x \bullet \mathtt{c} \bullet \mathtt{c})(\mathtt{d} \bullet \mathtt{c} \bullet \mathtt{c}) = \mathtt{d} \bullet \mathtt{c}$.

Remark 1.14. The condition on variables ensures that the result of an action on a closed term is a closed term, because $\mathrm{Var}(u) \subseteq \mathrm{Var}(v)$, and that the action is injective on its definition domain, because $\mathrm{Var}(v) \subseteq \mathrm{Var}(u)$.

Moreover, the action is compatible with the product of flows: for l and k two flows and t a term, $l(k(t)) = (lk)(t)$ and both are defined at the same time.

By adding a formal element $\perp$ (representing the failure of unification) to the set of flows, one could turn the product into a completely defined operation, making the set of flows an *inverse monoid*. However, we will need to consider the wider algebra of *sums* of flows that is easily defined directly from the partially defined product. An analogy can give an insight on this need: when considering logic programs, one wants to manipulate *set of clauses* and not only a single clause. All the same, we want here to compute thanks to sets of flows, formally represented as sums of flows in our algebra, although the coefficient will play a minor role (we will essentially take them all to be 1, as detailed in Definition 1.17).

We therefore now lift the structure to a $*$-algebra by considering formal sums of flows with complex coefficients and extending all our operations by linearity.

Definition 1.15 (wirings, unification algebra)**.** *Wirings* are $\mathbb{C}$-linear combinations of flows endowed with the following operations (with $\lambda_i, \mu_j \in \mathbb{C}$, l_i, k_j two flows and $\overline{\lambda}$ the complex

[2]The precise connections with logic programming, that comes with a relaxed definition of flows, were subsequently exposed [ABPS14, ABS15].

conjugate of λ):

$$\left(\sum_i \lambda_i l_i\right)\left(\sum_j \mu_j k_j\right) := \sum_{\substack{i,j \text{ such that} \\ (l_i k_j) \text{is defined}}} \lambda_i \mu_j (l_i k_j)$$

$$\left(\sum_i \lambda_i l_i\right)^\dagger := \sum_i \overline{\lambda}_i l_i^\dagger$$

We write $\mathcal{U}$ the set of wirings and refer to it as the *unification algebra*.

Remark 1.16. Indeed, $\mathcal{U}$ is a unital $*$-algebra: it is a $\mathbb{C}$-algebra, considering the product defined above, with an involution $(.)^\dagger$ and a unit I (Definition 1.10).

To study computation and concrete programs (which do not involve complex coefficients) we need to restrict the large algebraic framework defined and to consider wirings whose only coefficient is 1.

Definition 1.17 (concrete wirings)**.** A wiring is *concrete* whenever it is a sum of flows with all coefficients equal to 1. Given a set of wirings E we write E^+ the set of all concrete wirings of E, and will omit to write the coefficients.

We can then consider further the notion of *isometric wiring*. As they act on closed terms as a partial injection (Lemma 1.21), they can be considered as behaving in a reversible way. On the other hand, they satisfy the algebraic property of partial isometries, that is $WW^\dagger W = W$.

Definition 1.18 (isometric wiring)**.** A concrete wiring $\sum_i u_i \leftharpoonup t_i$ is *isometric* if the u_i are pairwise disjoint (Definition 1.6) and t_i are pairwise disjoint.

Example 1.19. The sum of flows $(\mathtt{c} \bullet x \leftharpoonup x \bullet \mathtt{d}) + (\mathtt{d} \bullet \mathtt{c} \leftharpoonup \mathtt{c} \bullet \mathtt{c})$ is an isometric wiring. Note that a wiring containing a single flow will always be a partial isometry.

It will be convenient to consider the action of wirings on closed terms, making them linear operators on the vector space spawned by closed terms. We therefore extend the definition of action on closed terms (Definition 1.12) to wirings.

Definition 1.20 ($\mathbb{T}$, action on closed terms)**.** Let $\mathbb{T}$ be the free $\mathbb{C}$-vector space spawned by closed terms. Wirings act on base vectors of $\mathbb{T}$ in the following way:

$$\left(\sum_i \lambda_i l_i\right)(t) := \sum_{\substack{i \text{ such that} \\ l_i(t) \text{ is defined}}} \lambda_i \big(l_i(t)\big) \quad \in \mathbb{T}$$

which extends by linearity into an action on the whole $\mathbb{T}$.

Lemma 1.21 (isometric action)**.** *Let F be an isometric wiring and t a closed term. We have that $F(t)$ and $F^\dagger(t)$ are either 0 or another closed term t' (seen as an element of $\mathbb{T}$). It follows that any isometric wiring induces a partial injection on the set of terms.*

Proof. A wiring F is isometric if and only if $F^\dagger$ is so we can focus on $F(t)$: because $F = \sum_i u_i \leftharpoonup t_i$ with the t_i pairwise disjoint, then t match at most one of the t_i and therefore the sum $F(t)$ can contain at most one element, with coefficient 1.

Then the action of an isometric wiring on closed terms is a partial function, with a partial inverse given by the action of $F^\dagger$. □

1.3. **Tensor Product and Permutations.** We now define the representation in the unification algebra $\mathcal{U}$ of structures that provide more expressivity. Thanks to the notion of tensor product, we will build wirings and subalgebras that are split into components computing independently. Unbounded tensor products will allow to represent a potentially unbounded number of data stores. Finite permutations have a natural representation in the algebra that acts on the unbounded tensor product, allowing to represent manipulation of these stores.

All this will provide enough room and computational principles to represent in Section 3.2 a basic model of computation, with pointers and internal states.

Definition 1.22 (tensor product)**.** Let $u \leftharpoonup v$ and $t \leftharpoonup w$ be two flows. Suppose we have chosen representatives of these renaming classes that have their sets of variables disjoint. We define their *tensor product* as $(u \leftharpoonup v) \dot{\otimes} (t \leftharpoonup w) := u \bullet t \leftharpoonup v \bullet w$. The operation is extended to wirings by bilinearity.

Given two $*$-algebras $\mathcal{A}, \mathcal{B}$, we define their tensor product as the $*$-algebra $\mathcal{A} \dot{\otimes} \mathcal{B}$ spawned by

$$\left\{ F \dot{\otimes} G \;\middle|\; F \in \mathcal{A}, G \in \mathcal{B} \right\}$$

This actually defines an embedding of the usual algebraic tensor product into $\mathcal{U}$, which means in particular that $(F \dot{\otimes} G)(P \dot{\otimes} Q) = (FP) \dot{\otimes} (GQ)$. As for $\bullet$, we will omit the parentheses for $\dot{\otimes}$ and write $\mathcal{A} \dot{\otimes} \mathcal{B} \dot{\otimes} \mathcal{C}$ for $\mathcal{A} \dot{\otimes} (\mathcal{B} \dot{\otimes} \mathcal{C})$.

Once we have the basic tensor product as a building block, we can define the unbounded one by putting together bigger and bigger tensor powers of the same $*$-algebra, with a variable in the end standing for the fact that the size is not specified in advance.

Definition 1.23 (unbounded tensor)**.** Let $\mathcal{A}$ be a $*$-algebra, we define the $*$-algebras $\mathcal{A}^{\otimes n}$ for all $n \in \mathbb{N}$ as (letting $\mathcal{I} := \{\lambda I \mid \lambda \in \mathbb{C}\}$, with $I = x \leftharpoonup x$ as in Definition 1.10)

$$\mathcal{A}^{\otimes 0} := \mathcal{I} \quad \text{and} \quad \mathcal{A}^{\otimes n+1} := \mathcal{A} \dot{\otimes} \mathcal{A}^{\otimes n}$$

and the $*$-algebra $\mathcal{A}^{\otimes \infty}$ spawned by $\bigcup_{n \in \mathbb{N}} \mathcal{A}^{\otimes n}$.

We consider that finite permutations can be composed even when their domain of definition do not match, and get a natural representation of them based on the binary function symbol $\bullet$.

Definition 1.24 (representation)**.** To a permutation $\sigma \in \mathfrak{S}_n$ we associate the flow

$$[\sigma] := x_1 \bullet x_2 \bullet \cdots \bullet x_n \bullet y \leftharpoonup x_{\sigma(1)} \bullet x_{\sigma(2)} \bullet \cdots \bullet x_{\sigma(n)} \bullet y$$

A permutation $\sigma \in \mathfrak{S}_n$ can act on the first n components of the unbounded tensor product (Definition 1.23) by swapping them and leaving the rest unchanged. The wirings $[\sigma]$ internalize this action: in the above definition, the variable y at the end stands for the components that are not affected.

Example 1.25. Let $\tau \in \mathfrak{S}_2$ be the permutation swapping the two elements of $\{1, 2\}$ and $U_1 \dot{\otimes} U_2 \dot{\otimes} U_3 \dot{\otimes} I \in \mathcal{U}^{\otimes 3} \subseteq \mathcal{U}^{\otimes \infty}$.

We have $[\tau] = x_1 \bullet x_2 \bullet y \leftharpoonup x_2 \bullet x_1 \bullet y$ and $[\tau](U_1 \dot{\otimes} U_2 \dot{\otimes} U_3 \dot{\otimes} I)[\tau]^\dagger = U_2 \dot{\otimes} U_1 \dot{\otimes} U_3 \dot{\otimes} I$.

In Section 3, we will consider the algebra spawned by these representations of permutations as the basic components of logarithmic space programs in $\mathcal{U}$.

Definition 1.26 (permutation algebra)**.** For $n \in \mathbb{N}$ we set $[\mathfrak{S}_n] := \{[\sigma] \mid \sigma \in \mathfrak{S}_n\}$ and $\mathcal{S}_n$ as the $*$-algebra spawned by $[\mathfrak{S}_n]$.

We define then the *permutation algebra* $\mathcal{S}$ as the $*$-algebra spawned by $\bigcup_{n\in\mathbb{N}} \mathcal{S}_n$.

2. Words, Observations and Normativity

The resolution algebra $\mathcal{U}$ embeds its own mechanism of execution, unification, and we saw how permutations could be represented in it.This is the general environment where the rest of this work is going to take place.

At this stage, there is no distinction between data and programs, functions and inputs. In this section, we single out two subsets of the algebra: one will represent data, the other corresponds to programs. In Section 2.2 we will see how to address through the notion of *normativity* the fact that many wirings can represent the same data in our algebraic view. This will lead to the definition of an acceptance predicate, based on nilpotency:

Definition 2.1 (nilpotency)**.** A wiring F is *nilpotent* if $F^n = 0$ for some $n \in \mathbb{N}$.

In the geometry of interaction models which are the intuitive starting point of this work, this corresponds to strong normalization, i.e. termination of computation. Note that this makes the acceptance only semi-decidable in general [Bag14]: one can always compute iterations of a wiring and eventually reach 0, but there is no general algorithm to decide if a wiring is *never* going to reach 0. However, we will consider in Section 3.1 a particularly simple kind of wirings with an acceptance problem simplified to the point it becomes a logarithmic space problem.

We will consider words on alphabet as our data, although more complex datatypes could be represented as long as they enjoy a representation in λ-calculus, following the same pattern.

2.1. Representing Computation: Words and Observations.

The representation of words over an alphabet, seen here as a set of constant symbols, in the resolution algebra directly comes from the translation of the representation of words in λ-calculus (or in linear logic) and their interpretation in geometry of interaction models [Gir89a, Gir95a].

This proof-theoretic origin is an useful guide for intuition, but we can give a direct definition of the notion.

Notation 2.2. *We fix distinguished constant symbols* $\mathtt{l}$, $\mathtt{r}$ *and* $\star$, *with* $\star \notin \Sigma$ *and we write* $u \leftrightharpoons v$ *the sum* $u \leftharpoonup v + v \leftharpoonup u$. *We also let* $\mathcal{I} := \{\lambda I \mid \lambda \in \mathbb{C}\}$ *as we did in Definition 1.23.*

Definition 2.3 (word representation)**.** From now on we suppose fixed a set $\mathtt{P}$ of constant symbols, the *position constants* and denote with $\mathtt{c}_i$ the symbols of the alphabet Σ.

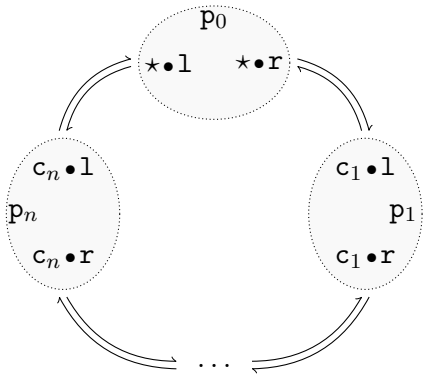
Let $W = \mathtt{c}_1 \dots \mathtt{c}_n$ be a word over Σ and $\mathtt{p}_0, \mathtt{p}_1, \dots, \mathtt{p}_n$ be distinct position constants. The *representation* $W(\mathtt{p}_0, \mathtt{p}_1, \dots, \mathtt{p}_n)$ of W with respect to $\mathtt{p}_0, \mathtt{p}_1, \dots, \mathtt{p}_n$ is an isometric

wiring (Definition 1.17), defined as

$$\begin{aligned} W(\mathtt{p}_0, \mathtt{p}_1, \dots, \mathtt{p}_n) = \ & \star\bullet\mathtt{r}\bullet x\bullet(\mathtt{p}_0\bullet y) \leftrightharpoons \mathtt{c}_1\bullet\mathtt{l}\bullet x\bullet(\mathtt{p}_1\bullet y) + \\ & \mathtt{c}_1\bullet\mathtt{r}\bullet x\bullet(\mathtt{p}_1\bullet y) \leftrightharpoons \mathtt{c}_2\bullet\mathtt{l}\bullet x\bullet(\mathtt{p}_2\bullet y) + \\ & \vdots \\ & \mathtt{c}_n\bullet\mathtt{r}\bullet x\bullet(\mathtt{p}_n\bullet y) \leftrightharpoons \star\bullet\mathtt{l}\bullet x\bullet(\mathtt{p}_0\bullet y) \end{aligned}$$

In this definition, the position constants $\mathtt{p}_0, \mathtt{p}_1, \dots, \mathtt{p}_n$ can be understood as the addresses of memory cells holding the symbols $\star, \mathtt{c}_1, \dots, \mathtt{c}_n$. This simulates the order naturally present when an input word is written on a tape (where each cell has one or two neighbours) in a context where the commutative addition cannot implement any form of order: we henceforth have to *tag* each symbol with a position.

More generally, this representation of words is to be understood as *dynamic*: we may think of a series of *movement instructions* from a symbol to the next or the previous for a kind of pointer machine. This is why each term of the form $\mathtt{c}_i \bullet \dots \bullet \mathtt{p}_i \bullet \dots$ comes in two distinct versions using either $\mathtt{r}$ or $\mathtt{l}$ (as "right/next", "left/previous") setting different responses to different directions in reading the input word. Moreover, the general shape of the wiring is circular, i.e. when reaching the end of the word, we return to the position holding $\star$. This can be pictured as follows:



This point of view will be at work in the proof of Theorem 3.11, where we will show that computations of a particular class of pointer machines can be represented in our context. In that perspective, the $\dots\bullet x\bullet\dots y\bullet\dots$ part will serve to preserve some information relative to the machine, such as its internal state or the positions of additional pointers.

To identify in which $*$-algebra all representations of word live, let us define some notations and sub-algebras.

Definition 2.4. We write, with a slight abuse of notation that identifies algebras and sets of symbols generating them,

- $\Sigma_{\mathtt{lr}}$ the $*$-algebra generated by flows of the form $\mathtt{s}\bullet\mathtt{d} \leftharpoondown \mathtt{s}'\bullet\mathtt{d}'$ with $\mathtt{s}, \mathtt{s}' \in \Sigma \cup \{\star\}$ and $\mathtt{d}, \mathtt{d}' \in \{\mathtt{l}, \mathtt{r}\}$
- $\mathtt{P}$ the $*$-algebra generated by flows of the form $\mathtt{p} \leftharpoondown \mathtt{p}'$ with $\mathtt{p}, \mathtt{p}' \in \mathtt{P}$,
- $\mathtt{Q}$ the $*$-algebra generated by flows of the form, $\mathtt{q} \leftharpoondown \mathtt{q}'$ with $\mathtt{q}, \mathtt{q}'$ being *state constants*, whose set is denoted $\mathtt{Q}$.

Proposition 2.5. *Any representation of a word* $W(\mathtt{p}_0, \mathtt{p}_1, \dots, \mathtt{p}_n)$ *lies in the* $*$*-algebra*

$$\mathcal{W} = \Sigma_{\mathtt{lr}} \dot{\otimes} \mathcal{I} \dot{\otimes} \mathtt{P}^{\otimes 1}$$

which we call the word algebra.

Let us turn now to the definition of *observations* that will correspond to programs computing on representations of words.

We give a general notion first, which we will instantiate in Section 3 to get a class of observations that captures logarithmic space computation, based on the representation of permutations over an unbounded tensor presented in the previous section.

Definition 2.6 (observations)**.** Given a $*$-algebra $\mathcal{A}$, an *observation by* $\mathcal{A}$ is an element of $\mathcal{O}[\mathcal{A}]^+$ where

$$\mathcal{O}[\mathcal{A}] = \Sigma_{\mathtt{lr}} \dot{\otimes} \mathcal{A}$$

and the $(\cdot)^+$ notation refers to the set of concrete (Definition 1.17) wirings obtained from $\mathcal{O}[\mathcal{A}]$. Moreover when an observation by $\mathcal{A}$ happens to be an isometric wiring, we will call it an *isometric observation* by $\mathcal{A}$.

In case $\mathcal{A}$ is the whole unification algebra $\mathcal{U}$ we call the elements of $\mathcal{O}[\mathcal{U}]^+$ simply *observations*.

2.2. **Independence from Representations: Normativity.** We now study how representations of words and observations interact, leading to a notion of acceptance. The basic idea is that an observation ϕ accepts a word W if the wiring $\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n)$ is nilpotent (Definition 2.1), but we want to make sure that the notion is independent of the choice of a specific representation of W.

This could be enforced at a basic syntactic level: we could forbid the observations to use the position constants, which are the only source of variability from one representation to the other. But we would like to give a more algebraic view of this idea, to tend to a more abstract vision: this leads to the notion of *normative pair* introduced by J.-Y. Girard [Gir12].

Definition 2.7 (automorphism)**.** An *automorphism* of a $*$-algebra $\mathcal{A}$ is an injective linear application $\varphi : \mathcal{A} \to \mathcal{A}$ such that for all $F, G \in \mathcal{A}$:

$$\varphi(FG) = \varphi(F)\varphi(G) \quad \text{and} \quad \varphi(F^\dagger) = \varphi(F)^\dagger$$

For instance $\varphi(U_1 \dot{\otimes} U_2) = U_2 \dot{\otimes} U_1$ defines an automorphism of $\mathcal{U} \dot{\otimes} \mathcal{U}$.

An automorphism φ therefore preserves the algebraic properties of elements of $\mathcal{A}$: in particular, $\varphi(A)$ is nilpotent if and only if A is.

Notation 2.8. *If* φ *is an automorphism of* $\mathcal{A}$ *and* ψ *is an automorphism of* $\mathcal{B}$*, we write* $\varphi \dot{\otimes} \psi$ *the automorphism of* $\mathcal{A} \dot{\otimes} \mathcal{B}$ *defined for all* $A \in \mathcal{A}, B \in \mathcal{B}$ *as*

$$(\varphi \dot{\otimes} \psi)(A \dot{\otimes} B) = \varphi(A) \dot{\otimes} \psi(B)$$

Keeping in mind that an automorphism is a transformation that preserves the algebraic properties, we define the notion of normative pair as a pair of $*$-algebras such that an automorphism of one of them can be extended to act as the identity on the other one.

Definition 2.9 (normative pair)**.** A pair $(\mathcal{A},\mathcal{B})$ of $*$-algebras is a *normative pair* whenever any automorphism φ of $\mathcal{A}$ can be extended into an automorphism $\overline{\varphi}$ of the $*$-algebra $\mathcal{E}$ generated by $\mathcal{A}\cup\mathcal{B}$ such that $\overline{\varphi}(B)=B$ for any $B\in\mathcal{B}\subseteq\mathcal{E}$.

A trivial example would be that of commuting $\mathcal{A}$ and $\mathcal{B}$: then any element of $\mathcal{E}$ can be written as a sum of AB with $A\in\mathcal{A}$ and $B\in\mathcal{B}$, which allows then to define $\overline{\varphi}(AB)=\varphi(A)B$ consistently. But this case is of little interest since when A,B commute $(AB)^n=A^nB^n$ so that there is no real interaction between A and B: they "pass through" each other without communicating.

The two following propositions set the basis for a notion of acceptance and rejection independent of the representation of a word, as soon as normative pairs are involved.

Proposition 2.10 (automorphic representations)**.** *Any two representations $W(\mathtt{p}_0,\dots,\mathtt{p}_n)$, $W(\mathtt{p}'_0,\dots,\mathtt{p}'_n)$ of the same word W are automorphic: there is an automorphism φ of $\mathtt{P}^{\otimes 1}$ such that*

$$(\mathrm{Id}_{\Sigma_{\mathtt{lr}}}\dot{\otimes}\varphi)\big(W(\mathtt{p}_0,\dots,\mathtt{p}_n)\big)=W(\mathtt{p}'_0,\dots,\mathtt{p}'_n)$$

Proof. Consider any bijection $f:\mathtt{P}\to\mathtt{P}$ such that $f(\mathtt{p}_i)=\mathtt{p}'_i$ for all i.

Then set $\varphi(x\bullet v\bullet y\leftharpoonup x\bullet w\bullet y)=x\bullet f(v)\bullet x\leftharpoonup x\bullet f(w)\bullet y$, extended by linearity. □

Proposition 2.11 (nilpotency and normative pairs)**.** *Let $(\mathcal{A},\mathcal{B})$ be a normative pair and φ an automorphism of $\mathcal{A}$. Let $F\in\Sigma_{\mathtt{lr}}\dot{\otimes}\mathcal{A}, G\in\Sigma_{\mathtt{lr}}\dot{\otimes}\mathcal{B}$ and $\psi=\mathrm{Id}_{\Sigma_{\mathtt{lr}}}\dot{\otimes}\varphi$.*

Then GF is nilpotent if and only if $G\psi(F)$ is nilpotent.

Proof. Let $\overline{\varphi}$ be the extension of φ as in Definition 2.9 and $\overline{\psi}=\mathrm{Id}_{\Sigma_{\mathtt{lr}}}\dot{\otimes}\overline{\varphi}$.

We have for all $n\neq 0$ that $(G\psi(F))^n=(\overline{\psi}(G)\overline{\psi}(F))^n=(\overline{\psi}(GF))^n=\overline{\psi}((GF)^n)$.

By injectivity of $\overline{\psi}$, $(G\psi(F))^n=0$ if and only if $(GF)^n=0$. □

In view of this last proposition, as we know that words are in $\mathcal{W}=\Sigma_{\mathtt{lr}}\dot{\otimes}(\mathcal{I}\dot{\otimes}\mathtt{P}^{\otimes 1})$ and observation by $\mathcal{B}$ are in $\mathcal{O}[\mathcal{B}]^+=(\Sigma_{\mathtt{lr}}\dot{\otimes}\mathcal{B})^+$, we understand that it is enough that $(\mathcal{I}\dot{\otimes}\mathtt{P}^{\otimes 1},\mathcal{B})$ constitutes a normative pair to get the expected result.

Corollary 2.12 (independence)**.** *If $(\mathcal{I}\dot{\otimes}\mathtt{P}^{\otimes 1},\mathcal{B})$ is a normative pair, W a word and ϕ an observation by B, the product $\phi W(\mathtt{p}_0,\dots,\mathtt{p}_n)$ is nilpotent for one choice of $(\mathtt{p}_0,\dots,\mathtt{p}_n)$ if and only if it is nilpotent for all choices of $(\mathtt{p}_0,\dots,\mathtt{p}_n)$.*

In the next section, we will consider a particular case of observations, namely where $\mathcal{B}=\mathtt{Q}\dot{\otimes}\mathcal{S}$, where $\mathtt{Q}$ is the algebra generated by the flows of state constants from Definition 2.4 and $\mathcal{S}$ is the permutation algebra from Definition 1.26.

Theorem 2.13 (normativity)**.** *For any $\mathcal{A},\mathcal{C}$ the pair $\big(\mathcal{I}\dot{\otimes}\mathcal{A}^{\otimes 1},\mathcal{C}\dot{\otimes}\mathcal{S}\big)$ is normative. In particular, $(\mathcal{I}\dot{\otimes}\mathtt{P}^{\otimes 1},\mathtt{Q}\dot{\otimes}\mathcal{S})$ is normative.*

Proof. Consider φ is an automorphism of $\mathcal{I}\dot{\otimes}\mathcal{A}^{\otimes 1}$. It can be written as

$$\varphi(I\dot{\otimes}G\dot{\otimes}I)=I\dot{\otimes}\psi(G)\dot{\otimes}I$$

for all G, with ψ an automorphism of $\mathcal{A}$. Now, the $*$-algebra generated by $\mathcal{I}\dot{\otimes}\mathcal{A}^{\otimes 1}$ and $\mathcal{C}\dot{\otimes}\mathcal{S}$ can be identified as finite sums of elements of the set

$$\big\{\sigma F\;\big|\;\sigma\in\mathcal{S}\text{ and }F\in\mathcal{A}^{\otimes\infty}\big\}$$

We set for $F = F_1 \dot{\otimes} \cdots \dot{\otimes} F_n \dot{\otimes} I \in \mathcal{A}^{\otimes n}$,

$$\tilde{\varphi}(F) = \psi(F_1) \dot{\otimes} \cdots \dot{\otimes} \psi(F_n) \dot{\otimes} I$$

which extends into an automorphism of $\mathcal{A}^{\otimes\infty}$ by linearity. Finally, we extend $\tilde{\varphi}$ to $\mathcal{A}$ by $\overline{\varphi}(\sigma F) = \sigma\tilde{\varphi}(F)$. □

Remark 2.14. This result is likely to be generalized: the permutation algebra *acts* on the infinite tensor product, and through this action the whole tensor product $\mathcal{A}^{\otimes\infty}$ is generated by $\mathcal{A}^{\otimes 1}$. With some adjustment it should be possible to show that given such a situation, one always get a normative pair.

We can then define the notion of the language recognized by an observation, thanks to Corollary 2.12 that makes it insensitive to a particular choice of position constants.

Definition 2.15 (language of an observation)**.** Let ϕ be an observation by $\mathcal{B}$ satisfying the hypothesis of Corollary 2.12,i.e. $\mathcal{B}$ is of the form $\mathcal{C} \dot{\otimes} \mathcal{S}$. The *language recognized by* ϕ is the following set:

$$\mathcal{L}(\phi) = \{W \text{ word over } \Sigma \mid \phi W(\mathtt{p}_0, \dots, \mathtt{p}_n) \text{ nilpotent for any } (\mathtt{p}_0, \dots, \mathtt{p}_n)\}$$

3. Wirings and Logarithmic Space

Now that we have defined our framework and showed how observations compute, we fix a specific class of observations and study the complexity of deciding whenever such one of its member accepts a word (Section 3.1). We then show in Section 3.2 that the languages recognized correspond exactly, depending on the isometricity of the observation (Definition 1.18), to the Logspace or NLogspace (written (N)Logspace if we don't want to be specific).

Definition 3.1 ($\mathfrak{S}$-observation)**.** We consider the $*$-algebra $\mathtt{Q} \dot{\otimes} \mathcal{S}$ as in Theorem 2.13 and call $\mathfrak{S}$-observation the observations by $\mathtt{Q} \dot{\otimes} \mathcal{S}$ (Definition 2.6). More explicitely, $\mathfrak{S}$-observations are finite sums of flows of the form

$$(\mathtt{c}' \bullet \mathtt{d}' \bullet \mathtt{q}' \leftharpoondown \mathtt{c} \bullet \mathtt{d} \bullet \mathtt{q}) \dot{\otimes} [\sigma]$$

where $\mathtt{c}, \mathtt{c}' \in \Sigma$, $\mathtt{d}, \mathtt{d}' \in \mathtt{LR}$, $\mathtt{q}, \mathtt{q}' \in \mathtt{Q}$ and σ is a permutation.

We already shown that the notion of acceptance (Definition 2.15) is well-defined since $(\mathcal{I} \dot{\otimes} \mathtt{P}^{\otimes 1}, \mathtt{Q} \dot{\otimes} \mathcal{S})$ is a normative pair.

Deciding nilpotency in this specific case amounts to build a finite-dimensional vector space where we can observe the "relevant computation" taking place in a finitary way (remember that the nilpotency problem is only semi-decidable in general). We introduce the notion of *separating space* (Definition 3.3) and give a logarithmic space-algorithm based on this notion.

That any (N)Logspace language, or predicate, can be decided by a $\mathfrak{S}$-observation will be proven thanks to *pointer machines* (Definition 3.6), a model of computation designed to be easily encoded. This model comes as a rephrasing of read-only Turing machines, or more precisely as a modification of two-way multi-head finite automata, known to capture (N)Logspace [Har72, Aub15]. Unification will act as a "hard-wired" way to represent execution, thanks to a dialogue between the representation of the input and the observation.

3.1. **Soundness of Observations.** The aim of this subsection is to prove the following theorem:

Theorem 3.2 (space soundness)**.** *Let ϕ be $\mathfrak{S}$-observation, its language $\mathcal{L}(\phi)$ is decidable in* NLogspace*. If moreover ϕ is isometric, then $\mathcal{L}(\phi)$ is decidable in* Logspace*.*

The proof is given later on, p. 14, but one should notice that the result stands for the complements of these languages, but as NLogspace = co-NLogspace by the Immerman-Szelepcsényi [Imm88, Sze88] theorem, this makes no difference. Indeed, if one looks closely to the definition of the language of an observation (Definition 2.15), one may notice that acceptation rests on the nilpotency of the wiring, which supposes that *all branches of computation ends*: observation looks like a "universally non-deterministic" model of computation.

This theorem will require the notion of *computation space*: finite dimensional[3] subspaces of the vector space spawned by closed terms $\mathbb{T}$ (Definition 1.20) on which we will be able to observe all the behaviour of certain wirings. It can be understood as the place where all the relevant computation takes place, which we call a *separating space*.

The rest of this subsection is devoted to the introduction of this notion of computation space, and to prove that a computation space can be computed from a observation and a word, and is indeed separating. Finally, we prove that deciding if a computation space is nilpotent—which is equivalent to an observation applied to an input being nilpotent—can be done with logarithmic space, thus proving Theorem 3.2.

Definition 3.3 (separating space)**.** A subspace E of $\mathbb{T}$ is *separating* for a wiring $F \in \mathbb{T}$ if $F(E) \subseteq E$ and if $F^k(E) = 0$ implies $F^k = 0$.

If we observe the computation of F on E, it cannot "step outside E". On the other hand, the fact that $F^k(E) = 0$ implies $F^k = 0$ means that it is enough to check that a certain iteration of F cancels on the space E to conclude that it cancels everywhere.

Definition 3.4 (computation space)**.** Let $\{\mathtt{p}_0, \dots, \mathtt{p}_n\}$ be a set of distinct position constant and ϕ a $\mathfrak{S}$-observation. Let $N(\phi)$ be the smallest integer and $\mathtt{S}(\phi)$ the smallest (finite) subalgebra of $\mathtt{Q}$ such that $\phi \in \Sigma_{\mathtt{lr}} \dot{\otimes} \mathtt{S}(\phi) \dot{\otimes} \mathcal{S}_{N(\phi)}$:

The *computation space* of ϕ associated to the positions $\mathtt{p}_i$, denoted $\mathtt{Comp}_\phi(\mathtt{p}_0, \dots, \mathtt{p}_n)$, is the subspace of $\mathbb{T}$ generated by closed terms of the form

$$\mathtt{c} \bullet \mathtt{d} \bullet \mathtt{q} \bullet (a_1 \bullet \cdots \bullet a_{N(\phi)} \bullet \star)$$

where $\mathtt{c} \in \Sigma \cup \{\star\}$, $\mathtt{d} \in \mathtt{LR}$, $\mathtt{q} \in \mathtt{S}(\phi)$ and $\forall 1 \leqslant i \leqslant N(\phi)$, $a_i \in \{\mathtt{p}_0, \dots, \mathtt{p}_n\}$.

Denoting $|A|$ the cardinal of A, the dimension of $\mathtt{Comp}_\phi(\mathtt{p}_0, \dots, \mathtt{p}_n)$ is

$$(|\Sigma| + 1) \times 2 \times |\mathtt{S}(\phi)| \times (n+1)^{N(\phi)}$$

which is polynomial in n.

Lemma 3.5 (separation)**.** *For any $\mathfrak{S}$-observation ϕ and any word W, $\mathtt{Comp}_\phi(\mathtt{p}_0, \dots, \mathtt{p}_n)$ is separating for the wiring $\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n)$.*

Proof. Immediate given how Comp was defined. □

[3] We recall that a vector space is finite dimensional if the cardinal of its basis is finite.

Now we proceed to prove Theorem 3.2: we provide an algorithm deciding the nilpotency of $\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n)$ in logarithmic space, based on the above lemma.

Proof. (of Theorem 3.2) We define the non-deterministic algorithm below. It takes as an input a word W of length n. Remark that the observation ϕ being a constant, one can compute once and for all $N(\phi)$ and $\mathtt{S}(\phi)$.

1: $D \leftarrow (|\Sigma|+1) \times 2 \times |\mathtt{S}(\phi)| \times (n+1)^{N(\phi)}$
2: $C \leftarrow 0$
3: pick a term $v \in \mathtt{Comp}_\phi(\mathtt{p}_0, \dots, \mathtt{p}_n)$
4: **while** $C \leq D$ **do**
5: **if** $(\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n))(v) = 0$ **then**
6: **return** ACCEPT
7: **end if**
8: pick a term $v' \in (\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n))(v)$
9: $v \leftarrow v'$
10: $C \leftarrow C + 1$
11: **end while**
12: **return** REJECT

All computation paths (the "pick" at lines 3 and 8 being non-deterministic choices) accept if and only if $(\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n))^n(\mathtt{Comp}_\phi(\mathtt{p}_0, \dots, \mathtt{p}_n)) = 0$ for some n lesser or equal to the dimension D of the computation space $\mathtt{Comp}_\phi(\mathtt{p}_0, \dots, \mathtt{p}_n)$. By Lemma 3.5, this is equivalent to $\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n)$ being nilpotent, as the computation space is a separating space for $\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n)$.

The terms chosen at lines 3 and 8 are representable by an integer of size at most D, and we need to store only two such terms at the same time, as one is replaced by the other at line 9, every time we go through the **while**-loop. We already mentioned that the dimension D of the computation space is polynomial in the size of the input (Definition 3.4). As C is bounded by D, both integers can be stored in a space logarithmic in n.

The computation of $(\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n))(v)$ at lines 5 and 8 and can be performed in logarithmic space by Theorem 1.9.

Moreover, if ϕ is an isometric wiring, $(\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n))(v)$ consists of a single term instead of a sum by Lemma 1.21, and there is therefore no non-deterministic choice to be made at line 8. It is then enough to run the algorithm enumerating all terms of $\mathtt{Comp}_\phi(\mathtt{p}_0, \dots, \mathtt{p}_n)$ at line 3 to determine the nilpotency of $\phi W(\mathtt{p}_0, \dots, \mathtt{p}_n)$. □

We can have a more graph-oriented view of the algorithm. Picture the elements of $\mathtt{Comp}_\phi(\mathtt{p}_0, \dots, \mathtt{p}_n)$ as vertices of a graph: they represent all the possible terms the computation could reach. The wiring induces the edges between those terms: if u is in the image of v then they both belong to $\mathtt{Comp}_\phi(\mathtt{p}_0, \dots, \mathtt{p}_n)$ and we can draw an edge between them. The iteration of this procedure gives a set of reachable terms, the trace of all possible computation. To know if this wiring is nilpotent, one only has to check whether there is a cycle in the graph obtained. This is a typical LOGSPACE problem, and that this graph can be built in LOGSPACE manly rests on the fact that matching is in LOGSPACE too. The algorithm above performs both tasks (building the graph and looking for cycles) at the same time.

3.2. **Completeness: Representing Pointer Machines as Wirings.** To prove the converse of Theorem 3.2, we will prove that wirings can encode a special kind of read-only multi-head Turing Machine: pointers machines. The definition of this model will be guided by our understanding of the wirings' way of computing: they don't have the ability to write or to store information, and acceptance will be defined as termination of all paths of computation.

For a survey of this topic, one may consult the first author's thesis [Aub13, Chapter 4], the main novelty of this part of our work is to notice that reversible computation is represented by isometric operators.

Definition 3.6 (pointer machine)**.** A *pointer machine* over an alphabet Σ is a tuple $(N, \mathtt{S}, \Delta)$ where

- $N \neq 0$ is an integer, the *number of pointers*,
- $\mathtt{S}$ is a finite set, the *states* of the machine,
- $\Delta \subseteq (\Sigma \times \mathtt{LR} \times \mathtt{S}) \times (\Sigma \times \mathtt{LR} \times \mathtt{S}) \times \mathfrak{S}_N$, are the *transitions* of the machine (we will write the transitions $(\mathtt{c}, \mathtt{d}, s) \to (\mathtt{c}', \mathtt{d}', s') \times \sigma$, for readability).

A pointer machine will be called *deterministic* if for any $A \in \Sigma \times \mathtt{LR} \times \mathtt{S}$, there is at most one $B \in \Sigma \times \mathtt{LR} \times \mathtt{S}$ and one $\sigma \in \mathfrak{S}_N$ such that $A \to B \times \sigma \in \Delta$. In that case we can see Δ as a partial function, and we say that the pointer machine is *reversible* if Δ is a partial injection.

We call the first of the N pointers the *main* pointer, it is the only one that can move (it will be moved by the representation of the integer, as we shall see below). The other pointers are referred to as the *auxiliary* pointers. An auxiliary pointer will be able to become the main pointer during the computation thanks to permutations.

Definition 3.7 (configuration)**.** Given the length n of a word $W = \mathtt{c}_1 \dots \mathtt{c}_n$ over Σ and a pointer machine $M = (N, \mathtt{S}, \Delta)$, a *configuration* C of (M, n) is an element of

$$\Sigma \times \mathtt{LR} \times \mathtt{S} \times \{0, 1, \dots, n\}^N.$$

The element of $\mathtt{S}$ is the state of the machine and the element of Σ is the symbol the main pointer points at. The element of $\mathtt{LR}$ is the direction of the next move of the main pointer, and the elements of $\{0, 1, \dots, n\}^N$ correspond to the positions of the (main and auxiliary) pointers on the input.

As the input tape is considered cyclic with a special symbol marking the beginning of the word (recall Definition 2.3), the pointer positions are *modulo* $n+1$ integers for an input word of length n.

Definition 3.8 (transition)**.** Let W be a word of length n and $M = (N, \mathtt{S}, \Delta)$ be a pointer machine. A *transition* of M on input W is a triple of configurations

$$\mathtt{c}, \mathtt{d}, s, (p_1, \dots, p_N) \xrightarrow{\texttt{MOVE}} \mathtt{c}', \overline{\mathtt{d}}, s, (p'_1, \dots, p'_N) \xrightarrow{\texttt{SWAP}} \mathtt{c}'', \mathtt{d}', s', (p'_{\sigma(1)}, \dots, p'_{\sigma(N)})$$

such that

(1) if $\mathtt{d} \in \mathtt{LR}$, $\overline{\mathtt{d}}$ is the other element of $\mathtt{LR}$,
(2) $p'_1 = p_1 + 1$ if $\mathtt{d} = \mathtt{l}$ and $p'_1 = p_1 - 1$ if $\mathtt{d} = \mathtt{r}$,
(3) $p'_i = p_i$ for $i \neq 1$,
(4) $\mathtt{c}$ (resp. $\mathtt{c}'$) is the symbol at position p_1 (resp. p'_1) of W,
(5) and $(\mathtt{c}', \overline{\mathtt{d}}, s) \to (\mathtt{c}'', \mathtt{d}', s') \times \sigma$ belongs to Δ.

There is no constraint on c'', but every time this value differs from the symbol pointed by $p'_{\sigma(1)}$, the computation will halt on the next MOVE phase, because there is a mismatch between the value that is supposed to have been read and the actual symbol of W stored at this position, and that would contradict the first part of item 4.

In terms of wirings, the MOVE phase corresponds to the application of the representation of the word, whereas the SWAP phase corresponds to the application of the observation. One

way to present it is to draw attention on the fact that, in the drawing page 9, there is no arrow *inside* the "domain" of a position constant. Stated differently, *observations only* can make the computation evolves from a direction constant to another, and *representation of the word only* can update the constant position.

Definition 3.9 (acceptance)**.** A pointer machine M accepts a word W of length n if for all configuration C_0 of (M, n), all sequences of transitions

$$\left(C_0 \xrightarrow{\texttt{MOVE}} C_0' \xrightarrow{\texttt{SWAP}} C_0'' = C_1 \xrightarrow{\texttt{MOVE}} \cdots \xrightarrow{\texttt{SWAP}} C_{k-1}'' = C_k\right)$$

are finite. We write $\mathcal{L}(M)$ the set of words accepted by M.

This means informally that a pointer machine accepts a word if it cannot ever loop, from whatever configuration it starts from. That a lot of paths of computation stops and accepts "wrongly" is no worry, since only rejection is meaningful: our pointer machines compute in a "universally non-deterministic" way, to stick to the acceptance condition of wirings, nilpotency.

Proposition 3.10 (space and pointer machines)**.** *If $L \in$ NLogspace, then there exist a pointer machine M such that $\mathcal{L}(M) = L$. Moreover, if $L \in$ Logspace then M can be chosen to be reversible.*

Proof. We proceed with a step-by-step transformation from Turing machines deciding L to pointer machines deciding the same language, through automata.

If $L \in$ NLogspace, then by the Immerman-Szelepcsényi [Imm88, Sze88] theorem there exists a Turing machine in co-NLogspace that decides it. If $L \in$ Logspace, as deterministic and reversible logarithmic space coincides [LMT00], we take a reversible Turing machine that decides it.

It is then possible to design two (non-)deterministic multi-head finite automata that recognize the same languages [Har72]. Those automata are read-only version of the Turing machines that are closer to pointer machines, but still differ on some features. We now prove that they can be re-arranged to fit the definition of pointer machines.

First, we modify their accepting condition to be "halt" and their rejection to be "loop": it is always possible to adjust automata so that no transition is defined from an accepting state, and so that rejection makes the automata never halts. The introduction of loops require some care: an "in-place loop" would prevent the pointer machine to ever stop no matter the input, so we implement loops thanks to a "re-initialization" ("go back to an initial state, with all your heads at position $\mathtt{p}_0$, reading $\star$"). Looping is in that setting more of a "check forever that you do not halt on that input" [AS15, Section 5.1][AS14, Section 6.2.3]. In the non-deterministic case, this amounts to define acceptance as "all branches of computation halt", and rejection as "at least one branch never halts".

Then, we transform the transition function so that at most one head moves at each transition. But this is not enough: the transition function of our pointer machines reads only one symbol at a time, the one pointed by the main pointer. We operate a sort of currying to the transition function of the automata: it reads only one symbol at a time, and the other symbols that were read by the other heads as well as the direction of their last move are encoded in the states.

Finally, we re-arrange the automata so that swapping the heads and moving them on the input are two different phases. We obtained a pointer machine. □

The first author recently proposed a recollection of the classical characterizations of complexity classes by automaton [Aub15]. Once the basics tricks of the transformation of a (N)Logspace-Turing machine into a two-ways multi-head automata are known, it becomes easy to use some classical theorems to get a pointer machine. The pointer machine obtained has more pointer than the automata had heads, and the number of state grew violently, but independently from the input, and without loosing computational power.

As our pointer machines are designed to be easily simulated by wirings, we get the expected result almost for free.

Theorem 3.11 (space completeness)**.** *If $L \in$ NLogspace, then there exist a $\mathfrak{S}$-observation ϕ such that $\mathcal{L}(\phi) = L$. Moreover, if $L \in$ Logspace then ϕ can be chosen isometric.*

Proof. There exists a pointer machine $M = (N, \mathtt{S}, \Delta)$ such that $\mathcal{L}(M) = L$ by Proposition 3.10. We associate to the set of states $\mathtt{S}$ a set of constants that we still write $\mathtt{S}$. To any element $D = (\mathtt{c}, \mathtt{d}, s) \to (\mathtt{c}', \mathtt{d}', s') \times \sigma$ of Δ we associate the flow

$$[D] = (\mathtt{c}' \bullet \mathtt{d}' \bullet s' \leftharpoondown \mathtt{c} \bullet \mathtt{d} \bullet s) \dot{\otimes} [\sigma]$$

which belongs to $\Sigma_{\mathtt{lr}} \dot{\otimes} \mathtt{Q} \dot{\otimes} \mathcal{S}_n$ and we define the $\mathfrak{S}$-observation $[M]$ as $\sum_{D \in \Delta} [D]$.

One can easily check that this translation preserves the language recognized (there is even a step by step simulation of the computation on the word W by the wiring $[M]W(\mathtt{p}_0, \dots, \mathtt{p}_n)$) and relates reversibility with isometricity: in fact, M is reversible if and only if $[M]$ is an isometric wiring. Then, if $L \in$ Logspace, M is deterministic and can always be chosen to be reversible [LMT00]. □

Conclusion

Related Works. The idea to consider the geometry of interaction representation of integers with implicit complexity perspectives is originally due to Girard [Gir12], where one of his motivation was to prove that no representation of the integers was "more standard" than any other (the "normativity" theorem).This approach diverges from the usual complexity results coming from linear logic, that entail a bound on complexity by restricting programs thanks to a type system. In this perspective, limitation on the computational power of observations (representations of programs) comes from *algebraic restrictions*.

A first series of work [Sei12c, Aub13, AS14, AS15] deepened those intuitions by making formal the interpretation of proofs of linear logic in the hyperfinite factor, a type II_1 von Neuman Algebra. Thanks to the representation of infinite operators by matrices, it was proven [Sei12c, AS14] that representations of programs in a specific sub-algebra were characterizing NLogspace. Later on, an additional restriction on the observations, phrased in terms of norm, was proven [Aub13, AS15] to characterize Logspace. As an additional result, it was also discovered that the observations' mechanism of computation was deeply related to automata theory [Aub13, AS15].

The present work and its previous version [AB14] constitute a bridge between this algebraic setting and a more syntactical one that followed. It still uses ad-hoc "pointer machines" and a full algebra to describe the computation of observations. In that perspective, normativity is still seen as mathematical feature, i.e. the existence of automorphisms to

switch from a representation to another without affecting observations, whereas it is a plain α-conversion in the following works.

Two changes in the perspective appeared later on: first, this whole construction could be rephrased in the latest formulation of geometry of interaction [Gir13], which fully uses first-order terms and unification, or matching, to represent linear logic and its execution procedure. This more syntactical presentation allows to isolate restriction on terms as syntactical conditions: *balanced* [ABPS14] and *unary* [ABS15] flows were proven to characterize respectively (N)Logspace and Ptime. Those innovative limits imposed on wiring were discovered thanks to a careful attention paid to automata theory, which is the second change in the perspective. The rephrasing of the *memoization technique* [Coo71]—that was originally invented to prove the Ptime-soundness of pushdown automata—applied to flows permitted to get the first time-bounded characterization of a complexity class in a geometry of interaction setting. Those works benefited from previous characterizations in terms of *unification algebra* [BP01], and constitutes a modern rephrasing of this work. The algebraic framework is lighter, for it uses semi-ring [Bag14] rather than full algebras.

As a by-product, this series of works is now closer to logic programming [Bag14, ABPS14, ABS15] and pretends to highlight with new perspectives this subject.

Future Directions. We built an algebra endowed with an evaluation mechanism relying on unification of first-order terms, that allows to seamlessly represent the execution of programs. Taking as a guiding intuition the functionnal representation of data as functions, we took the Curry-Howard interpretation of λ-terms as proofs to divide our algebra between inputs and observations. We separated them in two different sub-algebras that communicate "just enough": the input cannot interfere with the observation, the observation is insensitive to the choice of the representation (this is the normativity property), and yet they can represent decision of predicates. Using a specific sub-space that represents all possible computations, the computation space, we proved that deciding the outcome of the interaction between an observation and its input was decidable in (N)Logspace. This models, thanks to the representation of permutations and unbounded tensor product, has enough computational power to represent (N)Logspace. To prove it, we had to pay an extra-attention to the peculiarities of this model: it computes as a read-only model, whose heads read and move in a restricted way, and who accepts by halting. It was nevertheless possible to introduce pointer machines, mid-way between observations and two-ways multi-head automata, and to prove that observations could simulate this (N)Logspace-model of computation.

The language of the unification algebra gives a twofold point of view on computation, either through algebraic structures or pointer machines. We may therefore start exploring possible variations of the construction, combining intuitions from both worlds.

The algebraic setting allows for a number of modifications whose computational meaning is still unclear. We considered only the computational features provided by concrete wirings, but one could imagine that negative coefficients would provide a mechanism to interact "at distance" between branches of computation. That may offer the opportunity to represent parallel computation with a mechanism of synchronization, a branch of computation being able to get cancelled by another one. It is also worth mentioning that matrix computation is well-known to relate closely to parallel computation: observations could be evaluated in a parallel setting, providing complexity-bound on time rather than on space.

Pointer machines relate closely to automata theory, which is a vivid research field that should be inspiring. Apart from the PTIME-characterization provided by pushdown automata that was already explored [Bag14, ABS15], relations between our setting and pushdown systems, tree automata or asynchronous automata definitely ought to be studied. This angle could also provide intuitions to tackle the switching from complexity of predicates to complexity of functions, using transducers instead of automata.

Acknowledgement. The authors would like to thanks Jean-Yves Girard for inspiring hints he gave us during the writing of this article. They are also deeply grateful to Paolo Pistone and Thomas Seiller, with whom we pushed forward this promising line of work. Discussions with them modified our way of presenting this work, and we are much obliged.

REFERENCES

[AB14] Clément Aubert and Marc Bagnol. Unification and logarithmic space. In Gilles Dowek, editor, *RTA-TLCA*, volume 8650 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2014.

[ABPS14] Clément Aubert, Marc Bagnol, Paolo Pistone, and Thomas Seiller. Logic programming and logarithmic space. In Jacques Garrigue, editor, *APLAS*, volume 8858 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2014.

[ABS15] Clément Aubert, Marc Bagnol, and Thomas Seiller. Memoization for unary logic programming: Characterizing ptime. *ArXiv preprint*, abs/1209.3422:12, 2015.

[ADLR94] Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. Paths in the lambda-calculus. In *LICS*, pages 426–436. IEEE Computer Society, 1994.

[AS14] Clément Aubert and Thomas Seiller. Characterizing co-NL by a group action. *Mathematical Structures in Computer Science (FirstView)*, pages 1–33, 12 2014.

[AS15] Clément Aubert and Thomas Seiller. Logarithmic space and permutations. *Information and Computation*, 2015. To appear.

[Aub13] Clément Aubert. *Linear Logic and Sub-polynomial Classes of Complexity*. PhD thesis, Université Paris 13–Sorbonne Paris Cité, 2013.

[Aub15] Clément Aubert. An in-between "implicit" and "explicit" complexity: Automata. *ArXiv preprint*, abs/1502.00145, 2015.

[Bag14] Marc Bagnol. *On the Resolution Semiring*. PhD thesis, Aix-Marseille Université – Institut de Mathématiques de Marseille, 2014.

[BC92] Stephen J. Bellantoni and Stephen Arthur Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[BM10] Patrick Baillot and Damiano Mazza. Linear logic by levels and bounded time complexity. *Theoretical Computer Science*, 411(2):470–503, 2010.

[BMM11] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776–2796, 2011.

[BP01] Patrick Baillot and Marco Pedicini. Elementary complexity and geometry of interaction. *Fundamenta Informaticae*, 45(1–2):1–31, 2001.

[Coo71] Stephen Arthur Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1):4–18, 1971.

[Dan90] Vincent Danos. *La Logique Linéaire appliquée à l'étude de divers processus de normalisation (principalement du λ-calcul)*. PhD thesis, Université Paris VII, 1990.

[DEGV01] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[DJ03] Vincent Danos and Jean-Baptiste Joinet. Linear logic & elementary time. *Information and Computation*, 183(1):123–137, 2003.

[DKM84] Cynthia Dwork, Paris C. Kanellakis, and John C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[DLH10] Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. *Logical Methods in Computer Science*, 6(4), 2010.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[Gir89a] Jean-Yves Girard. Geometry of interaction 1: Interpretation of system F. *Studies in Logic and the Foundations of Mathematics*, 127:221–260, 1989.

[Gir89b] Jean-Yves Girard. Towards a geometry of interaction. In John W. Gray and Andre Scedrov, editors, *Proceedings of the AMS Conference on Categories, Logic and Computer Science*, volume 92 of *Categories in Computer Science and Logic*, pages 69–108. American Mathematical Society, 1989.

[Gir95a] Jean-Yves Girard. Geometry of interaction III: accommodating the additives. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Note Series, pages 329–389. Cambridge University Press, June 1995.

[Gir95b] Jean-Yves Girard. Light linear logic. In Daniel Leivant, editor, *LCC*, volume 960 of *Lecture Notes in Computer Science*, pages 145–176. Springer, 1995.

[Gir11] Jean-Yves Girard. Geometry of interaction V: logic in the hyperfinite factor. *Theoretical Computer Science*, 412(20):1860–1883, April 2011.

[Gir12] Jean-Yves Girard. Normativity in logic. In Peter Dybjer, Sten Lindström, Erik Palmgren, and Göran Sundholm, editors, *Epistemology versus Ontology*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 243–263. Springer, 2012.

[Gir13] Jean-Yves Girard. Three lightings of logic. In Simona Ronchi Della Rocca, editor, *CSL*, volume 23 of *Leibniz International Proceedings in Informatics*, pages 11–23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.

[Har72] Juris Hartmanis. On non-determinancy in simple computing devices. *Acta Informatica*, 1(4):336–344, 1972.

[Her30] Jacques Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Faculté des Sciences de Paris, 1930.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[Lau01] Olivier Laurent. A token machine for full geometry of interaction (extended abstract). In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 283–297. Springer, May 2001.

[Lei93] Daniel Leivant. Stratified functional programs and computational complexity. In Mary S. Van Deusen and Bernard Lang, editors, *POPL*, pages 325–333. ACM Press, 1993.

[LMT00] Klaus-Jörn Lange, Pierre McKenzie, and Alain Tapp. Reversible space equals deterministic space. *Journal of Computer and System Sciences*, 60(2):354–367, 2000.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Sch07] Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *LICS*, pages 411–420. IEEE Computer Society, 2007.

[Sei12a] Thomas Seiller. Interaction graphs: Additives. *ArXiv preprint*, abs/1205.6557, 2012.

[Sei12b] Thomas Seiller. Interaction graphs: Multiplicatives. *Annals of Pure and Applied Logic*, 163:1808–1837, December 2012.

[Sei12c] Thomas Seiller. *Logique dans le facteur hyperfini : géometrie de l'interaction et complexité*. PhD thesis, Université de la Méditerranée, 2012.

[Sei14] Thomas Seiller. A correspondence between maximal abelian sub-algebras and linear logic fragments. *ArXiv preprint*, abs/1408.2125, 2014.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.