An Algebraic Approach to Vectorial Programs

Charles Paperman
Univ. Lille, CNRS, INRIA, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Sylvain Salvati
Univ. Lille, CNRS, INRIA, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Claire Soyez-Martin
Univ. Lille, CNRS, INRIA, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

Vectorial programming, the combination of SIMD instructions with usual processor instructions, is known to speed-up many standard algorithms. Simple regular languages have benefited from this technology. This paper is a first step towards pushing these benefits further. We take advantage of the inner algebraic structure of regular languages and produce high level representations of efficient *vectorial programs* that recognize certain classes of regular languages.

As a technical ingredient, we establish equivalences between classes of *vectorial circuits* and logical formalisms, namely unary temporal logic and first order logic. The main result is the construction of compilation procedures that turns *syntactic semigroups* into vectorial circuits. The circuits we obtain are *small* in that they improve known upper-bounds on representations of automata within the logical formalisms. The gain is mostly due to a careful sharing of sub-formulas based on algebraic tools.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases Automata theory, Semigroups, Vectorisation

Digital Object Identifier 10.4230/LIPIcs.STACS.2023.51

Related Version *Full Version*: `https://hal.archives-ouvertes.fr/hal-03831752v2`

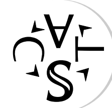
Acknowledgements We would like to thank the anonymous referees for helping to improve the paper a lot, Howard Straubing for proof-reading the main algebraic proofs of this paper, Michaël Hauspie for his advice, his support and his knowledge about SIMD and compilation and Corentin Barloy to have helped proof-read the paper.

1 Introduction

Finite state machines abstract the simplest class of programs. They are used everywhere: basic string manipulation functions of the `C` standard library like `memchr`, `strlen` or `strstr` are based on simple finite state automata, but also text-processing related tasks; checking the validity of encodings; text-mining; etc. As finite state machines are pervasive, implementing them efficiently is key in many softwares. The string related functions of the `C` standard library we mentioned earlier have greatly benefited from SIMD instructions built into modern CPUs. These functions can now process several characters per CPU cycle.

Single Instruction, Multiple Data (SIMD) executes an operation on several data in parallel, offering a form of low-level parallelism akin to Lamport's [13]. A function like `memchr` searches the first occurrence of a character in a string. SIMD instructions can check whether this character appears among several consecutive characters of the string in one go: each individual character is compared in parallel with the others. Other *vectorized algorithms* (see [11] for examples) benefit from these instructions. In the context of text-processing, impressive handcrafted SIMD based implementations have been proposed for string pattern matching [12], classical regular expression matching [33], Json parsing [14], checking correctness UTF-8 encoding [10], or DNA alignement in bioinformatics [6].

© Charles Paperman, Sylvain Salvati, and Claire Soyez-Martin;
licensed under Creative Commons License CC-BY 4.0



40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023).
Editors: Petra Berenbrink, Patricia Bouyer, Anuj Dawar, and Mamadou Moustapha Kanté; Article No. 51; pp. 51:1–51:23

Though many efforts have been put into compilers to solve the problem of *auto-vectorization* [18, 32, 19, 20, 8], these optimization methods rarely succeed in accelerating text algorithms with SIMD instructions. Finding auto-vectorization methods that deal with text algorithms would have a high impact. The challenge would be to produce, among others, the clever handcrafted code in the `C` standard library from the description of its underlying regular expression. Text processing, however, requires some form of sequentiality simply because, in many cases, information needs to be passed sideways. In this paper we consider two of them: prefix-or and addition.

The unreasonable power of binary addition. For sideways information passing, addition is very interesting: carry propagation can be used to compute long distance relations words. Moreover, *Big Int* instructions of modern processors compute it efficiently over large vectors.

Several papers already explored the use of addition in relation to regular languages: Myers [17] uses it to solve approximate string matching; Bergeron and Hamel [2, 1] show that counter-free automata can also benefit from addition; Serre [26] then characterizes counter-free languages in terms of addition; on the practical side, Cameron et al. [4] use explicitly addition when compiling parts of regular expressions of the form B^* when B is a set of letters.

This work starts with the definition of a notion of *vectorial circuit* that abstracts away from the details of CPU operations. Circuits make clear which operations are independent from one another and are objects of choice when it comes to introduce parallelism in some computation. We are confident that vectorial circuits can be compiled to obtain efficient programs that use SIMD instructions. As a first step in that direction, the main focus of the paper is then to construct small vectorial circuits from various presentations of counter-free regular languages.

The results of [2, 1] heavily rely on Krohn-Rhodes' Theorem. In this paper, the construction goes through Past-LTL logic and uses the equivalence between the Yesterday-Since operation and binary addition. The relation between these two operations is actually formalized in coq [21] (for technical reasons, in this formalization, we consider Forward-LTL and Next-Until instead of Past-LTL and Yesterday-Since). A consequence is that we can obtain concise vectorial circuits from Past-LTL formulae in the sense that they have the same size as the initial formula.

We also consider how to obtain concise vectorial circuits directly from automata. With Serre's result, it is theoretically possible to produce vectorial circuits for counter-free automata. However, only a double exponential upper bound in the size of the automaton is known on the size of Past-LTL formulas. This would make circuits far too large in practice. This upperbound comes from a construction of Wilke [35, Corollary 1] that builds formulas by induction over the structure of the *syntactic monoids* of the input automata. On other classes of automata, such as $FO_2[<]$ [5, 31], known transformations of automata into formulas are indirect and not constructive. They are of no use to actually compile automata into formulas or circuits.

Main contributions. We study a notion of *vectorial circuits* as an abstraction of SIMD programs. Vectorial circuits describe the shape of actual circuits for arbitrary size of inputs. In particular they are a uniform fragment of AC^0 (constant depth polysize boolean circuits). As for circuits, their expressivity depends on the set of authorized gates. Our main goal is to produce small vectorial circuits for particular classes of regular languages: the class of counter-free or $FO[<]$-languages and the class $\mathbf{DA}$ or $FO_2[<]$-languages. We measure the

size of output circuits with respect to the size of the syntactic monoids. Were we to consider automata as inputs that we would need to exponentiate our bounds (e.g. the languages of the form $A^k a A^*$ have syntactic monoids of size 2^k with minimal automata of size $O(k)$).

Concerning counter-free languages, we revisit Serre's result. We propose an algorithm that produces vectorial circuits (using bitwise boolean operations and addition) that are polynomial in the size of the input aperiodic monoid. Serre's result ensures here that the class of vectorial programs expressed with addition and bitwise boolean operations are equivalent to counter-free languages.

For the class **DA**, we replace addition with prefix-or to obtain a class of circuits that captures exactly that class. When transforming syntactic monoids of **DA** into vectorial circuits, the use of prefix-or makes circuits larger than they would be with addition. The transformation that we propose gives vectorial circuits which have exponential size in the $\mathcal{J}$-depth (see 3.1 for the definition of $\mathcal{J}$-depth) of the syntactic monoid.

We begin by presenting vectorial circuits in Section 2. We also show the links between vectorial circuits and fragments of logic. In Section 3, we introduce general strategies of evaluation of words. Then, in Section 4, we construct small programs that compute our strategies of evaluation. Sketches of all the main proofs can be found in the appendices. For the complete version, see [22].

2 Compiling regular languages into vectorial circuits

2.1 Algebraic preliminaries

We write $[n]$ for the set $\{0, \ldots, n-1\}$. Given a set E, we denote by $|E|$ its cardinality. Given some finite set Σ, the *alphabet*, words on Σ are finite sequences of elements of Σ. We write $|x|$ for the length of the word x. We denote by Σ^* the set of words on Σ.

Semigroups. A *semigroup* is a pair consisting of a set S and an associative binary operation $\cdot_S$ on S, called the inner operation of S. We usually write that the set S is a semigroup. A *monoid* is a triple $(M, \cdot_M, 1)$, where $(M, \cdot_M)$ is a semigroup and $1 \in M$ is an identity (or a neutral element) of M. We usually write that the set M is a monoid. We only work with *finite* semigroups and monoids. We thus designate *finite semigroups* (resp. finite monoids) when we mention semigroups (resp. monoids). Given a semigroup S, any element e of S satisfying $e \cdot_S e = e$ is called an *idempotent*. In a finite semigroup S, any element s of S admits an *idempotent power*, which is an element s^n (where $n > 0$ is an integer) that is idempotent, where s^n denotes the iterated product of s by itself n times. We use the usual notation s^ω to denote the idempotent power of s (ω is the minimum integer such that, for any element s, s^ω is the idempotent power of s). Given a semigroup S, we define $S^1 = S \cup \{1\}$ as the monoid formed by the semigroup to which an identity is added if necessary. For any subsets X and Y of S, we denote by $X \cdot_S Y$ the set $\{x \cdot_S y \mid x \in X, y \in Y\}$. Similarly, for any $x \in S$ and $Y \subseteq S$, we write $x \cdot_S Y$ and $Y \cdot_S x$ respectively for $\{x\} \cdot_S Y$ and $Y \cdot_S \{x\}$. Given a finite set Σ, we call Σ^+ the *free semigroup over* Σ with the concatenation as the associative binary operation. This is the only infinite semigroup that we consider. Given a semigroup S, we will denote by S^+ the free semigroup with the underlying set of S as alphabet.

Canonical morphism. We denote concatenation implicitly: given two words u, v, their concatenation is written uv. For instance, taking two elements x, y of S, xy denotes a word of S^+ of length 2. This notation **must not** be confused with $x \cdot_S y$ that denotes the element of S obtained by multiplying x and y with the inner operation of S. We **never use** concatenation

to mark the product within S. However, we relate words of the free semigroup S^+ to their value in S by means of *the canonical morphism*: $\pi_S \colon S^+ \to S$. It is the unique associative morphism verifying both the following properties: for every $x \in S$, $\pi_S(x) = x$ and, for every u, $v \in S^+$, $\pi_S(uv) = \pi_S(u) \cdot_S \pi_S(v)$.

Languages and semigroups. A link can be established between logics and semigroups by taking the *syntactic semigroup* of a language. This semigroup is defined as follows: given a language L on an alphabet Σ, the *syntactic congruence* of L in Σ^* is the relation $\sim_L$ defined on Σ^* such that, for any words $u, v \in \Sigma^*$, $u \sim_L v$ if and only if, for all words $x, y \in \Sigma^*$, $xuy \in L \Leftrightarrow xvy \in L$. The syntactic semigroup of L is the quotient $\Sigma^+/\sim_L$, and the syntactic monoid of L is the quotient $\Sigma^*/\sim_L$.

The link between logic and semigroups has already been well studied and gave birth to very nice algebraic characterizations of some well-known classes of languages. For more information on this topic, see the survey [30] along with the books [28] and [23]. Notably, the class of starfree languages is equivalent to the variety of aperiodic semigroups, the semigroups satisfying an equation of the form $\pi_S(x^{\omega+1}) = \pi_S(x^\omega)$, for any $x \in S$. We can also mention the class $\mathrm{FO}_2[<]$, which is equivalent to the variety **DA** of semigroups, the semigroups satisfying an equation of the form $\pi_S((xy)^\omega x(xy)^\omega) = \pi_S((xy)^\omega)$, for any $x, y \in S$. For an explanation of the name **DA**, see Section 3.1. We refer to [29] for a complete exposition of this class and its relationship to various logics.

2.2 Vectorial circuits

We call the words on the alphabet $\{0, 1\}$ *vectors*. For a vector $\mathbf{x}$, we may use the term *dimension* to refer to its length $|\mathbf{x}|$. We refer to vectors of dimension n as n-vectors. We let $\mathbf{1}_n$ and $\mathbf{0}_n$ respectively denote the sequence of n 1's and the sequence of n 0's. When n is irrelevant or obvious for the context, we may write $\mathbf{1}$ and $\mathbf{0}$. Two vectors $\mathbf{x} = x_0 \dots x_{n-1}$ and $\mathbf{y} = y_0 \dots y_{n-1}$, of dimension $n \in \mathbb{N}$, are said to be *disjoint* if, for any index $i \leq n-1$, $x_i \wedge y_i = 0$.

Vectorial circuits and their semantics. *Vectorial circuits* are labeled directed acyclic graphs. The nodes that have no input edge are called *input nodes*. The nodes with incoming edges are called *gates* and are labeled with *commutative operations*. The in-degree of a gate should be equal to the arity of its labeling operation while the out-degree can be arbitrary. We usually write input nodes and terms in bold-face fonts: $\mathbf{v}$, $\mathbf{v_1}, \dots$ *Output nodes* are distinguished nodes of the circuit. Generally they include the nodes that have no output edge. The *size of a circuit* is the number of its nodes.

Vectorial circuits can be seen as circuit templates that, for each n, instantiate a *concrete circuit* working on vectors of dimension n. Once the dimension is fixed to n, associating n-vectors to the input nodes and flowing the values through the gates (where the right function operating on n-vectors is used) yields output values in the output nodes. Take a circuit C with input nodes $\mathbf{i_1}, \dots, \mathbf{i_p}$ and output nodes $\mathbf{o_1}, \dots, \mathbf{o_r}$, given p n-vectors $\mathbf{x_1}$, $\dots$, $\mathbf{x_p}$, we write $C(\mathbf{x_1}, \dots, \mathbf{x_p})$ for the tuple of n-vectors $\mathbf{y_1}$, $\dots$, $\mathbf{y_r}$ that are respectively yielded in the output nodes $\mathbf{o_1}$, $\dots$, $\mathbf{o_r}$ when evaluating the C with the vector $\mathbf{x_1}$, $\dots$, $\mathbf{x_p}$ respectively associated to the input nodes $\mathbf{i_1}$, $\dots$, $\mathbf{i_p}$.

Operations for labeling gates. We use bitwise Boolean operations: the unary negation $\neg$ and the binary operations $\wedge$ and $\vee$, respectively the bitwise conjunction and disjunction.

Given a function $f : \{0, 1\}^+ \to \{0, 1\}$, we define the unary operation pref-f (resp. suf-f): given a n-vector $\mathbf{x} = b_0 \dots b_{n-1}$, with $b_0, \dots, b_{n-1}$ in $\{0, 1\}$, pref-$f(\mathbf{x})$ (resp. suf-$f(\mathbf{x})$) is the n-vector $\mathbf{z} = c_0 \dots c_{n-1}$ where for each $i \in [n]$, $c_i = f(b_0 \cdots b_i)$ (resp. $c_i = f(b_i \cdots b_{n-1})$). In this paper, we use the unary operations pref-$\vee$, suf-$\vee$, pref-$\wedge$ and suf-$\wedge$.

Binary vectors $\mathbf{x}$ of dimension n naturally represent numbers in $[2^n]$. We write $nb(\mathbf{x})$ for the number represented by $\mathbf{x}$ with the convention that its least significant bit is the left-most one. For a natural number k, we write $bin_n(k)$ to denote the vector of dimension n that represents k modulo 2^n.

The unary operations LSB (Least Significant Bit) and MSB (Most Significant Bit) replace by 0 respectively the left-most 1 and right-most 1 of their argument vector. For these two operations, when the argument vector is $\mathbf{0}_n$, the resulting vector is also $\mathbf{0}_n$. The binary operation $+$ is defined by the family $(\text{plus}_n)_{n\in\mathbb{N}}$ so that for any two vectors of dimension n, denoted by $\mathbf{x}$ and $\mathbf{y}$, $\text{plus}_n(\mathbf{x}, \mathbf{y}) = bin_n(nb(\mathbf{x}) + nb(\mathbf{y}))$.

We study two families of vectorial circuits:

- Sweeping-*vectorial circuits*, circuits built only with the operations, $\wedge$, $\vee$, $\neg$, pref-$\vee$, pref-$\wedge$, suf-$\vee$, suf-$\wedge$, LSB, MSB,
- and ADD-*vectorial circuits*, circuits built only with the operations of Sweeping-vectorial circuits and $+$.

Term notation for vectorial circuits. Trees are a particular kind of directed acyclic graphs. Circuits that are tree shaped are those where nodes have at most one out-going edge and exactly one output node. These particular circuits can be advantageously denoted by terms built with operations (respecting their arity) and input nodes. The term $(\mathbf{v_1} \wedge \neg \mathbf{v_2}) \vee (\neg \mathbf{v_3} \wedge \mathbf{v_4})$ represents the left-most circuit of Figure 1. Allowing input nodes to have several occurrences in terms gives access to some limited kind of sharing. This is exemplified with the central and right-most circuits of Figure 1. So as to fully capture such sharing capabilities with the term notation, we use equations: a term t that is to be shared is associated to a node $\mathbf{v}$ with the equation $\mathbf{v} = t$ and, when $\mathbf{v}$ is used in another term, this refers to the *shared* circuit t. For example, we write $\mathbf{v} = \text{pref-}\vee(\text{suf-}\vee(\mathbf{v_1}))$, $(\neg \mathbf{v} \wedge \mathbf{v_2}) \vee (\mathbf{v} \wedge \mathbf{v_3})$ to denote the third circuit in Figure 1.

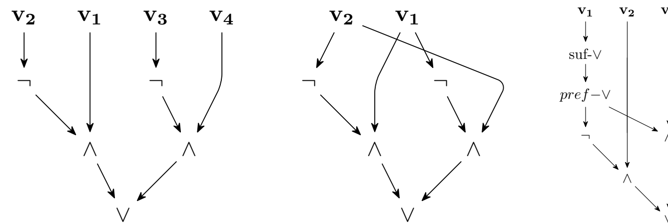


Figure 1 Graph representation of terms: $(\mathbf{v_1} \wedge \neg \mathbf{v_2}) \vee (\neg \mathbf{v_3} \wedge \mathbf{v_4})$; $(\mathbf{v_1} \wedge \neg \mathbf{v_2}) \vee (\neg \mathbf{v_1} \wedge \mathbf{v_2})$; and $\mathbf{v} = \text{pref-}\vee(\text{suf-}\vee(\mathbf{v_1}))$, $(\neg \mathbf{v} \wedge \mathbf{v_2}) \vee (\mathbf{v} \wedge \mathbf{v_3})$.

Terms also offer a convenient way for reusing certain circuits in several places: it suffices to change the input nodes at their leaves. We adopt a notation that denotes *parametrized circuits*: $\text{c}(\mathbf{v_1}, \dots, \mathbf{v_n}) := t$ where t is a term built with the nodes $\mathbf{v_1}, \dots, \mathbf{v_n}$. For circuits $t_1, \dots, t_n$, we write $\text{c}(t_1, \dots, t_n)$ the circuit described by the term obtained by replacing $\mathbf{v_1}, \dots, \mathbf{v_n}$ respectively with $t_1, \dots, t_n$ in t. For example, we define the bitwise exclusive-or as $\mathbf{v_1} \oplus \mathbf{v_2} := (\mathbf{v_1} \wedge \neg \mathbf{v_2}) \vee (\neg \mathbf{v_1} \wedge \mathbf{v_2})$. We can compose parametrized circuits to define others et construct complex circuits.

Parameterized circuits. We can compose parametrized circuits to define others:

- $\mathrm{NotZero}(\mathbf{v}) := \text{pref-}\vee(\text{suf-}\vee(\mathbf{v}))$
- $\mathrm{IsZero}(\mathbf{v}) := \neg\mathrm{NotZero}(\mathbf{v})$
- $\mathrm{IfThenElse}(\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}) := \mathbf{v} = \mathrm{NotZero}(\mathbf{v_1}), (\mathbf{v} \wedge \mathbf{v_2}) \vee (\neg\mathbf{v} \wedge \mathbf{v_3})$
- $\mathrm{Thr2}(\mathbf{v}) := \mathrm{NotZero}(\mathrm{LSB}(\mathbf{v}))$

The parametrized circuits $\mathrm{NotZero}(\mathbf{v})$, $\mathrm{IsZero}(\mathbf{v})$, $\mathrm{Thr2}(\mathbf{v})$ perform tests on their input: on an n-vector, they return $\mathbf{1}_n$ when the test succeeds and $\mathbf{0}_n$ when it fails. $\mathrm{NotZero}(\mathbf{v})$ tests whether its input vector contains an occurrence of 1, $\mathrm{IsZero}(\mathbf{v})$ tests whether its input vector is $\mathbf{0}$ and $\mathrm{Thr2}(\mathbf{v})$ tests whether its input vector contains at least two occurrences of 1. The parametrized circuit $\mathrm{IfThenElse}(\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3})$, tests whether $\mathbf{v_1}$ is $\mathbf{0}$ and in this case outputs $\mathbf{v_2}$. Otherwise, it outputs $\mathbf{v_3}$. With parameterized circuits, we can construct complex circuits. An example is the circuit $\mathrm{IfThenElse}(\mathbf{v_1} \oplus \mathbf{v_2}, \mathbf{v_3}, \mathbf{v_2} \oplus \mathbf{v_3})$ which is depicted in Figure 2.

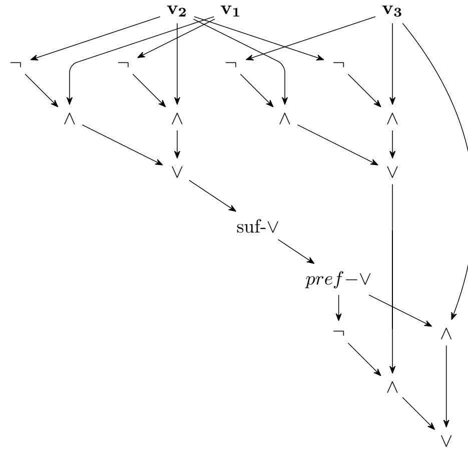


Figure 2 Circuit represented by $\mathrm{IfThenElse}(\mathbf{v_1} \oplus \mathbf{v_2}, \mathbf{v_3}, \mathbf{v_2} \oplus \mathbf{v_3})$.

The addition Lemma. We say that a vector $\mathbf{x}$ is contained in a vector $\mathbf{y}$ if for any position i, $\mathbf{x}_i = 1$ implies $\mathbf{y}_i = 1$. Let $\mathbf{x}, \mathbf{y}$ be two disjoint vectors of dimension n and $\mathbf{z}$ a vector of dimension n that contains both $\mathbf{x}$ and $\mathbf{y}$. We denote by $\mathbf{v} = \mathrm{Successor}(\mathbf{x}, \mathbf{y}, \mathbf{z})$ the vector such that for all $i < n$, $\mathbf{v}_i = 1$ if and only if $\mathbf{x}_i = 1$, there exists $j < i$ such that $\mathbf{y}_j = 1$ and, for all $k \in \mathbb{N}$ such that $j < k < i$, $\mathbf{z}_k = 0$. In other words, $\mathrm{Successor}(\mathbf{x}, \mathbf{y}, \mathbf{z})$ indicates the positions marked in $\mathbf{x}$ that follow a marked position of $\mathbf{y}$, with no other position marked by $\mathbf{z}$ in-between.

▶ **Lemma 1** (Addition lemma). *Let* $\mathbf{x}, \mathbf{y}$ *be two disjoint vectors of dimension* n *and* $\mathbf{z}$ *a vector of dimension* n *that contains both* $\mathbf{x}$ *and* $\mathbf{y}$. *Then, we have* $\mathrm{Successor}(\mathbf{x}, \mathbf{y}, \mathbf{z}) := (\mathbf{y} + (\mathbf{y} \vee \neg\, \mathbf{z})) \wedge \mathbf{x}$

This lemma has a rather tedious proof. So as to relieve the reader from checking its details, we provide a formalization and a proof this Addition Lemma in Coq [21]. The proof consists of two steps. First, we show that vectorial circuits built with the Next-Until modality of LTL, aka XU, and logical gates, are equivalent to circuits built with addition and logical gates. This equivalence is proved by constructing circuits based on XU and logical gates to encode addition and circuits based on addition and logical gates to encode XU. The intuition behind this equivalence is that carry propagation and XU both propagate information sideways. Second, encoding XU in first order-logic and using the previous equivalence allows us to relate the computation of the circuit of Lemma 1 to the specification of $\mathrm{Successor}(\mathbf{x}, \mathbf{y}, \mathbf{z})$ and prove the relation correct.

Vectorial circuits as language recognizers and functions from words to finite sets. Given a fixed alphabet Σ, we say that a vectorial circuit C *recognizes* a set of words in Σ^+ when it has a unique output node and there is a bijection between the letters $a_1, \ldots, a_p$ of Σ and its input nodes $\mathbf{a_1}, \ldots, \mathbf{a_p}$. We consider a particular bijection: given a word u of length n, we write $\mathbb{1}_a(u)$ for the n-vector $\mathbf{x}$ so that $\mathbf{x}_i = 1$ if and only if $u_i = a$ for every i in $[n]$. We say that u is *accepted* or *recognized* by the circuit when $C(\mathbb{1}_{a_1}(u), \ldots, \mathbb{1}_{a_p}(u)) \neq \mathbf{0}$. As a shorthand, we write $\text{enc}(u)$ for the tuple $(\mathbb{1}_{a_1}(u), \ldots, \mathbb{1}_{a_p}(u))$ and thus $C(\text{enc}(u))$ for $C(\mathbb{1}_{a_1}(u), \ldots, \mathbb{1}_{a_p}(u))$.

Vectorial circuits can also represent functions f from Σ^+ to a finite domain E. It suffices to consider circuits C which have a bijection between their input nodes and the letters of Σ, but also a bijection between the elements $e_1, \ldots, e_r$ of E and their output nodes $\mathbf{e_1}, \ldots, \mathbf{e_r}$. We say that C *represents* f when for every u, the output $(\mathbf{z_1}, \ldots, \mathbf{z_r}) = C(\text{enc}(u))$ is such that, for every i in $\{1, \ldots, r\}$, $\mathbf{z_i} \neq \mathbf{0}$ if and only if $f(u) = e_i$.

Example. Consider the alphabet $\Sigma = \{a, b, c\}$. The language $c^*a\Sigma^*$ is recognized by the Sweeping-vectorial circuit $C = \neg(\text{suf-}\vee(\text{pref-}\vee(\text{suf-}\vee(F_a) \wedge \mathbf{b})))$, where the circuit $F_a = \text{LSB}(\mathbf{a}) \oplus \mathbf{a}$ gives the position of the first occurrence of a. The language $\Sigma^*ac^*a\Sigma^*$ is recognized by the ADD-vectorial circuit $C = (\mathbf{a} + \neg\mathbf{b}) \wedge \mathbf{a}$, which is equal to $(\mathbf{a} + (\mathbf{a} \vee \neg(\mathbf{a} \vee \mathbf{b}))) \wedge \mathbf{a} = \text{Successor}\,(\mathbf{a}, \mathbf{a}, \mathbf{a} \vee \mathbf{b})$. An example is showed in Table 1; note that every 1 in the last vector indicates, as we want, each letter a such that there is no letter b between this position and the previous letter a, and no other letter is indicated.

Table 1 Example of application of the operation $(\mathbb{1}_a + \neg\mathbb{1}_b) \wedge \mathbb{1}_a$.

$\mathbb{1}_b$	0	0	1	0	0	0	0
$\mathbb{1}_a$	0	1	0	1	0	1	1
$\mathbb{1}_a + \neg\mathbb{1}_b$	1	0	1	0	0	1	1
$(\mathbb{1}_a + \neg\mathbb{1}_b) \wedge \mathbb{1}_a$	0	0	0	0	0	1	1

2.3 Going through first-order logic

The computational model we propose is actually very close to some classical fragments of logic over words. For a full exposition of this subject, we refer to [5]. Informally, we can describe some regular languages with the help of first-order formulas. In those formulas, quantifications range over positions of the word and atomic predicates can check either numeric constraints between positions (typically their order) or the label of the position. For instance, the formula $\exists x, y,\ a(x) \wedge a(y) \wedge \forall z,\ ((x < z \wedge z < y) \Rightarrow c(z))$ describes the language over the alphabet $\Sigma = \{a, b, c\}$ which words belong to $D = \Sigma^*ac^*a\Sigma^*$.

In this context, linear temporal logic (LTL for short) has an interesting role that we will rely on. LTL, even restricted to future or past operators, allows to define the same languages as full first-order logic [9]. For instance, the language D can be described with the following LTL formula: $\text{F}(a \wedge \text{X}(c\text{U}a))$.

This class of languages has many equivalent characterizations. In [25], Schützenberger proved for regular languages the equivalence between aperiodic languages and *star-free* languages. Later, McNaughton and Papert proved that star-free languages are equivalent with LTL and first-order logic [15, 5]. In a similar fashion to McNaughton and Papert, we can prove that languages computed by ADD-vectorial circuits are exactly the starfree languages. Both directions of the equivalence have been proved by Serre [26, Theorem 1.] by relying on a syntactic logical rewriting of LTL formulas.

▶ **Proposition 2.** *Let L be a regular language. The following propositions are equivalent.*
1. *L is computed by an* ADD*-vectorial circuit.*
2. *L is starfree.*
3. *L is definable in Forward-LTL.*

Proof. The equivalence between 1 and 2 is from [26], Corollary 1. In this paper, the vectorial computational model consists of straight-line expressions over a basis which includes addition, Boolean operations and right shift. This basis is equivalent to ADD-vectorial circuits. The equivalence between 2 and 3 is standard. It can be proved by combining theorems proved by Kamp [9] and McNaughton and Papert [15]. The former proves the equivalence between being definable in Forward-LTL and First-Order logic for regular languages and the later the equivalence between First-Order logic and star-free languages. ◀

Another fragment of interest is the class of languages definable by two-variable first-order logic, or equivalently by LTL without the Until operation but with neXt and Yesterday operations. This fragment has been studied a lot and is well understood [9, 5, 34, 36].

In the proof of the next result, we rely on the known equivalence between $\mathrm{FO}_2[<]$ and a logic called $\mathrm{TL}[X_a, Y_a]$. More formally, the languages describable in $\mathrm{FO}_2[<]$ are exactly the languages describable by the logic $\mathrm{TL}[X_a, Y_a]$ presented in [5]. This logic uses only two kinds of operators: X_a, for a any letter, verifies that there exists a position greater than the current one which holds an a, and if so, it sets the new current position at the position of the closest a. If no occurrence of a is found, the word is not recognized. The operator Y_a, for any letter a, is the symmetrical operator: it operates the same way except for the fact that it searches for the closest occurrence of a before the current position.

▶ **Proposition 3.** *A language is recognized by a* Sweeping*-vectorial circuit if and only if it is definable in $TL[X_a, Y_a]$.*

Proof. We rely on the equivalence between $\mathrm{FO}_2[<]$ and $\mathrm{TL}[X_a, Y_a]$, to prove the result. First, we show that $\mathrm{TL}[X_a, Y_a]$ is captured by Sweeping-vectorial circuits, i.e. that any formula of $\mathrm{TL}[X_a, Y_a]$ defines a function from words to Boolean vectors that can be modeled by these circuits; here we interpret a formula of $\mathrm{TL}[X_a, Y_a]$ as a function from words to Boolean vectors, by considering the truth vector of a formula on every position of the input word. Clearly, Boolean operations are equivalent in both models. Now, suppose that we have a Sweeping-vectorial circuit C_α that is equivalent to a $\mathrm{TL}[X_a, Y_a]$ formula α, i.e. given a word $u \in \Sigma^+$ and a vector $\mathbf{pos}$ (of the same dimension as u) that has a unique 1 in some position x, we have $C_\alpha(\mathbf{pos}, \mathrm{enc}(u)) \neq \mathbf{0}$ if and only if $u, x \models \alpha$. Then, for any letter $a \in \Sigma$, the formula $\beta = X_a\alpha$ can be modeled by the circuit C_β defined as follows: for any word $u \in \Sigma^*$ and any vector $\mathbf{pos}$ having a unique 1, we begin by finding the first letter a strictly after the position marked by $\mathbf{pos}$, by sequentially computing the following vectors: $\mathbf{authorizedPos} = \mathrm{LSB}(\text{pref-}\vee(\mathbf{pos}))$, $\mathbf{next} = \mathbf{authorizedPos} \wedge \mathbb{1}_a(u)$ and $\mathbf{nextPos} = \mathbf{next} \oplus \mathrm{LSB}(\mathbf{next})$. This gives us the new position vector to give as parameter to the new circuit. Then we can define $C_\beta(\mathbf{pos}, \mathrm{enc}(u)) := C_\alpha(\mathbf{nextPos}, \mathrm{enc}(u))$, which is equivalent to β. Finally, we can model the formula $\gamma = Y_a\alpha$ using the same formulas, except for the vector $\mathbf{authorizedPos}$, which is computed as $\mathbf{authorizedPos} = \mathrm{MSB}(\text{suf-}\vee(\mathbf{pos}))$, in order to get the positions situated strictly before the position x marqued by $\mathbf{pos}$.

Now, we prove that Sweeping-vectorial circuits are captured by formulas in $\mathrm{FO}_2[<]$. Since Sweeping-vectorial circuits output a truth vector indicating if some property is true for each position in the input word, we must consider $\mathrm{FO}_2[<]$ formulas which can take into account a starting position. Thus, we consider only $\mathrm{FO}_2[<]$ formulas with one free variable, which represents the position in which we begin to evaluate the formula in the input word.

Clearly, Boolean operations are equivalent in both models. We can also interpret pref-$\vee$ as an $\mathrm{FO}_2[<]$ formula: consider an $\mathrm{FO}_2[<]$ formula φ with one free variable and suppose that it is computed by a circuit C_φ. Then, the formula $\varphi_{\text{pref-}\vee}$ with one free variable defined by $\varphi_{\text{pref-}\vee}(y) := \exists x \leq y,\, \varphi(x)$ has its truth vector equal to the output of pref-$\vee(C_\varphi)$. The other prefix and suffix operations can be dealt with in a similar way. Finally, we can interpret LSB and MSB as $\mathrm{FO}_2[<]$ formulas. Indeed, consider an $\mathrm{FO}_2[<]$ formula φ with one free variable and suppose that it is computed by a circuit C_φ. Then, the formula φ_{LSB} with one free variable defined by $\varphi_{\mathrm{LSB}}(y) := \varphi(y) \wedge \exists x < y,\, \varphi(x)$ has its truth vector equal to the output of $\mathrm{LSB}(C_\varphi)$. MSB is equivalent to the same formula where $x < y$ is replaced by $x > y$. ◀

▶ Remark 4. We can always build a vectorial circuit of size linear in the size of the equivalent formula. Conversely, if we consider formulas with sharing of sub-formulas, we can also build an equivalent circuit of size linear in the size of the formula.

2.4 Direct compilation scheme

In the previous section, we have seen the relationship between vectorial circuit classes and LTL logic formulas. However, the proofs of those results do not give satifying algorithms for building a vectorial circuit recognizing a given formula. Indeed, the proofs going from the automata to the formulas are either indirect and not constructive, and they do not give any upperbound on the sizes of the formulas, or they are constructive but give formulas (and thus circuits) of too large sizes. Our goal is to reduce the size of the formulas we obtain through semigroups, so that we can obtain tractable algorithms for the languages that have a small syntactic monoid. Because of this point of view, all complexity measures of our circuits are provided in terms of the semigroup size.

▶ **Theorem 5.** *Let S be a semigroup in* **DA** *of $\mathcal{J}$-depth d (see Section 3.1 for the definition of $\mathcal{J}$-depth). We can construct a* Sweeping*-vectorial circuit of size $O(2^d|S|^2)$ that computes the operation π_S.*

Moving from Sweeping-vectorial circuits to Unary-LTL could cost an exponential blowup, which in total gives a doubly-exponential blowup for constructing a Unary-LTL formula from a semigroup in **DA**. Note that the classical constructions do not provide upperbounds on the minimum size of a Unary-LTL formula equivalent to a given semigroup in **DA**. However, these constructions already have considerable sizes of formulas.

We conjecture that no Sweeping-vectorial circuit of polynomial size exists and believe that it is an interesting open question to provide lowerbounds for those circuit models.

▶ **Theorem 6.** *Let S be an aperiodic semigroup of $\mathcal{J}$-depth d. We can construct an* ADD*-vectorial circuit of size $O(d|S|^3)$ that computes the operation π_S.*

Proofs' organisation. We chose to separate the proofs into two main parts. In the first one, we introduce the notion of evaluation program, which is a very generic tool allowing to replace a sub-word by a single letter equal to its product, and we define two generic evaluation strategies which are evaluation programs. These strategies were chosen according to the properties of the classes of semigroup that will be considered later but, supposing that we can provide an efficient encoding of those strategies, they are valid on any semigroup. Then, in the second part, we provide vectorial circuits encoding the two evaluation strategies on the classes of semigroups that we are interested in.

3 Semigroups evaluation strategies

3.1 Green's relations

We refer the reader to [24, 23] for a complete exposition of algebraic automata theory. We remind here some more basic notations and definitions about semigroups. Consider a function $F\colon S \to \mathcal{P}(S)$, where $\mathcal{P}(S)$ denotes the power set of S. We write $x \mathrel{\mathcal{F}} y$ when $F(x) = F(y)$; $x \leq_{\mathcal{F}} y$ when $F(x) \subseteq F(y)$; and $x <_{\mathcal{F}} y$ when $x \leq_{\mathcal{F}} y$ and $F(x) \neq F(y)$. The relation $\mathcal{F}$ is an equivalence relation and $\leq_{\mathcal{F}}$ is a partial pre-order. We also write $\mathcal{F}(x) = \{y \mid y \mathrel{\mathcal{F}} x\}$, the *$\mathcal{F}$-class of* x. We say that a semigroup is *$\mathcal{F}$-trivial* when $\mathcal{F}(x)$ is a singleton for any element $x \in S$. Green's relations are defined with the following functions: $R\colon x \mapsto x \cdot_S S^1$, $L\colon x \mapsto S^1 \cdot_S x$, $J\colon x \mapsto S^1 \cdot_S x \cdot_S S^1$, $H\colon x \mapsto R(x) \cap L(x)$. Note that the respective relations obtained from R, L, J and H are denoted by $\mathcal{R}$, $\leq_{\mathcal{R}}$, $<_{\mathcal{R}}$, $\mathcal{L}$, $\leq_{\mathcal{L}}$, $<_{\mathcal{L}}$, $\mathcal{J}$, $\leq_{\mathcal{J}}$, $<_{\mathcal{J}}$,$\mathcal{H}$, $\leq_{\mathcal{H}}$ and $<_{\mathcal{H}}$. In finite semigroups, the relation $\mathcal{J}$ is equal to the relation $\mathcal{D}$, which is the union of $\mathcal{L}$ and $\mathcal{R}$. From this relation $\mathcal{D}$ comes the name of the class $\mathbf{DA}$, which indicates the class of semigroups which regular $\mathcal{D}$-classes are aperiodic semigroups.

The $\mathcal{J}$-depth of a semigroup. Let S be a semigroup. The *$\mathcal{J}$-depth* of a $\mathcal{J}$-class is the length of a maximal strictly decreasing sequence of $\mathcal{J}$-classes to it. Formally, given a semigroup S and a $\mathcal{J}$-class J, we say that J is of $\mathcal{J}$-depth i if there exist i $\mathcal{J}$-classes $J_1 >_{\mathcal{J}} J_2 \ldots >_{\mathcal{J}} J_i$ such that $J_i = J$, but there exists no decreasing sequence $J'_1 >_{\mathcal{J}} J'_2 \ldots >_{\mathcal{J}} J'_{i+1}$ such that $J'_{i+1} = J$. The $\mathcal{J}$-depth of a semigroup is the maximum $\mathcal{J}$-depth of its $\mathcal{J}$-classes. For any semigroup, there exists a unique $\mathcal{J}$-class of maximum $\mathcal{J}$-depth. Given d the $\mathcal{J}$-depth of S, for each integer i such that $1 \leq i \leq d$, we denote by $D_i(S)$ the union of all the $\mathcal{J}$-classes of depth i and we denote by S_i the sub-semigroup composed exactly of all the elements of S of $\mathcal{J}$-depth at least i.

$\mathcal{J}$-constant words. Let S be a semigroup. A word $s_0 \cdots s_k$ in S^+ is *left $\mathcal{J}$-constant* if, for any index i such that $0 \leq i \leq k$, we have $\pi_S(s_0 \cdots s_i) \mathcal{J} s_0$. Symetrically, $s_0 \cdots s_k$ is *right $\mathcal{J}$-constant* if, for any index i such that $0 \leq i \leq k$, $\pi_S(s_i \cdots s_k) \mathcal{J} s_k$. Finally, a word in S^+ is $\mathcal{J}$-constant if it is both left and right $\mathcal{J}$-constant. The latter property is equivalent to having a left $\mathcal{J}$-constant word such that $s_0 \mathcal{J} s_k$.

3.2 Evaluation programs

In this paper, we consider words over some semigroup S. Our goal is to compute the product in S of the letters composing these words. For any word $u \in S^+$, this amounts to computing $\pi_S(u)$. This computation can be performed by vectorial circuits. Instead of directly building these circuits, we first pay attention to evaluation strategies that we call *evaluation programs*. These strategies form an overlay of abstraction over the intricacy of circuits. They are meant to modularize the construction of vectorial circuits.

Given a semigroup S, an evaluation program over S transforms words in S^+ by replacing some of the factors by their values (through the canonical morphism) in S. In this section, we consider a fixed semigroup S.

▶ **Definition 7** (Partial evaluation). *A* partial evaluation step *over S is a relation over S^+ denoted by $\to_S$ and defined as $uvw \to_S u\pi_S(v)w$ for any v in S^+ and $u, w \in S^*$. We denote by $\to_S^+$ the transitive closure of $\to_S$. We say that v* is a partial evaluation of *u when $u \to_S^+ v$.*

Note that if v is a partial evaluation of u over S, then $\pi_S(u) = \pi_S(v)$. Usually, the context makes it clear which semigroup is considered. Thus, we generally leave the semigroup implicit and only say that v is a partial evaluation of u. Note that each word u is a partial evaluation of itself.

▶ **Definition 8** (Partial evaluation programs). *A* partial evaluation program *over S is a partial function $P : S^+ \to S^+$ such that, for any word $u \in S^+$ that is in the domain of P, $P(u)$* is a partial evaluation of u*. If the domain of P is S^+, then we say that P is* total.

The function π_S is an example of a partial evaluation (which is, in this case, total). Another example is the function $\mathrm{LProd}_S\colon S^+ \to S^+$ that performs the product of the first two elements of the input, if there are at least two elements, and otherwise returns the input word. In symbols, it is defined by $\mathrm{LProd}_S(s_0 \cdots s_n) = \pi_S(s_0 s_1) s_2 \cdots s_n$ for any word $u = s_0 \cdots s_n$ of length at least two, and $\mathrm{LProd}_S(s_0) = s_0$ for any element $s_0 \in S$. Similarly, we define RProd_S as the partial evaluation program which performs the product of the two last elements, if there are at least two elements, and otherwise returns the input word. Evaluation programs are closed under composition.

3.3 Waterfall evaluation programs

In this section, we are designing a first set of specific evaluation programs. These programs work by evaluating the semigroup in a top-down fashion (with respect to the $\mathcal{J}$-order). We first detect each maximal subword whose product is maximal for the $\mathcal{J}$-order, evaluate those subwords and multiply the results with the letters that immediately follow.

▶ **Definition 9** ($\mathcal{J}$-maximal falling words). *A word u in S^+ is* $\mathcal{J}$-maximal falling *whenever for every $p, s \in S^*$ and $v, w \in S^+$ so that $u = pvws$, we have $\pi_S(vw) \in S_2$.*

Note that when u is $\mathcal{J}$-maximal falling and $|u| > 1$, $\pi_S(u) \in S_2$.

▶ **Definition 10** ($\mathcal{J}$-maximal decomposition). *Consider a word $u \in S^+$. If $u \notin S_2^+$, let $t \in \mathbb{N}$ and some words $w_0, \ldots, w_{t+1} \in S^*$, $v_0, \ldots, v_t \in S^+$, that define a decomposition $u = w_0 v_0 w_1 \cdots v_t w_{t+1}$. This decomposition is called* $\mathcal{J}$-maximal *if we have*

- *for any integer i such that $0 \leq i \leq t+1$, w_i is a word in S_2^**
- *for any integer i such that $1 \leq i \leq t$, v_i is a maximal subword of u verifying $\pi_S(v_i) \in D_1(S)$. More formally, if we consider the decomposition $u = pav_ibs$, with $p, s \in S^*$, and $a, b \in S \cup \epsilon$ (a is the letter preceding v_i, if it exists, and b is the letter following v_i, if it exists) then, if $a \neq \epsilon$, $\pi_S(av_i) <_{\mathcal{J}} \pi_S(v_i)$ and, if $b \neq \epsilon$, $\pi_S(v_i b) <_{\mathcal{J}} \pi_S(v_i)$*

We call $t+1$ the size of the decomposition.

If $u \in S_2^+$, the $\mathcal{J}$-maximal decomposition of u is composed of only one element equal to u itself. In this case, the decomposition is unique and by convention of size 0.

▶ Remark 11. The property of the (v_i)'s implies that each v_i is a word of $D_1(S)^+$. Indeed, the product of a word is at most $\mathcal{J}$-equivalent to the letter of greatest $\mathcal{J}$-depth, so all the letters must be of $\mathcal{J}$-depth 1 for the product to be in $D_1(S)$.

We will use the following useful technical lemma.

▶ **Lemma 12.** *Let S be a semigroup. Let x, y be $\mathcal{J}$-equivalent elements of S and z another element of S.*

- *If $\pi_S(xy) \mathcal{J} x$ and $\pi_S(zxy) <_{\mathcal{J}} x$, then $\pi_S(zx) <_{\mathcal{J}} x$.*
- *If $\pi_S(xy) \mathcal{J} x$ and $\pi_S(xyz) <_{\mathcal{J}} x$, then $\pi_S(yz) <_{\mathcal{J}} x$.*

We are now proving the existence and uniqueness of $\mathcal{J}$-maximal decomposition. The proof works by considering a variant of $\mathcal{J}$-maximal decomposition by enforcing the maximality constraint only at the right and showing that the two variants are actually equivalent.

▶ **Lemma 13.** *Let S be a semigroup. For any finite word $u \in S^+$, there exists a unique $\mathcal{J}$-maximal decomposition of u.*

Proof. We focus on proving the existence of such a decomposition. Unicity follows from the proof of existence. First, note that $S = D_1(S) \uplus S_2$. Thus, for any word $u \in S^+$, there necessarily exists a unique decomposition of u of the form $u = w_0 v_0 w_1 \cdots w_t v_t w_{t+1}$ such that $w_0, w_{t+1} \in S_2^*$, for each integer i such that $1 \leq i \leq t$, $w_i \in S_2^+$, and, for each $i \in [t+1]$, $v_i \in D_1(S)^+$. Informally, this is a decomposition of u based only on the $\mathcal{J}$-depth of each letter. Thanks to Remark 11, we know that the words w_i that are not empty in any $\mathcal{J}$-maximal decomposition of u correspond exactly to the words w_i of this decomposition. Thus, we only need to prove that any word over $D_1(S)$ can be decomposed into a unique sequence of maximal subwords whose product is in $D_1(S)$.

Consider a fixed word $u \in D_1(S)^+$. We prove that there exists a decomposition of u of the form $u = v_0 \cdots v_s$, where each word v_i is maximal in u such that $\pi_S(v_i) \in D_1(S)$.

If $\pi_S(u) \in D_1(S)$, then both existence and unicity follow from the definition.

Suppose now that $\pi_S(u) \in S_2$. Then we define a decomposition of u with weaker properties than the $\mathcal{J}$-maximal decompositions, and we prove that it is a $\mathcal{J}$-maximal decomposition of u (with empty words as the (w_i)'s). This decomposition is of the form $u = v_0 v_1 \cdots v_t$ for some integer $t \in \mathbb{N}$ and is defined from left to right such that, for each integer $i \in [t+1]$, $\pi_S(v_i) \in D_1(S)$ and, for each integer $i \in [t]$, $\pi_S(v_i x_{i+1}) \in S_2$, where x_{i+1} is the first letter of v_{i+1}. By construction, this decomposition exists and is unique. We prove that this decomposition satisfies the properties of the (v_i)'s given in Definition 10, that is, we prove that each word v_i is maximal such that $\pi_S(v_i) \in D_1(S)$. To do that, we only need to prove that, for each integer i such that $1 \leq i \leq t$, $\pi_S(y_{i-1} v_i) \in S_2$, where y_{i-1} is the last letter of the word v_{i-1}.

Thus, consider some integer $i \in [t]$. We focus on the subword $v_i v_{i+1}$ in order to prove that $\pi_S(y_i v_{i+1}) \in S_2$, where y_i is the last letter of v_i. This fact is a consequence of Lemma 12. Since we know that $\pi_S(v_i x_{i+1}) \in S_2$, where x_{i+1} is the first letter of v_{i+1}, and that $\pi_S(v_i) \in D_1(S)$, Lemma 12 implies that $\pi_S(y_i x_{i+1}) \in S_2$. Thus, $\pi_S(y_i v_{i+1}) \in S_2$. Thus, this decomposition is the unique $\mathcal{J}$-maximal decomposition of u. ◀

Now, we need to refine the $\mathcal{J}$-maximal decomposition. Keeping the same notation for u, and its $\mathcal{J}$-maximal decomposition $u = w_0 v_0 \cdots v_t w_{t+1}$, we set, for any $0 \leq i \leq t$, $v_i = x_i v_i'$ ($x_i \in S$ is the first letter of v_i), $w_i' = w_i x_i$ and $w_i' = y_i w_i''$ ($y_i \in S$ is the first letter of w_i'). The evaluation program Collapse_S takes a word $u \in S^+$ and performs at once all the products $\pi_S(v_i' y_{i+1})$, the operation $\pi_S(v_t')$ if $w_{t+1} = \epsilon$, or $\pi_S(v_t' y_{t+1})$ otherwise, with $w_{t+1}' = y_{t+1} w_{t+1}$ ($y_{t+1} \in S$ is the first letter of w_{t+1}). In symbols:

$\text{Collapse}_S(u) = w_0' \pi_S(v_0' y_1) w_1'' \cdots \pi_S(v_i' y_{i+1}) w_{i+1}'' \cdots w_t'' z$, where z denotes either $\pi_S(v_t')$ if w_{t+1} is the empty word, or $\pi_S(v_t' y_{t+1}) w_{t+1}'$. The image of any word $u \in S^+$ by Collapse_S is a $\mathcal{J}$-maximal falling word.

Given an element $s \in D_1(S)$, we also define the partial evaluation program $\text{Falling}_S(s)$ which takes a word $u \in S^+$ which is a $\mathcal{J}$-maximal falling word such that the last letter of u is in S_2, and returns a partial evaluation of u in which there is no occurrence of s. Given a word u, we call s-decomposition of u the decomposition of the form $u = w_0 s^{k_0} x_0 w_1 \cdots s^{k_t} x_t w_{t+1}$ where the k_i's are positive integers, the x_i's are non-s elements of S, except for x_t which

can also be equal to ϵ if $w_{t+1} = \epsilon$, and the w_i's are words without any occurrence of s. The s-decomposition of a word always exists and is unique. Given the s-decomposition of some word $u \in S^+$, keeping the same notation, we define

$$\text{Falling}_S(s)(u) = w_0 \pi_S(s^{k_1} x_1) \cdots \pi_S(s^{k_t} x_t) w_t$$

▶ **Lemma 14.** *Let S be a semigroup of $\mathcal{J}$-depth d. The evaluation program π_S is equal to the composition of $O(|S|)$ evaluation programs among* RProd_S*,* Collapse_{S_i} *and* $\text{Falling}_{S_i}(s)$ *for all $1 \leq i \leq d$ and $s \in D_1(S_i)$.*

▶ Remark 15. Waterfall evaluation programs have some ressemblance with the *factorizations forest* of Simon [3]. Indeed, our programs create a factorization forest for each word they are applied to. Moreover, the proof of the factorization forests theorem uses an induction on $\mathcal{J}$-classes, as we do for our programs. However, they are not quite the same. A waterfall evaluation program can be applied to any word on the right alphabet, whereas the factorization forests theorem proves the existence of a forest of bounded depth for a fixed word. This theorem is used to prove the existence of a formula corresponding to a given semigroup in two cases: the classes $\mathcal{B}\Sigma_1$ and Σ_1. To our knowledge, there are no proofs for the class **DA** or the class of aperiodic semigroups. Lastly, such proofs amount to consider all the formulas of some quantifier depth that depends on the forest depth, a technique that resembles Wilke's proof for **DA** [35, Corollary 1] and also gives awful upper bounds.

3.4 Sweeping evaluation programs

We introduce an evaluation program which processes words in a more lateral fashion – from left to right or right to left. In this subsection, S is a semigroup of $\mathcal{J}$-depth d.

In the proofs of Section 4, we will perform evaluations that produce left (resp. right) $\mathcal{J}$-constant prefixes (resp. suffixes). We define those programs depending on the $\mathcal{J}$-depth of the semigroup that is considered. First, we introduce a *left splitting* higher order operation that applies an evaluation program over a prefix of the input word defined by some constraints over Green's relations. Formally, for any integer $i \leq d$, we define the operation $\text{LSplit}_{S,i}$ as follows. Consider a word $u = s_0 \cdots s_k \in S^+$ and an evaluation program P defined at least on all left $\mathcal{J}$-constant words of depth i. If s_0 is not of $\mathcal{J}$-depth i, we set $\text{LSplit}_{S,i}\langle P\rangle(u) = u$. Otherwise, there exist two uniquely defined words $p \in S^+$, $s \in S^*$ such that $u = ps$, where p is left $\mathcal{J}$-constant and either s is empty or, if we denote by $x \in S$ its first letter, $\pi_S(px) <_{\mathcal{J}} \pi_S(p)$. In this case, $\text{LSplit}_{S,i}\langle P\rangle(u) = P(p)s$. We define similarly the symmetric operation $\text{RSplit}_{S,i}$. Finally, we define the partial function JProd_S, that is the restriction of π_S over words that are $\mathcal{J}$-constant. Formally, JProd_S is defined only on $\mathcal{J}$-constant words and, given $u \in S^+$ such a word, $\text{JProd}_S(u) = \pi_S(u)$.

A *sweeping evaluation program* is an evaluation program built with the following operations: LProd_S, RProd_S, the partial function JProd_S, and the higher order operations $\text{LSplit}_{S,i}$ and $\text{RSplit}_{S,i}$ for any integer i such that $1 \leq i \leq d$.

Example. Consider a semigroup S and the sweeping evaluation program $P = \text{LProd}_S \circ \text{LSplit}_{S,1}\langle\text{JProd}_S\rangle$. The program P first executes the operation $\text{LSplit}_{S,1}\langle\text{JProd}_S\rangle$. To do so, it begins by looking at the beginning of the input word w. If the first letter of w is not in $D_1(S)$, then P computes only $\text{LProd}(w)$, which is the word obtained by multiplying the two first letters of w. Otherwise, P decomposes w into two words $p \in D_1(S)^+$, $s \in S^*$ such that $u = ps$ and, if s is not the empty word, $\pi_S(px) \in S_2$, where x is the first letter of s. Then, P applies the operation JProd_S to the prefix p. Then, the result of $\text{LSplit}_{S,1}\langle\text{JProd}_S\rangle(w)$ is the word $\pi_S(p)s$. Finally, P applies the operation LProd_S to the previous result, multiplying the first two letters of $\pi_S(p)s$, if s is not empty.

▶ **Lemma 16** (Sweeping evaluation programs). *Let S be a semigroup. There exists a sweeping evaluation program computing π_S. Moreover, there exists such a program that is equal to the composition of $O(2^d)$ operations.*

4 Proof of the main results

We recall the proof strategy. Given a regular language which is aperiodic (resp. in **DA**), we prove that we can compute π_S with a ADD- (resp. Sweeping-) vectorial circuit. To build such circuits, we rely on respectively waterfall and sweeping evaluations programs. As the considered vectorial circuit classes are closed under functional composition, it is sufficient to prove that every basic operation in our evaluation programs is computable with the desired vectorial circuit class. We provide those arguments in this section and the corresponding section of the appendix.

Vectorial encoding of a partial evaluation of a word. Our sweeping and waterfall evaluation programs perform operations on partial evaluations of a word, so we need to provide an explanation on how we encode these partial evaluations. From now on, a partial evaluation will always designate a partial evaluation of a word, usually the initial word that needs to be processed with the evaluation program. Informally, a vectorial representation of a partial evaluation is a set of vectors, each vector corresponding to a semigroup element. The size of the vectors is equal to the size of the initial word on which we apply the evaluation program. We need the vectors we employ to always have the same size throughout the execution, so our definition needs to be more general than indicator vectors. Each bit set to one denotes the presence of the element in the partial evaluation, and the order of the bits set to one determines the order of the letters in the word. As in the case of characteristic functions of letters, in the vector encoding of a word, vectors in an encoding do not overlap, however their union may not cover all the positions. More formally, a *vectorial encoding of a partial evaluation* is a mapping $\mathbf{c}\colon S \to \{0,1\}^n$, for some integer $n \geq 1$, such that we have the following constraint: for any $s, t \in S$ such that $s \neq t$, we have $\mathbf{c}(s) \wedge \mathbf{c}(t) = \mathbf{0}$. Note that, by definition, a word is a partial evaluation of itself, so $\mathrm{enc}(u)$ is a vectorial encoding of a partial evaluation, with the additional property that $\bigvee_{a \in S} \mathbb{1}_a(u) = \mathbf{1}$.

Given such a function $\mathbf{c}$ outputting vectors of dimension n, we can interpret it as a word of length at most n by respecting the order of appearance of the bits. Formally, a word $s_0 \cdots s_k \in S^{\leq n}$ is represented by a vectorial encoding $\mathbf{c}$ of dimension n if, for every index $j \leq k$, there exists some integer i such that $0 \leq i < n$, $\mathbf{c}(s_j)_i = 1$ and $|\{t \in \mathbb{N} \mid t < i \text{ and } \bigvee_{s \in S} \mathbf{c}(s)_t = 1\}| = j - 1$.

In this section, we use only circuits of the following form: the input is a vectorial encoding of a partial evaluation $\mathbf{c}\colon S \to \{0,1\}^n$ (for some $n \in \mathbb{N}^+$) representing some word $u \in S^{\leq n}$, and the output is a vectorial encoding $\mathbf{out}\colon S \to \{0,1\}^n$ representing the partial evaluation of u obtained by applying the operation corresponding to the circuit. To prove our theorems, we construct vectorial circuits using this encoding that implement the partial evaluation functions we presented earlier.

Example. For example, consider the semigroup $S = \{a, b, c\}$ with the inner multiplication as follows: $\pi_S(ac) = \pi_S(ca) = c$, $\pi_S(bc) = \pi_S(cb) = c$, $\pi_S(aa) = b$. The word $w = aaabac$ has the vectorial encoding $\mathbf{a} = 111010$, $\mathbf{b} = 000100$, $\mathbf{c} = 000001$. With our circuits, if we multiply the last two letters of w using this encoding, we will obtain the encoding $\mathbf{a} = 111000$, $\mathbf{b} = 000100$, $\mathbf{c} = 000001$. As expected, it represents the word $w' = aaabc$: the first four

elements of the vectors are the same and still represent the word $aaab$, the fifth element is a 0 in all vectors so it represents no letter, and the sixth element is a 1 in $\mathbf{c}$, so it represents the letter c. Since we started from a word of length 6, the length of the vectors is still 6 but some of the indices do not represent a letter anymore.

Sketch of proof of Theorem 6. To prove Theorem 6, we consider a fixed aperiodic semigroup S, and we denote by d its $\mathcal{J}$-depth. Thanks to Lemma 14, it is sufficient to provide ADD-vectorial circuits computing the operations $\mathrm{Collapse}_S$, RProd_S and $\mathrm{Falling}_S(s)$ (for any aperiodic semigroup S and $s \in D_1(S)$) over some vectorial encoding of any partial evaluation of a word. Once we have those, we proceed as in Lemma 14, in which we prove that, for any integer i such that $1 \leq i \leq d$, π_{S_i} is equal to the composition of $\pi_{S_{i+1}}$ and $O(|S|)$ operations among $\mathrm{Collapse}_{S_i}$, RProd_S and $\mathrm{Falling}_S(s)$ (for any $s \in D_i(S)$). More precisely, the evaluation program for π_S uses $|S|$ operations of the form $\mathrm{Falling}_{S'}(s)$ (where $s \in S$), d operations of the form RProd_S and d operations of the form $\mathrm{Collapse}_{S_i}$ (recall that d is the $\mathcal{J}$-depth of S). Now, we can obtain the result by using the following lemmas:

▶ **Lemma 17.** *For any aperiodic semigroup S, we can compute* $\mathrm{Collapse}_S$ *over any vectorial encoding of a partial evaluation with an* ADD*-vectorial circuit of size $O(|S|^3)$.*

▶ **Lemma 18.** *For any aperiodic semigroup S, we can compute* RProd_S *on any vectorial encoding of a partial evaluation of a word over S with an* ADD*-vectorial circuit of size $O(|S|^2)$.*

▶ **Lemma 19.** *Let S be an aperiodic semigroup of $\mathcal{J}$-depth d. For any element $s \in D_1(S)$, we can compute* $\mathrm{Falling}_S(s)$ *over any vectorial encoding of a partial evaluation in its domain with an* ADD*-vectorial circuit of size $O(d|S|)$.*

With this, we can conclude that π_S can be computed with an ADD-vectorial circuit of size $O(d|S|^3)$.

Sketch of proof of Theorem 5. To prove Theorem 5, we consider a fixed semigroup $S \in \mathbf{DA}$, and we denote by d its $\mathcal{J}$-depth. Thanks to Lemma 16, it is sufficient to provide Sweeping-vectorial circuits computing the base operations over some vectorial encoding of any partial evaluation of a word. Those operations are LProd_S, RProd_S, JProd_S and, for each integer i such that $1 \leq i \leq d$, the operations $\mathrm{LSplit}_{S,i}$ and $\mathrm{RSplit}_{S,i}$. Once we have those, we proceed as in Lemma 16, in which we prove that π_S is equal to the composition of $O(2^d)$ operations among JProd_S, LProd_S, RProd_S and $\mathrm{LSplit}_{S,i}\langle E \rangle$ (for any integer i such that $1 < i \leq d$ and some sweeping program E composed of the same operations, that are taken into account in the $O(2^d)$). To prove this, we have $\pi_S = P_{d,l} = P_{d,r}$ where, for each integer i such that $1 \leq i \leq d$, the program $P_{i,l}$ computes π_S on the maximal prefix of $\mathcal{J}$-depth at most i (included), and the symmetric program $P_{i,r}$ which acts on suffixes instead of prefixes. Our proof constructs Sweeping-vectorial circuits for those programs, by induction on the depth i. Now, we can obtain the result by using the following lemmas:

▶ **Lemma 20.** *For any semigroup $S \in \mathbf{DA}$, we can compute* JProd_S *over any vectorial encoding of a partial evaluation in its domain with a* Sweeping*-vectorial circuit of size $O(|S|^2)$.*

▶ **Lemma 21.** *For any semigroup $S \in \mathbf{DA}$, we can compute* LProd_S *and* RProd_S *over any vectorial encoding of a partial evaluation in their domains with* Sweeping*-vectorial circuits of size $O(|S|^2)$.*

▶ **Lemma 22.** *Let S be a semigroup in* $\mathbf{DA}$ *of $\mathcal{J}$-depth d, i be an integer such that $1 \leq i \leq d$, let P be a sweeping evaluation program defined at least on all left $\mathcal{J}$-constant words of depth i, and suppose that we have a* Sweeping-*vectorial circuit of size s_P that computes P over any vectorial encoding of a partial evaluation. Then we can compute* $\mathrm{LSplit}_{S,i}\langle P\rangle$ *and* $\mathrm{RSplit}_{S,i}\langle P\rangle$ *over any vectorial encoding of a partial evaluation in their respective domains with* Sweeping-*vectorial circuits of size $O(|S|^2 + s_P)$.*

Thus, following the construction in the proof of Lemma 16, for each integer i such that $1 \leq i < d$, we have a Sweeping-vectorial circuit computing $P_{i+1,l}$ (or $P_{i+1,r}$, these are symmetrical operations) given some circuits computing $P_{i,l}$ and $P_{i,r}$, and its size is equal to $O(|S|^2 + 2c_i)$, where c_i is the size of the circuits computing $P_{i,l}$ and $P_{i,r}$. Since, by the construction of Lemma 16, we also have $c_1 = O(|S|^2)$, we then have $c_i = (2^i - 1)|S|^2$ for each integer i such that $1 < i \leq d$. Thus, the circuit computing $\pi_S = P_{d,l}$ is of size $O(2^d|S|^2)$.

5 Conclusion

We introduce vectorial circuits as abstractions of low level hardware architectures. As circuits, they put forward dependencies between steps of computation and thus opportunities of parallelism. Taking stock on previous work [16, Theorem 1], the next step is then to adapt these circuits to a streaming context. This streaming setting is closer to text processing problems as only small chunks of input data can hold at the level of CPUs. We shall then study how to take advantage of SIMD instructions to implement these machines. The vast number of SIMD instructions give many possibilities to compile vectorial circuits for streaming text. The challenge is then to find the right set of instructions, combine them sufficiently well together and obtain efficient programs. Here again, we hope that algebra can back our efforts up.

Concerning the circuits themselves, we can consider extensions with operations that have natural correspondence in CPU instructions such as *shifts*, *prefix-xor*, etc. The question is then to understand what classes of regular languages we can describe. Hopefully, the combinations of particular sets of operations could correspond to well-studied algebraic operations [27]. Instead of operations, we can also consider the use of arbitrary constants, (i.e. particular vectors that can be used as gates inputs). We think that using constants in circuits can be related to that of arbitrary monadic advices [7].

In the paper, we limit ourselves to consider vectorial circuits as recognizers. As these circuits produce outputs, they should be more adequately viewed as transducers. On the theoretical side, there is a need to explore their expressivity. This requires an adaptation of the algebraic tools we use, i.e. an understanding of classes of logical transducers. On the practical side, in the context of streaming, this point of view calls for composing stream processing programs. This probably requires to explore ideas from synchrone programming in combination with vectorial circuits.

This paper also calls for more foundational work concerning circuit complexity. In particular we conjecture that the bounds provided in Theorem 5 are tight. However, finding an adequate lower bound is a challenging open problem.

References

1 Anne Bergeron and Sylvie Hamel. Cascade decompositions are bit-vector algorithms. In *International Conference on Implementation and Application of Automata*, pages 13–26. Springer, 2001.

2 Anne Bergeron and Sylvie Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(01):53–65, 2002.

3 Mikolaj Bojańczyk. Factorization forests. In *International Conference on Developments in Language Theory*, pages 1–17. Springer, 2009.

4 Robert D Cameron, Thomas C Shermer, Arrvindh Shriraman, Kenneth S Herdy, Dan Lin, Benjamin R Hull, and Meng Lin. Bitwise data parallelism in regular expression matching. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 139–150. IEEE, 2014.

5 V. Diekert, P. Gastin, and M. Kufleitner. A survey on small fragments of first-order logic over finite words. *Internat. J. Found. Comput. Sci.*, 19:513–548, 2008.

6 Michael Farrar. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, November 2006.

7 Nathanaël Fijalkow and Charles Paperman. Monadic second-order logic with arbitrary monadic predicates. *ACM Transactions on Computational Logic (TOCL)*, 18(3):1–17, 2017.

8 Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

9 Johan Anthony Wilem Kamp. *Tense logic and the theory of linear order*. University of California, Los Angeles, 1968.

10 John Keiser and Daniel Lemire. Validating UTF-8 in less than one instruction per byte. *Software: Practice and Experience*, 51(5):950–964, 2021.

11 Donald Ervin Knuth. The art of computer programming: Bitwise tricks & techniques. *Binary Decision Diagrams*, 4, 2009.

12 M Oguzhan Külekci. Filter based fast matching of long patterns by using SIMD instructions. In *Stringology*, pages 118–128, 2009.

13 Leslie Lamport. Multiple byte processing with full-word instructions. *Communications of the ACM*, 18(8):471–475, 1975.

14 Geoff Langdale and Daniel Lemire. Parsing gigabytes of JSON per second. *The VLDB Journal*, 28(6):941–960, 2019.

15 McNaughton, Robert, Papert, and Seymour A. *Counter-Free Automata (MIT research monograph no. 65)*. The MIT Press, 1971.

16 Filip Murlak, Charles Paperman, and Michal Pilipczuk. Schema validation via streaming circuits. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 237–249, 2016.

17 Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

18 Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 151–160. IEEE, 2011.

19 Dorit Nuzman and Ayal Zaks. Autovectorization in GCC–two years later. In *Proceedings of the 2006 GCC Developers Summit*, volume 6, 2006.

20 Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 2–11, 2008.

21 Charles Paperman, Sylvain Salvati, and Claire Soyez-Martin. Addition Lemma, September 2022. URL: `https://hal.archives-ouvertes.fr/hal-03787033`.

22 Charles Paperman, Sylvain Salvati, and Claire Soyez-Martin. An algebraic approach to vectorial programs. Complete version of the paper, January 2023. URL: `https://hal.archives-ouvertes.fr/hal-03831752v2`.

23 J.E. Pin and European Mathematical Society Publishing House ETH-Zentrum SEW A27. *Handbook of Automata Theory: Volume I: Theoretical Foundations; Volume II: Automata in Mathematics and Selected Applications*. EMS Press, 2021.

24 Jean Eric Pin. *Varieties of formal languages*, volume 184. Springer, 1986.

25 M. P. Schützenberger. On finite monoids having only trivial subgroups. *Information and control*, 8:190–194, 1965.

26 Olivier Serre. Vectorial languages and linear temporal logic. *Theoretical computer science*, 310(1-3):79–116, 2004.

27 Howard Straubing. Finite semigroup varieties of the form V*D. *Journal of Pure and Applied Algebra*, 36:53–94, 1985.

28 Howard Straubing. *Finite automata, formal logic, and circuit complexity*. Springer Science & Business Media, 2012.

29 Pascal Tesson and Denis Thérien. Diamonds are forever: The variety DA. In *Semigroups, algorithms, automata and languages*, pages 475–499. World Scientific, 2002.

30 Denis Therien and Pascal Tesson. Logic meets algebra: the case of regular languages. *Logical Methods in Computer Science*, 3, 2007.

31 Denis Thérien and Thomas Wilke. Over words, two variables are as powerful as one quantifier alternation. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 234–240, 1998.

32 Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337. IEEE, 2009.

33 Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 631–648, 2019.

34 Philipp Weis and Neil Immerman. Structure theorem and strict alternation hierarchy for FO^2 on words. In *International Workshop on Computer Science Logic*, pages 343–357. Springer, 2007.

35 Thomas Wilke. Classifying discrete temporal properties. In *Annual symposium on theoretical aspects of computer science*, pages 32–46. Springer, 1999.

36 James Worrell, Rastislav Lenhardt, and Michael Benedikt. Two variable vs. linear temporal logic in model checking and games. *Logical Methods in Computer Science*, 9, 2013.

A Proofs for Section 3 (Semigroups evaluation strategies)

Every word in S^+ admits a unique $\mathcal{J}$-maximal decomposition of a word. This property hinges on the Localisation Theorem of Clifford and Miller [24, Proposition 1.6, page 48].

▶ **Lemma 23** (Localisation Theorem)**.** *Let S be a semigroup and x, y be in S. We have $xy\mathcal{J}x$ if and only if there exists an idempotent e in $R(y) \cap L(x)$.*

▶ **Lemma 12.** *Let S be a semigroup. Let x, y be $\mathcal{J}$-equivalent elements of S and z another element of S.*

- *If $\pi_S(xy)\mathcal{J}x$ and $\pi_S(zxy) <_{\mathcal{J}} x$, then $\pi_S(zx) <_{\mathcal{J}} x$.*
- *If $\pi_S(xy)\mathcal{J}x$ and $\pi_S(xyz) <_{\mathcal{J}} x$, then $\pi_S(yz) <_{\mathcal{J}} x$.*

Proof. Both cases are symmetric, so we only prove the first one. Suppose that $\pi_S(zx)\mathcal{J}x$, $\pi_S(xy)\mathcal{J}x$ and $\pi_S(zxy) <_{\mathcal{J}} x$. Lemma 23 implies that there is no idempotent in $R(y) \cap L(zx)$. But, as $\pi_S(zx)\mathcal{J}x$, by definition of $\mathcal{L}$, we have $\mathcal{L}(zx) = \mathcal{L}(x)$. Therefore $R(y) \cap L(x)$ does not contain an idempotent. Finally Lemma 23 entails $xy <_{\mathcal{J}} x$, a contradiction. ◀

▶ **Lemma 14.** *Let S be a semigroup of $\mathcal{J}$-depth d. The evaluation program π_S is equal to the composition of $O(|S|)$ evaluation programs among* RProd_S*,* $\mathrm{Collapse}_{S_i}$ *and* $\mathrm{Falling}_{S_i}(s)$ *for all $1 \leq i \leq d$ and $s \in D_1(S_i)$.*

Proof. For any integer i such that $1 \leq i \leq d$, we define $O_i = (s_1, \dots, s_k)$ to be any enumeration of $D_1(S_i)$. Given such an enumeration, we define the following operation:

$$\mathrm{Falling}_{S_i}[O_i] = \mathrm{Falling}_{S_i}(s_k) \circ \cdots \circ \mathrm{Falling}_{S_i}(s_1)$$

We define an intermediate partial evaluation program f_i which is a restriction of π_S to the domain $S_i^+ \cup S$. In symbols, for any word $u \in S_i^+ \cup S$, $f_i(u) = \pi_S(u)$. Note that f_1 is equal to π_S on any word of S^+. We prove by induction over $j = d - i$, with $i \in [d]$, that

$$f_j = f_{j+1} \circ \mathrm{Falling}_{S_j}[O_j] \circ \mathrm{RProd}_S \circ \mathrm{Collapse}_{S_j}$$

with the convention that f_{d+1} is the identity. The base case (f_{d+1}) being fixed, we only need to prove the result by induction on i for $0 \leq i \leq d-1$. We remark that the image of $\mathrm{Collapse}_{S_j}$ produces a $\mathcal{J}$-maximal falling word, and then the application of RProd_S guarantees that the last element is not in $D_1(S)$. Finally, the application of $\mathrm{Falling}_{S_i}(s_i)$ on the resulting elements produces new elements that are only in S_{i+1} and removes all occurrences of the element s_i. Hence by applying $\mathrm{Falling}_{S_i}[O_i]$ we obtain a word in S_{i+1}^*, which concludes the proof. ◀

▶ **Lemma 16** (Sweeping evaluation programs). *Let S be a semigroup. There exists a sweeping evaluation program computing π_S. Moreover, there exists such a program that is equal to the composition of $O(2^d)$ operations.*

To prove Lemma 16, we introduce an intermediate operation. Let i be an integer such that $1 \leq i \leq d$. We denote by $P_{i,l}$ the *left sweeping evaluation program* of $\mathcal{J}$-depth i, which computes π_S on the maximal prefix of $\mathcal{J}$-depth at most i (included). Formally, given a word $u = s_0 \cdots s_k \in S^+$, $P_{i,l}(u)$ is equal to u if s_0 is of $\mathcal{J}$-depth strictly greater than i. Otherwise, there exist $p \in S^+$, $q \in S^*$ such that $u = pq$, where either q is empty, or $x \in S$ is its first letter and then $\pi_S(p)$ is of $\mathcal{J}$-depth at most i and $\pi_S(px)$ is of $\mathcal{J}$-depth strictly greater than i. In this case, $P_{i,l}(u) = \pi_S(p)q$. We define symmetrically $P_{i,r}$, the *right sweeping evaluation program* of $\mathcal{J}$-depth i.

The next lemma allows to conclude the proof of Lemma 16 since $\pi_S = P_{d,l} = P_{d,r}$.

▶ **Lemma 24.** *For any integer i such that $1 \leq i \leq d$, there exist sweeping evaluation programs computing $P_{i,l}$ and $P_{i,r}$.*

Proof of Lemma 24. We will prove by induction on the $\mathcal{J}$-depth i that we can implement a left (resp. right) sweeping evaluation program $P_{i,l}$ (resp. $P_{i,r}$). In this proof, we consider a word $u = s_0 \cdots s_{k-1}$ over S. For the base case, we first suppose that $i = 1$, i.e. we consider maximal $\mathcal{J}$-classes. Thus, if s_0 is of $\mathcal{J}$-depth 1, we will compute the product of the unique prefix $s_0 \cdots s_p$ of u such that $\pi_S(s_0 \cdots s_p) \mathcal{J} s_0$ and $\pi_S(s_0 \cdots s_{p+1}) <_{\mathcal{J}} s_0$. If s_{p+1} does not exist, we want to compute $\pi_S(u)$. Note that $s_0 \cdots s_p$ is $\mathcal{J}$-constant, hence we can apply JProd_S to it. Thus, we can compute the base case $P_{1,l}$ using the program $\mathrm{LSplit}_{S,1}\langle \mathrm{JProd}_S \rangle$. Note that this program is well defined since JProd_S is in particular defined on all $\mathcal{J}$-constant words of depth 1, which are exactly the left $\mathcal{J}$-constant words of depth 1. Symmetrically, $P_{1,r} = \mathrm{RSplit}_{S,1}\langle \mathrm{JProd}_S \rangle$. To prove the induction case, we will rely on the following fact (see 3.1 for the definition of a left $\mathcal{J}$-constant word):

▶ **Fact.** *For any left $\mathcal{J}$-constant word $v \in S^+$ of $\mathcal{J}$-depth i, the word* $\mathrm{RProd}_S \circ P_{i-1,r}(v)$ *is $\mathcal{J}$-constant.*

Proof. The result is obtained from the fact that the last element of $w = \mathrm{RProd}_S \circ P_{i-1,r}(v)$ is necessarily of $\mathcal{J}$-depth i. Indeed, by definition, the word $x = P_{i-1,r}(v)$ is such that the product of its last two elements (if there are at least two elements) is at least of $\mathcal{J}$-depth i. Since we supposed that v is left $\mathcal{J}$-constant of $\mathcal{J}$-depth i, it is garanteed that this product is defined and is exactly of $\mathcal{J}$-depth i. Thus, both the first and last elements of w are of $\mathcal{J}$-depth i, as well as $\pi_S(w)$. Thus the product of any prefix or suffix of w will be of $\mathcal{J}$-depth i, and in the same $\mathcal{J}$-class as the first and last elements of w, which corresponds to the definition of $\mathcal{J}$-constant. ◀

For the induction step, we assume to have sweeping evaluation programs $P_{i,r}$ and $P_{i,l}$ for any integer $i < d$. We prove the result for $P_{i+1,r}$ and $P_{i+1,l}$. These two cases being symmetrical, we only show the result for $P_{i+1,l}$. Let $v = \mathrm{LProd}_S \circ P_{i,l}(u)$. If $|P_{i,l}(u)| \neq 1$, we have necessary that the first letter of v is of $\mathcal{J}$-depth strictly greater than i. Otherwise $v = P_{i,l}(u) = \pi_S(u)$. We are going to split v with respect to the $\mathcal{J}$-depth i and apply the program $E = \mathrm{JProd}_S \circ \mathrm{RProd}_S \circ P_{i,r}$ to the prefix. Indeed, after the split, and thanks to the previous Fact, we can apply JProd_S over the factor $\mathrm{RProd}_S \circ P_{i,r}(p)$, where p is the prefix obtained after the split. Indeed, this factor is $\mathcal{J}$-constant. To conclude, $P_{i+1,l} = \mathrm{LSplit}_{S,i+1}\langle E\rangle \circ \mathrm{LProd}_S \circ P_{i,l}$. Thus, each P_i is defined using $O(2^i)$ operations. ◀

B Proofs for Section 4 (Proof of the main results)

We introduce two vectors that we will use extensively in our circuits: given S a semigroup and $\mathbf{c}$ a vectorial encoding of a partial evaluation, we define the universe vector $\mathbf{U} = \bigvee_{s \in S} \mathbf{c}(s)$ We also define the vector marking the end of the vectors: $\mathbf{End} = \neg\mathrm{MSB}(\mathbf{1})$.

Proofs for Theorem 6. In our proofs involving aperiodic semigroups, we will rely on some classical equivalent characterizations of this variety of semigroups.

▶ **Proposition 25** ([25]). *Let S be a semigroup. The following conditions are equivalent:*
- *S is aperiodic*
- *there exists an integer ω such that for all $s \in S$, $\pi_S(s^\omega) = \pi_S(s^{\omega+1})$*
- *All $\mathcal{H}$-classes of S are trivial*

Here is a technical property of aperiodic semigroups that will be useful in the proofs.

▶ **Lemma 26.** *Let S be an aperiodic semigroup. Suppose that $u = s_0 \cdots s_k \in S^+$ is a $\mathcal{J}$-constant word. Then, $\pi_S(u)$ is the unique element of $\mathcal{R}(s_0) \cap \mathcal{L}(s_k)$. If $k > 0$, this also implies that $\pi_S(u) = \pi_S(s_0 s_k)$.*

▶ **Lemma 17.** *For any aperiodic semigroup S, we can compute* $\mathrm{Collapse}_S$ *over any vectorial encoding of a partial evaluation with an* ADD*-vectorial circuit of size $O(|S|^3)$.*

Proof. Consider a word $u \in S^+$ and its $\mathcal{J}$-maximal decomposition $u = w_0 v_1 x_1 \cdots v_t x_t w_t v_{t+1}$. Our goal is to compute a vectorial encoding of the partial evaluation $u' = w_0 \pi_S(v_1 x_1) w_1 \cdots w_{i-1} \pi_S(v_i x_i) w_i \cdots w_t \pi_S(v_{t+1})$. We proceed as follows. We start by computing, for each $s \in D_1(S)$, the vector $\mathbf{SecondEl}(s)$ marking the positions that indicate an s at the beginning of some sub-word v_i. This is done using the operation Successor and the fact that the first letter of each v_i either is the first letter of the word or is such that the product with

the previous letter stays in $D_1(S)$. Similarly, for each $s \in D_1(S)$, we compute the vector $\mathbf{BlockFall}(s)$ marking each letter x_i that follows an occurrence of s. This is done using the operation Successor and the fact that each x_i is after a block of letters which product stays in $D_1(S)$. We then compute the products in the vectors $\mathbf{Prod}(p)$, for any $p \in S_2$. To do so, we define the set $F_p = \{(s,t,x) \in (D_1(S))^2 \times S \mid \pi_S(\alpha \cdot x) = p$, where $\alpha = \mathcal{R}(s) \cap \mathcal{L}(t)\}$. This set is exactly what we need since, thanks to Lemma 26, we know that the product of each subword v_i is only determined by s and t. Thus, this set gives all the triplets (s,t,x) such that, if a subword v_i has s as its first letter, t as its last letter, and if $x_i = x$, then $\pi_S(v_i x_i) = p$. We compute $\mathbf{Prod}(p)$ as a union of calls to Successor over the set F_p. Note that each triplet $(s,t,x) \in (D_1(S))^2 \times S$ can only appear in at most one set F_p, so the size of the union of all these sets is only $O(|D_1(S)|^2 * |S|) \leq O(|S|^3)$. Finally, we compute the output vectors by adding the products computed earlier and removing all the letters used in these products. ◀

▶ **Lemma 18.** *For any aperiodic semigroup S, we can compute* RProd_S *on any vectorial encoding of a partial evaluation of a word over S with an* ADD*-vectorial circuit of size $O(|S|^2)$.*

Proof. Let $u \in S^+$ be a partial evaluation such that u is a $\mathcal{J}$-maximal falling word. To compute the vectorial encoding of $\mathrm{RProd}_S(u)$, we begin by labeling the last element by its value: the vector $\mathbf{LastEl}(t)$ is equal to $\mathbf{End}$ if and only the last letter of u is a t. Then, we use $O(|S|^2)$ calls to Successor to determine the value of the product of the last two elements. We replace the last two elements by that value if the size of u is at least 2, using the vector $\mathrm{Thr2}(\mathbf{U})$ and the IfThenElse circuit we presented earlier in the article. Since that circuit is of constant size, the circuit for RProd_S is of size $O(|S|^2)$. ◀

Before proving Lemma 19, we prove the following technical lemma.

▶ **Lemma 27.** *Let S be an aperiodic semigroup and $u \in S^+$ a $\mathcal{J}$-maximal falling word over S such that its last letter is not an element of $D_1(S)$. Consider a fixed element $s \in D_1(S)$ and write u as its s-decomposition $w_0 s^{k_1} x_1 w_1 \cdots w_{t-1} s^{k_t} x_t w_t$. Then, we can use an* ADD*-vectorial circuit of size $O(|S|)$ which takes the vectorial encoding as input and produces a vectorial encoding of the word $w_0 s^{k_1 - 1} \pi_S(s x_1) \cdots s^{k_t - 1} \pi_S(s x_t) w_t$.*

Proof. With a call to Successor and $O(|S|)$ conjunctions, we can obtain the vectors $\mathbf{Last}(t)$, for $t \in S \setminus \{s\}$, that mark the occurrences of t preceeded by an occurrence of s. Then, we can replace thos elements by the product $\pi_S(st)$. Instead of removing the corresponding occurrences of s, we remove the first s of each block, which is equivalent but far easier to do in our model. Those occurrences of s can be marked using a single call to Successor. ◀

▶ **Lemma 19.** *Let S be an aperiodic semigroup of $\mathcal{J}$-depth d. For any element $s \in D_1(S)$, we can compute* $\mathrm{Falling}_S(s)$ *over any vectorial encoding of a partial evaluation in its domain with an* ADD*-vectorial circuit of size $O(d|S|)$.*

Proof. Let u be a word of S^+ and the set $(\mathbf{c}(t))_{t \in S}$ be a vectorial encoding of u. Since we know that the last element of u is not in $D_1(S)$, we know that each block of occurrences of s is followed by at least one element. By applying Lemma 27, we can reduce by 1 the size of each of these blocks. Moreover, the semigroup S is aperiodic, so by Proposition 25 there necessarily exists an integer ω_s such that $\pi_S(s^{\omega_s+1}) = \pi_S(s^{\omega_s})$. Thus, if we apply Lemma 27 ω_s times, the only occurrences of s that will be left will be any letter s that was originally followed by at least ω_s other occurrences of s. Since $\pi_S(s^{\omega_s+1}) = \pi_S(s^{\omega_s})$, we can just forget those occurrences without changing anything else. Note that, for any $t \in S$, we have $\omega_t \leq d$, where d in the $\mathcal{J}$-depth of S, so we apply Lemma 27 at most d times. ◀

Proofs for Theorem 5 In our proofs involving semigroups in $\mathbf{DA}$, we will rely on some classical equivalent characterizations of the variety $\mathbf{DA}$.

▶ **Proposition 28** ([29, Theorem 2]). *Let M be a monoid. The following are equivalent:*

- *M is in the variety $\mathbf{DA}$*
- *if J is a regular $\mathcal{J}$-class of M, then J is an aperiodic semigroup*
- *$\forall x, y, z \in M, (xyz)^\omega y(xyz)^\omega = (xyz)^\omega$ (this is the algebraic characterization of $\mathbf{DA}$)*

Any semigroup in $\mathbf{DA}$ is aperiodic, so we will also be able to use Proposition 25. Moreover, the class $\mathbf{DA}$ admits the following nice property that we will rely on in the proofs.

▶ **Lemma 29** (Algebra folklore). *Let S be a semigroup in $\mathbf{DA}$ and R an $\mathcal{R}$-class of S. Then there exist two sets $T, K \subseteq S$ such that $S = T \uplus K$ and, for all $x \in R$, we have $\forall s \in T, xs\mathcal{R}x$ and $\forall s \in K, xs <_{\mathcal{J}} x$. Moreover, both T and K are sub-semigroups of S such that, if we denote by J is the $\mathcal{J}$-class containing R, $J \subseteq T$ if R is regular and $J \subseteq K$ otherwise. It follows that if S is a monoid, then T is also a monoid.*

Before proving the lemmas necessary for Theorem 5, we define some intermediary operations. These operations will all be of the same form: for any $i \in \mathbb{N}$, we define the operation $\text{Value}_{i,S}$ that identifies the ith semigroup element that occurs in the word represented by the input vectors. Formally, we define this operation as follows:

▶ **Definition 30.** *Let S be a semigroup in $\mathbf{DA}$ and let $(\mathbf{c}(s))_{s\in S}$ be a vectorial encoding of some word $u \in S^+$. For each $s \in S$ and $i \in \mathbb{N}$, we define the vector $\text{Value}_{i,S}(s)$ that is equal to $\mathbf{1}$ if and only if there exists an integer j such that the jth element of $\mathbf{c}(s)$ is a 1 and the position j is the ith position of the vector $\mathbf{U}$ to hold a 1. Otherwise, $\text{Value}_{i,S}(s) = \mathbf{0}$.*

▶ **Lemma 31.** *For any integer $i \geq 1$, we can compute the function $\text{Value}_{i,S}$ over any vectorial encoding of a partial evaluation with a* Sweeping*-vectorial circuit of size $O(i + |S|)$.*

Proof. Given a set of input vectors $I = (\mathbf{c}(s))_{s\in S}$, $\text{Value}_{i,S}(I)$ is a set of vectors $(\mathbf{out}(s))_{s\in S}$ such that, for each element $s \in S$, $\mathbf{out}(s)$ is computed as follows. We begin by removing the first $i-1$ bits set to 1 in the union of the inputs by defining the vectors $\mathbf{U}_0 = \mathbf{U}$ and, $\forall j < i-1$, $\mathbf{U}_{j+1} = \text{LSB}(\mathbf{U}_j)$. Then, for each $x \in S$, we set to 0, in $\mathbf{c}(x)$, the $i-1$ first bits set to 1 in $\mathbf{U}$ by computing the vector $\mathbf{rm}(x) = \mathbf{c}(x) \wedge \mathbf{U}_{i-1}$. Now, to detect the element associated to the ith bit set to 1 in $\mathbf{U}$, we only need to detect the value associated to the first bit set to 1 in $\bigvee_{x\in S} \mathbf{rm}_x$, which is done as follows: for any $s \in S$, we compute the vector $\mathbf{out}(s)$ that is full of ones if and only if the position of the first bit set to 1 in $\mathbf{U}_{i-1}$ (that is the union of the vectors $\mathbf{rm}(x)$) is set to 1 in the vector $\mathbf{rm}(s)$. Thus, $\mathbf{out}(s) = \text{Eq}(\text{pref-}\vee(\mathbf{rm}(s)), \text{pref-}\vee(\mathbf{U}_{i-1}))$. ◀

▶ **Lemma 20.** *For any semigroup $S \in \mathbf{DA}$, we can compute JProd_S over any vectorial encoding of a partial evaluation in its domain with a* Sweeping*-vectorial circuit of size $O(|S|^2)$.*

Proof. Let $u = u_0 \cdots u_k$ be a word of S^+. To compute $\text{JProd}_S(u)$, we want to detect the first and last bits set to 1 in $\mathbf{U}$ in order to compute an encoding of the word composed only of the element $\pi_S(u_0u_k)$. The first element is directly indicated by the vectors $\text{Value}_{1,S}(s)$ for each element $s \in S$. Now we detect the last element by computing the similar vectors $\mathbf{Last}(s)$ for each $s \in S$: $\mathbf{Last}(s) = \text{Eq}(\text{suf-}\vee(\mathbf{c}(s)), \text{suf-}\vee(\mathbf{U}))$. With these vectors, we know the value of the product: the product is $s \in S$ if and only if $\text{Value}_{1,S}(t) \wedge \mathbf{Last}(p)$ is equal to $\mathbf{1}$ for some $(t, p) \in S^2$ such that $\mathcal{R}(t) \cap \mathcal{L}(p) = \{s\}$. We set the last bit of the corresponding output vector to 1, and the rest to 0. ◀

▶ **Lemma 21.** *For any semigroup $S \in \mathbf{DA}$, we can compute* LProd_S *and* RProd_S *over any vectorial encoding of a partial evaluation in their domains with* Sweeping-*vectorial circuits of size $O(|S|^2)$.*

Proof. The two operations are symmetrical, so we present only the circuit for LProd_S. Let u be a word of S^+. We can use $\mathrm{Value}_{1,S}$ and $\mathrm{Value}_{2,S}$ to compute vectors that give the values of the first and second elements. If $\forall s \in S$, $\mathrm{Value}_{2,S}(s) = \mathbf{0}$, then $|u| = 1$ and there is nothing to do. Thus, we use a circuit IfThenElse. If $|u| > 1$, we perform the product by computing a vector **PosSec** with a unique 1 at the position of the second letter, which takes only a constant number of gates, then we remove the first two elements of the input vectors and add **PosSec** to the vector corresponding to the product. This last operation takes $O(|S|^2)$ gates since we need to check all the pairs of elements of S to compute the product. ◀

▶ **Lemma 22.** *Let S be a semigroup in $\mathbf{DA}$ of $\mathcal{J}$-depth d, i be an integer such that $1 \leq i \leq d$, let P be a sweeping evaluation program defined at least on all left $\mathcal{J}$-constant words of depth i, and suppose that we have a* Sweeping-*vectorial circuit of size s_P that computes P over any vectorial encoding of a partial evaluation. Then we can compute* $\mathrm{LSplit}_{S,i}\langle P\rangle$ *and* $\mathrm{RSplit}_{S,i}\langle P\rangle$ *over any vectorial encoding of a partial evaluation in their respective domains with* Sweeping-*vectorial circuits of size $O(|S|^2 + s_P)$.*

Proof. The two operations are symmetrical, so we only present the circuit for $\mathrm{LSplit}_{S,i}\langle P\rangle$. Let $u = u_0 \cdots u_k$ be a word of S^+. We want to detect the first element u_i such that $\pi_S(u_0 \cdots u_i)$ is of $\mathcal{J}$-depth at least $i+1$ in order to replace the prefix of $u_0 \cdots u_{i-1}$ by its image through P. To do that, we begin by checking if the first element of the word is of $\mathcal{J}$-depth i by computing the vectors $\mathrm{Value}_{1,S}(s)$ for all $s \in D_i(S)$. The union of those vectors is then used in a circuit IfThenElse: if the union is $\mathbf{0}$, nothing is done. Otherwise, we want to find the first position such that the product of the prefix is of $\mathcal{J}$-depth at least $i+1$. Thanks to Lemma 29, we know that the set of elements that make that product fall in a $\mathcal{J}$-class of greater $\mathcal{J}$-depth depends only on the $\mathcal{R}$-class of the prefix, which is uniquely determined by the first element, since that element is necessarily of $\mathcal{J}$-depth i. Using the vectors $\mathrm{Value}_{1,S}(s)$ we computed, we can determine the $\mathcal{R}$-class of the prefix. Depending on this $\mathcal{R}$-class, we search for the first letter of the word that belongs to the set K defined by Lemma 29, using calls to pref-$\vee$, and we mark all letters before its position: these letters are exactly the prefix we need to consider. Computing all these masks for each $\mathcal{R}$-class takes $O(|S|)$ gates. Then, we mask the input vectors and use the results as inputs for the circuit C_P. Finally, we reassemble the results with the suffixes that were not considered in C_P to get an encoding of $\mathrm{LSplit}_{S,i}\langle P\rangle(u)$. ◀