Beyond Generalized Multiplicities: Register Machines over Groups

Artiom Alhazov[1], Rudolf Freund[2], Sergiu Ivanov[3]

[1] Vladimir Andrunachievici Institute of
Mathematics and Computer Science
Academiei 5, Chişinău, MD-2028, Moldova
E-mail: artiom@math.md

[2] TU Wien, Institut für Logic and Computation
Favoritenstraße 9–11, 1040 Wien, Austria
E-mail: rudi@emcc.at

[3] IBISC, Université Évry, Université Paris-Saclay
23, boulevard de France, 91034 Évry, France
E-mail: sergiu.ivanov@univ-evry.fr

Summary. Register machines are a classic model of computing, often seen as a canonical example of a device manipulating natural numbers. In this paper, we define register machines operating on general groups instead. This generalization follows the research direction started in multiple previous works. We study the expressive power of register machines as a function of the underlying groups, as well as of allowed ingredients (zero test, partial blindness, forbidden regions). We put forward a fundamental connection between register machines and vector addition systems. Finally, we show how registers over free groups can be used to store and manipulate strings.

1 Introduction

Register machines are traditionally seen as a model of computing manipulating non-negative numbers. However, quite some time ago integer numbers were already considered as the base set for register contents [8]. Such machines are traditionally called blind as long as they do not allow testing registers for zero, except eventually testing all registers for zero at the end. The computational power of such blind register machines is inferior to that of "conventional" register machines over natural numbers [2]. If the register machine is not allowed to go below zero, but can neither explicitly test its registers for zero, it is called partially blind.

Even further, we need not restrict the definition of the model to numbers. For example, Section 3 of [2] gives a very general definition of register machines whose registers may contain elements of any set A. However, going this far up the abstraction scale loses too much structure: almost nothing can be said about such general

constructs. In this paper, we focus on a level of abstraction which is in between the two: we consider register machines over finitely presented groups. This generalization comes in as a natural sequel to multiple previous works. For example, [9] introduced integer vector addition systems by lifting the traditional restriction on the vectors to only contain non-negative components. Subsequently, [7] generalized P systems (compartmentalized multiset rewriting systems [18]) to allow multiplicities of objects to come from Abelian groups instead of just natural numbers. Finally, papers [2, 3] come back on the less general case of integer multiplicities and show a number of new properties of integer vector addition systems and blind register machines.

As almost any work on register machines, studies on register machines over groups have multiple interesting consequences for P systems. The present paper lays the ground for further exploration of P systems with generalized multisets and raises a number of important questions, for example, about the ways in which multiplicities from non-commutative groups can be interpreted. As we will show later, registers containing elements of the free group can be used to emulate strings; what would be the meaning of string multiplicities in P systems?

In this work, we define register machines over arbitrary finite families of groups, with or without zero test, as well as partially blind register machines, and register machines with forbidden regions (Section 3). Each of these ingredients is meant to generalize individual features which appear in the classic definition of register machines. We then study the computational power of the variants we define: we compare the generating and the accepting modes, single- and multi-register machines, vector addition systems with and without states (Section 4).

As it often happens, all of the results we present in this paper originate from the fruitful discussions the team of authors had during the Brainstorming Week on Membrane Computing 2019 in Seville, Spain. Even though this work is not explicitly situated within the domain of membrane computing, we believe that it may have an important influence on the study of generalized multiplicities in P systems. We would therefore like to thank the organizing team for giving us the opportunity to work on these exciting topics.

2 Preliminaries

In this paper, we use the symbols $\mathbb{R}$, $\mathbb{Z}$, and $\mathbb{N}$ to refer to the set of real numbers, integer numbers, and the set of natural numbers including 0.

For an alphabet V, by V^* we denote the free monoid generated from the elements of V under the operation of concatenation, i.e., containing all possible strings over V. The *empty string* is denoted by λ. The family of all recursively enumerable sets of strings is denoted by RE, the corresponding family of recursively enumerable sets of Parikh sets (vectors of natural numbers) and of number sets is denoted by $PsRE$ and NRE, respectively. For an extensive introduction to the theory of formal languages, we recommend [18, 19].

Given a set A, a total function $f : A \times A \to A$ is called a *binary operation* over A. We will use the infix notation afb to refer to $f(a, b)$, for $a, b \in A$. A relation over a set A is any subset $R \subseteq A \times A$. As for binary operations, we will also use the infix notation aRb for $(a, b) \in R$.

A relation $\leq \subseteq A \times A$ is a called a *total order* if the following statements hold for every three elements $a, b, c \in A$:

- *antisymmetry*: if $a \leq b$ and $b \leq a$ then $a = b$,
- *transitivity*: if $a \leq b$ and $b \leq c$ then $a \leq c$,
- *totality*: either $a \leq b$ or $b \leq a$.

For $a, b \in A$ and a total order $\leq$ on A, we will sometimes write $b \geq a$ as equivalent to $a \leq b$, and use $a < b$ ($a > b$) to denote that $a \leq b$ and $a \neq b$ ($a \geq b$ and $a \neq b$).

2.1 Groups and Group Presentations

Groups

A group is the structure $G = (G', \circ)$ where G' is the set of elements (the underlying set) and $\circ : G' \times G' \to G'$ a binary operation over G' satisfying the following properties (*group axioms*):

- *closure*: for any $a, b \in G'$, $a \circ b \in G'$,
- *associativity*: for any $a, b, c \in G'$, $(a \circ b) \circ c = a \circ (b \circ c)$,
- *identity*: there exists a (unique) element $e \in G'$, called the *identity*, such that $e \circ a = a \circ e = a$ for all $a \in G'$, and
- *invertibility:* for any $a \in G'$, there exists a (unique) element a^{-1}, called the *inverse* of a, such that $a \circ a^{-1} = a^{-1} \circ a = e$.

The group G is called *commutative* or *Abelian*, if for any $a, b \in G'$, $a \circ b = b \circ a$.

A *subgroup* of the group $(G, \circ)$ is any group $(H, \circ)$ with $H \subseteq G$ and the same group operation $\circ$.

For any element $b \in G'$, the order of b is the smallest number $n \in \mathbb{N}$ such that $b^n = e$ provided such n exists, and then we write $ord\,(b) = n$. If no such n exists, $\{b^n \mid n \geq 1\}$ is an infinite subset of G' and we write $ord\,(b) = \infty$.

In the following, we will often use the same symbol G to refer both to a group and to its underlying set.

Representations of groups

The definitions and examples from group theory we exhibit now follow the exposition given in [1] and [2], based on the notions in [10]. In what follows, we will use strings for representing group elements.

For any set B, the set B^{-1} is defined to contain the symbols representing the "inverses" of the elements of B, i.e., $B^{-1} = \left\{b^{-1} \mid b \in B\right\}$. B (not containing the identity) is called a *generator set* of the group G if every element a from G can be

written as a finite product/sum of elements from $B \cup B^{-1}$, i.e., $a = b_1 \circ \cdots \circ b_m$ for $b_1, \ldots, b_m \in B \cup B^{-1}$. In this paper, we restrict ourselves to finitely presented groups, i.e., having a finite presentation $\langle B \mid R \rangle$ with B being a finite generator set and moreover, R being a finite set of relations among these generators. Informally, the group $G = \langle B \mid R \rangle$ is the largest one generated by B subject only to the group axioms and the relations in R. We will restrict ourselves to relations of the form $b_1 \circ \cdots \circ b_m = e$ with $b_1, \ldots, b_m \in B$; omitting the identity e we write $b_1 \circ \cdots \circ b_m$, which then is called *relator*.

Example 1. The free group $F(B) = (I(B), \circ)$ can be written as $\langle B \mid \emptyset \rangle$ (or even simpler as $\langle B \rangle$) because it has no restricting relations.

Example 2. The *cyclic group* of order n has the presentation $\langle \{a\} \mid \{a^n\} \rangle$ (or, omitting the set brackets, as $\langle a \mid a^n \rangle$). It is also known as $\mathbb{Z}_n$ or as the quotient group $\mathbb{Z}/n\mathbb{Z}$.

Example 3. $\mathbb{Z}$ is a special case of an Abelian group generated by 1 and its inverse -1, i.e., $\mathbb{Z}$ is the free group generated by $B = \{1\}$. $\mathbb{Z}^d$ is the Abelian group generated by the unit vectors $(0, \ldots, 1, \ldots, 0)$ and their inverses $(0, \ldots, -1, \ldots, 0)$. It is well known that every finitely generated Abelian group is a direct sum of a torsion group and a free Abelian group, where the torsion group may be written as a direct sum of finitely many quotient groups of the form $\mathbb{Z}/p^k\mathbb{Z}$, with p a prime and $k \in \mathbb{N}$, and the free Abelian group is a direct sum of finitely many copies of $\mathbb{Z}$.

Example 4. A very well-known example of a non-Abelian group is the hexagonal group with the finite presentation $\langle a, b, c \mid a^2, b^2, c^2, (abc)^2 \rangle$. The relators a^2, b^2, and c^2 indicate that all three generators a, b, and c are self-inverse.

Remark 1. In this paper, we will restrict ourselves to finitely generated groups, for which the word equivalence problem $u = v$ is decidable, i.e., there exists a decision procedure telling us whether $u \circ v^{-1} = e$ for two strings u and v. In this case, we call G *recursive* or *computable*. If the set of relators R in a presentation $\langle B \mid R \rangle$ of G is computable (recursive), we call this a computable (recursive) presentation. Clearly, any finitely presented group is computable.

A group $(G, +)$ in which the group operation can be interpreted as addition is called additive. For such groups, the inverse of $b \in G$ is often written as $-b$, the neutral element e as 0, and the sum $a + (-b)$ as $a - b$, whenever no ambiguity arises. Another kind of groups are multiplicative groups, in which the group operation can be thought of as multiplication. For such groups, the inverse of $b \in G$ is usually written as b^{-1}, and the group operation as multiplication: $a \cdot b$ or ab.

For Abelian groups, further shortcut notation is introduced to capture chained applications of the operation to a single element. Consider $z \in \mathbb{Z}$ and $a \in G$. The *scalar product* of a by z is defined as follows (using either additive or multiplicative notation):

$$za = \begin{cases} a^z = \sum_{i=1}^{z} a, & z > 0, \\ a^0 = 0 \text{ (group identity)}, & z = 0, \\ (-a)^{-z} = \sum_{i=1}^{z} (-a), & z < 0. \end{cases}$$

A *linearly* or *totally ordered group* is construct $(A, +, \leq)$ where $(A, +)$ is a group, $\leq \subseteq A \times A$ is a total order on A and, for any triple $a, b, c \in A$, the fact that $a \leq b$ implies that $c + a \leq c + b$ and $a + c \leq b + c$.

2.2 Register Machines

Register machines are well-known universal devices for computing (generating or accepting) sets of vectors of natural numbers. The article [13] is one of the reference works on the universality of register machines.

Definition 1. *A* register machine *is the construct* $M = (m, B, l_0, l_h, P)$*, where*

- m *is the number of registers,*
- B *is a set of labels bijectively labeling the instructions in the set* P*,*
- $l_0 \in B$ *is the initial label,*
- $l_h \in B$ *is the final label, and*
- P *is the set of instructions.*

The labeled instructions in P *can be of the following forms:*

- $p : (ADD(r), q, s)$*, with* $p \in B \setminus \{l_h\}$*,* $q, s \in B$*,* $1 \leq r \leq m$*.*
  Increment the value of register r *and non-deterministically jump to instruction* q *or* s*.*
- $p : (SUB(r), q, s)$*, with* $p \in B \setminus \{l_h\}$*,* $q, s \in B$*,* $1 \leq r \leq m$*.*
  If the value of register r *is not zero then decrement the value of register* r *(*decrement *case) and jump to instruction* q*, otherwise jump to instruction* s *(*zero-test *case).*
- $l_h : HALT$*.*
  Stop the execution of the register machine.

A *configuration* of a register machine is the tuple $C = (q, r_1, \ldots, r_m)$, in which $r_1, \ldots, r_m$ are the values of the registers and q is the current instruction label which indicates which is the *next* instruction to execute. This label is often called the *state* of the machine. An n-step computation of a register machine is a sequence of configurations $(C_i)_{0 \leq i \leq n}$ in which the configuration C_{i+1} is obtained from C_i by applying to C_i the instruction given by the current instruction label of C_i. The first configuration C_0 of a computation is usually referred to as the *initial* configuration and its current instruction label must be l_0. If, in the last configuration C_n, the current instruction label is be l_h, C_n is called a *halting* configuration and the whole computation is called a halting computation.

A register machine M can be seen as an accepting device, a generating device, or as a device computing functions or relations. In the *accepting* case, the first k registers of M are designated as input registers, and are initialized to a k-vector of natural numbers v (the input vector). If there exists a halting computation of M starting with this initial configuration, then v is accepted by M. Without losing generality, we may only consider computations in which all registers are empty in the halting configuration.

On the other hand, in the *generating* case, a single initial configuration is fixed for all computations of M, the first k registers are designated as the output registers, and for every halting computation of M, the k-vector contained in the output registers in the halting configuration is said to be generated by M. Without losing generality, we may only consider those computations of M in which all registers with indices greater than k are empty in the halting configuration.

Finally, we can designate input *and* output registers (these may be disjoint) and see M as establishing a binary *relation* between the contents of the input registers in the initial configurations and the output registers in the halting configurations. If this relation is functional, i.e., M associates at most one output vector to any input vector, M can be seen as defining a *function*.

In this paper, we will only consider the accepting and the generating cases. We will denote by $\mathcal{L}_{acc}(M)$ (respectively, by $\mathcal{L}_{gen}(M)$) the set of input vectors accepted (respectively, generated) by the register machine M. Similarly, for a family $\mathcal{X}$ of register machines, we will denote by $\mathcal{L}_{acc}(\mathcal{X})$ (respectively, by $\mathcal{L}_{gen}(\mathcal{X})$) the family of sets of vectors accepted (respectively, generated) by the register machines in the family $\mathcal{X}$. We will use the same notations to denote the sets of languages accepted (respectively, generated) by any other computing device M or any other family of computing devices $\mathcal{X}$. In case the operating mode is fixed by the definition of the device (e.g., vector addition systems always generate), we omit the corresponding subscript.

We use the notation RM to refer to the family of register machines defined as above. It is folklore (e.g., see [16]) that $\mathcal{L}_{acc}(RM) = PsRE$. Similarly, register machines generate any recursively enumerable set of vectors of natural vectors, $\mathcal{L}_{gen}(RM) = PsRE$. A proof sketch: consider $L \in PsRE$, then build the machine M such that it first non-deterministically generates a vector, and then runs a sequence of instructions recognizing precisely the vectors in L.

Blind and Partially Blind Machines

Several papers consider weaker kinds of register machines: blind and partially blind register machines, for example, see [2, 6, 8].

In *partially blind* register machines, the SUB instruction has the form $p : (SUB(r), q)$: if the register r is not empty, it is decremented and the register machine moves to state q, otherwise the machine crashes—the computation stops in a non-halting configuration, yielding no result. In *blind* register machines, the regis-

ters are allowed to contain negative values, meaning that the decrement instruction always succeeds. However, valid computations of a blind machine are required to have 0 in all non-output registers in halting configurations. The definitions of blind and partially blind machines may vary from source to source: notably some sources define blind register machines as partially blind, but without the zero check at the end [6].

In this paper, we will give uniform definitions of various types of register machines.

A General Model for Register Machines

For the record, we recall here a very general definition of a register-machine-like device given in [3].

Definition 2. *A* register-machine-like device over the set A *is the tuple* $M_A = (m, A, B, l_0, l_h, P)$, *where*

- $m \in \mathbb{N}$ *is the number of registers,*
- A *is the set of values the registers may contain,*
- B *is a finite set of instruction labels,*
- l_0 *is the initial label,*
- l_h *is the final label,*
- P *is a mapping associating an instruction to every label in* B.

An instruction p *is a function* $p : A^m \to A^m \times 2^Q$ *associating to every* m*-tuple of values from* A *another* m*-tuple of such values and a set of new instruction labels from* B. *A configuration* $C \in B \times A^m$ *of* M_A *is a tuple combining an instruction label and the values of the* m *registers of* M_A.

2.3 Vector Addition Systems (VAS)

A *vector addition system* (VAS) of dimension $n \in \mathbb{N}$ is defined to be the pair $(\mathbf{w}_0, W)$, where $\mathbf{w}_0 \in \mathbb{N}^n$ is the start vector, and W is a finite set of vectors from $\mathbb{Z}^n$, called addition vectors. An addition vector $\mathbf{w} \in W$ is said to be applicable to a vector $\mathbf{x} \in \mathbb{N}^n$ if $\mathbf{x} + \mathbf{w} \in \mathbb{N}^n$, i.e., if all the components of the vector $\mathbf{x} + \mathbf{w}$ are non-negative. A VAS evolves from the start vector $\mathbf{w}_0$ by sequentially adding applicable addition vectors from W.

A *vector addition system with states* (VASS) is a VAS equipped with a finite state control. Essentially, state labels are assigned to addition vectors and a graph of states is given which defines the possible sequences of application of addition vectors.

An extended model lifting the restriction that the valid vectors must have non-negative components has recently been defined in [9] and studied in [3]: An *integer*

vector addition system ($\mathbb{Z}$-VAS) of dimension $n \in \mathbb{N}$ is the pair $(\mathbf{w}_0, W)$, where $\mathbf{w}_0 \in \mathbb{Z}^n$ is the start vector and $W \subseteq \mathbb{Z}^n$ is finite set of addition vectors. A $\mathbb{Z}$-VAS evolves from $\mathbf{w}_0$ by sequentially applying the addition vectors from W. The set of vectors generated by a $\mathbb{Z}$-VAS is defined to be the set of reachable vectors.

An *integer vector addition system with states* ($\mathbb{Z}$-VASS) is a $\mathbb{Z}$-VAS equipped with a state control and is defined as a tuple $(\mathbf{w}_0, Q, q_0, q_h, p, \delta)$, where $\mathbf{w}_0 \in \mathbb{Z}^n$ is the start vector, Q is a finite set of state labels, $q_0 \in Q$ is the starting state, $q_h \in Q$ is the halting state, $p : Q \setminus \{q_h\} \to \mathbb{Z}^n$ is a function assigning a vector to every state from $Q \setminus \{q_h\}$, and $\delta : Q \to 2^Q$ is a state transition function assigning to each state the set of possible successor states. A $\mathbb{Z}$-VASS starts in $\mathbf{w}_0$ and in state q_0, applies the addition vector $p(q_0)$, and non-deterministically moves into one of the states from $\delta(q_0)$. This process is iteratively repeated, until the halting state q_h is reached. The vector language generated by a $\mathbb{Z}$-VASS is defined as the set of all vectors which are reachable in the halting state q_h.

It was shown in [11] that VASS are equivalent in expressive power to VAS (without states): any n-dimensional VASS can be simulated by an $(n+3)$-dimensional VAS. On the other hand, in [3, Section 6], it is proved that $\mathbb{Z}$-VASS are strictly more powerful than $\mathbb{Z}$-VAS. This is one first example showing that changing the nature of the objects on which a model of computing operates can affect its expressive power in important ways.

3 Register Machines over Groups

3.1 General Definition

In this section we extend the definition of register machines to allow their registers to contain elements of arbitrary finitely presented groups.

Definition 3. *Let $m \in \mathbb{N}$ and take the finite family of finitely presented groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. A* $\mathcal{G}$-register machine *(or a register machine over the family $\mathcal{G}$) is the construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$, where*

- *B is the set of labels bijectively labeling the instructions in the set P,*
- *$l_0 \in B$ is the initial label,*
- *$l_h \in B$ is the final label, and*
- *P is the set of instructions.*

The labeled instructions in P can be of the following forms:

- *$p : (ADD(r, b), T)$, with $p \in B \setminus \{l_h\}$, $T \subseteq B$, $1 \leq r \leq m$, $b \in B_r$.*
  Add the generator b of the group $G_r = \langle B_r \mid R_r \rangle$ to the current contents of the register r, then non-deterministically jump to one of the instructions in T.

- $l_h : HALT$.
  Stop the execution of the register machine.

A configuration of a $\mathcal{G}$-register machine is, like in the case of a classical register machine, the tuple $C = (q, r_1, \ldots, r_m)$, in which $r_i \in G_i$, $1 \leq i \leq m$, are the values of the registers, and q is the current instruction label which indicates the next instruction to execute. This label is called the state of the machine. We define the computations, halting, generating, and accepting for register machines over the family $\mathcal{G}$ in the same way as for conventional register machines in Section 2.2. In particular, $k \leq m$ registers are designated as input registers in the accepting case, (respectively, as output registers in the generating case), meaning that the $\mathcal{G}$-register machine accepts (respectively, generates) vectors of the form $(g_1, \ldots, g_k)$, where g_j, $1 \leq j \leq k$, belongs to a group $G_i \in \mathcal{G}$, $1 \leq i \leq m$, where different indices i are assigned to different indices j.

Remark 2. Note that the ADD instructions as we define them here allow a non-deterministic choice between more than two target states, as different from the classical definition, in which only two target states are allowed. We allow a set of possible target states because it simplifies the formulations of many properties and results, without critically affecting the power of the model: indeed, multiple target states can be easily simulated by a chain of dummy branching instructions.

Example 5. Consider the following family of 3 groups $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$, where $\mathbb{Z}$ is the usual Abelian group of integer numbers which can be presented in this way: $\mathbb{Z} = \langle 1 \mid a{+}b{+}(-a){+}(-b) \rangle$. A $\mathcal{Z}^3$-register machine $M_{\mathbb{Z}^3}$ is *almost* a blind 3-register machine: indeed, the three registers of $M_{\mathbb{Z}^3}$ may contain any integer number, an increment of a register r, $r \in \{1, 2, 3\}$, is done by the operation $ADD(r, 1)$, and a decrement by the operation $ADD(r, -1)$. Nevertheless, $M_{\mathbb{Z}^3}$ is *not* a blind register machine, because no zero check is performed at the end of a computation: the only restriction on the halting configuration is to have l_h as the current instruction label.

Given a finite family of finitely generated computable groups $\mathcal{G}$, we will use the notation $\mathcal{G}$-RM to refer to the family of $\mathcal{G}$-register machines. We will sometimes also use the notation $*$-$RM = \bigcup_{\mathcal{G}} \mathcal{G}$-$RM$.

Remark 3. We observe that, in contrast to the original definition of register machines, the definition of $\mathcal{G}$-register machines only introduces increment instructions $p : (ADD(r, b), T)$ and no decrement instructions $p : (SUB(r, b), q, s)$, as decrementing by an element $e \in B_i$ corresponds to incrementing by $-e$. On the other hand, there is no direct check for zero in these ADD-instructions.

Remark 4. Register machines over groups as we define them here are somewhat similar to previous works on automata operating on groups (e.g. [17]). However, in our work, we rather focus on generalizing the ingredients forming register machines and describing them in a general setting, instead of analyzing their power as language recognizers.

Remark 5. We could extend the classical model of register machines to operate on other algebraic structures than groups. In this paper, we choose to focus on groups because these objects are rather well studied and there have already been previous works on using groups as a substrate for computation (e.g., [5]).

Vector Addition Systems over Groups

Even though register machines and vector addition systems are traditionally seen as quite different models and research on one often does not discuss the other (see, for example, the classic works [13] and [4]), the connection between the two is clearly rather strong, especially when considered in a more general setting. For example, $\mathbb{Z}$-VASS are equivalent in power to blind register machines [3]. In the present paper, we explicitly enforce this connection by defining vector addition systems over groups in terms of register machines over groups.

Definition 4. *Consider a finitely generated computable group $(G, \circ)$. A* vector addition system with states over G *(a G-VASS) is a tuple (g_0, M), where $g_0 \in G$ is the start element and M is a (G)-register machine (i.e., a machine with a single register over the group G) working in generating mode and whose only register is initialized with g_0.*

Definition 5. *Consider a finitely generated computable group $(G, \circ)$. A* vector addition system over G *(a G-VAS) is a G-VASS with the following structure on the instructions of the underlying register machine:*

- *$l_0 : (ADD(1, 0), B) \in P$: the initial instruction does not modify the contents of the register, but allows non-deterministically jumping to any other instruction, including the halting instruction.*
- *all instructions labelled by $l \in B \backslash \{l_0, l_h\}$ have the form $l : (ADD(0, g), B \backslash \{l_0\})$, with $g \in G$: the underlying machine can jump from any non-initial instruction to any other non-initial instruction, including the halting instruction.*

Example 6. Integer vector addition systems (with states), as introduced in [9] and studied in [3], are vector addition systems (with states) over the product group $\mathbb{Z}^n = \mathbb{Z} \times \cdots \times \mathbb{Z}$. Indeed, the elements of $\mathbb{Z}^n$ are n-vectors of integer numbers, and the state control of the $(\mathbb{Z}^n)$-register machine corresponds to the state control of the integer VASS. On the other hand, since the register machine associated with a VAS over $\mathbb{Z}^n$ can halt at any time, any vector it reaches belongs to the generated language.

Given a finitely generated computable group G, we will use the notations $G\text{-}VASS$ and $G\text{-}VAS$ to refer to the families of G-VASS and G-VAS, respectively. Since vector addition systems are only considered as generating devices, we will use the notations $\mathcal{L}(G\text{-}VASS)$ and $\mathcal{L}(G\text{-}VAS)$ to refer to the families of sets of elements of G generated by G-VASS and G-VAS, respectively.

3.2 Blindness, Partial Blindness, and the Zero Test

A $\mathcal{G}$-register machine as defined in the previous section is quite "blind": it has no mechanism to make the choice of the new instruction depend on the values of the registers. A classical way to introduce such a dependence is by allowing an explicit zero-test instruction.

Definition 6. *Let $m \in \mathbb{N}$ and take the finite family of finitely generated computable groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. A* $\mathcal{G}$-register machine with zero test *is the construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ such that $M_{\mathcal{G}}$ is a $\mathcal{G}$-register machine, and the set P is also allowed to contain instructions of the following form:*

- $p : (0TEST(r), s, z)$, *with $p \in B \backslash \{l_h\}$, $s, z \in B$, $1 \leq r \leq m$.*
  Test if the current value of register r is equal to the neutral element of the group G_r; if yes, jump to instruction z, if not, jump to instruction s.

Configurations, computations, halting, generating, and accepting for $\mathcal{G}$-register machines with zero test are defined as for $\mathcal{G}$-register machines in Section 3.

Example 7. Consider the same family of 3 copies of the group of integers as in Example 5, $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$. A $\mathcal{Z}^3$-register machine with zero test is *almost* like a conventional register machine: increment and decrement are done by the instructions $ADD(r, 1)$ and $ADD(r, -1)$, and zero test by the $TEST(r)$ instructions. However, the registers of a $\mathcal{Z}^3$-register machine with zero test are allowed to contain arbitrary integers.

For a finite family of finitely generated computable groups $\mathcal{G}$, we will use the notation $\mathcal{G}$-RM_0 to refer to the family of $\mathcal{G}$-register machines with zero test.

Allowing an explicit zero test instruction is known to strictly increase the computational power of register machines (e.g., [8]). A much weaker way of introducing a dependency between the contents of the registers and the choice of instructions is the terminal zero test.

Definition 7. *Let $m \in \mathbb{N}$ and take the finite family of finitely generated computable groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. A* $\mathcal{G}$-register machine with a terminal zero test *(a* blind $\mathcal{G}$-register machine*) is a construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ such that $M_{\mathcal{G}}$ is a $\mathcal{G}$-register machine and, in any halting configuration, all registers which have not been explicitly designated as output must contain the neutral element of the corresponding group.*

Computations, halting, generating, and accepting for $\mathcal{G}$-register machines with a terminal zero test are defined as for $\mathcal{G}$-register machines in Section 3, with the additional requirement of emptiness of the working registers, as indicated in the previous definition.

We use the notation $\mathcal{G}$-BRM to refer to the family of $\mathcal{G}$-register machines with a terminal zero test.

Remark 6. Using the notation *BRM* refers to the original definition of blind register machines which we take over for the general case of $\mathcal{G}$-register machines.

Example 8. Consider the family of groups $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$. A $\mathcal{Z}^3$-register machine with a terminal zero test is a blind register machine, as usually defined in the literature (e.g., [3, 8]).

Remark 7. Sheila Greibach's original paper introducing blind and partially blind register machines [8] considers them as recognizers of *strings*: these devices read the string from the beginning to the end, using registers to store internal information. Later works (e.g., [13]) tend to discard the string recognizer aspect, and instead treat register machines as devices manipulating numbers exclusively. In general, it is quite easy to encode any string as a number (using, for example, a prime number encoding over the alphabet), therefore restricting registers machines to numbers does not critically affect their expressiveness. In Section 3.4, we show how to recover string-related behavior in register machines over groups with forbidden regions.

3.3 Partial Blindness

The types of register machines over groups we have defined up to now do not directly generalize the classical register machines, which can be seen as defined over the monoid of natural numbers $(\mathbb{N}, +)$: addition over the natural numbers is not invertible, because negative numbers do not belong to $\mathbb{N}$. To capture this restriction, we directly draw inspiration from the definitions of VAS and conventional partially blind register machines, and define partially blind register machines over totally ordered groups.

Definition 8. *Let $m \in \mathbb{N}$ and take the finite family of finitely generated computable groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Suppose one of the groups G_j, $1 \leq j \leq m$, is totally ordered with the total order $\leq_j$. A $\mathcal{G}$-*register machine with a partially blind register j *is a construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ such that $M_{\mathcal{G}}$ is a $\mathcal{G}$-register machine whose register j is only allowed to contain values $a_j \in G_j$ with the property $0_j \leq_j a_j$.*

Configurations, computations, halting, generating, and accepting for $\mathcal{G}$-register machines with some partially blind registers are defined as for $\mathcal{G}$-register machines in Section 3, with the additional restriction on the values of the partially blind registers.

Example 9. Consider the family of groups $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$. The group $(\mathbb{Z}, +)$ is totally ordered with respect to the natural order. A $\mathcal{Z}^3$-register machine partially blind in all of its 3 registers is a 3-register partially blind register machine in the conventional sense, but without the final zero test.

For a finite family of finitely generated computable groups $\mathcal{G}$, we use the notation $\mathcal{G}$-PB_ARM, with $A \subseteq \{1, \ldots, m\}$, to refer to the family of $\mathcal{G}$-register machines with partially blind registers with indices from A. This supposes that the groups of $\mathcal{G}$ with indices from A are totally ordered.

When $A = \{1, \ldots, m\}$, i.e., all the registers are partially blind, we will omit the subscript A from the notations, and we will refer to the register machine itself as being *partially blind*. We use the particular notation $\mathcal{G}$-$PBRM$ to refer to the family of register machines with all registers blind, and with the final zero test at the end of successful computations.

Note finally that $\mathcal{G}$-register machines with all registers blind and with the zero test instruction directly generalize classic register machines.

3.4 Forbidden Regions

While $\mathcal{G}$-register machines with partially blind registers are a generalization which is rather close to conventional register machines, imposing a total order on a group is a rather strong condition: for example, it entails the absence of elements of a finite order [14], thus excluding cyclic groups from consideration. Notice, however, that the total order is essentially used to define a forbidden subset of elements. We can therefore define another generalization of conventional register machines which imposes less constraints on the group.

Definition 9. *Let $m \in \mathbb{N}$ and take the finite family of finitely generated computable groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Let $\mathcal{F} = (F_i)_{1 \leq i \leq m}$ be a family of subsets of the groups in $\mathcal{G}$: $F_i \subseteq G_i$, $1 \leq i \leq m$. A $\mathcal{G}$-*register machine with forbidden regions *$\mathcal{F}$ is a construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ such that $M_{\mathcal{G}}$ is a $\mathcal{G}$-register machine whose register i is only allowed to contain the elements in $G_i \setminus F_i$, $1 \leq i \leq m$.*

Configurations, computations, halting, generation, and acceptance for $\mathcal{G}$-register machines with forbidden regions are defined as for $\mathcal{G}$-register machines in Section 3, with the additional restriction on the values of registers: if a forbidden value appears in a register, the computation crashes without producing any output.

Example 10. Consider the family of groups $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$ and the family $\bar{\mathcal{N}}^3 = (\bar{N}, \bar{N}, \bar{N})$, where $\bar{N} = \mathbb{Z} \setminus \mathbb{N}$. A $\mathcal{Z}^3$-register machine with the forbidden regions $\mathcal{N}^3$ is also a $\mathcal{Z}^3$-register machine partially blind in all of its registers.

Since groups suitably abstract a large number of objects and since forbidden regions can be used to carve particular "shapes" out of a given group, multiple connections with different domains can be traced for register machines over groups and with forbidden regions.

Example 11. The dihedral group D_n is the group of symmetries of a regular polygon with n sides and can be presented as follows: $D_n = \langle r, s \mid r^n, s^2, (sr)^2 \rangle$ [20].

The infinite dihedral group D_∞ can be seen as the group of symmetries of integers and can be presented as $D_\infty = \langle r, s \mid s^2, (sr)^2 \rangle$. The Cayley graph of this presentation can be depicted as follows [5]:

$$\begin{array}{ccccccccccccc}
\ldots & sr^2 & \xleftarrow{r} & sr & \xleftarrow{r} & s & \xleftarrow{r} & sr^{-1} & \xleftarrow{r} & sr^{-2} & \ldots \\
 & s \downarrow\uparrow s & & s \downarrow\uparrow s & & s \downarrow\uparrow s & & s \downarrow\uparrow s & & s \downarrow\uparrow s & \\
\ldots & r^{-2} & \xrightarrow{r} & r^{-1} & \xrightarrow{r} & e & \xrightarrow{r} & r & \xrightarrow{r} & r^2 & \ldots
\end{array}$$

In this picture, the lower and the upper lines are going into opposite directions, which nicely fits as a representation of double-stranded DNA molecules, i.e., the lower line going from the left 5′-end to the right 3′-end, whereas the complementary upper line goes from the right 5′-end to the left 3′-end [5]. Thus, if the family $\mathcal{G}$ contains D_∞, a $\mathcal{G}$-register machine can be seen as operating on a DNA molecule. Forbidding the region $F = \{r^k \mid k \in \mathbb{Z}\} \subset D_\infty$ can be seen as restricting the register machine to operate on one of the strands of the molecule (the upper one on the figure).

Example 12. Take a finite alphabet of symbols V and consider the free group $\langle V \mid \emptyset \rangle = (I(V), \circ)$ over V. It follows from the definition of the free group that it contains two types of elements:

- strings from the syntactic monoid V^*: $a_1 \circ \cdots \circ a_n$, such that $a_1 \ldots a_n \in V^*$;
- strings which include the inverses $\{a^{-1} \mid a \in V\}$ of the elements of V.

Take now the singleton family of groups $\mathcal{G} = (\langle V \mid \emptyset \rangle)$ and the singleton family of forbidden regions $\mathcal{F} = (F)$, with F containing all the elements of G of the second type. Then the only register of a $\mathcal{G}$-register machine M with the forbidden regions $\mathcal{F}$ will contain strings from V^* in any successful computation.

Remark 8. On a historical side-note, register machines as originally introduced by Minsky in [15] came out as a consequence of reducing the tape alphabet of Turing machines to two symbols, including the empty symbol. Such a reduction imposes unary encoding of the working values and essentially transforms the tape into a series of registers [12]. By generalizing register machines from natural numbers to groups, we come back to computing devices operating on strings.

For a finite family of finitely generated computable groups $\mathcal{G}$, we will use the notation $\mathcal{G}$-$RM_{\neg\mathcal{F}}$ to refer to the family of $\mathcal{G}$-register machines with the forbidden regions $\mathcal{F}$.

Vector Addition Systems with Forbidden Regions

Since we define vector addition systems over groups as particular cases of register machines, the idea of forbidden regions can be easily transported to VAS.

Definition 10. *Consider a finitely generated computable group $(G, \circ)$ and a subset $F \subseteq G$. A* vector addition system over G (respectively, with states) with the forbidden region F *is a vector addition system (respectively, with states) whose underlying (G)-register machine belongs to (G)-$RM_{(F)}$.*

Example 13. A $\mathbb{Z}^n$-VAS with the forbidden region $F = \{(x_1, \ldots, x_n) \mid \exists i : x_i < 0\}$ is an n-component vector addition system as classically defined.

Given a finitely generated computable group G, we will use the notation G-$VASS_{\neg F}$ (respectively, G-$VAS_{\neg F}$) to refer to the family of G-VASS (respectively, G-VAS) with the forbidden region F.

4 Expressive Power of RM and VAS over Groups

In this section we will a give a series of results characterizing the power of register machines over groups with or without ingredients. We start by considering the simplest case: no ingredients and singleton group families.

4.1 Singleton Group Families

For register machines with no ingredients, there is little difference between considering non-singleton and singleton group families. In this section, we will use the notation $\mathcal{C} = (C_k)_{0 \leq k \leq n}$ to refer to an n-step computation of a $\mathcal{G}$-register machine, where C_k is a vector of elements of the groups in $\mathcal{G}$ collecting the contents of the registers at step k. Notice that this definition of computation and configurations discards the instruction label, as opposed to the more general definition given in Section 3.

Proposition 1. *Consider a $\mathcal{G}$-register machine M over a non-singleton family $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, $m > 1$. Then there exists a singleton family $\mathcal{G}^0 = (G)$, a family of projections $\pi = (p_i : G \to G_i)_{1 \leq i \leq m}$, and a $\mathcal{G}^0$-register machine M^0 such that, for any n-step computation $\mathcal{C}$ of M there exists an n-step computation $\mathcal{C}^0$ of M^0 with the following property:*

$$C_k[j] = p_j(C_k^0),\ 1 \leq j \leq m,$$

where $C_k^0 \in \mathcal{C}^0$, $C_k \in \mathcal{C}$, and $C_k[j]$ is the j-th element of the vector C_k.

Proof. It suffices to take the group G to be the direct product [10] of the groups in $\mathcal{G}$: $G = \prod_{i=1}^{m} G_i$. The $\mathcal{G}^0$-register machine M^0 will thus have a single register containing vectors of values of the groups in $\mathcal{G}$. Any $ADD(j, b)$ instruction of M will be represented in M^0 by an instruction $ADD(1, \mathbf{b})$, where $\mathbf{b} = (e_1, \ldots, e_{j-1}, b, e_{j+1}, \ldots, e_m)$ is a vector consisting of the neutral elements of the groups in $\mathcal{G}$, except for the j-th element. □

The converse statement is not true, because any vectors from the direct product of G can appear in the ADD instructions of M^0, thus affecting multiple components of the vector from G at once. However, unsurprisingly, any computing step of M^0 can still be simulated by M in *multiple* steps.

Proposition 2. *Consider the non-singleton family of groups $\mathcal{G} = (G_i)_{1\leq i\leq m}$, $m > 1$, and take the singleton family $\mathcal{G}^0 = (G)$, where G is the direct product of the groups in $\mathcal{G}$ and $\pi = (p_i : G \to G_i)_{1\leq i\leq m}$ is the corresponding family of projections. Then, for any $\mathcal{G}^0$-register machine M^0 there exists a $\mathcal{G}$-register machine M such that, for any n-step computation $\mathcal{C}^0$ of M^0, there exists an n'-step computation $\mathcal{C}$ of M, $n' > n$, with the property:*

$$C_0[j] = p_j(C_0^0) \text{ and } C_{n'}[j] = p_j(C_n^0),\ 1 \leq j \leq m,$$

where C_0 and $C_{n'}$ are the first and the last configurations of the computation $\mathcal{C}$, and C_0^0 and C_n^0 are the first and the last configurations of the computation $\mathcal{C}^0$.

Proof (sketch). M simulates the instruction $p : (ADD(0, \mathbf{b}), T)$ of M^0 by the following sequence of instructions:

$$\begin{aligned} p_0 &: \big(ADD\big(0, p_0(\mathbf{b})\big), \{p_1\}\big), \\ p_j &: \big(ADD\big(j, p_j(\mathbf{b})\big), \{p_{j+1}\}\big), \quad 1 < j < m, \\ p_m &: \big(ADD\big(m, p_m(\mathbf{b})\big), T\big). \end{aligned}$$

This ensures that M simulates M^0 with a constant-time slowdown and proves the statement of the proposition. □

The two previous propositions imply that, in a somewhat counter-intuitive way, blind single-register machines are a little more efficient than multi-register machines, because the former may require less computational steps to achieve a given configuration than the latter. This statement, however, becomes false with the addition of some of the ingredients we considered in the previous sections. Indeed, the zero test in a single-register machine over a direct product of groups requires that all components of the combined register should be zero; it is impossible to individually test the components. Similarly, transposing the total orders on some or all of the groups of the family $\mathcal{G}$ to their direct product is not generally possible. Forbidden regions are, on the other hand, more flexible and can be directly carried over from individual groups to components of the elements of the product.

The conclusion we make from these arguments is that, when no additional ingredients are considered, the power of register machines over groups does not depend on the number of registers.

Theorem 1. *Consider the family of finitely presented computable groups $\mathcal{G} = (G_i)_{1\leq i\leq m}$ and a $\mathcal{G}$-register machine M. Then there exists a $\mathcal{G}^0$-register machine M^0 over the singleton family $\mathcal{G}^0 = (\prod_{i=1}^m G_i)$ such that $\mathcal{L}_X(M) = \mathcal{L}_X(M^0)$, with $X \in \{acc, gen\}$.*

4.2 Generation and Acceptance: No Ingredients

As a consequence of the definition of VASS over groups, Theorem 1 implies that any $\mathcal{G}$-register machine working in the generating mode can be simulated by a VASS over the direct product of the groups in $\mathcal{G}$.

Corollary 1. *Consider the family of finitely presented computable groups $\mathcal{G} = (G_i)_{1\leq i\leq m}$ and a $\mathcal{G}$-register machine M. Then there exists a G-VASS A over the product $G = \prod_{i=1}^{m} G_i$ such that $\mathcal{L}_{gen}(M) = \mathcal{L}(A)$.*

The similar statement for register machines in accepting mode does not hold. In fact, an accepting register machine either accepts or rejects any contents of the input registers.

Proposition 3. *Consider the family of finitely generated computable groups $\mathcal{G}$, a subfamily $\mathcal{G}_{in}$ and a $\mathcal{G}$-register machine M whose input registers correspond exactly to the groups from $\mathcal{G}_{in}$. Then $\mathcal{L}_{acc}(M) \in \left\{\emptyset, \prod_{G\in\mathcal{G}_{in}} G\right\}$.*

Proof. M can accept the empty language by never reaching the halting state. Suppose now that it accepts some input vector $\mathbf{x} \in \prod_{G\in\mathcal{G}_{in}} G$. Since the state transitions of M do not depend on the values of its registers, and since no particular conditions are checked at halting, the sequence of actions applied to accept $\mathbf{x}$ can be applied to accept any other $\mathbf{x}' \in \prod_{G\in\mathcal{G}_{in}} G$, meaning that M will accept all possible vectors in $\prod_{G\in\mathcal{G}_{in}} G$. □

We therefore conclude that generation is at least as powerful as acceptance for $\mathbb{G}$-register machines without any additional ingredients.

Theorem 2. *Consider the family of finitely generated computable groups $\mathcal{G}$. Then $\mathcal{L}_{acc}(\mathcal{G}\text{-}RM) \subseteq \mathcal{L}_{gen}(\mathcal{G}\text{-}RM)$.*

Proof. According to Proposition 3, it suffices to show how to generate the empty language and the language of all vectors over the groups corresponding to the output registers. The empty language can be generated by never reaching the halting state. The language of all vectors can be generated by non-deterministically adding the corresponding generators and their inverses to the output registers. □

The inclusion from the previous theorem is not strict. The following example shows a case in which the generating and accepting power are equal.

Example 14. Consider the singleton group $\mathbf{1}$ containing the single element e and a group family $\mathcal{G}$ containing $\mathbf{1}$. Then the languages accepted by $\mathcal{G}$-register machines with the input register containing elements from $\mathbf{1}$ is equal to the languages generated by $\mathcal{G}$-register machines with the output register containing elements from $\mathbf{1}$. Indeed, the only two possible languages which can be accepted or generated are $\emptyset$ and $\{e\}$. As discussed previously, the first language is accepted/generated by never reaching the halting state, and the second language is accepted/generating by halting immediately.

On the other hand, generating $\mathcal{G}$-register machines are not restricted to generating all possible combinations of values of their output registers, as the following example shows.

Example 15. Consider the generating ($\mathbb{Z}$)-register machine M with two states and whose only non-halting state is associated with the instruction $ADD(1,1)$ adding 1 to the contents of its only register. When the register of M is initialized to 0, M only generates the set of natural numbers $\mathbb{N}$.

The difference in power between the accepting mode and the generating mode puts forward an asymmetry in the definition of the two semantics: in the generating mode, the registers have the "knowledge" about their initial values, whereas in the accepting mode, no information about the register contents whatsoever is available.

4.3 Generation and Acceptance: The Zero Test

In this subsection we exhibit that allowing the zero test instruction equalizes the power of the accepting and generating modes. In line with the usual terminology, we will use the term "increment register i by b" to refer to composing the contents of register i with the generator b, and the term "decrement i by b" to refer to composing the contents of register i with the *inverse* of the generator b.

Lemma 1. *Let $m \in \mathbb{N}$ and take the finite family of finitely presented groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Then there exists another family of finitely presented groups G' such that $\mathcal{L}_{gen}(\mathcal{G}\text{-}RM_0) \subseteq \mathcal{L}_{acc}(\mathcal{G}'\text{-}RM_0)$.*

Proof. Let $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ be a $\mathcal{G}$-register machine with zero test. We consider $M_{\mathcal{G}}$ as a generating device, where $1 \leq j \leq k$ are the output registers. We now construct a register machine with zero test $M_{\mathcal{G}'} = (\mathcal{G}', B', l_0', l_h', P')$ with

$$\mathcal{G}' = (G_1, \ldots G_k, G_1, \ldots G_k, G_{k+1}, \ldots G_{m+k}),$$

i.e., every output register of $M_{\mathcal{G}}$ appears in two copies, and the first copy is designated as an input register of $M_{\mathcal{G}'}$, which now becomes an accepting device with the input registers $1 \leq k \leq m$. Given any input in the input registers, $\mathcal{G}'$ simulates $M_{\mathcal{G}}$ in the registers $k+1, \ldots, m+k$ representing the registers $1, \ldots, m$ using the instructions in P' with each register j in an instruction of P replaced by the corresponding register $k+j$ in the instructions of P'. With $M_{\mathcal{G}}$ reaching l_h, also $M_{\mathcal{G}'}$ reaches l_h. After that, in a final procedure, $\mathcal{G}'$ checks if the contents of register j equals the contents of register $j+k$ for every $1 \leq j \leq k$. In the success case, $\mathcal{G}'$ enters the final label l_h'.

For $j = 1, \ldots, k$, starting with p_1, sequences of instructions
$p_j : (0TEST(j), \hat{p}_j, p_j')$,
$p_j' : (0TEST(k+j), p_j', p_{j+1})$,
$\hat{p}_j : (ADD(j, -b), \{\bar{p}_j\})$,

$\bar{p}_j : (0TEST(j+r), \tilde{p}_j, \bar{p}_j)$, and
$\tilde{p}_j : (ADD(j+k, -b), \{p_j\})$
simultaneously decrement related registers j and $j+k$, $1 \leq j \leq k$, down to zero. In this construction, we define an instance of the rule $\hat{p_j}$ and an instance of the rule $\tilde{p_j}$ for every generator b of the group G_j, which allows testing registers j and $j+k$ for equality independently of the number of generators of the corresponding (finitely generated) group. In the success case, i.e., if both have been checked to be equal, the procedure continues with the next pair of registers. At the end, in the success case, we take $p_{k+1} = l'_h$. In the failure case, an infinite loop is entered. We leave the remaining details of the construction to the interested reader. For example, to allow non-deterministic branching from a label p to q and s, without modifying the registers, we can use a working register r and a generator b as well as the sequence of instructions $p : (ADD(r, b), \{p'\})$ and $p' : (ADD(r, -b), \{q, s\})$. Moreover, if there is no working register in $M_{\mathcal{G}}$, we add one in $M_{\mathcal{G}'}$.

We conclude that the set generated by $M_{\mathcal{G}}$ equals the set accepted by $M_{\mathcal{G}'}$. □

Lemma 2. *Let $m \in \mathbb{N}$ and take the finite family of finitely presented groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Then there exists another family of finitely presented groups G' such that $\mathcal{L}_{acc}(\mathcal{G}\text{-}RM_0) \subseteq \mathcal{L}_{gen}(\mathcal{G}'\text{-}RM_0)$.*

Proof. We now start with an accepting $\mathcal{G}$-register machine with zero test $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ and construct a generating register machine with zero test $M_{\mathcal{G}'} = (\mathcal{G}', B', l'_0, l'_h, P')$, again using a similar construction of additional registers as in the proof of Lemma 1. $M_{\mathcal{G}'}$ randomly generates two copies of the output in registers i and $i+k$, $1 \leq i \leq k$. Then $M_{\mathcal{G}'}$ simulates an accepting computation of $M_{\mathcal{G}}$ in the registers $k+1, \ldots, m+k$ of $M_{\mathcal{G}'}$. In case l_h is reached in that way, instead of halting $M_{\mathcal{G}'}$ finally decreases all working registers $k+1, \ldots, m+k$ to zero:

For $j = k+1, \ldots, m+k$, starting with $p_{k+1} = l_h$, sequences of instructions
$p_j : (0TEST(j), \hat{p_j}, p_{j+1})$ and
$\hat{p}_j : (ADD(j, -b), \{p_j, p_j\})$
are carried out in a deterministic way, finishing with the HALT-instruction with label $l'_h = p_{m+k}$. We conclude that the set accepted by $M_{\mathcal{G}}$ equals the set generated by $M_{\mathcal{G}'}$. □

As an immediate consequence of the two preceding lemmas, we conclude that the generating and the accepting power of register machines over groups with the zero test instruction is equal.

Theorem 3. $\mathcal{L}_{acc}(*\text{-}RM_0) = \mathcal{L}_{gen}(*\text{-}RM_0)$.

A result similar to the one stated in Lemma 2 also holds true for blind register machines:

Corollary 2. *Let $m \in \mathbb{N}$ and take the finite family of finitely presented groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Then there exists another family of finitely presented groups G' such that $\mathcal{L}_{acc}(\mathcal{G}\text{-}BRM) \subseteq \mathcal{L}_{gen}(\mathcal{G}'\text{-}BRM)$.*

Proof. We can use the same construction as described in the proof of Lemma 2, except for the final procedure decrementing all working registers to zero, which is not needed as by definition acceptance for blind register machines requires all working registers to be zero. □

4.4 Vector Addition Systems over Groups

One of the classical results on vector addition systems is that, in the conventional definition of the model, adding a state control does not increase the power, because the states can be simulated using 3 additional components of vectors [11, Lemma 2.1]. This result can be naturally generalized to vector addition systems over groups with forbidden regions.

Theorem 4. *Consider a finitely generated computable group G, the product $G' = G \times \mathbb{Z}^3$, its subset $F = \{(g, a, b, c) \in G' \mid a < 0 \text{ or } b < 0 \text{ or } c < 0\}$, and an arbitrary G-VASS A. Then there exists a G'-VAS with the forbidden region F whose computations modulo the natural projection $p : G' \to G$ are exactly the computations of A.*

Proof. The construction from the proof of [11, Lemma 2.1] can be directly carried over to our setting: the three components over $\mathbb{Z}$ with forbidden negative values can be used to encode the current state and to compute the next one unambiguously. We do not recall the cited construction here because it is rather technical and out of the scope of this article. □

We can immediately generalize this result by replacing $\mathbb{Z}$ in the previous statement by a different group into which one can injectively (monomorphically) embed $\mathbb{Z}$.

Theorem 5. *Consider a finitely generated computable group G, and a totally ordered group Z such that there exists an injective homomorphism of totally ordered groups $i : \mathbb{Z} \to Z$. Take the product $G' = G \times Z^3$, its subset $F = \{(g, a, b, c) \in G' \mid a <_Z i(0) \text{ or } b <_Z i(0) \text{ or } c <_Z i(0)\}$, and an arbitrary G-VASS A. Then there exists a G'-VAS with the forbidden region F whose computations modulo the natural projection $p : G' \to G$ are exactly the computations of A.*

Proof. The injective homomorphism i delimits a totally ordered subgroup of Z which is isomorphic to $\mathbb{Z}$, allowing to perform the same operations as in the proof of [11, Lemma 2.1]. □

Remark 9. The result [11, Lemma 2.1] as well as the two generalizations we give here do *not* necessarily state the equality between the families of languages generated by VAS with and without states. Indeed, the language generated by a VASS is usually taken to contain all the vectors which the VASS reaches while also being in a terminal or halting state, while the language of a VAS is often taken to be simply its reachability set. In Definition 5, this behavior is captured by allowing the underlying register machine of a G-VAS to halt at any time.

A consequence of the fact that only the elements a G-VASS produces in its halting state contribute to the generated language is that a G-VASS can generate the empty language $\emptyset$ by never reaching the halting state. On the other hand, the language of a G-VAS always includes at least the start element. This observation together with the fact that we define G-VAS as a particular case of G-VASS implies the following statement.

Proposition 4. *For any finitely generated computable group G, it holds that $\mathcal{L}(G\text{-}VAS) \subsetneq \mathcal{L}(G\text{-}VASS)$.*

Since this strict inclusion is rather trivial and does not reflect the intrinsic computing power of vector addition systems, in the rest of this section we will only consider G-VASS generating non-empty languages. In this setting, the increase in power due to the state control depends strongly on the underlying group. For example, $\mathbb{Z}^n\text{-}VAS \subsetneq \mathbb{Z}^n\text{-}VASS$, as shown in [3, Lemmas 6 and 7]. On the other hand, it follows trivially from Proposition 3 and Example 14 that adding states to vector addition systems over the singleton group $\mathbf{1}$ does not increase the power. We generalize these observations in the following statement.

Theorem 6. *If a finitely generated computable group G contains an element of order greater than 2, then $\mathcal{L}(G\text{-}VAS) \subsetneq \mathcal{L}(G\text{-}VASS) \setminus \{\emptyset\}$.*

Proof. Suppose that g is the element of G whose order is greater than 2. Suppose that there exists such a G-VAS A with the start element $g_0 \in G$ that $\mathcal{L}(A) = \{e, g\}$, where e is the neutral element of G. Then there exist two elements $h_0, h_1 \in G$ such that $g_0 h_0 = e$ and $g_0 h_1 = g$, and A executes the operations corresponding to adding h_0 to g_0 to generate e, and corresponding to adding h_1 to g_0 to generate g. Since $ord(g) > 2$, $g \neq e$, and either $h_0 \neq e$, or $h_1 \neq e$, or both. Let $h \in \{h_0, h_1\}$ such that $h \neq e$. Then, if A executes the sequence of actions corresponding to h_0, and afterwards the one corresponding to h, it will generate $g_0 h_0 h = h$. If $h \notin \{e, g\}$, then $\mathcal{L}(A) \supsetneq \{e, g\}$, which is a contradiction.

Now suppose that $h \in \{e, g\}$. By construction, $h \neq e$, so $h = g$. Suppose that A carries out the sequence of actions corresponding to h_0, then the sequence corresponding to h, and then the same sequence again. It would generate $g_0 h_0 h h = h^2 = g^2$. By hypothesis, $ord(g) > 2$, meaning that $g^2 \notin \{e, g\}$. But in this case $\mathcal{L}(A) \supsetneq \{e, g\}$, which is again a contradiction.

We conclude the proof by remarking that the language $\{e, g\}$ can be generated by a G-VASS with the starting element e and whose underlying register machine contains the single instruction $l : (ADD(1, g), \{l_h\})$. □

It follows immediately from the previous theorem that the state control already makes a difference for vector addition systems over $\mathbb{Z}_3 = \mathbb{Z}/3\mathbb{Z}$, the group of addition modulo 3. Indeed, it is impossible to construct a $\mathbb{Z}_3$-VAS generating $\{0, 1\}$: any attempt would end up putting the element 2 into the generated language.

Corollary 3. $\mathcal{L}(\mathbb{Z}_3\text{-}VAS) \subsetneq \mathcal{L}(\mathbb{Z}_3\text{-}VASS) \setminus \{\emptyset\}$.

On the other hand, the state control does not increase the power of vector addition systems over the two-element group $\mathbb{Z}_2 = \mathbb{Z}/2\mathbb{Z}$.

Proposition 5. $\mathcal{L}(\mathbb{Z}_2\text{-}VAS) = \mathcal{L}(\mathbb{Z}_2\text{-}VASS) \setminus \{\emptyset\}$.

Proof. Only the following non-empty languages over $\mathbb{Z}_2$ exist: $\{0\}$, $\{1\}$, and $\{0,1\}$. The first two ones can be generated by a $\mathbb{Z}_2$-VAS whose underlying register is initialized to 0 or 1, respectively, and whose underlying register machine always halts immediately.

The third one is generated by a $\mathbb{Z}_2$-VAS with the start element $g_0 \in \{0,1\}$ and with the underlying register machine containing only one instruction $l : (ADD(1,1), \{l, l_h\})$. □

Even though Theorem 6 gives a sufficient criterion for the state control to strictly augment the expressive power of G-VAS, we do not claim that this criterion is necessary. Establishing a necessary and sufficient criterion is left as an open problem.

We conclude this discussion about the frontier between the power of G-VAS and G-VASS by recalling that the paper [3] considers *uniform families* of VAS, $\mathbb{Z}\text{-}VAS_{\cup}$, which are essentially an extension of vector addition systems allowing a finite number of start vectors instead of only one of them. In a similar fashion, we can consider uniform families of G-VAS. We denote these by $G\text{-}VAS_{\cup}$. For a finite group H, languages generated by uniform families of H-VAS turn out to be the same as those generated by H-VASS.

Proposition 6. *For a finite group H, $\mathcal{L}(H\text{-}VAS_{\cup}) = \mathcal{L}(H\text{-}VASS)$.*

Proof. Since H is finite, H-VASS generate finite languages. Hence, any given H-VASS A can be "simulated" by a uniform family of H-VAS without any addition elements (the underlying register machine halts immediately) and whose start elements form exactly $\mathcal{L}(A)$. □

5 Generating and Accepting Strings

In this section, we will show how the idea to use the free group constrained to the syntactic monoid introduced in Example 12 can be used to turn register machines over groups into devices recognizing and generating strings. We assume that the reader is familiar with regular and context free grammars, finite-state and push-down automata, as well as Turing machines. For an extensive introduction to the domain of formal languages, we refer to [19].

For an alphabet V, we will use the symbol $\mathcal{V}$ to refer to the free group $\langle V \mid \emptyset \rangle$ whenever no ambiguity occurs. We will also use the notation $F_{\mathcal{V}}$ to refer to the elements of the free group $\mathcal{V}$ which do not appear in the syntactic monoid V^*: $F_{\mathcal{V}} = \{x \in \mathcal{V} \mid x \notin V^*\} = \mathcal{V} \setminus V^*$.

Our first result shows that a register machine with only one register containing values from $\mathcal{V}$ can simulate a regular grammar. The symbol REG_V stands for the class of all regular languages over the alphabet V.

Theorem 7. $\mathcal{L}_{gen}\big((\mathcal{V})\text{-}RM_{\neg(F_\mathcal{V})}\big) = REG_V$.

Proof. Consider an arbitrary regular language L and the regular grammar $G = (N, V, P, S)$ generating it, where N is the set of non-terminal symbols, V is the set of terminal symbols, $N \cap V = \emptyset$, P is the set of productions, and S is the starting symbol. We will construct a $(\mathcal{V})$-register machine with the forbidden region $F_\mathcal{V}$ which will generate the language L. We associate the instruction labels $l(A)$ (defined below) with every non-terminal $A \in N$ and we construct the program of M in the following way:

- for every rule $A \to aB$, $A, B \in N$, $a \in V$, we add a fresh label l_A to the set $l(A)$ and the instruction $l_A : (ADD(1, a), l(B))$ to the program of M;
- for every rule $A \to a$, $A \in N$, $a \in V$, we add a fresh label l_A to the set $l(A)$ and the instruction $l_A : (ADD(1, a), \{l_h\})$ to the program of M;
- for every rule $A \to \lambda$, $A \in N$, we add a fresh label l_A to the set $l(A)$ and the instruction $l_A : (ADD(1, \lambda), \{l_h\})$ to the program of M, where λ is the empty string and the neutral element of the group $\mathcal{V}$.

The set of instruction labels of M is therefore $B = \bigcup_{A \in N} l(A)$. Without losing generality, we may assume that $l(S)$ only contains one element, which will serve as the starting label for M.

By construction, M faithfully simulates the regular grammar G by reflecting the current non-terminal symbol in the instruction label, by adding the corresponding symbol to the only register containing the generated string, and by non-deterministically jumping to one of the instruction labels corresponding to the new non-terminal symbol if a rule $A \to aB$ is applied. □

A symmetric result can be proved for the accepting mode, except that in this case we need the terminal zero test to ensure that the input string has been read completely. In this statement, we combine the notation BRM and the subscript $\neg(F_\mathcal{V})$ to refer to register machines with both the terminal zero test and forbidden regions.

Theorem 8. $\mathcal{L}_{acc}\big((\mathcal{V})\text{-}BRM_{\neg(F_\mathcal{V})}\big) = REG_V$.

Proof. Consider a regular language L and a (non-deterministic) finite automaton $FA = (Q, V, \delta, q, F)$ recognizing it, where Q is the set of states, V is the set of input symbols, $\delta : Q \times V \to 2^Q$ is the transition function giving a set of target states based on the current state and the symbol on the tape, q is the starting state, and $F \subseteq Q$ is the set accepting states. We construct a $(\mathcal{V})$-register machine with the forbidden region $F_\mathcal{V}$ which accepts the reverse image of the language L, i.e., $\mathcal{L}_{acc}(M) = \{s^R \mid s \in L\} = L^R$, where s^R is the reverse of s.

We denote $l_h(p) = \{l^h\}$ if $p \in F$ and $l_h(p) = \emptyset$ otherwise. We define the following mapping from the set of states of FA to the set of labels of M:

$$l(p) = \{p_a \mid p \in Q, a \in V, \delta(p,a) \neq \emptyset\} \cup l_h(p).$$

We also use the natural extension $l(Q') = \bigcup_{p \in Q} l(p)$, for $Q' \subseteq Q$.

For every pair $p \in Q$ and $a \in V$ for which $\delta(p,a) \neq \emptyset$, we add the following to M:

1. the label p_a to the set of labels B;
2. the instruction $p_a : \big(ADD(1, a^{-1}),\ l(\delta(p,a)) \big)$ to the program.

Finally, we add the instruction $l_0 : (ADD(1,\lambda), l(q))$ to M, where q is the starting state of FA.

To recognize the string $s^R \in L^R$, M first non-deterministically jumps to one of the labels q_a by performing the instruction l_0 which does not modify the register. In the following step, the machine performs $ADD(1, a^{-1})$. If the string in its register has the form wb, $w \in V^*$, $b \in V \setminus \{a\}$, then this operation results in the forbidden string wba^{-1} and M aborts. Otherwise the value of the register becomes w, M non-deterministically jumps to one of the labels in $l(\delta(q,a))$, and repeats the same procedure.

Whenever the machine simulates a jump to a state $p \in F$, it may choose to jump to the instruction l^h and halt. If it does so while the register does not contain the empty string λ, the terminal zero test will fail and the computation will abort. If, on the other hand, it does not jump to l^h after updating its register to λ, the subsequent instruction $ADD(1, a^{-1})$ aborts the computation.

We conclude the proof by recalling that regular languages are closed under the reverse image. □

Registers containing elements from $\mathcal{V}$ can also be used as stacks, allowing to simulate pushdown automata. We only give sketches of the proofs of the following results, the omitted details being very similar to those appearing in the previous proof.

Theorem 9. *Consider two alphabets V and R, the corresponding free groups $\mathcal{V}$ and $\mathcal{R}$, and the regions $F_{\mathcal{V}} = \mathcal{V} \setminus V^*$ and $F_{\mathcal{R}} = \mathcal{R} \setminus R^*$. Then $(\mathcal{V}, \mathcal{R})$-register machines with terminal zero test and with the family of forbidden regions $(F_{\mathcal{V}}, F_{\mathcal{R}})$ accept all context-free languages that can be accepted by a pushdown automaton with the tape alphabet V and the stack alphabet R.*

Proof (sketch). The proof idea is very close to that employed in Theorem 8: we construct a register machine M with two registers. The first register contains the input string which it non-deterministically "reads" by appending symbols a^{-1} and aborting when the symbol was not guessed correctly. The second register contains the stack. M pushes a symbol $z \in R$ on the stack by performing $ADD(2, z)$ and

pops a symbol by non-deterministically performing $ADD(2, z^{-1})$. At the end of the computation, both registers must be empty for M to accept. We conclude the sketch of the proof by recalling that context-free languages are closed under the mirror image. □

The proof of Lemma 2 can be directly generalized to register machines with forbidden regions, yielding the following corollary for the generating mode.

Theorem 10. *Consider two alphabets V and R, the corresponding free groups $\mathcal{V}$ and $\mathcal{R}$, and the regions $F_{\mathcal{V}} = \mathcal{V} \setminus V^*$ and $F_{\mathcal{R}} = \mathcal{R} \setminus R^*$. Then $(\mathcal{V}, \mathcal{R})$-register machines with terminal zero test and with the family of forbidden regions $(F_{\mathcal{V}}, F_{\mathcal{R}})$ generate all context-free languages that can be accepted by a pushdown automaton with the tape alphabet V and the stack alphabet R.*

Finally, we remark that two registers containing strings over the same alphabet can be used to directly simulate the tape of a Turing machine. Our construction is very similar to an automaton with two independent stacks.

Without losing generality, we will only consider Turing machines whose input is placed entirely to the left of their head.

Theorem 11. *Consider the alphabet V, the free group $\mathcal{V}$ over V, and the region $F_{\mathcal{V}} = \mathcal{V} \setminus V^*$. $(\mathcal{V}, \mathcal{V})$-register machines with the zero test instruction and the family of forbidden regions $(F_{\mathcal{V}}, F_{\mathcal{V}})$ can directly simulate a Turing machine by starting with the initial tape contents in the first register and halting with the final tape contents in the first register.*

Proof (sketch). A register machine M simulates a given Turing machine T by keeping the string representing the tape contents to the left of the head in the first register, and the reverse of the tape contents to the right of the head in the second register. At every step, M checks if the second register is not empty, and if not, non-deterministically guesses the symbol T is reading. To write a symbol a on the tape, M simply performs $ADD(2, a)$. To simulate a move of the head of T to the left, M non-deterministically reads a symbol from the first register and adds it to the second register. To simulate a move to the right, M non-deterministically reads a symbol a from the second register and adds a to the first register. If the second register is empty, M simulates the action of T corresponding to reading an empty tape cell. If the head should move to the right, M adds the symbol representing the empty tape cell (different from the empty string λ) to the first register. Similarly, if M must simulate a move of the head to the left while its first register is empty, it adds the symbol representing the empty tape cell to the second register. □

6 Conclusion and Open Problems

In this paper we focused on generalizing the model of register machines to operate on groups instead of natural or integer numbers, thus continuing previous works

aiming at generalizing related models of computing, such as vector addition systems and P systems [2, 3, 7, 9]. Generalizing register machines to groups allowed us to put forward the fundamental connection between vector addition systems and register machines, as well as to reveal an unexpected possibility to operate on registers containing strings, without any encoding.

The definitions and basic tools exhibited in this paper illustrate some of the consequences of the way in which register machines are generalized. One interesting class of problems which is still left open is the role of the nature of the underlying group in defining the frontiers of computational power. For example, Theorem 6 approaches one such separation between vector addition systems with and without states, but does not give a crisp borderline. On the other hand, the impact of the group being commutative is still to be explored.

References

1. Alhazov, A., Aman, B., Freund, R., Păun, Gh.: Matter and anti-matter in membrane systems. In: Jürgensen, H., Karhumäki, J., Okhotin, A. (eds.) Descriptional Complexity of Formal Systems – 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8614, pp. 65–76. Springer (2014). https://doi.org/10.1007/978-3-319-09704-6_7
2. Alhazov, A., Belingheri, O., Freund, R., Ivanov, S., Porreca, A.E., Zandron, C.: Purely catalytic P systems over integers and their generative power. In: Leporati, A., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) Membrane Computing – 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10105, pp. 67–82. Springer (2016). https://doi.org/10.1007/978-3-319-54072-6_5
3. Alhazov, A., Belingheri, O., Freund, R., Ivanov, S., Porreca, A.E., Zandron, C.: Semilinear sets, register machines, and integer vector addition (p) systems. In: Proceedings of the 17th International Conference on Membrane Computing, CMC 2016. pp. 27–42 (2016)
4. Büning, H.K., Lettmann, T., Mayr, E.W.: Projections of vector addition system reachability sets are semilinear. Theoretical Computer Science **64**(3), 343–350 (May 1989). https://doi.org/10.1016/0304-3975(89)90055-8
5. Freund, R.: Control mechanisms for array grammars on Cayley grids. In: Durand-Lose, J., Verlan, S. (eds.) Machines, Computations, and Universality – 8th International Conference, MCU 2018, Fontainebleau, France, June 28-30, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10881, pp. 1–33. Springer (2018). https://doi.org/10.1007/978-3-319-92402-1_1
6. Freund, R., Ibarra, O.H., Păun, Gh., Yen, H.C.: Matrix languages, register machines, vector addition systems. In: Gutiérrez-Naranjo, M., Riscos-Núñez, A., Romero Campero, F., Sburlan, D. (eds.) Proceedings of the Third Brainstorming Week on Membrane Computing. pp. 155–168. University of Sevilla (2005)
7. Freund, R., Ivanov, S., Verlan, S.: P systems with generalized multisets over totally ordered abelian groups. In: Rozenberg, G., Salomaa, A., Sempere, J.M., Zandron, C. (eds.) Membrane Computing – 16th International Conference, CMC 2015, Valencia, Spain, August 17–21, 2015, Revised Selected Papers. Lecture Notes in Computer

Science, vol. 9504, pp. 117–136. Springer (2015). https://doi.org/10.1007/978-3-319-28475-0_9
8. Greibach, S.A.: Remarks on blind and partially blind one-way multicounter machines. Theoretical Computer Science **7**(3), 311–324 (1978). https://doi.org/10.1016/0304-3975(78)90020-8
9. Haase, C., Halfon, S.: Integer vector addition systems with states. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) Reachability Problems: 8th International Workshop, RP 2014, Oxford, UK, September 22–24, 2014. Proceedings, pp. 112–124. Springer (2014). https://doi.org/10.1007/978-3-319-11439-2_9
10. Holt, D.F., Eick, B., O'Brien, E.A.: Handbook of Computational Group Theory. CRC Press (2005)
11. Hopcroft, J., Pansiot, J.J.: On the reachability problem for 5-dimensional vector addition systems. Theoretical Computer Science **8**(2), 135–159 (1979). https://doi.org/10.1016/0304-3975(79)90041-0
12. Ivanov, S.: On the power and universality of biologically-inspired models of computation. (Étude de la puissance d'expression et de l'universalité des modèles de calcul inspirés par la biologie). Ph.D. thesis, University of Paris-Est, France (2015), `https://tel.archives-ouvertes.fr/tel-01272318`
13. Korec, I.: Small universal register machines. Theoretical Computer Science **168**(2), 267–301 (1996). https://doi.org/10.1016/S0304-3975(96)00080-1
14. Levi, F.W.: Ordered groups. In: Proceedings of the Indian Academy of Sciences. vol. A16, pp. 256–263 (1942)
15. Minsky, M.: Recursive unsolvability of Post's problem of tag and other topics in the theory of Turing machines. Annals of Mathematics, second series **74**, 437–455 (1961)
16. Minsky, M.L.: Computation. Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ (1967)
17. Mitrana, V., Stiebe, R.: Extended finite automata over groups. Discrete Applied Mathematics **108**(3), 287 –300 (2001). https://doi.org/10.1016/S0166-218X(00)00200-6
18. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
19. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, 3 volumes. Springer, New York, NY, USA (1997)
20. Thomas W. Judson, R.A.B.: Abstract Algebra: Theory and Applications (2018)