Theoretical Informatics and Applications
Theoret. Informatics Appl. **35** (2001) 419–435

COMMUTATIVE IMAGES OF RATIONAL LANGUAGES AND THE ABELIAN KERNEL OF A MONOID*

Manuel Delgado[1]

Abstract. Natural algorithms to compute rational expressions for recognizable languages, even those which work well in practice, may produce very long expressions. So, aiming towards the computation of the commutative image of a recognizable language, one should avoid passing through an expression produced this way. We modify here one of those algorithms in order to compute directly a semilinear expression for the commutative image of a recognizable language. We also give a second modification of the algorithm which allows the direct computation of the closure in the profinite topology of the commutative image. As an application, we give a modification of an algorithm for computing the Abelian kernel of a finite monoid obtained by the author in 1998 which is much more efficient in practice.

Mathematics Subject Classification. 20M35, 68Q99.

Introduction

For basic notions related to rational sets we refer the reader to [16] or [4]. An algorithm to compute a rational expression for the language recognized by a finite automaton, the one developed in [5], is implemented in the computer program AMoRE [15]. This algorithm, while working very well in practice, produces expressions whose size grows exponentially, as we shall see. So, when we have applications in mind that require the computation of the commutative image of a recognizable language, we should avoid the usage of the rational expressions produced by such an algorithm. Through simple modifications of the algorithm,

Keywords and phrases: Rational language, semilinear set, profinite topology, finite monoid.

* *The author gratefully acknowledges support of FCT through the* Centro de Matemática da Universidade do Porto *and the FCT and POCTI Project POCTI/32817/MAT/2000 which is comparticipated by the European Community Fund FEDER.*

[1] Centro de Matemática, Universidade do Porto P. Gomes Teixeira, 4099-002 Porto, Portugal; e-mail: `mdelgado@fc.up.pt`

© EDP Sciences 2002

we show that we can compute an expression for the commutative image in the n-generated free Abelian group of the language recognized by a finite automaton without explicitly computing a rational expression for that language. This way of computing the commutative image of a recognizable language is much more efficient in practice than the obvious one using the rational expression. We show that even an expression for the closure in the profinite group topology of that commutative image can be computed this way.

The application we are interested in is to solve, in a relatively efficient way, the problem of computing the Abelian kernel of a finite monoid. This problem is the Abelian counterpart of the Rhodes Type II conjecture, which was solved independently by Ash [2] and Ribes and Zalesskiĭ [20]. Many decidability problems in the theory of finite semigroups can be reduced to the computation of kernels of finite monoids as may be seen, for instance, in [12]. The validity of an algorithm proposed by Pin and Reutenauer [18] (proved in [20]) was the successful realization of a topological approach to computing kernels introduced by Pin [17]. An algorithm to compute the Abelian kernel of a finite monoid was given by the author a few years ago in [8]. The algorithm proposed there involves, for each element of the monoid, the computation of a rational expression for the language (over a finite alphabet) recognized by a finite automaton, followed by the computation of an expression for the closure, under the profinite group topology, of its commutative image. We use the modification proposed here to compute directly this expression, obtaining this way an algorithm that works much better in practice.

The implementation in GAP [21, 26] of the original algorithm allowed computations of some examples, but only of monoids of small order. Bigger examples were handled as the algorithm was improved. The ability to compute the Abelian kernels of some monoids led the author and Fernandes to the results of [9].

After a section of preliminaries recalling some facts on rational subsets, Abelian groups and finite monoids, this paper is divided as follows:

In Section 2 we state an algorithm to compute a semilinear expression from a rational expression of a language. Then we recall an algorithm (the original one) to test whether or not an element of a finite monoid belongs to the Abelian kernel of the monoid.

In Section 3 we describe in some detail an algorithm to compute a rational expression for the language recognized by a finite automaton.

In Section 4 we make some comments on the tractability of the algorithm previously given. In particular we look at the number of steps needed as well as at the growth of the size of the rational expression obtained.

In Section 5 we propose a modification of the algorithm and discuss what is gained with it. The algorithm to test whether a given element of a finite monoid belongs to its Abelian kernel is then improved.

In Section 6 we recall some well-known results and discuss how their usage allows the computation of the Abelian kernel of a finite monoid without applying the algorithm to every element, thus saving time.

1. Preliminaries

This section starts with some words on rational subsets of A^*, $\mathbb{N}^n$ and $\mathbb{Z}^n$. Then it recalls some facts about Abelian groups and ends with a few more words on finite monoids.

The elements of $\mathbb{Z}^n$ will sometimes be called vectors. Let $A = \{a_1, \dots, a_n\}$ be an alphabet with n letters and let $\gamma : A^* \to \mathbb{Z}^n$ be the canonical homomorphism (*i.e.*, the homomorphism defined by $\gamma(a_i) = (0, \dots, 0, 1, 0, \dots, 0)$, the i-th vector of the standard basis of the n-generated free Abelian group $\mathbb{Z}^n$). Observe that the image of γ is contained in the n-generated free commutative monoid $\mathbb{N}^n$. We endow $\mathbb{Z}^n$ with the profinite topology (*i.e.*, the weakest topology rendering all homomorphisms into finite groups continuous) and write $\overline{X}$ to denote the closure of the subset $X \subseteq \mathbb{Z}^n$. All topological notions used in this paper may be found in [27].

For $a \in \mathbb{N}^n$ (respectively, $a \in \mathbb{Z}^n$), we denote by a^* or $a\mathbb{N}$ (resp. $\langle a \rangle$ or $a\mathbb{Z}$) the submonoid of $\mathbb{N}^n$ (resp. subgroup of $\mathbb{Z}^n$) generated by a. For a subset L of a monoid (resp. a group) the notation L^* (resp. $\langle L \rangle$) is also used with the more general meaning "submonoid generated by L" (resp. "subgroup generated by L").

Let L be a non-empty rational subset of A^*, *i.e.*, a non-empty rational language. The set $\gamma(L)$, being the homomorphic image of a rational set, is a rational subset of $\mathbb{N}^n$, therefore it is a *semilinear* set, *i.e.*, a finite union of sets of the form $a + b_1\mathbb{N} + \cdots + b_p\mathbb{N}$, with $a, b_1, \dots, b_p \in \mathbb{N}^n$, which are called *linear*.

The following has been proved in [8].

Proposition 1.1. *The closure of a rational subset of* $\mathbb{Z}^n$ *is rational. Furthermore, this closure can be computed using the following formulas, where* S *and* T *are rational subsets of the free Abelian group given by rational expressions:*

(1) $\overline{S} = S$ *if* S *is finite;*
(2) $\overline{S \cup T} = \overline{S} \cup \overline{T}$*;*
(3) $\overline{S + T} = \overline{S} + \overline{T}$*;*
(4) $\overline{S^*} = \langle S \rangle$*, the subgroup of* $\mathbb{Z}^n$ *generated by* S*.*

As an immediate consequence we have the following corollary:

Corollary 1.2. *For* $a, b_1, \dots, b_r \in \mathbb{N}^n$*, the closure of the subset* $a + b_1\mathbb{N} + \cdots + b_r\mathbb{N}$ *of* $\mathbb{Z}^n$ *is the coset* $a + b_1\mathbb{Z} + \cdots + b_r\mathbb{Z}$ *of the subgroup of* $\mathbb{Z}^n$ *generated by the elements* $b_1, \dots, b_r$*.*

In order to keep the paper as self-contained as possible, we recall here some well-known results concerning Abelian groups. See [22] or [7] for details.

A subgroup H of $\mathbb{Z}^n$ generated by m vectors can be specified by giving an $m \times n$ matrix A whose rows are the m given vectors. The subgroup H is the set of all integral linear combinations of the rows of A and is denoted by $S(A)$. These linear combinations are the vectors uA where u ranges over $\mathbb{Z}^m$. If a matrix B is obtained from the matrix A through *integer row operations* (*i.e.*, interchanging rows, multiplying a row by -1 or adding an integral multiple of a row to another row), then $S(A) = S(B)$. Two $m \times n$ matrices are said to be *row equivalent*

over $\mathbb{Z}$ if one of them can be transformed into the other by a finite sequence of row operations.

Let A be an $m \times n$ matrix and suppose that A has r nonzero rows. We say that A is in *Hermite normal form* (HNF) if the following conditions are satisfied:

(1) the first r rows are nonzero;
(2) for $1 \leq i \leq r$ let a_{ij_i} be the first nonzero entry in the i-th row of A. Then $j_1 < j_2 < \cdots < j_r$;
(3) $a_{ij_i} > 0, 1 \leq i \leq r$;
(4) if $1 \leq k < i \leq r$, then $0 \leq a_{kj_i} < a_{ij_i}$.

The following proposition states that it is possible to specify representatives for the row equivalence classes of integer matrices.

Proposition 1.3. *Each integer matrix B is row equivalent over $\mathbb{Z}$ to a unique matrix A in Hermite normal form.*

It turns out that if a subgroup H of $\mathbb{Z}^n$ is such that $H = S(B)$ for a matrix B which is row equivalent to a matrix A in Hermite normal form, then the non-zero rows of the matrix A form a basis of the subgroup H. In particular, a subgroup of $\mathbb{Z}^n$ can be generated by n or fewer elements. It follows also that to test whether two finitely generated subgroups of $\mathbb{Z}^n$ are equal it suffices to compute the bases of the subgroups given by matrices in HNF and test their equality.

In the literature already mentioned, and also in [24], one may find polynomial time algorithms to compute normal forms for integer matrices. Some of these algorithms, particularly those based in Storjohann's work, are implemented in GAP [26].

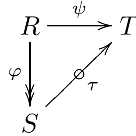
There are several ways to give a finite monoid in practice. One of them is to give it as a submonoid, generated by a set A, of a monoid U, called universe, for which is known an algorithm to compute the product and an algorithm to test the equality of two elements. Another is to give it as the transition monoid of a finite A-automaton. Alternatively, a presentation $\langle A \mid R \rangle$ may be given. Of course, the multiplication table is another way to give a finite monoid.

Let M be a finite monoid. Since M is finite, there exists a finite alphabet $A = \{a_1, \ldots, a_n\}$ and an onto homomorphism $\varphi : A^* \to M$, thus M may be seen as an A-monoid. To avoid arbitrariness, the set A above could be taken equal to M, but usually A can be much smaller, as is clear from the discussion above about how a finite monoid can be given. We fix M, A and φ.

Note that the complexity of an algorithm concerning finite monoids depends in general on the way monoids are given. For instance, determining the number of elements of a monoid is trivial when the multiplication table is given and may require a lot of work in other cases.

The *right Cayley graph* of a finite monoid M relative to a generating set A is the graph $\Gamma_{M,A}$ (or simply Γ_M, if the generating set is understood) having M as set of vertices and, for each vertex s and generator a, an edge labeled by a from s to sa.

A *relational morphism* of monoids is a relation $\tau : S \multimap T$ between the monoids S and T with domain S which is a submonoid of $S \times T$. A relational morphism always has a factorization in terms of the inverse of an onto homomorphism and a homomorphism. Such a factorization is conveniently described through a diagram such as the following:



A pair $(s, t) \in S \times T$ belongs to τ if and only if there exists $u \in R$ such that $s = \varphi(u)$ and $t = \psi(u)$.

For a relational morphism $\tau : S \multimap T$, the set $\{s : (s, t) \in \tau\}$ is denoted by $\tau^{-1}(t)$.

In this paper we will assume that the right Cayley graph of a monoid may be computed efficiently (see [11]) and do not care about the way the monoid is given. (See Rem. 5.4 below.)

Let **Ab** be the class of all finite Abelian groups. The *Abelian kernel of M* is $K_{\mathbf{Ab}}(M) = \bigcap \tau^{-1}(1)$, with the intersection being taken over all groups $G \in \mathbf{Ab}$ and all relational morphisms of monoids $\tau : M \multimap G$. Of course this definition may be given for any class of finite groups. In the Rhodes Type II Conjecture the class of all finite groups was considered and the terminology *kernel of M* was used.

The following result proved in [8] gives an alternative definition of Abelian kernel: the set of elements $x \in M$ such that $0 \in \overline{\gamma(\varphi^{-1}(x))}$. Note that the Abelian kernel does not depend on the choice of A and φ.

Theorem 1.4. *An element $x \in M$ belongs to the Abelian kernel of M if and only if $0 \in \overline{\gamma(\varphi^{-1}(x))}$.*

Our aim is to give an algorithm that allows us to test, in practice, whether or not $0 \in \overline{\gamma(\varphi^{-1}(x))}$.

2. Semilinear expressions and the original algorithm

Let n be a positive integer. A *linear expression over n* is an expression of the form $a + b_1N + \cdots + b_pN$, where $a, b_1, \ldots, b_p \in \mathbb{N}^n$. We say that the expression $a + b_1N + \cdots + b_pN$ has *size p*. We define the *length* of the linear expression $a+b_1N+\cdots+b_pN$ as being $|a|+|b_1|+\cdots+|b_p|$, where, for $x = (x_1, \ldots, x_n) \in \mathbb{N}$, $|x|$ denotes the maximum of the set $\{|x_1|, \ldots, |x_n|\}$. A *semilinear expression* is either the empty set $\emptyset$ or a finite union of linear expressions. The *size* (respectively *length*) of the semilinear expression $\bigcup_{i=1}^{r} u_{0,i} + u_{1,i}N + \cdots + u_{p_i,i}N$ is the maximum of the sizes (respectively lengths) of the linear expressions involved in the expression and its *width* is r, the number of linear expressions involved. The size, the length and the width of $\emptyset$ are 0. Taking, if necessary, some $u_{i,j}$ equal to the null vector, we

may suppose that a semilinear set is a finite union of linear sets with the same size. Therefore, in the above expression all p_i may be supposed to be equal.

As *rational expressions* over an alphabet (*i.e.*, finite expressions involving letters, unions, products or the operation $-^*$) are used to represent rational languages, we use semilinear expressions to represent semilinear sets. The linear expression $a+b_1N+\cdots+b_pN$ over n represents the linear set $a+b_1\mathbb{N}+\cdots+b_p\mathbb{N}$ of $\mathbb{N}^n$. A semilinear expression represents the union of the linear sets represented by the linear expressions involved. Semilinear expressions representing the same semilinear set are said to be *equivalent*. Similarly, rational expressions representing the same rational language are also said to be *equivalent*.

The next proposition (also proved in [8]) gives an algorithm to effectively compute a semilinear expression for the semilinear set $\gamma(L)$ from a rational expression for the rational language L.

Suppose that we are given a non-empty rational language L. Observe that L is either finite or of the form $L_1 \cup L_2$, $L_1 \cdot L_2$ or L_1^*, with L_1 and L_2 rational languages. Suppose that semilinear expressions for $\gamma(L_1)$ and $\gamma(L_2)$ are known, say $\bigcup_{i=1}^r u_{0,i}+u_{1,i}N+\cdots+u_{p,i}N$ and $\bigcup_{j=1}^s v_{0,j}+v_{1,j}N+\cdots+v_{q,j}N$, respectively.

Proposition 2.1. *With the notation just introduced, the following formulas hold:*

(1) *if L is finite, then* $\bigcup_{x\in L}\gamma(x)$ *is a semilinear expression for $\gamma(L)$;*

(2) $\bigcup_{k=1}^{r+s} w_{0,k}+w_{1,k}N+\cdots+w_{t,k}N$, *where*

$$w_{l,k}=\begin{cases} u_{l,k} & \text{if } \ k\le r \text{ and } l\le p\\ v_{l,k-r} & \text{if } \ k-r\le s \text{ and } l\le q\\ 0 & \text{otherwise,}\end{cases}$$

is a semilinear expression for $\gamma(L_1\cup L_2)$;

(3)

$$\bigcup_{1\le i\le r,1\le j\le s}((u_{0,i}+v_{0,j})+u_{1,i}N+\cdots+u_{p,i}N+v_{1,j}N+\cdots+v_{q,j}N)$$

is a semilinear expression for $\gamma(L_1\cdot L_2)$;

(4)

$$\sum_{1\le i\le r}(0\bigcup(u_{0,i}+u_{0,i}N+u_{1,i}N+\cdots+u_{p,i}N))$$

is a semilinear expression for $\gamma(L_1^*)$.

Proof. The formula given in (1) is obvious.

Clearly $(\bigcup_{i=1}^r u_{0,i}+u_{1,i}N+\cdots+u_{p,i}N)\cup(\bigcup_{j=1}^s v_{0,j}+v_{1,j}N+\cdots+v_{q,j}N)$ is a semilinear expression for $\gamma(L_1)\cup\gamma(L_2)=\gamma(L_1\cup L_2)$ and is obviously equivalent to the semilinear expression given in (2).

Since γ is a homomorphism, we have $\gamma(L_1 \cdot L_2) = \gamma(L_1) + \gamma(L_2)$. Therefore a semilinear expression for $\gamma(L_1 \cdot L_2)$ is $(\bigcup_{i=1}^{r} u_{0,i} + u_{1,i}N + \cdots + u_{p,i}N) + (\bigcup_{j=1}^{s} v_{0,j} + v_{1,j}N + \cdots + v_{q,j}N)$ and it is equivalent to the semilinear expression given in (3).

As γ is a homomorphism, the equality $\gamma(L_1^*) = \gamma(L_1)^*$ holds. Now, we may use the following two easy observations:

- $(X \cup Y)^* = X^* + Y^*$ for $X, Y \subseteq \mathbb{N}^n$;
- the submonoid generated by the linear set $a + b_1\mathbb{N} + \cdots + b_p\mathbb{N}$ is $0 \bigcup (a + a\mathbb{N} + b_1\mathbb{N} + \cdots + b_p\mathbb{N})$;

to obtain the semilinear expression for $\gamma(L_1^*)$ given in (4).

(We observe that the set $0 \bigcup (a + a\mathbb{N} + b_1\mathbb{N} + \cdots + b_p\mathbb{N})$ considered above is strictly contained in $a\mathbb{N} + b_1\mathbb{N} + \cdots + b_p\mathbb{N}$ (since it does not contain b_1, for example), which at first sight seemed to be the submonoid generated by $a + b_1\mathbb{N} + \cdots + b_p\mathbb{N}$.) $\square$

We observe that if the widths of the semilinear expressions for $\gamma(L_1)$ and $\gamma(L_2)$ are r and s respectively, then the semilinear expression for $\gamma(L_1 \cdot L_2)$ given by Proposition 2.1 has width rs, thus the growth of the width of the semilinear expression obtained when we apply this algorithm to a rational expression is very high in general. The growth of the size of the semilinear expression obtained is also very large in general (this is due to parts (3) and (4) of the proposition). Observe that the sizes of the expressions obtained grow in general, even when we try to substitute semilinear expressions by smaller equivalent semilinear expressions: for example, the submonoid of $\mathbb{N}^2$ generated by the elements $(2,0), (1,1)$ and $(0,2)$ is not generated by two elements, therefore the linear set $(2,0)\mathbb{N} + (1,1)\mathbb{N} + (0,2)\mathbb{N}$ can not be represented by a linear expression of size 2. We remark that the subgroup of $\mathbb{Z}^2$ generated by the 3 elements above is in fact generated by less than 3 elements (as happens with any other subgroup of $\mathbb{Z}^2$), thus the situation, in terms of the size of the expressions obtained, may be a bit different when we consider cosets of subgroups of $\mathbb{Z}^n$ instead of linear sets.

Corollary 1.2 suggests that, when we are interested in computing closures, we will mainly be concerned with unions of cosets of subgroups, instead of semilinear sets. This will allow us to surpass, in some sense, the problem concerning the size of the semilinear expressions stated above. In fact, we do not need more than n elements of $\mathbb{Z}^n$ to describe a subgroup of $\mathbb{Z}^n$ and it is feasible in practice to find such elements (it basically suffices to compute the HNF of a matrix).

Next we recall the algorithm obtained in [8] to test whether or not an element of a finite monoid belongs to its Abelian kernel. The *input* of the algorithm is a finite monoid M together with an element x of M; the *output* is **Yes**, if x belongs to the Abelian kernel of M and **Not**, otherwise.

Algorithm 2.2.

- *Construct the Cayley graph Γ_M of M.*
  Recall that we are assuming that this construction may be done in an efficient way. Next consider the automaton $\Gamma_M(x)$ obtained from Γ_M taking 1 for initial state and x for final state.

- *Compute a rational expression for the recognizable language* $\varphi^{-1}(x)$*. (This is the language recognized by* $\Gamma_M(x)$ *and a rational expression for it may be computed using, for example, Algorithm 3.1.) Then use Proposition 2.1 to compute a semilinear expression for* $\gamma(\varphi^{-1}(x))$ *and substitute* N *by* $\mathbb{Z}$ *whenever it appears in the expression (cf. Cor. 1.2).*
- *Test if* $0 \in \overline{\gamma(\varphi^{-1}(x))}$*.*
  Taking into account the expression that we have obtained for $\overline{\gamma(\varphi^{-1}(x))}$*, this means testing if a Diophantine system has some solution. There are polynomial algorithms, in terms of the absolute values of the coefficients, to solve Diophantine systems, admitting integer (eventually negative) solutions. See, for example [6]. This part may also be performed using normal forms for integer matrices which are already implemented in GAP [26].*

3. Computing rational and semilinear expressions

We begin this section recalling an already mentioned algorithm due to Brzozowski and McCluskey [5] for computing a rational expression for the language recognized by a finite automaton. Since we are aiming to give a variant of it to compute an expression for the commutative image of a recognizable language, we will explain it in some detail.

The definition of *generalized transition graph* (abbreviated: GTG) $\mathcal{G}$ over a given alphabet may be obtained from the definition of automaton by requesting that: $\mathcal{G}$ has a single initial state q_I and a single final state q_F, with $q_F \neq q_I$; given two states of $\mathcal{G}$ there is exactly one edge beginning in one of them and ending in the other; the labels of the edges of $\mathcal{G}$ are rational sets (instead of letters, as happens in the automata case) or, more precisely, rational expressions representing them. For states p, q of $\mathcal{G}$ we denote by $\lambda(p,q)$ the label of the single edge beginning in p and ending in q. A word w is *recognized* by $\mathcal{G}$ if and only if there is a finite sequence $q_I = p_0, \ldots, p_n = q_F$ of states of $\mathcal{G}$ and a factorization $w = u_1 \cdots u_n$ of w such that, for $1 \leq i \leq n$, u_i belongs to $\lambda(p_{i-1}, p_i)$. The *language recognized* by $\mathcal{G}$ is the set of words recognized by $\mathcal{G}$.

Let $\mathcal{A}$ be a finite automaton and let Q be its set of states. The *output* of the following algorithm, whose *input* is $\mathcal{A}$, is the label of the single edge from the initial state to the final state of the GTG obtained at the end. It will be a rational expression for the language recognized by $\mathcal{A}$.

Algorithm 3.1.

- *Construct the following generalized transition graph* $\mathcal{G}$*: the set of states is* $Q' = Q \cup \{q_I, q_F\}$ *where* q_I *and* q_F *do not belong to* Q*; the edges are labeled in the following way: the label of an edge from* q_I *to any initial state of* $\mathcal{A}$ *is the empty word and the same happens with the label of the edge from any final state of* $\mathcal{A}$ *to* q_F*. The remaining edges adjacent to* q_I *or to* q_F *are labeled by* $\emptyset$*, the empty set. The label* $\lambda(p,q)$*,* $p, q \in Q$*, is the set (eventually* $\emptyset$*) of letters labeling the edges from* p *to* q *in* $\mathcal{A}$*.*

- *Execute the following cycle:*
  While $Q \neq \emptyset$*,*
  – *choose* $q \in Q$*;*
  – destroy *the loop in* q*; then* eliminate q.

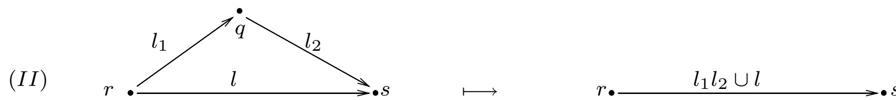
The way the *destruction* of a loop and the *elimination* of a state are performed is indicated in what follows.

Destroy a loop at q: if the label of the loop at q is non-empty and $p \in Q \setminus \{q\}$, replace the label $\lambda(q,p)$ of the edge from q to p by $(\lambda(q,q))^*\lambda(q,p)$ and the label $\lambda(q,q)$ of the loop in q by $\emptyset$.

The following may help to visualize this operation:

Eliminate a state q (with $\lambda(q,q) = \emptyset$): for all states r, s of $\mathcal{G}$ such that $r, s \neq q$, the label $\lambda(r,s)$ of the edge from r to s is replaced by $\lambda(r,q)\lambda(q,s) \cup \lambda(r,s)$. The state q and all edges adjacent to q are then removed.

The following may help to visualize this operation:



We observe that the language recognized by the generalized transition graph $\mathcal{G}$ constructed in the first step of the algorithm is precisely the language recognized by $\mathcal{A}$. We observe also that the language recognized by a GTG obtained from a GTG by destruction of a loop at some state followed by the elimination of this state is the language recognized by the original GTG. So, the output of Algorithm 3.1 is a rational expression for the language recognized by the automaton $\mathcal{A}$ given. In particular, the rational language given by Algorithm 3.1 is independent of the choices made in the second step of the algorithm.

The following lemma is straightforward and will be used freely. (Analogous lemmas could be stated after Algorithms 3.3 and 5.2.)

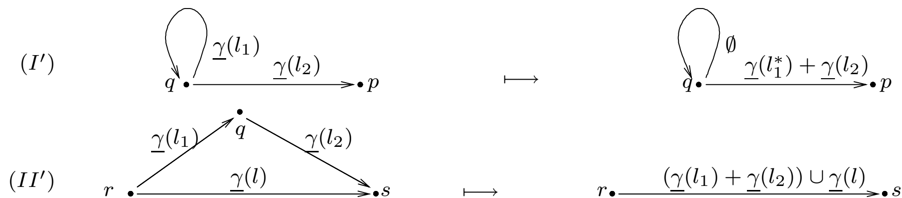
Lemma 3.2. *The substitution of a label by an equivalent rational expression while applying Algorithm 3.1 does not modify the set represented by its output.*

Consider now the following slight modification of Algorithm 3.1. Denote by $\underline{\gamma}(l)$ the semilinear expression given by Proposition 2.1 for the image by γ of the rational language with rational expression l. The *input* and *output* of the following algorithm are as in Algorithm 3.1.

Algorithm 3.3.

- *Replace each label* a *in* $\mathcal{A}$ *by* $\gamma(a)$ *(i.e., consider commutative labels).*
- *Construct a generalized transition graph as in the first step of Algorithm 3.1.*

- *Execute the cycle given in the second step of Algorithm 3.1, taking into account the change of the meaning of* destroy *and of* eliminate *which should be clear from the following pictures.*



The following proposition may be easily proved using Proposition 2.1 (*cf.* proof of Prop. 5.3).

Proposition 3.4. *The expression produced by Algorithm 3.3 with the automaton $\mathcal{A}$ as input is a semilinear expression for the image by γ of the language recognized by $\mathcal{A}$.*

4. Tractability

An algorithm is usually said to be tractable if the computations can be carried out in time that is a polynomial function of the size of the input. We do not believe that the algorithm given above to compute the Abelian kernel of a finite monoid is tractable in this sense, as may be inferred from this section.

In practice the output of Algorithm 3.1 is produced in a very reasonable time. The number of substitutions of labels involved to remove a state q in a GTG with $n+2$ states is not greater than $(n+1)^2$. In fact, it involves at most $n+1$ substitutions to destroy the loop at q plus $n(n+1)$ substitutions for the elimination of q. So the number of substitutions involved in the computation of a rational expression for the language recognized by a finite automaton with n states is bounded by the polynomial $n(n+1)^2$.

But there is a problem with this algorithm: the size of the rational expression obtained, in terms of the number of symbols involved, grows very fast as we shall see. In fact, unless we are able to simplify the expression (substituting rational expressions by equivalent smaller ones) when taking products or unions, the growth of the size of the rational expression is exponential in terms of $|Q||A|$.

It is not difficult to produce an automaton with a few hundred states over an alphabet with less than a dozen letters such that the rational expression produced by Algorithm 3.1 (and written in the usual readable form) occupies more than 10 Mbyte of computer memory. If one uses the algorithm given in Proposition 2.1 to produce a semilinear expression from such a rational expression, one has to wait a long time before obtaining a result.

To each rational expression R we associate two natural numbers $l(R)$ and $l'(R)$ defined as follows:

- $l(R)$ is the number of letters involved in the expression R, counted with multiplicities. (For example, $l(aaa^* \cup b) = 4$.)
- $l'(R)$ is defined recursively as follows:
  - $l'(R)$ is the number of elements of the set represented by R, if it is finite;
  - if $R = R_1 \cup R_2$, then $l'(R) = l'(R_1 \cup R_2) = l'(R_1) + l'(R_2)$;
  - if $R = R_1 \cdot R_2$, then $l'(R) = l'(R_1 \cdot R_2) = l'(R_1) \times l'(R_2)$;
  - if $R = R_1^*$, then $l'(R) = l'(R_1^*) = 1$.

Observe that $l'(R)$ is at most the width of the semilinear expression $\underline{\gamma}(R)$. The "at most" instead of "equal" is due to the fact that the width of $\underline{\gamma}(R^*)$ is, in general, greater that 1.

So, given a rational expression R, if we produce $\underline{\gamma}(R)$ using the algorithm given by Proposition 2.1, the semilinear expression obtained has width at least $l'(R)$. We have not found any reason to believe that one can find, in general, an equivalent semilinear expression with smaller width.

Either of the numbers defined above may, for a particular expression, be greater than the other. For example, if $R_1 = a_1a_2$ and $R_2 = (a_1 \cup a_2) \cdot (b_1 \cup b_2 \cup b_3)$ we have $l(R_1) = 2 > 1 = l'(R_1)$ and $l(R_2) = 5 < 6 = 2 \times 3 = l'(R_2)$.

Lemma 4.1. *Let $\mathcal{G}$ be a GTG having at least 4 states and such that, for all non-initial and non-final states p, q, $l(\lambda(p,q)) = k$ and $l'(\lambda(p,q)) = k'$ $(k, k' \geq 1)$. Let $\overline{q}$ be a non-initial and non-final state and suppose $\mathcal{G}_1$ is obtained from $\mathcal{G}$ by destruction of a loop in $\overline{q}$ followed by the elimination of $\overline{q}$. Then, representing by $\lambda_1(r,s)$ the label of the edge connecting r to s in the GTG $\mathcal{G}_1$, we have $l(\lambda_1(r,s)) = 4k$ and $l'(\lambda_1(r,s)) = k'^2 + k'$, for all states r and s of $\mathcal{G}_1$ different from the initial and final states.*

Proof. Observe that $\lambda_1(r,s) = \lambda(r,s) \cup \lambda(r,\overline{q})\lambda(\overline{q},\overline{q})^*\lambda(\overline{q},s))$. □

Consider the cyclic group G of order $n \geq 2$. Denote its elements by $a_0, \ldots, a_{n-1}$, and assume that the product is given by $a_ia_j = a_{i+j\,(\mathrm{mod}\,n)}$. Let $A = G$ and consider the onto homomorphism $\varphi : A^* \to G$ defined by $\varphi(a_i) = a_i$. (Observe that we could have taken a singleton set $A = \{a_1\}$.) The Cayley graph Γ_G of G is such that given two states a_i and a_j there is always an edge, labeled by $a_{j-i\,(\mathrm{mod}\,n)}$, from a_i to a_j. As above, we transform this graph into an automaton by distinguishing two states: let a_0 (the neutral element of G) be the initial state and choose a final state x. Denote by $\Gamma_G(x)$ the automaton obtained this way. Constructing, as indicated in Algorithm 3.1, the GTG $\mathcal{G}$ from $\Gamma_G(x)$, we obtain a GTG satisfying the conditions of the preceding lemma with $k = k' = 1$. Observe that when we eliminate a state, the GTG obtained still remains in the conditions of the preceding lemma unless it has only 3 states. The rational expression R obtained, using Algorithm 3.1, after the elimination of the n states, is such that $l(R) = 4^n$ and $l'(R) \geq 2^{2^{\cdot^{\cdot^{2}}}}$ (2 is raised $n-2$ times to the power 2).

We may thus conclude that the size of R, in terms of the number of symbols involved, grows exponentially as a function of $|G|$. The width of $\underline{\gamma}(R)$ grows even faster, for $n \geq 4$.

The exponential growth of rational expressions given automata for several complexity measures is also proved in [10]. In general, the rational expression produced applying Algorithm 3.1 to an automaton $\mathcal{A}$ is much larger than the equivalent rational expression produced applying the same algorithm to the minimal automaton of $\mathcal{A}$. So, in general, it is worth to start minimalizing the automaton before executing Algorithm 3.1. A polynomial time algorithm to minimalize an automaton may be found in [1].

5. Computing the closure with a modified algorithm

In the beginning of this section we give a more direct way to compute an expression for $\overline{\gamma(\varphi^{-1}(x))}$. Then we discuss the advantages of such a modification.

Let us start by proving the following lemma.

Lemma 5.1. *Let* $L = \bigcup_{i=1}^{r} u_{0,i} + u_{1,i}\mathbb{N} + \cdots + u_{p,i}\mathbb{N}$ *be a semilinear subset of* $\mathbb{Z}^n$. *Then* $\overline{\langle L \rangle} = \overline{L^*}$.

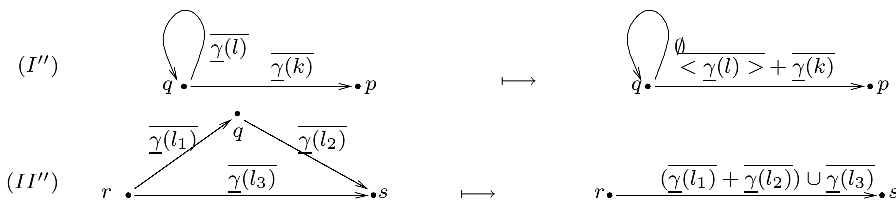
Proof. Observe that, as a consequence of Corollary 1.2, both sets are equal to $\sum_{1 \leq i \leq r}(u_{0,i}\mathbb{Z} + u_{1,i}\mathbb{Z} + \cdots + u_{p,i}\mathbb{Z})$. □

Next we give a new modification of Algorithm 3.1. The *input* and *output* are as in Algorithm 3.1. It has also the same steps (repeated here anyway), only the meaning of *destroy* and of *eliminate* are changed.

Algorithm 5.2.

- *Replace each label* a *in* $\mathcal{A}$ *by* $\gamma(a)$ *(or by* $\overline{\gamma(a)}$*, since* $\gamma(a)$ *is finite).*
- *Construct a generalized transition graph as in the first step of Algorithm 3.1.*
- *Execute the cycle given in the second step of Algorithm 3.1, taking into account the change of the meaning of* destroy *and of* eliminate *which should be clear from the following pictures.*

The changes should be clear from the following pictures.



Let $\mathcal{A}$ be a finite automaton. Denote by $\mathcal{L}(\mathcal{A})$ the language recognized by $\mathcal{A}$ and by $\mathcal{CL}(\mathcal{A})$ the semilinear set given by the output of Algorithm 5.2 when considered with input $\mathcal{A}$.

Proposition 5.3. *Using the notation just introduced, we have* $\mathcal{CL}(\mathcal{A}) = \overline{\underline{\gamma}(\mathcal{L}(\mathcal{A}))}$.

Proof. We will prove that the expressions produced using our algorithms are equivalent.

Let $\mathcal{G}_0$ (resp. $\mathcal{G}'_0$) be the generalized transition graph constructed in the first (resp. second) step of Algorithm 3.1 (resp. Algorithm 5.2). Denote by $\mathcal{G}_k$ (resp. $\mathcal{G}'_k$) the generalized transition graph obtained from $\mathcal{G}_0$ (resp. $\mathcal{G}'_0$) after the elimination of k states using Algorithm 3.1 (resp. Algorithm 5.2, applied to the same states).

The property

For any edge of $\mathcal{G}'_k$ *the label is of the form* $\overline{\underline{\gamma}(l)}$, *where* l *is the label of the corresponding edge of* $\mathcal{G}_k$

obviously holds if $k = 0$.

Suppose that k is less than the number of states of $\mathcal{A}$. The label of any edge of $\mathcal{G}'_{k+1}$ is of the form

$$\overline{\underline{\gamma}(l_1)} + \overline{\langle\underline{\gamma}(l_2)\rangle} + \overline{\underline{\gamma}(l_3)} \cup \overline{\underline{\gamma}(l_4)},$$

where the $\overline{\underline{\gamma}(l_i)}$ are labels of appropriate edges of $\mathcal{G}'_k$. The corresponding edges in $\mathcal{G}_k$ and $\mathcal{G}_{k+1}$ are respectively l_1, l_2, l_3, l_4 and $l_1 l_2^* l_3 \cup l_4$.

As a consequence of Lemma 5.1, Proposition 2.1 and Proposition 1.1, we have $\overline{\underline{\gamma}(l_1)} + \overline{\langle\underline{\gamma}(l_2)\rangle} + \overline{\underline{\gamma}(l_3)} \cup \overline{\underline{\gamma}(l_4)}$ is equivalent to $\overline{\underline{\gamma}(l_1 l_2^* l_3 \cup l_4)}$. The conclusion follows by induction. □

Next we discuss how we can, in practice, take advantage of the use of this algorithm to produce shorter equivalent semilinear expressions. (We make use of a lemma analogous to Lem. 3.2.)

Each time a new subgroup appears we compute a basis for it (given by a matrix in HNF). In this way we bound the size of the semilinear expression obtained.

We have to observe that the HNF of a matrix A, besides being computable in polynomial time, has its entries bounded in magnitude by a polynomial in $||A||$, the maximum of the magnitudes of the entries of A. See [24]. But in our algorithm, we have to compute repeatedly the Hermite normal forms of matrices whose entries contain the entries of matrices in HNF previously computed. Thus, the magnitude of the entries may grow exponentially, and so, the size of the rational expression obtained may grow exponentially in terms of the number of elements of the monoid.

When a new union occurs, we throw away the superfluous cosets. This way we may produce expressions whose width is smaller, but, contrary to what happened with the size, we have not found any reason to believe that one can produce, in general, an equivalent semilinear expression with smaller width and therefore to have a certain control over the width. To test the inclusion of a coset in another coset, say $a + H_1 \subseteq b + H_2$, it suffices to test whether the subgroup $\langle((a-b) + H_1) \cup H_2\rangle$ is equal to H_2. According to the comment following Proposition 1.3, this can be done by comparing bases given by a matrices in HNF for the subgroups in cause. Now observe that a basis for H_2 given by a matrix in HNF has already been computed, as was a finite set of generators for $\langle((a-b) + H_1) \cup H_2\rangle$. We

may now use one of the efficient algorithms implemented in GAP to compute a basis for $\langle((a-b)+H_1)\cup H_2\rangle$ given by a matrix in HNF.

The Algorithm 2.2 may now be improved as follows: the second step may be substituted by Algorithm 5.2 which is much more efficient in practice. The discussion above shows also that, in general, we obtain shorter expressions in this way and thus we may execute the third step more quickly.

Remark 5.4. Depending on the way the monoid is given, we could try to use other methods to compute a rational expression for $\varphi^{-1}(x)$ and do the same kind of modifications to produce directly an expression for $\overline{\gamma(\varphi^{-1}(x))}$. For example, if the monoid is given as the transition monoid of an automaton $\mathcal{A}$, other than its Cayley graph, $\varphi^{-1}(x)$ is simply the intersection of the languages recognized by the various automata $\mathcal{A}_q(x)$ obtained from $\mathcal{A}$ by considering, for each state q of $\mathcal{A}$, q as the initial state and qx as the final state. In order to compute the intersection, we could use the product of these automata. But the automaton obtained is big and may have no advantages relative to the Cayley graph. So, unless we find a more efficient way to compute intersections of languages, this method doesn't seem to be better. In addition, we have to start computing a list of the monoid elements, which comes for free when we compute the Cayley graph.

Remark 5.5. We might be tempted to use an algorithm due to Lenstra, Lenstra and Lovász [13] (or modifications of it (see [7, 19, 22])) to compute LLL-reduced bases and even reduced representatives for the cosets (see also [3]). We have to observe that the vectors in an LLL-reduced basis are not the shortest possible (for some norm), but such a basis can be computed in polynomial time. We do not believe that this would improve our algorithm, since the comparison of subgroups that we make several times could not be done just by testing equality between LLL-bases, as is the case with basis given by matrices in HNF.

6. Computing the Abelian kernel

We may now give an algorithm to compute the Abelian kernel of a finite monoid M.

Algorithm 6.1.

- *Let* $K=\emptyset$.
- *Execute the following cycle:*
  for $x\in M$ *do*
  - *use Algorithm 2.2 with the improvement referred before Remark 5.4 to test whether or not* $x\in K_{\mathbf{Ab}}(M)$*;*
  - *if* $x\in K_{\mathbf{Ab}}(M)$*, add* x *to* K.
- *Return* K.

It is clear, from the above section, that Algorithm 5.2, although much faster than the original Algorithm 3.1 followed by Proposition 2.1, is not a fast algorithm and therefore neither is Algorithm 6.1. Some improvements may easily be performed, as we explain in the sequel. The following is almost obvious.

Proposition 6.2. *$K_{\mathbf{Ab}}(M)$ contains the set of idempotents of M.*

Using it, we may leave the idempotents, which are easily computed, out of the cycle in Algorithm 6.1.

It is also easy to verify the following:

Proposition 6.3. *$K_{\mathbf{Ab}}(M)$ is a submonoid of M.*

So, if a subset $K \subseteq K_{\mathbf{Ab}}(M)$ is already computed, and, in the cycle of Algorithm 6.1 a new element x of the Abelian kernel is found, we may close for submonoid (*i.e.*, compute $(K \cup \{x\})^*$) obtaining, possibly, new elements in $K_{\mathbf{Ab}}(M)$. These new elements may, of course, be left out of the cycle. We observe that this process (although polynomial) has to be handled with care, since the computation of a submonoid with a large number of generators may take a lot of time. So, it is advisable to keep as few generators as possible each time one computes $(K \cup \{x\})^*$. (We have to remark that we have not found an efficient algorithm to compute a minimal set of generators of a monoid. Instead, we use low time consuming *ad-hoc* methods to throw away some superfluous generators.)

Another result that may be used with the same purposes is the following:

Proposition 6.4. *$K_{\mathbf{Ab}}(M)$ is closed under* weak conjugation *(i.e., if $s, t \in M$ are such that $sts = s$ or $tst = t$ and $u \in K_{\mathbf{Ab}}(M)$, then $sut \in K_{\mathbf{Ab}}(M)$).*

The proof is analogous to the corresponding result for the kernel case, which may be found, for example, in [25].

So, we may compute a list of conjugated pairs (*i.e.*, pairs (s, t) of elements of M such that $sts = s$ or $tst = t$) and close for weak conjugation and for submonoid each time a new element of the Abelian kernel is found. Since the list of conjugated pairs may be large, this (polynomial time) process must also be handled with care. In fact, in the course of the computations some of the conjugated pairs may become superfluous. For example, a pair corresponding to an idempotent is not needed, and the same happens with the pairs of elements already known to be in $K_{\mathbf{Ab}}(M)$.

The following result, the famous Type II theorem, gives a polynomial time algorithm to compute the kernel of a finite monoid and was proved in [2, 20].

Theorem 6.5. *The kernel of a finite monoid M is the smallest submonoid of M closed for weak conjugation.*

It results trivially from the definition, but results also from Proposition 6.3, Proposition 6.4 and Theorem 6.5, that the Abelian kernel of a finite monoid contains the kernel of that monoid. So, we could start the search for new elements in the Abelian kernel from the kernel of that monoid. So, fast algorithms to compute the kernel would be useful for us. (We observe that the algorithm given by Th. 6.5 suffers from the problems with the number of generators and conjugated pairs referred above.)

Fortunately, for some classes of finite monoids, the computation of the kernel may be executed more rapidly than using Theorem 6.5. For example, it is well-known that the kernel of a monoid whose idempotents commute consists of its idempotents. Computing the idempotents of a finite monoid and testing whether

they commute is fast. Among the examples of monoids whose idempotents commute are the, largely studied, inverse monoids.

In [23] a fast geometric method for computing the kernel of a finite monoid is given and may be a good alternative, but as far as the author knows, no computer program has yet been written for this algorithm.

To conclude, we summarize this discussion proposing the following algorithm to compute the Abelian kernel of a finite monoid M.

Algorithm 6.6.

- *Compute the kernel K of M and let $L = \emptyset$. (L stands for the set of elements that we already know to be outside the Abelian kernel.)*
- *Execute the following cycle:*
  While $M \setminus (K \cup L) \neq \emptyset$,
  - *choose $x \in M \setminus (K \cup L)$ and use Algorithm 5.2 to test whether or not $x \in K_{\mathbf{Ab}}(M)$;*
  - if $x \notin K_{\mathbf{Ab}}(M)$, then L *becomes* $L \cup \{x\}$, else K *becomes the submonoid closed under weak conjugation generated by* $K \cup \{x\}$.
- *Return K.*

7. Comments

The algorithm described in this paper is being implemented in GAP [26] by the author. It uses algorithms to produce normal forms of matrices that, as observed earlier, are already implemented in GAP. Of course, we are also using the facilities already available in GAP to work with semigroups.

The implementation process began when the author was visiting LIAFA at the University of Paris 7, using GAP3 [21] and the package Monoid [14]. During this phase we used monoids of injective partial transformations, which were being studied by V. H. Fernandes, to perform some tests. With the original algorithm we were unable to compute Abelian kernels of monoids with more than 30 elements, while using the improvements presented we managed to compute the Abelian kernel of monoids with about 4000 elements. It took a couple of days in a 550 MHz machine with 512Mb RAM memory. The computation of several examples gave the necessary intuition which led to a joint work with Fernandes [9].

I wish to thank Jorge Almeida for introducing me to the problem of computing the Abelian kernel of a finite monoid and for many helpful discussions and comments. I wish also to thank Jean-Eric Pin for encouraging me to implement the algorithm and for his suggestion to consider commutative variables. Many thanks also to Ben Steinberg for many helpful discussions and comments. I wish also to thank the anonymous referee for his/her valuable sugestions.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analisys of Computer Algorithms.* Addison Wesley (1974).

[2] C.J. Ash, Inevitable graphs: A proof of the type II conjecture and some related decision procedures. *Internat. J. Algebra and Comput.* **1** (1991) 127-146.

[3] L. Babai, On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica* **6** (1986) 1-13.

[4] J. Berstel, *Transductions and Context-free Languages.* Teubner, Stuttgart (1979).

[5] J. Brzozowski and E. McCluskey, Signal flow graph techniques for sequential circuit state diagrams. *IEEE Trans. Electronic Comput.* **12** (1963) 67-76.

[6] T.J. Chou and G.E. Collins, Algorithms for the solution of systems of linear Diophantine equations. *SIAM J. Comput.* **11** (1982) 687-708.

[7] H. Cohen, *A Course in Computational Algebraic Number Theory.* GTM, Springer Verlag (1993).

[8] M. Delgado, Abelian pointlikes of a monoid. *Semigroup Forum* **56** (1998) 339-361.

[9] M. Delgado and V.H. Fernandes, Abelian kernels of some monoids of injective partial transformations and an application. *Semigroup Forum* **61** (2000) 435-452.

[10] A. Ehrenfeucht and P. Zeiger, Complexity measures for regular expressions. *J. Comput. System Sci.* **12** (1976) 134-146.

[11] V. Froidure and J.-E. Pin, Algorithms for computing finite semigroups, edited by F. Cucker and M. Shub. Berlin, *Lecture Notes in Comput. Sci.* (1997) 112-126.

[12] K. Henckell, S. Margolis, J.-E. Pin and J. Rhodes, Ash's type II theorem, profinite topology and Malcev products: Part I. *Internat. J. Algebra and Comput.* **1** (1991) 411-436.

[13] A.K. Lenstra, H.W. Lenstra Jr. and L. Lovász, Factoring polynomials with rational coefficients. *Math. Ann.* **261** (1982) 515-534.

[14] S. Linton, G. Pfeiffer, E. Robertson and N. Ruškuc, *Monoid Version 2.0.* GAP Package (1997).

[15] O. Matz, A. Miller, A. Pothoff, W. Thomas and E. Valkema, *Report on the program AMoRE.* Tech. Rep. 9507, Christian Albrechts Universität, Kiel (1995).

[16] J.-E. Pin, *Varieties of Formal Languages.* Plenum, New-York (1986).

[17] J.-E. Pin, A topological approach to a conjecture of Rhodes. *Bull. Austral. Math. Soc.* **38** (1988) 421-431.

[18] J.-E. Pin and C. Reutenauer, A conjecture on the Hall topology for the free group. *Bull. London Math. Soc.* **23** (1991) 356-362.

[19] M. Pohst and H. Zassenhaus, *Algorithmic Algebraic Number Theory.* Cambridge University Press (1989).

[20] L. Ribes and P.A. Zalesskiĭ, On the profinite topology on a free group. *Bull. London Math. Soc.* **25** (1993) 37-43.

[21] M. Schönert *et al.*, *GAP – Groups, Algorithms, and Programming*, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule. Aachen, Germany, fifth Edition (1995).

[22] C.C. Sims, *Computation with Finitely Presented Groups.* Cambridge University Press (1994).

[23] B. Steinberg, Finite state automata: A geometric approach. *Trans. Amer. Math. Soc.* **353** (2001) 3409-3464.

[24] A. Storjohann, *Algorithms for matrix canonical forms*, Ph.D. thesis. Department of Computer Science, Swiss Federal Institute of Technology (2000) `http://www.scg.uwaterloo.ca/ astorjoh/publications.html`

[25] B. Tilson, *Type II redux*, edited by S.M. Goberstein and P.M. Higgins. Reidel, Dordrecht, *Semigroups and their applications* (1987) 201-205.

[26] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.2.* Aachen, St Andrews (1999), `http://www-gap.dcs.st-and.ac.uk/ gap`

[27] S. Willard, *General Topology.* Addison Wesley (1970).

Communicated by J.-E. Pin.
Received March 6, 2001. Accepted March 1, 2002.