Analysis of a simple reversible language (previous version of TCS **290** 3, 2003)

Armando B. Matos [a] *

[a]DCC-FC & LIACC, Universidade do Porto

Rua do Campo Alegre 823, 4150-180 Porto, Portugal

Very simple reversible programming languages can be useful for the study of reversible transformations. For this purpose we characterise the language SRL, a simple reversible language, and analyse its properties. The language SRL is similar to the "loop" languages that have been used by several authors to characterise the set of primitive recursive functions. In a sense, the class of SRL-definable transformations is the image of the class of primitive recursive functions in the "reversible world". There are however important differences: SRL has domain $\mathbb{Z}$ instead of $\mathbb{N}$, programs written in SRL are always reversible. The reversibility of linear homogeneous SRL programs is related to the fact that the corresponding set of matrices has the algebraic structure of a group. We show that such programs implement exactly the linear transformations belonging to the group of integer positive modular matrices, while in ESRL, an extended version of SRL, the set of transformations that can be implemented by linear homogeneous programs corresponds exactly the group of integer modular matrices.

Keywords. Loop languages, group of modular matrices, reversibility.

*The work presented in this paper has been partially supported by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundao para a Cincia e Tecnologia*, *Programa POSI* and *Programa PRAXIS*

1. Introduction

In this paper we study the properties of a very simple reversible programming language, similar to the LOOP($\mathbb{N}$) language which has been used by several authors ([9, 10, 13, 11]) to characterise the class of primitive recursive functions.

Commonly used languages are not reversible; for instance, any language containing the assignment instruction is not reversible. In a general purpose language irreversible computations can be simulated by reversible computations at the cost of extra space and time ([1, 7]).

Here we use another approach and restrict ourselves to programming languages that are inherently reversible: for *any* program P there is a program P^{-1} such that the composition P; P^{-1} is the identity. The program P^{-1} is called the reverse (or inverse) of P. A very simple algorithm transforms P into P^{-1}.

These languages are usually quite restricted but have the advantage that it is immediate to define the reverse of a program; a program and its reverse typically run in exactly the same time and uses exactly the same amount of memory[2], hence, in these languages, one-way functions, a basic ingredient of modern public key cryptography, do not exist.

This paper is organised as follows. After some preliminaries in Section 2, we define in Section 3 SRL($\mathbb{Z}$), a simple reversible language. Then we present in Section 4 several examples[3]. Programs where the maximum nesting of "for" loops is 1 are called *linear*. In Section 5 we study in detail these programs and show that a linear transformation is implementable in SRL($\mathbb{Z}$) iff the associated matrix is positive modular (Theorem 1). It is also shown (Theorem 2) that it is decidable whether two linear SRL programs are equivalent. For ESRL, an extended version of SRL, all modular matrix transformations can be implemented (Section 6). A SRL($\mathbb{Z}$) program can be seen as a parametric transformation as well as a numeric computation. Finally in Section 7 we summarise the main contributions of this work and mention some areas for future research.

[2]Reversible Turing machines (introduced by Lecerf [6] and independently later by Bennett [1, 2], see also the formalisation in [8]) can be considered languages of this kind.

[3]All the examples presented in this paper have been tested by a simulator written in Prolog.

2. Preliminaries

$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ and $\mathbb{C}$ denote respectively the set of nonnegative integers, the set of integers, the set of real numbers and the set of complex numbers. The determinant of a square matrix M is denoted by $|M|$. Following [4] we say that an integer square matrix is *modular* if its determinant is ± 1 and *positive modular* if its determinant is 1. Integer matrices have were studied by several authors since about 1860. It is well known that, for each $n > 0$, the following sets have the algebraic structure of a group relatively to the operation of matrix multiplication: the set of unitary (complex) matrices, the set of regular (nonsingular) real matrices, the set of integer positive modular matrices, the set of integer modular matrices, the set of permutation matrices, the set of even permutation matrices. It follows for instance that the inverse of an integer positive modular is still an integer matrix with determinant 1.

We will study register languages. The initial and final (after the execution of a program) values of the register number i will be respectively denoted by r_i and ${r_i}'$; lowercase letters a, b,..., r, s will also be used to denote registers. The vector of registers used by a program is denoted by $\mathcal{R}$ and the transformation corresponding to a program P is also denoted by P, so that $P(\mathcal{R})$ is the transformed vector of registers. The assignment of an expression E to register r is denoted by $r \leftarrow E$. The reverse (or inverse) of a program P is P^{-1}; the identity (or null program) is denoted by i_D. $I_1; I_2$ and I^n denote respectively the composition of instruction I_1 with instruction I_2 and the composition of n (identical) instructions I. Two programs are *equivalent*, $P \equiv Q$, if the register transformation associated with P and Q are identical, that is $P(\mathcal{R}) = Q(\mathcal{R})$.

3. The languages LOOP($\mathbb{N}$) and SRL($\mathbb{Z}$)

The class of primitive recursive functions has been characterised by several authors as the set of functions that can be implemented in an appropriate programming language, see for instance the languages used for that purpose in [9, 10, 13, 11]. All these languages are equivalent; in particular they only include loop instructions such that *the number of repetitions of the corresponding block of code is fixed in advance*; this implies in particular that every program halts, so that every function that can be defined in those languages is total; of course, not every total function is primitive recursive; in fact, by an easy and well known diagonalisation argument we can show that no finitely described model of computation can define exactly the class of total functions. We will begin by defining LOOP($\mathbb{N}$), a very small such language. Later we define and study SRL($\mathbb{Z}$) (simple reversible language), a reversible language very similar to LOOP($\mathbb{N}$). They will turn out to have very different properties.

3.1. The language LOOP($\mathbb{N}$)

Let us briefly characterise the LOOP($\mathbb{N}$) language. The memory is an unbounded set of registers capable of storing an arbitrary non-negative integer. Thus the domain is $\mathbb{N}$. Registers $r_1 \ldots r_k$ are the input arguments, the register r_0 is the output. Every non input register is assumed to contain 0 at the beginning of the computation. The instructions are: "`INC` r" (increment register r by one), "`DEC` r" (decrement register r by one, using the convention $0 - 1 = 0$), "`FOR` r $\{P\}$" (execute r times the LOOP program P which can not change r).

Clearly every function implemented by a LOOP($\mathbb{N}$) program is total. In fact, as already mentioned, the class of functions that can be programmed in the language LOOP($\mathbb{N}$) is exactly the class of primitive recursive functions. The LOOP($\mathbb{N}$) language is not reversible: if after the execution of the instruction "`DEC` r", the register r contains 0, we cannot deduce the initial value of r (it can be either 0 or 1).

3.2. SRL($\mathbb{Z}$): a simple reversible language

In this paper we study SRL($\mathbb{Z}$), a *reversible* language similar to LOOP($\mathbb{N}$). Our language differs from LOOP($\mathbb{N}$) in two main aspects: (i) each register can contain an arbitrary (possibly negative) integer, that is, the domain is $\mathbb{Z}$ and (ii) nothing is assumed about the initial contents of the registers. These design decisions have to do with the reversibility of the language; in particular, if immediately after the execution of a program P (without changing any register), the reverse program P^{-1} is executed, the memory contents is exactly as in the beginning, $P\,;\,P^{-1}(\mathcal{R}) = \mathcal{R}$.

Henceforth we assume that the domain of the two languages is known and write LOOP and SRL instead of LOOP($\mathbb{N}$) and SRL($\mathbb{Z}$) respectively. We now describe the syntax and semantics of SRL. Then we show that every program in SRL can be reversed by a very simple procedure.

The *domain* is the set $\mathbb{Z}$ of integers. The memory consists of registers r_0, $r_1 \ldots$

A SRL *program* is a finite sequence of instructions of the form:

Increment: `INC` r_i. Semantics: $r_i \leftarrow r_i + 1$.

Decrement: `DEC` r_i. Semantics: $r_i \leftarrow r_i - 1$.

Loop: `FOR` r_i $\{P\}$. Semantics: If $r_i \geq 0$, the SRL program P is executed r_i times; If $r_i < 0$, the SRL program P is executed $-r_i$ times[4]. The value of r_i cannot be changed by the instruction; it can only appear in P as the register of a nested `FOR`.

Composition: If P_1 and P_2 are programs, $P_1; P_2$ is also a program.

The *reverse* P^{-1} of a program P is defined inductively as follows: (i) the reverse of "`INC` r_i" is "`DEC` r_i", (ii) the reverse of "`DEC` r_i" is "`INC` r_i", (iii) the reverse of "`FOR` r_i $\{P\}$" is "`FOR` r_i $\{P^{-1}\}$", (iv) the reverse of "$P_1; P_2$" is "${P_2}^{-1}; {P_1}^{-1}$". Using the semantics of SRL it easy to see that P^{-1} is in fact the reverse of P, that is, for every program P we have $P; P^{-1} = P^{-1}; P = i_D$.

We will also use the language ESRL($\mathbb{Z}$), extended an extended version of SRL($\mathbb{Z}$): there is an additional instruction that changes the sign of a register, "$r \leftarrow -r$". It is clear that the language ESRL($\mathbb{Z}$) is also reversible, the reverse of an instruction "$r \leftarrow -r$" being itself.

3.3. On the relationship between primitive recursive functions and functions implementable in SRL

The languages LOOP and SRL are very similar so that we expect that the classes of primitive recursive and SRL-definable functions may have interesting similarities. There are however important differences: (i) the domains are different ($\mathbb{N}$ and $\mathbb{Z}$ respectively), (ii) PR functions are usually not reversible, (iii) a "SRL-definable function" should be seen not as a function but rather as a register transformations, being similar to a gate as used for instance in quantum computation (which are also reversible). Obviously no one

[4]This corresponds to the following interpretation which is crucial for the reversibility of a SRL program: if $n < 0$, executing n times a certain program is the same thing as executing $-n$ times the reverse of the program.

of the classes includes the other. However there may be interesting properties relating relating the two classes but as far as we know, that relationship has not yet been studied in detail[5].

4. Some programs in SRL

We now consider a few examples of programs in SRL showing that functions like the sum, difference and product can be easily implemented if we assume that the initial values of all auxiliary registers (used, non-input registers) is 0. However, in general, SRL computations should rather be seen as register transformations.

Example 1 *The following program computes $a+b$:* `FOR b {INC a}`*; the transformation is $a'=a+b$, $b'=b$. Linear computations may be represented by matrices (most programs do not correspond to linear transformations). For this case, the matrix is* $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$*. The reverse of this program can be found by two methods: by reversing the program or by finding the inverse matrix:* `FOR b {DEC a}`*; the corresponding transformation is $a'=a-b$, $b'=b$ and the matrix is* $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$.

Example 2 *The product of a and b can be computed by the following program* if we assume that the initial values of c and d are 0*:* `FOR a {FOR b {INC c}}`*; the transformation is $a'=a$, $b'=b$, $c'=c+ab$. The reverse program is "*`FOR a {FOR b {DEC c}}`*":*

$$\begin{tikzcd}
a \arrow[r, "x"] & a \arrow[r, "x"] & a \\
b \arrow[r, "y"] & b \arrow[r, "y"] & b \\
c \arrow[r, "z+xy"] & c+ab \arrow[r, "z-xy"] & (c+ab)-ab=c
\end{tikzcd}$$

Example 3 *It is possible to swap the values in registers a and b using an additional variable c:*

```
FOR a {INC c}; FOR c {DEC a}; FOR b {INC a}; FOR a {DEC b};
FOR c {INC b}; FOR b {DEC c}; FOR c {DEC a}; FOR c {INC b;INC b}
```

[5]As an example of a property relating the two languages, it is not difficult to see that SRL computations can be simulated in LOOP using the following bijection $f:\ \mathbb{Z}\rightarrow\mathbb{N}$: $f(n)=n$ if $n\geq 0$ and $f(n)=-2n-1$ otherwise. This corresponds to a (somewhat contrived) reversible sub-language of LOOP that simulates reversible computations in the domain $\mathbb{Z}$.

The transformation is $a' = b$, $b' = a$, $c' = -c$ corresponding to the matrix $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. *The values of a and b were swapped; however c changed sign. It is also possible to swap two disjoint pairs of registers. However, swapping only a pair of registers (without changing any other) is not possible.*

5. Linear programs

In the previous Section (see Example 3) we mentioned that it is not possible to swap the values of two registers without changing other registers – for instance swapping two other registers or changing the sign of another. We will now show that this inability has a reason: a single swap (or a single change of sign) is not possible with a *linear* SRL program. This follows from Theorem 1, the main result of this paper, which characterises the set of transformations that can be implemented by a linear SRL program.

A SRL program is *linear* if every instruction has one of the following forms: "`INC a`", "`DEC a`", "`FOR a {INC b}`" and "`FOR a {DEC b}`". An equivalent definition is: a program is linear if the `FOR` loops cannot be nested.

A ESRL program is linear if every instruction has one of the following forms: "`INC a`", "`DEC a`", "$\mathtt{a} \leftarrow -\mathtt{a}$", "`FOR a {INC b}`" and "`FOR a {DEC b}`". Notice that the instructions of the form "$\mathtt{FORa}\,\{\mathtt{b} \leftarrow -\mathtt{b}\}$" are not allowed.

5.1. Transformations that can be implemented by linear SRL programs

We will prove that linear SRL programs implement exactly linear transformations corresponding to positive modular matrices. We begin with a few Lemmas. The proof of the first three is easy and is not given.

Lemma 1 *Let $r_1, \ldots, r_n$ be the register used by a linear SRL program P. The transformation associated with P is linear, that is it has the form $r_i = t_{ij}r_j + c_i \quad (i = 1, 2, \ldots, n)$ where all $t_{i,j}$ and c_i are integers.*

Denote by $\mathcal{R}$ and $\mathcal{R}'$ respectively the column vectors $[r_1 \cdots r_n]^t$ and $[r'_1 \cdots r'_n]^t$. Program P implements a transformation $\mathcal{R}' = T_P\mathcal{R} + C$. We say that a transformation

$f(\mathcal{R}) = T_P\mathcal{R} + C$ is *feasible* if it corresponds to some linear SRL program.

Lemma 2 *If the transformation $f(\mathcal{R}) = T_P\mathcal{R} + C$ is feasible, then for every other constant column matrix C', the transformation $f(\mathcal{R}) = T_P\mathcal{R} + C'$ is also feasible.*

Lemma 3 *If a transformation is feasible, there is a program P that implements it and has all* `FOR` *instructions at the beginning and all* `INC` *and* `DEC` *instructions at the end.*

Lemma 4 *If a linear program only has at the outermost level* `FOR` *instructions, the corresponding transformation is homogeneous.*

In view of Lemmas 2 and 4 we will discuss only linear programs having only `FOR` instructions at the outermost level. Let us call *homogeneous* to these programs.

Lemma 5 *The transformation matrix associated with an homogeneous linear program is positive modular.*

Proof. Easy from the fact that the matrices associated with "`FOR a {INC b}`" and "`FOR a {DEC b}`" are respectively $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$. Both have determinant 1. $\diamond$

Recall example 3. Swapping two registers and changing the sign of a register correspond respectively to the matrices $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and $[-1]$. In both cases the determinant is -1; in view of Lemma 5 linear programs cannot implement these transformations.

We will now see that the converse of Lemma 5 is also true as expressed in the following result whose first part is proved in Appendix A (the proof for the equivalence between ESRL and modular transformations is similar).

Theorem 1 *The set of functions implementable by linear SRL programs is exactly the set of linear transformations $f(\mathcal{R}) = M\mathcal{R} + C$ where M is positive modular. The set of functions implementable by linear ESRL programs is exactly the set of linear transformations $f(\mathcal{R}) = M\mathcal{R} + C$ where M is modular.*

5.2. The equivalence of linear SRL programs is decidable

An important question related to SRL programs is whether it is decidable if two programs implement the same transformation. For linear SRL programs the answer is yes; this is similar to the case of LOOP programs ([9]).

Theorem 2 *It is decidable whether two linear SRL programs implement the same register transformation.*

Proof. Let P and P' be two linear SRL programs, let $\mathcal{R}$ be the column vector of the set of registers used in either P or P'; denote by $M\mathcal{R}+C$ and $M'\mathcal{R}+C'$ the corresponding transformations; use Lemma 3 to obtain equivalent programs where all `FOR` instructions are at the beginning. The matrices M and M' can be computed as the product of the matrices corresponding to the `FOR` instructions and the independent coefficient column matrices C and C' are easily obtained as explained in the proof of Lemma 2. Clearly the programs P and P' are equivalent if $M=M'$ and $C=C'$. ◇

6. Some notes on SRL programs and group theory

In this Section we consider again general, not necessarily linear, SRL programs. Recall the definition of SRL program equivalence given in Section 2. Let us denote by $[P]$ the equivalence class associated with program P. Consider the set $\mathcal{P}$ of those classes of equivalence. We first define an algebraic structure $(\mathcal{P},\odot)$ where "$\odot$" denotes the composition of transformations (not the composition of programs) and show that it is a group.

If $C_1=[P]$ and $C_2=[Q]$ are two equivalence classes, then define $C_1\odot C_2=[P;Q]$. The operation "$\odot$" is associative due to the associativity of program composition. Let $I=[i_D]$ be the identity class. For each program P there is a reverse program P^{-1} such that $[P;P^{-1}]=[i_D]$.

For linear homogeneous programs P using no more than the first n registers, the group $G_1=(\mathcal{P},\odot)$ is isomorphic to the group of $n\times n$ positive modular matrices with the operation of matrix multiplication (in particular, it is well known that the inverse of such a matrix still has determinant 1 and all its entries are integer). For ESRL, linear homogeneous programs P we have the following result whose proof is similar to the proof of Theorem 1 given in Appendix A.

Theorem 3 *The set of transformations implemented by linear homogeneous ESRL programs is isomorphic to the group of modular $n\times n$ matrices.*

A permutation of the registers is a particular case of a linear homogeneous transformation. For SRL we get a group isomorphic to the group of $n \times n$ even permutation matrices and for ESRL we get a group isomorphic to the group of all $n \times n$ permutation matrices.

More generally, consider a class of total reversible programs and say that two total programs P and P' are equivalent and write $P \equiv P'$ if they change the memory in the same way (all inputs are parametric, the programs are executed with no "preparation" of the register contents). In any such reversible programming language the classes of the equivalence relation "$\equiv$" have the algebraic structure of a group.

7. Conclusions and future work

We have defined SRL, a simple reversible language with domain $\mathbb{Z}$ and studied the algebraic transformations that can be implemented with the linear part of his language.

We have shown that linear SRL programs implement exactly the linear transformations $M\mathcal{R} + C$ where M is positive modular. This implies that not all linear reversible transformations of the registers are possible in linear SRL; in ESRL, the extended version of SRL, all linear transformations $A\mathcal{R} + C$ where A is modular are possible.

General SRL programs implement a much richer class of reversible transformations. In this case, and as far as we know, no detailed study has yet been done. Thus many problems remain open; it would be for instance interesting to characterise the class of transformations that can be implemented with general SRL and ESRL programs (including the hierarchy corresponding to the bounding of the maximum nesting of `FOR` loops) and to relate the classes of SRL- and ESRL-definable functions with the class of primitive recursive functions.

REFERENCES

1. Charles H. Bennett, *Logical reversibility of computation*, IBM Journal of Research and Development, **6**, pp 525–532, 1973.
2. Charles H. Bennett, *The thermodynamics of computation – a review*, International Journal of Theoretical Physics, **21**, pp 905–940, 1982.
3. David Deutsch, *Quantum theory, the Church-Turing principle, and the universal quan-*

tum computer, In Proceedings of the Royal Society of London, **400**, pp 97–117, London, 1985.

4. Loo Keng Hua, *Introduction to Number Theory*, Springer-Verlag, 1982.
5. Richard E. Korf, *Inversion of applicative programs*, Proceedings of the IJCAI 1981, pp 1007–1009.
6. Y. Lecerf, *Machines de Turing reversibles. Recursive insolubilit en $n \in \mathbb{N}$ de l'quation $u = \theta^n$ ou θ est un isomorphisme de codes*, Comptes Rendus, **257**, pp 2597–2600, 1963.
7. Ming Li and John Tromp and Paul Vitnyi, *Reversible simulation of irreversible computation*, Physica D, **120**, pp 168–176, 1998.
8. Armando B. Matos, *A Turing machine model suitable for the characterisation of determinism and reversibility*, LIACC & Departamento de Cincia de Computadores, FCUP, 1999.
9. A. R. Meyer and D. M. Ritchie, *The complexity of loop programs*, Proceedings of 22nd National Conference of the ACM, pp 465–469, 1967.
10. A. R. Meyer and D. M. Ritchie, *Computational complexity and program structure*, IBM Research Report RC 1817, 1967.
11. R. Sommerhalder and S.C. van Westrhenen, *The Theory of Computability: Programs, Machines, Effectiveness and Feasibility*, Addison Wesley, International Computer Science Series, 1988,
12. T. Toffoli and Norman Margolus, *Invertible cellular automata: a review*, Physica D **45**, pp 229–253, 1993.
13. D. Tsichritzis, *The equivalence problem of simple programs*, Journal of the ACM, **17** (4), pp 729–738, 1970.

A. Appendix: proof of Theorem 1

One of the directions of Theorem 1 is given by Lemma 5. So we have only to prove that every homogeneous transformation with determinant 1 can be implemented in SRL. In this Appendix by *program* and *transformation* we mean respectively a linear homogeneous program and a linear homogeneous transformation. By *triangular* matrix we mean a lower (or upper) triangular square matrix, that is, a square containing only zeros to the right and above (or to the left and below) of the main diagonal.

A.1. A schema of the proof

Let M be a given positive modular matrix and let P be a program that implements M ; at this stage we do not know whether P exists; we will prove that it does.
In A.4 we will show how to define for every M a SRL program P' and a matrix M_t such that, if P exists, M_t, the matrix associated with "$P; P'$", is triangular; in a diagram: $M \ \rightarrow \ M_t, P'$.
In A.5 we prove that every positive modular triangular matrix can be implemented in SRL. Thus M_t is implemented by some (known) program D.

Let M' be the matrix associated with P'. From M we can get M' and then M_t such that $M'M = M_t$ (recall[6] that$M'M$ corresponds to P; P'); the matrices M' and M_t can be implemented in SRL respectively by the programs P' and T; so $M = M'^{-1}M_t$ can also be implemented in SRL namely by the program $P = T$; P'^{-1} which of course, is also positive modular. Before presenting the details of the proof, we give in A.2 a simple example which may be helpful.

The construction described in this proof can be seen as a definition of a program in SRL whose inverse transforms a given integer matrix (with determinant 1) into integer Hermite normal form, by a series of elementary transformations, see for instance [4].

A.2. A simple example of the proof construction

Let us define a program that implements the modular matrix $\begin{bmatrix} 3 & 5 \\ 1 & 2 \end{bmatrix}$ that corresponds to the register transformation $a' = 3a+5b$, $b' = a+2b$. We can triangularise the matrix M by the following sequence of transformations: (i) subtract line 2 from line 1, (ii) subtract

[6]The usual conventions for function composition and program concatenation are opposite, that is, if f and g are functions and P and Q are programs, then fg (function composition) means "f after g" while P; Q (program concatenation) means "Q after P". Then, if the register transformation associated with P and Q are respectively f_P and f_Q, the transformation associated with P; Q if $f_Q f_P$.

line 2 from line 1, (iii) subtract line 1 from line 2:

$$\begin{bmatrix} 3 & 5 \\ 1 & 2 \end{bmatrix} \to \begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \to \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \to \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Subtracting line 2 from line 1 corresponds to

$$\texttt{FOR}\ b\ \{\ \texttt{DEC}\ a\,\} \quad \leftrightarrow \quad \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \quad \leftrightarrow \quad \begin{cases} a' = a - b \\ b' = b \end{cases}$$

and similarly for the operation "subtract line 1 from line 2".

Thus we have $P' = \overbrace{\texttt{FOR}\ b\ \{\ \texttt{DEC}\ a\,\};}^{1}\ \overbrace{\texttt{FOR}\ b\ \{\ \texttt{DEC}\ a\,\};}^{2}\ \overbrace{\texttt{FOR}\ a\ \{\ \texttt{DEC}\ b\,\};}^{3}$. In A.4 it is explained a general algorithm to get P' from M. The transformation corresponding to P' is the product (in reverse order) of the matrices correspondint to its 3 parts:

$$M' = \overbrace{\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}}^{3} \times \overbrace{\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}}^{2} \times \overbrace{\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}}^{1} = \begin{bmatrix} 1 & -2 \\ -1 & 3 \end{bmatrix}$$

It can be verified that $M'M$ is in fact triangular

$$M'M = M_t = \begin{bmatrix} 1 & -2 \\ -1 & 3 \end{bmatrix} \times \begin{bmatrix} 3 & 5 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

and easily implemented by a SRL program, as explained in A.5: $T = \texttt{FOR}\ b\ \{\ \texttt{INC}\ a\,\}$. Notice that $P'^{-1} = \overbrace{\texttt{FOR}\ a\ \{\ \texttt{INC}\ b\,\};}^{3'}\ \overbrace{\texttt{FOR}\ b\ \{\ \texttt{INC}\ a\,\};}^{2'}\ \overbrace{\texttt{FOR}\ b\ \{\ \texttt{INC}\ a\,\};}^{1'}$ thus we get the desired program, $P = T;\ P'^{-1}$: `FOR` b { `INC` a } `FOR` a { `INC` b } `FOR` b { `INC` a } `FOR` b { `INC` a } which implements the unimodular matrix M as can be easily verified by considering in sequence the transformation of the registers a and b by each of the 4 lines of the previous program: $a' = a+b$, $b' = a'+b = a+2b$, $a'' = a'+b' = 2a+3b$ and $a''' = a''+b' = 3a+5b$, which is the desired transformation (where a''' and b' are the transformed values of respectively a and b).

A.3. Proof: Preliminaries

To prove the following result it is enought to consider the composition:
$\{c \leftrightarrow d;\ a' \leftarrow -a\}; \{c \leftrightarrow d;\ b' \leftarrow -b\}$ (see Example 3).

Lemma 6 *Let a and b be two distinct registers. The transformation $a' = -a$, $b' = -b$ can be implemented in SRL.*

Lemma 7 *Let M be the transformation matrix that corresponds to a linear homogeneous SRL program. If we replace in M a line by the sum of itself with another line, the resulting matrix can also be implemented in SRL. If we replace in M a line by the difference between itself and another line, the resulting matrix can also be implemented in SRL.*

Proof. Suppose that P implements a linear homogeneous transformation which is, for registers a and b, the following $a' = \Sigma m_{a,i} r_i$, $b' = \Sigma m_{b,i} r_i$. If after P we add the instruction "`FOR` b {`INC` a}", we get a program that implements the transformation $a'' = \Sigma(m_{a,i} + \Sigma m_{b,i}) r_i$, $b'' = \Sigma m_{b,i} r_i$. The matrix that corresponds to this transformation can be obtained by replacing in M the a-line by the sum of a-line with b-line. The other case (difference of two lines) is similar, just put instead the instruction "`FOR` b {`DEC` a}" after P. ◇

A.4. Proof: finding a SRL program that triangularises a transformation

We now show that, if there is a program P that implements the matrix M, there is also a program P' such that P; P' implements a triangular matrix M_t. This program P' and the diagonal matrix M_t are the output of an algorithm which is described below. It should be noted that such an algorithm is not intended to be written in SRL. It is just a method to find P' and M_t from M.

Given M, the program P' will be defined as a finite sequence of identical sections, each transforming the corresponding matrix so as to make an entry equal to zero.

The method is now described. Let M be an $n \times n$ integer matrix with entry $m_{i,j}$ on line i, column j. In the following algorithm the sentence "make the entry $m_{...} = 0$" corresponds to the algorithm "`make_zero`" described below.

```
Algorithm triangularise($M$):
for $i = n$ downto 0
  for $j = 0$ to i-1
    Use lines $i$ and $j$ to make the entry $m_{j,i} = 0$
```

Lines of the matrix will be added or subtracted as described by the following algorithm in order to get $b = 0$. The elements a and b correspond in algorithm "`triangularise`" respectively to the diagonal element element $d_i = m_{i,i}$ and to the element $m_{j,i}$.

```
Algorithm make_zero($a, b$)
while $a \neq 0 \wedge b \neq 0$
  reduce $|a|$ or $|b|$ by subtracting or adding
  one of the numbers $a$, $b$ to the other
if $a = 0$, make $a \leftarrow b$; $b \leftarrow 0$
```

The additions and subtractions of numbers will in fact correspond to additions and subtractions of the corresponding matrix lines.It is easy to see that this program always halts. In terms of the matrix the last line of the algorithm, "$a \leftarrow b$; $b \leftarrow 0$" corresponds to: (i) Sum the b-line to the a-line, (ii) Subtract a-line from the b-line.

Example of Algorithm `make_zero` for the initial values: $a = 6$, $b = 2$:

Step:	1	2	3	4	5	6
a	6	4	2	0	2	2
b	2	2	2	2	2	0

From Lemma 7 we know that we can implement in SRL the operations of adding one line to another and subtracting one line from another. The following result follows.

Lemma 8 *If M is the matrix corresponding to the linear homogeneous transformation implemented by a program P, there is a program P' such that the matrix corresponding to the composition P; P' is triangular.*

A.5. Proof: Triangular matrices with determinant 1 can be implemented in SRL

We now show that if the matrix of a transformation is triangular and has determinant one, the transformation can be implemented in SRL. Let M be a triangular matrix; as the determinant is 1, all the elements in the main diagonal are either 1 or -1 and the number of negative elements is even. We will now show through an example how to implement such a triangular matrix in SRL.

Consider the transformation $a' = a$, $b' = 2a - b$, $c' = -a + 3b - c$. The diagonal elements are 1, -1, -1. In order to remove the negative diagonal elements (there is an even number of them) we use Lemma 6 to get a program P that changes the sign of b and c (it uses but does not modify some auxiliary registers). We now have to implement the transformation $a' = a$, $b' = -2a + b$, $c' = a - 3b + c$. In general, using Lemma 6 an appropriate number of times, we get a matrix having only 1's on the main diagonal.

A transformation corresponding to a triangular matrix can be easily implemented in SRL as shown for the example:

```
FOR a {DEC b; DEC b}                         % b' = -2a + b
FOR b {DEC c; DEC c; DEC c}                  % c' =  6a -3b + c
FOR a {DEC c; DEC c; DEC c; DEC c; DEC c} % c' =   a -3b + c
```