Register reversible languages

(work in progress)

Armando B. Matos
armandobcm@yahoo.com

April 5, 2014

Topics discussed in this work (selected from the table of contents): the languages ⋆SRL - syntax, formal inversion and semantics - parametric composition - depth of a program - executing a program - non implementable transformations - examples of implementable transformations - partitions - program composition - simulation of recursive Boolean circuits - simulation by primitive recursive functions - a RAM machine that runs ⋆SRL - equivalence and commutativity - "in the language of group theory" - the order of a program - normal forms - execution time. Appendices (programs in Prolog): checking ⋆SRL programs - representation of some of programs presented in this work - interpreter code - assembly language interpreter with inversion.

Contents

Abstract

We begin by quoting Abramsky [Abr01]: "R. Landauer [Lan61] has demonstrated that it is only the logically irreversible operations in a physical computer that necessarily dissipate energy by generating a corresponding amount of entropy for every bit of information that gets irreversibly erased; the logically reversible operations can in principle be performed dissipation-free. At the basic level, however, matter is governed by classical mechanics and quantum mechanics, which are reversible."

In a not too distant future the miniaturization of circuits will reach the "power dissipation" physical barrier, which can (only?) be overcome by the usage of reversible computers. Quantum computers are, of course, a particular case of reversible computers.

Reversible programming languages are an important tool for the design and analysis of logically reversible computations. In this work we study *register* reversible languages.

Motivation: Characterization of the primitive recursive functions by a register language. A well known and very important sub-class of recursive (total) functions is the class of "primitive recursive" functions. As shown by A. R. Meyer and D. M. Ritchie, this class of functions can also be characterized by a specific (non reversible) register language, called "Loop". This means that every program written in Loop (together with the specification of the input and output registers) defines a primitive recursive function and, reciprocally, that every primitive recursive is computed by some Loop program.

Reversible transformations and the language SRL. Inspired by the work of Meyer and Ritchie [MR67a], we describe a simple and *reversible* register language called SRL (Simple Reversible Language), that characterize a set of transformations (bijections) of tuples of $\mathbb{Z}^n$ into tuples of $\mathbb{Z}^n$, where n is the number of registers used by the program; these transformations may be seen as the "primitive recursive functions in the reversible world". More precisely, it can be shown [Mat12] that the class of primitive recursive bijections is not enumerable, so that it can not be considered a "model of computation". The best that can

be done is to look for interesting enumerable sub-classes of primitive recursive bijections.

The language SRL is inherently reversible and does not need extra "ad-hoc" features (like, for instance, an extra memory tape) to insure reversibility. Programs written in SRL may be interpreted by an "instantly invertible" finite memory machine, in which the modification of a control bit by the operator causes the instantaneous inversion of the time direction (if going backwards, the computation finishes in the initial instruction). A slightly more powerful language (ESRL, Extended Simple Reversible Language) is also studied; it includes also the "`swap`(x, y)" instruction.

Reversible transformations: inductive definition. Similarly to what happens with the primitive recursive functions, it is also possible to define inductively the class of SRL transformations, without mentioning any particular programming language, as (roughly) the smallest set of transformations that includes the following operations

1) increment a register by 1;
2) decrement a register by 1;
3) compose two SRL transformations;
4) parametric composition, in which the value of a variable x determines the number of times that a given SRL transformation P is composed with itself, say $\overbrace{P;\, P;\, \ldots\, P}^{v[x]\ P\text{'s}}$, where $v[x]$ is the value contained in the register x. The transformation P can not mention x. As $v[x]$ may be a negative integer, we give an appropriate definition for "execute $v[x]$ times the transformation P" when $v[x] < 0$.

It should be emphasized that the inductive definition and the language based definition are in fact quite similar. And this is true for both concepts: primitive recursive functions and SRL reversible transformations. For instance, computing the value of a function $f(\overline{x}, y)$ using the definition of primitive recursion (see for instance [Odi89, Dav85, Rob47]), is essentially identical to the execution of a Loop program with input registers $\overline{x}$ and y.

Comparision with the reversible logical gates. Notice that, although the programs in SRL and ESRL are somewhat similar to the Fredkin and Toffoli's

reversible logical circuits, there are two important differences: in SRL or ESRL registers may contain an arbitrary integer (and not just 0 or 1) and the computation model is uniform.

Decidability problems. We state an important problem, "program equivalence", whose decidability is still unknown. It is proved that it is decidability class of the following problems is the same, in the sense that either all these problems are decidable or all belong to Π_1^0, the class of problems that are complements of semi-decidable problems.

1. "Program equivalence" problem: given two programs P and Q, is $P \equiv Q$?".
2. "Commutativity" problem: given two programs P and Q, is $PQ \equiv QP$?".
3. "Kernel" problem: given a program P is $P \equiv \varepsilon$?" (similar to the word problem of Group theory).

Another problem, "is n the order of the program P?", is at least as difficult as these three problems.

Two-variable SRL programs: class of transformations. Linear SRL programs have been studied previously by the author. Here, we characterize SRL programs with at most two variables, showing that they implement exactly the so called IP (invertible and polynomial) transformations.

Notes on the execution time. Due to the "`swap`(x, y)" instruction, the value stored in a register can change abruptly in a single instruction step. In order to measure how the values stored in the registers change as the computation proceeds, even with the presence of the "`swap`(x, y)" instruction, we define an appropriate global time function and study its properties.

Relation with Group Theory. Some relations and similarities with Group Theory concepts are also established. We study for instance a particular computable *normal form* for SRL programs. Unfortunately, two programs with different normal forms may be equivalent. The existence of a "complete" computable normal form (two programs having the same normal form are equivalent) would of course imply the recursiveness of the problems 1–3.

Chapter 1

Introduction

Contrarily to what has previously believed, only non reversible computations imply the dissipation of heat [Lan61]; see also [Ben73, Ben07]. This fact may have important consequences in the future of computing: the development of physically reversible computers may significantly reduce the energy that is currently spent in computations.

In our opinion the theoretical study of the mathematics of reversible computations should be based on a reversible model of computation with the following characteristics

- The model should be as *simple* as possible, but not trivial. In this work we have chosen a register machine as computation model.
- Each program (or circuit) should be easily inverted.
- The inversion of the execution direction of a program should be possible without extra auxiliary memory or other artifacts.

Register machines have been widely used as algorithmic models of computation, see for instance [BB96, Mor98, Odi89]. Associated with a register machine there are "register languages" in which the programs appropriate for that machine can be written. The purpose of this work is to study the properties of very simple register languages that are both reversible (each program has an inverse) and total (every computation halts). An example of a total, but not reversible,

register language is Loop [MR67a], whose programs (called "loop programs" in [MR67a]) correspond exactly to the set of primitive recursive functions. Examples of *Boolean* reversible register circuits are described in [FT82].

When designing a reversible language it is important to guarantee

1) The reversibility of the data modifying instructions, "what were the previous values of the registers?".

2) The reversibility of the program flow of control, "what instruction was executed before this one?".

These remarks also apply to reversible Turing machines.

Regarding point 2), it should not be possible, in principle, to reach a certain program label L from two different labels because, without additional information, it might be not possible to "go backwards" from L. See for instance [Ben07], page 3. Two examples of computation models where this merge of control is impossible are the Loop language [MR67a] (which is not reversible) and Fredkin's billiard ball computer [FT82].

In this work we define two reversible register languages, SRL and ESRL, where each register can contain an arbitrary, possibly negative, integer. Loops, but not conditional jumps, are possible – just as in the language Loop [MR67a]; the transformations implemented by these languages can be seen as the

reversible "primitive recursive" transformations

By ⋆SRL we mean "SRL and ESRL". In the following table we briefly compare the ⋆SRL languages with the Loop language (that characterizes the class of primitive recursive functions). In the Loop language, after the execution of the "$\texttt{for}^{(\star)}$" instruction, the loop variable is set to 0.

Language:	**Loop**	**SRL**	**ESRL**
Total	YES	YES	YES
Reversible	NO	YES	YES
Register contents	a non-negative integer	a (possibly negative) integer	a (possibly negative) integer
Input	a tuple of registers	the tuple of all registers	the tuple of all registers
Output	a register	the tuple of all registers	the tuple of all registers
Instructions	`inc`, `dec`, `for`$^{(\star)}$	`inc`, `dec`, `for`	`inc`, `dec`, `for`, `swap`

1.1 Reversibility and register languages

In a register language, like Loop and ⋆SRL, each program uses only a predetermined number of registers, so that, using the uniform[1] model, see for instance [BB96, CLR01], we can say that, independently of the values of the inputs, each program uses a fixed amount of memory. This does not happen, for instance, with Turing machines, where the number of tape cells used in a computation depends in general on the input; for non halting computations, that number may be not bounded.

We study reversible languages or models of computation (see for instance, [Ben73, FT82, Lec63]) in which there is an algorithm that inverts each program (or circuit or automaton) P, producing the inverse program P^{-1} that satisfies

$$P; P^{-1} \equiv P^{-1}; P \equiv \varepsilon$$

[1] In complexity theory the word "uniform" has at least two meanings: a model (such as an algorithm) which has a fixed description – it is not parametrized in the length of the input (circuits are not uniform models), and (ii) a method for analyzing the execution time of a computation, where it is assumed that the execution time of a machine instruction is the same, independently of the of the magnitude of the values contained in the registers.

where ";" denotes the concatenation operation (program sequencing), "ε" denotes the identity program, and "$\equiv$" denotes program equivalence, see Definition 3, page 16. As the empty program "ε" is total, every program P must also be total. The inverse of a program should be very easy to define and should use exactly the same registers; no additional memory should be needed to run the program in the reverse direction; this does not happen with some "reversibilizations" of computation model, like, for instance in [Ben73, LTV98].

As far as we know, the ⋆SRL languages (described in [Mat03], and in Chapter 3) are the simplest non trivial reversible and total register languages; we think they are an interesting tool for

- The study of the fundamental properties of reversible computations.
- The specification of reversible computations.

Regarding to the last goal, any program written in ⋆SRL automatically corresponds to a reversible and often non trivial transformation; many reversible transformations described in the literature, such as the pair

$$\mathtt{y} := \mathtt{y} + \mathtt{x}, \quad \mathtt{y} := \mathtt{y} - \mathtt{x}$$

described in page 2 of [Ben07], are easily implemented by ⋆SRL programs.

1.2 Reversible register languages

The language Loop [MR67a] characterizes exactly the class of primitive recursive functions. Each register contains a non-negative integer. This language is not reversible, essentially for two reasons

(i) If some register x contains 0 after the execution of an instruction of the form "`dec` x" (decrement the value stored in register x by 1), it is impossible to know the previous value of x: it may either be 0 or 1.

(ii) The projection operation, which in this case consists in the selection of a single register as output (ignoring all the others) is widely used.

In [Mat03], we modify Loop so as to obtain two reversible languages, namely SRL (Simple Reversible Language) and ESRL (Extended SRL), that solve the problems (i) and (ii) as follows

(i) Replace $\mathbb{N}$ by $\mathbb{Z}$, so that a register can contain negative integers.

(ii) The output (and the input) of a program is always the tuple of all the registers mentioned in the program.

In order to get some idea of the languages $\star$SRL, the reader may look at the first examples of Section 4.2, page 22.

Consider the following dsirable characteristics of a register reversible language.

1. The set of registers is denumerable, say x_1, x_2,... Each register x_i contains an arbitrary, possibly negative, integer value, and these values can be used[2] as "loop counters" in instructions like "`for` x $\{P\}$" (execute x times the program P, which can not modify the value contained in x). Because of this, it would make no sense to use real, instead of integer, register values.

2. Each instruction is elementary in the sense that it only modifies a finite number of registers (specified in the instruction). As a consequence, the number of registers mentioned in any program is finite.

3. Every program implements a bijection $\mathbb{Z}^n \to \mathbb{Z}^n$ in the registers it uses.

4. From every program P it is possible to effectively compute a program P^{-1}, called the inverse of P, such that

$$P; P^{-1} \;=\; P^{-1}; P \;=\; \varepsilon$$

   where "ε" is the empty program (which is total).

5. Every program implements a total function; thus, every program halts.

6. There are no pre-fixed input registers nor garbage output registers.

[2]The meaning of this instruction for negative x is explained in Section 3.5, page 16.

The languages ⋆SRL satisfy all these properties. Assumptions 1 and 2 are quite general and apply to (essentially) all register languages. The other assumptions are not independent. For instance, assumptions 3 and 4 imply assumptions 5 and 6. For instance,

- If 4 holds, the assumption 5 also holds, because ε is a total program.
- Assume that 3 and 4 are satisfied. We prove that the assumption 6 must also be satisfied. Consider the computation $P^{-1};P$ that we represent as follows

  $$\overline{x} \xrightarrow{P} \overline{y} \xrightarrow{P^{-1}} \overline{x}$$

  and suppose that the input of P has more elements than the output, $|\overline{x}| > |\overline{y}|$. Some of the registers are thus discarded in the output of P; denote them by $\overline{z}$. From $\overline{y}$ we can compute $\overline{x}$ (using P^{-1}) and, thus, compute $P(\overline{x})$, and, as a result, we get both $\overline{y}$ and $\overline{z}$ from $\overline{y}$. Thus, the function implemented by P would not be surjective, and the assumption 3 would not be satisfied. So we must have $|x| = |y|$.
- It is obvious that if the function implemented by P is not injective, it is impossible to obtain $\overline{x}$ from $\overline{y} = P(\overline{x})$.

1.2.1 Representing a pair of integers by an integer

Although it is easy to define effective bijections $b : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, the inclusion of instructions of this form in a reversible language would imply the existence of "garbage" registers at the output stage, as the following example shows

$$\begin{cases} x' &= b(x,y) \\ y' &= f(x,y) \end{cases}$$

where f is some function. Note that x' uniquely determines both x and y (because b is a bijection) and it follows that it also determines y'. Thus, this is not surjective transformation.

1.2.2 The number of registers does not change

Through this work it is assumed that the number of registers used by a ⋆SRL program does not change. We state this explicitly.

Assumption 1 (fixed number of registers) *The number of registers used by a* ⋆SRL *program is fixed. During the execution, no new registers can be created or disposed.*

For instance, without this assumption, the reasoning in Section 1.2.1 would not be valid.

Chapter 2

Preliminaries and notation

Most of the notation used in this work is either standard or self explanatory.

LHS and RHS denote respectively the "left-hand side" and the "right-hand side" of an equation or inequality. By "almost all" we mean "all except a finite number". A set is "denumerable" if it is either finite or enumerable [RK66].

Consider a function $f : A \to B$.

- "f is injective" if $x \neq y \Rightarrow f(x) \neq f(y)$.
- "f is surjective" if $\forall z \in B, \exists x \in A,\ f(x) = z$.
- "f is bijective" if it is both injective and surjective.

Let x_i be a register of some program. The initial value of x_i, also called the input x_i, is also denoted by x_i, while the final value of x_i is denoted by x_i'. If the program P uses registers x_1, x_2,..., x_n, we write

$$P(x_1, x_2, \ldots, x_n) = (x_1', x_2', \ldots, x_n')$$

As an example of this notation, see Example 1 in Section 4.2, page 22.

A tuple of registers $(x_1, \ldots, x_n)$ will be denoted by $\overline{x}$. The initial value of the tuple $\overline{x}$ is also denoted by $\overline{x}$, while the final value is denoted by $\overline{x}'$.

To select the ith element of the tuple $(x_1, \ldots, x_n)$ we use the projection operator "$|_i$", $(x_1, \ldots, x_n)|_i = x_i$.

We will also use the notation "$|$" to specify initial values for the registers; for instance, $P|_{[x=0]}$ denotes the output of the program P when the register x has the initial value 0.

When there is no possibility of confusion, the set of tuples of the form $(x_{i_1}, x_{i_2}, \ldots x_{i_n})$, where the set $\{i_1, i_2 \ldots i_n\}$ is fixed, will be denoted simply by $\mathbb{Z}^n$.

When convenient, programs will be parenthesized, so that we may for instance write $(P; Q)$ instead of $P; Q$.

Chapter 3

The languages ⋆SRL

In Sections 3.1, 3.3, and 3.4 the syntax, the semantics and the inversion algorithm of the programs in ⋆SRL are described informally, while in Section 3.5 those concepts are defined more rigourously.

The registers x_1, x_2,..., will also be denoted by arbitrary lowercase letters, a, b,.... The language SRL described in Section 3.1 (page 11) has three instructions, while the language ESRL has the extra instruction "$\texttt{swap}(x, y)$" ("swap" instruction). Most of our results apply to both languages.

The set of programs in ⋆SRL will be denoted by $\mathcal{P}$ and particular programs by P, Q....

We distinguish two cases for the meaning of the sentence "the program P modifies the variable x": during the computation and at the end of the computation.

Definition 1 *A program P never modifies a register x (meaning that x never changes during the execution of P) if it does not contain the following instructions: "$\texttt{inc}\, x$", "$\texttt{dec}\, x$", "$\texttt{swap}(x, -)$", and "$\texttt{swap}(-, x)$". The logical negation of "never modifies" is modifies (at some step).*

Definition 2 *A program P does not modify a register x if, after the computation, the value of x never changes, $\forall_x\, (\forall_{\overline{y}} P(x, \overline{y})|_1 = x$ where $\overline{y}$ represents the tuple of other registers and, for simplicity it is assumed that x is the first register (index 1). The logical negation of "does not modify" is modifies.*

For instance, the program "`dec` y; `inc` x; `inc` y; `for` z(`dec` x)"

- Never modifies z.
- Does not modify z.
- Modifies (at some step) y.
- Does not modify y.
- Modifies (at some step) x.
- Modifies x.

Clearly "modifes" implies "modifies (at some step)" and "never modifies" imply "does not modify".

3.1 Syntax of the languages ⋆SRL

We list the instructions of the languages SRL ("simple reversible language") and ESRL ("extended simple reversible language"), see [Mat03].

Language SRL:

- Instruction "`inc` x", called "increment instructions". The value stored in register x is incremented by 1.
- Instruction "`dec` x", called "decrement instructions". The value stored in register x is decremented by 1.
- Instruction "`for` $x(P)$", called "loop instructions". The symbol x denotes a register and P denotes a SRL program that never modifies x (Definition 1 above); for instance, the instruction may be "`for` x(`for` x(`inc` y))" but not "`for` x(`inc` x; `dec` x)". The program P is executed x times; we will explain in Section 3.5, page 16, how to interpret this when x contains a negative value. An instruction "`for` $x(P)$" is called a "for" or "loop" instruction; x and P are called the *loop variable* and the *loop program* respectively.

The language ESRL has also the extra instruction "`swap`(x, y)" that swaps the values stored in registers x and y.

The reason for introducing the instruction "swap"

It can be shown, see [Mat03] and 5.3 (page 29) in Section 4.2, that it is possible to implement in the language SRL a program equivalent to

$$\texttt{swap}(x, y); \texttt{swap}(w, z)$$

(it swaps two pairs of registers); however, it is not possible to implement in SRL a swap of a single pair of registers,

$$\texttt{swap}(x, y)$$

More generally ([Mat03]), with linear (Definition 5, page 18) SRL programs it is possible to implement all linear transformations with determinant is 1, while with linear ESRL programs all the linear transformations with determinant ± 1 are implementable.

3.2 The registers of a program

3.2.1 On the names of the registers

In general the names of the registers are not important. If, for instance, the program P mentions the registers a, b, and c, these letters may in fact correspond to arbitrary distinct registers, for instance $a \to x_2$, $b \to x_5$, and $c \to x_9$. However, one must be careful with name clashes; for instance, when composing two programs P and Q in parallel (see Section 7, page 44), there can be no register common to both programs.

3.2.2 Two kinds of registers

Given a program P, the set of registers mentioned in P is denoted by $\mathcal{R}(P)$. It is possible that some of these registers remain (for every input) unchanged. Associated to every program P there are two, in general non disjoint, sets of registers

– The set $\mathcal{L}(P)$ of loop (or "for") registers formed by the registers x, such that P has some instruction of the form "`for` $x(Q)$", where Q is a program.

– The set $\mathcal{V}(P)$ of registers that are modified (at some step) by P in the sense of Definition 1 (page 10), that is, the set of registers x, such that P contains some instruction of the form "`inc` x", "`dec` x", "`swap`(x, y)", or "`swap`(y, x)", for some register y.

Registers not in $\mathcal{V}(P)$, namely the registers in $\mathcal{L}(P) \setminus \mathcal{V}(P)$, are never modified by P. Consider for instance the program (1), page 13; we have $\mathcal{L}(P) = \{x, y, z\}$ and $\mathcal{V}(P) = \{x, y\}$.
A program may use registers that are always left unchanged; as an example, see the example 6 in Section 4.2, page 24.

3.2.3 A program as a tree

We can represented an arbitrary SRL program by an ordered tree, as follows

– A sequence of instructions $I_1; \ldots; I_n$ with $n \geq 2$ is represented by a tree consisting of a node having as sons the n ordered sub-trees corresponding to the instructions $I_1, \ldots, I_n$.

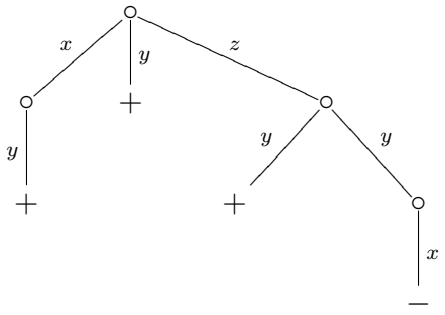
– Instructions "`inc` x" and "`dec` x" are represented by an edge labelled x whose lower vertex is a leaf labelled "+" or "−", respectively.

– The instruction "`for` $x(P)$" is represented by an edge labelled x whose lower vertex connects to the representation of P (which is not a leaf, unless P is null).

As an example, consider the following program

$$\texttt{for}\, x(\texttt{inc}\, y);\ \texttt{inc}\, y;\ \texttt{for}\, z(\texttt{inc}\, y;\ \texttt{for}\, y(\texttt{dec}\, x)) \tag{1}$$

The corresponding tree is



Notice that, in a path starting in the root and finishing in a leaf, a label (corresponding to a register) can occur more than once. That happens for example in the tree corresponding to the program "$\texttt{for}\, x(\texttt{for}\, x(\texttt{inc}\, y))$".

3.3 The meaning of a program

Let $\mathbb{Z}^\infty$ denote the set of infinite tuples of integer (possibly negative) values,

$$(x_1,\, x_2,\, \ldots)$$

We consider *permutations* $f : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$ where only finitely many register may be modified, that is, for every f we have, for almost all $i \in \mathbb{N}$,

$$\forall x_1,\, x_2, \ldots\ f(x_1,\, x_2, \ldots)|_i = x_i$$

The set of permutations of $\mathbb{Z}^\infty$ in $\mathbb{Z}^\infty$ is denoted by $\mathcal{G}(\mathbb{Z}^\infty \to \mathbb{Z}^\infty)$, while $\mathcal{G}(\mathbb{Z}^\infty \to \mathbb{Z}^\infty)^I$ denotes the set of permutations in which only the registers indexed by the finite set I may be modified. A particular permutation $s \in \mathcal{G}(\mathbb{Z}^\infty \to \mathbb{Z}^\infty)^I$ can be represented by

$$s(x_1, x_2, \ldots, x_n) = (x'_1, x'_2, \ldots, x'_n)$$

where, in this case, $I = \{1, 2, \ldots, n\}$. In general, the indices in I are, of course, arbitrary.

The permutations mentioned above (if $n \geq 1$ there are always infinitely many of

them) should not be confused with the *wire permutations* in which each input value r_i always occur at the output $r_{\pi(i)}$ where $\pi : \mathbb{N} \to \mathbb{N}$ is some *permutation of the indices of the registers.*

Each program P induces a permutation on $\mathcal{G}(\mathbb{Z}^\infty \to \mathbb{Z}^\infty)^{\mathcal{R}(P)}$. This permutation is called the *meaning* of P and is denoted by $[\![P]\!]$.

In particular, the meaning of the identity program ε is the identity permutation ι, $[\![\varepsilon]\!] = \iota$. There are of course other programs P equivalent to ε, that is, such that $[\![P]\!] = [\![\varepsilon]\!] = \iota$; an example is P="$\texttt{inc}\, x$; $\texttt{dec}\, x$".

An open problem: are there permutations $\pi \in \mathcal{G}(\mathbb{Z}^\infty \to \mathbb{Z}^\infty)^I$, such that any program that implements π must use additional registers (besides x_i for $i \in I$)? Of course, after every computation, the value contained in those additional registers must be left unchanged: $x_i = x'_i$ for every $i \notin I$. If yes, what is the minimum cardinality of I?

3.4 The inverse of a program

The inverse P^{-1} of a program P is obtained by the following rules.

1. $(\texttt{inc}\, x)^{-1} = \texttt{dec}\, x$
2. $(\texttt{dec}\, x)^{-1} = \texttt{inc}\, x$
3. $(\texttt{swap}(x, y))^{-1} = \texttt{swap}(x, y)$
4. $(\texttt{for}\, x(P))^{-1} = \texttt{for}\, x(P^{-1})$
5. $(P;\, Q)^{-1} = Q^{-1};\, P^{-1}$

We present an example of a program and its inverse

$$\begin{array}{ll} \text{Program:} & \texttt{for}\, x(\texttt{inc}\, y);\ \texttt{inc}\, y;\ \texttt{for}\, z(\texttt{inc}\, y;\ \texttt{for}\, y(\texttt{dec}\, x)) \\ \text{Inverse:} & \texttt{for}\, z(\texttt{for}\, y(\texttt{inc}\, x);\ \texttt{dec}\, y);\ \texttt{dec}\, y;\ \texttt{for}\, x(\texttt{dec}\, y) \end{array}$$

It is easy to show that, for any program P, we have

$$(P;\, P^{-1}) \equiv (P^{-1};\, P) \equiv \varepsilon$$

where ε is the empty program and "$\equiv$" denotes equivalence, see Definition 3, page 16.

3.5 Syntax, inversion and semantics: formal definition

Previously we have defined the following characteristics of the languages $\star$SRL: syntax, inversion of a program, and semantics. However, in a rigorous inductive definition, those aspects must be defined simultaneously.

Let us begin by stating formal definitions related to the semantics of a program

Definition 3 *Two programs P and Q are equivalent, and we write $P \equiv Q$ if they induce the same permutation $f : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$, that is, if $[\![P]\!] = [\![Q]\!]$. The empty program is denoted by ε, while the identity permutation $\mathbb{Z}^\infty \to \mathbb{Z}^\infty$ is denoted by ι.*

We now define simultaneously the syntax, the inverse, and the meaning of a program.

Language SRL:

> Instruction "$\texttt{inc}\, x$". The inverse is $(\texttt{inc}\, x)^{-1} = \texttt{dec}\, x$. The meaning is: increment by 1 the value contained in x.
>
> Instruction "$\texttt{dec}\, x$". The inverse is $(\texttt{dec}\, x)^{-1} = \texttt{inc}\, x$. The meaning is: decrement by 1 the value contained in x.
>
> Instruction "$\texttt{for}\, x(P)$" where P never modifies x (see Definition nmodi, page 10). The value of x is not altered by the execution of this instruction. The inverse is $(\texttt{for}\, x(P))^{-1} = \texttt{for}\, x(P^{-1})$. The meaning is: execute x times the program P; if x is negative, this should be interpreted as "execute $-x$ times the program P^{-1}".

The language ESRL also contains the following instruction.

Instruction $\texttt{swap}(x, y)$. The inverse is $(\texttt{swap}(x, y))^{-1} = \texttt{swap}(x, y)$ (the same instruction). The meaning is: exchange the values stored in the registers x and y.

Property 1 *Let P be any program in $\star$*SRL *and let P^{-1} be its formal (or syntactic) inverse, as defined above. Then*

$$[\![P;\, P^{-1}]\!] = [\![P^{-1};\, P]\!] = \iota$$

Proof. Use structural induction on P. □

However, it may happen that $[\![P;\, Q]\!] = \iota$ and Q is not the formal inverse of P. For instance, $P =$"$\texttt{inc}\, x;\ \texttt{inc}\, y;\ \texttt{dec}\, x$" and $Q =$"$\texttt{dec}\, y$".

3.5.1 Parametric composition

The loop instruction "$\texttt{for}\, x(P)$" is of particular interest. Its semantics can be described as "parametric composition", because it corresponds to the execution of

$$\overbrace{P;\, P; \ldots ; P}^{x \text{ times}}$$

As we said before, the contents of the register x can never be modified by P (in the sense of Definition 1, page 10). Moreover, after the execution of "$\texttt{for}\, x(P)$", the value of x remains unchanged.

3.5.2 Depth of a program

Definition 4 *The* depth *of a program is its maximum nesting of loop instructions; where a* `swap` *instruction is considered to have depth 1.* Linear *programs are programs with depth 1.*

A program has depth 0 if it only contains (possibly zero) increment and decrement. The program "$\texttt{for}\, x(\texttt{inc}\, y)\texttt{swap}(x, y)$" is linear, while the program "$\texttt{for}\, x(\texttt{swap}(x, y))$" has depth 2.

Definition 5 *The class of programs which have depth at most n is denoted by S_n for SRL programs, or E_n for ESRL programs. The programs in S_1 and E_1 (in this class there can be no* `swap` *instructions inside loops) are called* linear.

3.5.3 Executing a program P a negative number of times

When describing the semantics of ⋆SRL, we said that, when x is negative, executing x times a program P is the same thing as executing $-x$ times the program P^{-1}. This is a natural definition, and a number of simple program properties hold, such as

– If the register z contains $x+y$, we have "`for` $x(P)$; `for` $y(P)$" $\equiv$ "`for` $z(P)$" for every program P that never modifies x, nor y, nor z (Definition 1, page 10).

This interpretation of "`for` $x(P)$" when x is negative is related to the following notation: if n is an integer constant, the concatenation of n identical programs P will be denoted by P^n. For $n < 0$, P^n is defined as $(P^{-1})^{-a}$. The following property holds, even when n or m are negative.

$$P^n;\ P^m \ \equiv\ P^{n+m}$$

We have, for instance

$$P^n;\ P^{-n};\ \equiv\ P^0 \ =\ \varepsilon$$

Note however that, assigning a meaning to "execute P a negative number of times", which is a generalization of the traditional loop-like instructions, only seems to make sense for reversible languages.

During the execution of a program P, a part Q of P may sometimes be executed in the forward direction and, other times, in the inverse direction – when Q occurs in an instruction "`for` $x(Q)$", and x is negative. As we shall see in Chapter 13, there is a simple way to implement these "inverse computations" in a low level machine.

Chapter 4

Transformations

4.1 Non implementable transformations

A transformation $f : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$ is *implementable* in SRL if there is some SRL program P that implements to it, $[\![P]\!] = f$. We allow P to use additional registers whose values, at the end of the execution, remain unchanged.

A transformation f can not be implemented in $\star$SRL if: f is not a bijection, a sub-sequence of f (with arguments which are not "too large") grows faster than any primitive recursive functions, and f could be used as an algorithm for an undecidable problem.

4.1.1 Transformations that are not bijections

Many transformations are not implementable, just because they are not bijections. It should be recalled that the principle 1 (page 7) of the conservation of the number of registers is assumed through all this work.

We give some examples of non implementable transformations. By "..." we mean that "it doesn't matter".

$a' = 2a,\ b' = \ldots$ (not surjective, a' is necessarily even).

$a' = f(a,b)$, $b' = \ldots$ where f is bijection $\mathbb{Z}^2 \to \mathbb{Z}$ (not surjective, a' "fixes" b').

$a' = a^2$ (neither injective nor surjective, a' is the square of an integer).

$a' = ab$, $b' = b$ (not surjective; for instance $a' = 1$ and $b' = 2$ is not possible).

Non cloning theorem

The following result, well known in quantum mechanical computations, also applies to ⋆SRL transformations.

Theorem 1 (Non-cloning theorem) *Consider a* ⋆SRL *program with n registers $x_1, \ldots, x_n$, denoted by $\overline{x}$. It is impossible, either during the computation, or at the end of it, that two distinct registers always (that is, independently of the input values of $\overline{x}$) have the same value. In particular, it is impossible to have at the end of the computation $y' = z' = x$, where y and z are distinct registers.*

The proof is trivial: a transformation $x'_i = x'_j = \ldots$ with $1 \leq i < j \leq n$ is not surjective.

4.1.2 Sub-sequences that grow too fast

We define a bijection $g : \mathbb{Z} \to \mathbb{Z}$ such that the sub-sequence $g(2), g(4), g(6)\ldots$ grows asymptotically faster than any primitive recursive function. In follows that it can not be implemented by a ⋆SRL program.

Consider the following function defined for non-negative integers n.

$$f(n) = 2^{2^{\cdots^2}} \quad \text{where the number of 2's equals } n$$

The first values of f are

n :	1	2	3	4
$f(n)$:	2	4	16	65 536

The value of $f(5)$ has 19 729 digits! It is well known (see for instance [Her69]) that $f(n)$, which is related to the Ackermann function [MP11], grows faster than any primitive recursive function, and it follows that it also grows faster (for positive values of n) than any bijective transformation $\mathbb{Z} \to \mathbb{Z}$ implemented by a $\star$SRL program (even if the program uses other registers initialized with any fixed input values). Using the bijection f, define the function $g : \mathbb{Z} \to \mathbb{Z}$ as follows

1. For $n = 1, 2, \ldots$ make $g(2n) = f(n)$ (these values are underlined in the example below).
2. Define $g(2n+1)$ for $n \in \mathbb{N}$ with the remaining, that is, not in the co-domain of f, positive integers 1, 3, 5, 6,..., 15, 17...
3. Define $g(n) = n$ for $n \leq 0$

Clearly g defines a bijection $\mathbb{Z} \to \mathbb{Z}$ which is not implementable, because a subsequence of the co-domain grows too fast.

This construction is illustrated below

n :	...	-2	-1	0	1	2	3	4	5	6	7	8	...
$g(n)$:	...	-2	-1	0	1	$\underline{2}$	3	$\underline{4}$	5	$\underline{16}$	6	$\underline{65\,536}$	...

4.1.3 Uncomputable transformations

A bijection $g : \mathbb{Z} \to \mathbb{Z}$ that "solves" algorithmically an unsolvable decision problem, can not be implemented by any programming language.

Let i_1, i_2,... be the indices of the Turing machines that halt when the input is its index, or in symbols, $\{i_k\}(i_k) \downarrow$, where $\{i\}$ is the Turing machine with index i and "$\downarrow$" means "halt". Define the function g as follows

1. $g(i_k)$= next odd positive integer (underlined in the example below).
2. For $n \in \mathbb{N}^+ \setminus \{i_1, i_2, \ldots\}$, $g(n)$ is the next even positive integer for the positive odd values of n (marked with "$\star$" in the example below).
3. $g(n) = n$ for $n \leq 0$

Suppose for instance that the first Turing machines that "self-halt" have indices 2, 5, 7. Then, some values of g are

$$\begin{array}{rccccccccccccc} n: & \ldots & -2 & -1 & 0 & 1^\star & \underline{2} & 3^\star & 4^\star & \underline{5} & 6^\star & \underline{7} & 8^\star & \ldots \\ g(n): & \ldots & -2 & -1 & 0 & 2^\star & \underline{1} & 4^\star & 6^\star & \underline{3} & 8^\star & \underline{5} & 10^\star & \ldots \end{array}$$

The function $g(n)$ is a bijection by construction. Moreover, g is not computable because the statement "$g(i)$ is odd" is equivalent to the statement "the Turing machine with index i and input i halts".

4.2 Examples of implementable transformations

We now give some examples of implementable transformations. The inverse programs and inverse transformations are sometimes shown. The notation S_0, S_1,..., E_0, E_1,... is explained in Definition 5, page 18.

1. The S_0 program "`inc` a; `inc` a; `dec` b" implements the transformation

$$\begin{cases} a' &= a+2 \\ b' &= b-1 \end{cases}$$

2. The S_2 program "`for` a(`for` a(`inc` b))" implements the transformation

$$\begin{cases} a' &= a \\ b' &= b+a^2 \end{cases}$$

   The inverse program is "`for` a(`for` a(`dec` b))" and the inverse transformation is

$$\begin{cases} a &= a' \\ b &= b'-a'^2 \end{cases}$$

3. The S_1 program

$$\begin{array}{llll}
\texttt{for}\, a(\texttt{inc}\, c); & \texttt{for}\, c(\texttt{dec}\, a); & \texttt{for}\, b(\texttt{inc}\, d); & \texttt{for}\, d(\texttt{dec}\, b); \\
\texttt{for}\, c(\texttt{inc}\, b); & \texttt{for}\, d(\texttt{inc}\, a); & & \\
\texttt{for}\, a(\texttt{inc}\, c); & \texttt{for}\, c(\texttt{dec}\, a); & \texttt{for}\, b(\texttt{inc}\, d); & \texttt{for}\, d(\texttt{dec}\, b); \\
\texttt{for}\, c(\texttt{inc}\, b); & \texttt{for}\, d(\texttt{inc}\, a); & & \\
\texttt{for}\, a(\texttt{dec}\, c); & \texttt{for}\, b(\texttt{dec}\, c); & \texttt{for}\, a(\texttt{dec}\, d); & \texttt{for}\, b(\texttt{dec}\, d)
\end{array}$$

implements the transformation $\texttt{swap}(a, b)$; $\texttt{swap}(c, d)$, that is,

$$\begin{cases} a' &= b \\ b' &= a \\ c' &= d \\ d' &= c \end{cases}$$

This is an example of a wire permutation.

4. The S_1 program

$$\begin{array}{llll}
\texttt{for}\, a(\texttt{inc}\, c); & \texttt{for}\, c(\texttt{dec}\, a); & \texttt{for}\, b(\texttt{inc}\, a); & \texttt{for}\, a(\texttt{dec}\, b); \\
\texttt{for}\, c(\texttt{inc}\, b); & \texttt{for}\, b(\texttt{dec}\, c); & \texttt{for}\, c(\texttt{dec}\, a); & \texttt{for}\, c(\texttt{inc}\, b;\, \texttt{inc}\, b)
\end{array}$$

implements the transformation $\texttt{swap}(a, b)$; , $c' = -c$, that is,

$$\begin{cases} a' &= b \\ b' &= a \\ c' &= -c \end{cases}$$

A graphic representation of this program can be seen in Figure 1; the transformation $(a, b, c) = (1, 2, 3)$ into $(a, b, c) = (2, 1, -3)$ is also illustrated.

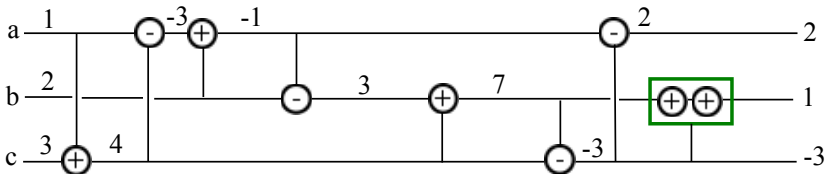


Figure 1: Diagram representing the SRL program of Example 4, page 23.

This is an example of a linear and homogeneous transformation, see Section 5.3 below and reference [Mat03]. As such, it can be described by a square matrix, namely

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

where the row and column order is: a, b, c.

5. Let P be the program of Example 4. Then, the program "`for` $r(P)$" implements the transformation

$$\begin{cases} \text{if } r \text{ is odd:} & a' = b,\ b' = a,\ c' = -c \\ \text{if } r \text{ is even:} & a' = a,\ b' = b,\ c' = c \end{cases}$$

It is called a "controlled swap and inversion".

6. Consider the program 4 and denote it by ex-sig(a, b, c). We can change the sign of the registers c and d, $c' = -c$, $d' = -d$ with

$$\text{ex-sig}(a, b, c);\ \text{ex-sig}(a, b, d)$$

The registers a and b are used, but not modified at the end of the computation. This program corresponds to the matrix transformation

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Notice that the *first* "instruction" to be executed, namely "ex-sig(a, b, c)", corresponds to the *second* matrix (immediately before the "=" sign).

7. The arithmetic signs of two registers can be changed without using additional registers (as in Example 6), as the following program shows. Transformation: $a' = -a$, $b' = -b$:

$$\begin{array}{ll} \texttt{for}\ a(\texttt{inc}\ b); & \texttt{for}\ b(\texttt{dec}\ a;\ \texttt{dec}\ a); \\ \texttt{for}\ a(\texttt{inc}\ b); & \texttt{for}\ b(\texttt{dec}\ a;\ \texttt{dec}\ a) \end{array}$$

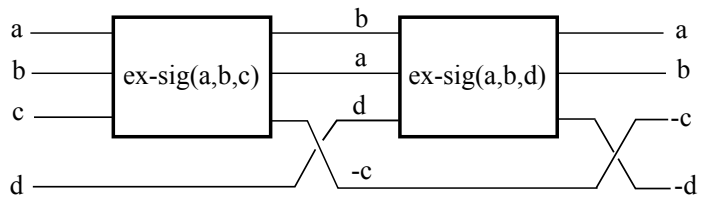


Figure 2: Diagram representing the sequencing of two "ex-sig" programs.

The inverse program is

$$\begin{array}{ll} \texttt{for}\, b(\texttt{inc}\, a;\ \texttt{inc}\, a) & \texttt{for}\, a(\texttt{dec}\, b); \\ \texttt{for}\, b(\texttt{inc}\, a;\ \texttt{inc}\, a); & \texttt{for}\, a(\texttt{dec}\, b) \end{array}$$

and the inverse transformation is $a = -a'$, $b = -b'$.

8. Consider again the program 4, ex-sig(a, b, c). We can swap the registers a and b and also the registers c and d, "$\texttt{swap}(a, b)$; $\texttt{swap}(c, d)$", with the following program that does not modify (at the end of the computation) the register e.

$$\text{ex-sig}(a, b, e);\ \text{ex-sig}(c, d, e)$$

See Figure 3. In this figure the "wires" c and d are "crossed" in order that the arguments of the right "ex-sig" box are consecutive; otherwise, it would be impossible to draw "ex-sig(c, d, e)" as a continuous "box". This drawing technique is also used in other figures, and it *does not* correspond to a wire permutation.

9. Consider the S_2 program

$$\texttt{for}\, r(\texttt{for}\, b(\texttt{inc}\, a);\ \texttt{for}\, a(\texttt{inc}\, b))$$

Let $F_i(x, y)$, where $i \in \mathbb{Z}$ be the value of the ith term of a Fibonacci like

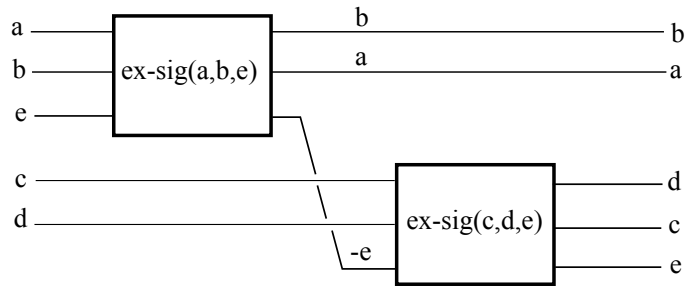


Figure 3: Diagram representing a sequencing of two "ex-sig" programs that implements the swap of two pairs of registers. The register e is used, but it is not modified (at the end of the computation) by the program.

sequence defined by

$$\begin{cases} F_0(x,y) &= x \\ F_1(x,y) &= y \\ F_n(x,y) &= F_{n-1}(x,y) + F_{n-2}(x,y) \end{cases}$$

For instance, $F_n(0,1)$, as a function of n is the usual Fibonacci sequence

$n:$	$\ldots$	-3	-2	-1	1	0	1	2	3	4	5	6	7	8	$\ldots$
$F_n(0,1):$	$\ldots$	-3	2	-1	1	0	1	1	2	3	5	8	13	21	$\ldots$

Note that, given x and y, $F_n(x,y)$ is defined for every $n \in \mathbb{Z}$. Then, the transformation corresponding to the program above is

$$\begin{cases} a' &= F_{2r}(a,b) \\ b' &= F_{2r+1}(a,b) \\ r' &= r \end{cases}$$

Diagrams that represent this program and its inverse can be seen in Figure 4

This example shows that with a simple program with only two levels of "loop nesting" (thus in the class S_2) we obtain a transformation with two exponential output values (the values a' and b').

Figure 4: Left: diagram representing the SRL program P of Example 9, page 25. Right: diagram representing the program P^{-1}.

The inverse program is

$$\texttt{for}\, r(\texttt{for}\, a(\texttt{dec}\, b);\, \texttt{for}\, b(\texttt{dec}\, a))$$

Chapter 5

Sub-classes of ⋆SRL programs

We study some specific families of ⋆SRL programs, namely depth 0 programs (Definition 4, page 17), programs without `for` instructions, depth 1 programs, and programs with two registers.

5.1 SRL programs with depth 0

SRL programs with depth 0 contain only `inc` and `dec` instructions[1]. These instructions commute and it is easy to see that these programs implement exactly the following family of transformations

$$\begin{cases} x'_1 &= x_1 + c_1 \\ x'_2 &= x_2 + c_2 \\ \dots & \dots \quad \dots \\ x'_n &= x_n + c_n \end{cases}$$

where $n \in \mathbb{N}$ and $c_1, \dots, c_n$ are integer (possibly negative) constants.

Example. Example 1, page 22. □

The equivalence problem for deph 0 programs is decidable.

[1] Note that every ESRL program has depth at least 1.

5.2 Programs without loops

These programs may include the instruction `swap`. By induction on the number of instructions of the program it is easy to show that these programs (sequences of `inc`, `dec` and `swap` instructions) implement exactly the following family of transformations

$$\begin{cases} x'_1 &= x_{i_1} + c_1 \\ x'_2 &= x_{i_2} + c_2 \\ \dots & \dots \quad \dots \\ x'_n &= x_{i_n} + c_n \end{cases}$$

where $\begin{pmatrix} 1 & 2 & \dots & n \\ i_1 & i_2 & \dots & i_n \end{pmatrix}$ is an arbitrary permutation.

Example. The program "`inc` x; `inc` x; `swap`(x, y)" which implements the transformation

$$\begin{cases} x' &= y \\ y' &= x + 2 \end{cases}$$

□

The equivalence problem for deph 0 programs is decidable.

5.3 Linear (depth 1) programs

If the domain is $\mathbb{Z}$, a linear system of equations has an unique integer solution if and only if the determinant associated with the system of equations is ± 1, that is, if the corresponding matrix is "integer positive modular", see for instance [Hua82]. As an example consider the following two systems of equations.

$$\begin{cases} x' &= 7x + 6y \\ y' &= 8x + 7y + 1 \end{cases} \qquad \begin{cases} x' &= 1x + 0y + 4 \\ y' &= 1x - 2y + 5 \end{cases} \tag{2}$$

The matrix associated with the first system is $\begin{bmatrix} 7 & 6 \\ 8 & 7 \end{bmatrix}$, its determinant is 1, and the inverse is $\begin{bmatrix} 7 & -6 \\ -8 & 7 \end{bmatrix}$. We get

$$\begin{cases} x &= 7x' - 6y' + 6 \\ y &= -8x' + 7y' - 7 \end{cases}$$

The first system of equations has an unique integer solution. The matrix associated with the second system has determinant -2, so that it is not invertible.

Programs ⋆SRL can be run "backwards" in time. This implies that linear ⋆SRL programs correspond to linear systems of equations with a unique solution, see [Mat03].

An ⋆SRL program P is called *linear* if loop instructions contains neither other loop instruction nor `swap` instructions; "linear" is other name for the classes S_1 or E_1. It is called *homogeneous* if the constant terms of the register transformations are 0. In [Mat03] it is shown that

1. Homogeneous linear SRL programs implement exactly the linear transformations belonging to the group PM_n of *integer positive modular* matrices (integer matrices with determinant +1).
   Two examples of linear homogeneous SRL programs: Example 4 (page 23) and Example 3 (page 22).

2. Homogeneous linear ESRL programs (recall that `swap` instructions can not occur inside loops) implement exactly the linear transformations belonging to the group IM_n of *integer modular* (or "unimodular") matrices, that is, $\mathrm{GL}_n(\mathbb{Z})$, the general linear group over the integers, see for instance [Sch98] (integer matrices with determinant ±1).

Both groups mentioned above are subgroups of the general linear group $\mathrm{GL}_n(\mathbb{R})$, and $\mathrm{IM}_n(\mathbb{R})$ is a subgroup of the special linear group $SL_n(\mathbb{R})$ (see for instance [Art91]). Thus, in particular,

$$\mathrm{PM}_n \leq \mathrm{IM}_n = \mathrm{GL}_n(\mathbb{Z}) \leq \mathrm{GL}_n(\mathbb{R})$$

where "≤" means "is a subgroup of".

5.4 SRL programs with two variables

In this section we characterize the transformations that can be implemented by SRL programs that use at most two variables. First, a form of polynomial transformations between two (integer) variables is defined, the *invertible and polynomial* (IP) transformations (Definition 6, page 34). Then we prove that every such transformation can be implemented by a SRL program (Lemma 1, page 35) and that every SRL program implements an IP transformation, see Lemma 2 and Theorem 2 (page 36). We conclude with some additional observations.

5.4.1 Solving non linear equations with two integer variables

An example

Consider the following non linear system of equation with two integer variables

$$\begin{cases} x' &= x + y^2 + 1 \\ y' &= y - 2(x + y^2 + 1)^2 \end{cases} \tag{3}$$

Notice that all the coefficients are integers. Given values for x and y, we get values for x' and y'. For instance,

$$(1, 2) \to (6, -70)$$

where the transformation is $(x, y) \to (x', y')$.

We ask if a given integer system of equations (with integer coeficients and integer solutions) is soluble, and, if the answer is affirmative, whether the solution is unique.

In the linear case, the solution of the last question is well known: The system has an unique integer solution if and only if the determinant associated with the system of equations is 1 (± 1 for the ESRL language), that is, if the corresponding matrix is "integer positive modular", see for instance [Hua82].

A method for obtaining the solution of (3)

The programming language SRL will be used to prove that the system of equations (3), which is non linear, has an unique integer solution. The method we will use consists in the following steps.

1. Prove that the transformation $(x, y) \to (x', y')$ corresponding to the system (3) can be implemented by a SRL program P. This may be difficult, and we do not know if there is an algorithm that implements this step.
2. Invert the program P.
3. Find the transformation (system of equations) $(x', y') \to (x, y)$ that corresponds to P^{-1}. This is the unique (integer) solution of (3).

We now use this method to solve the non linear system (3) (page 31).

Step 1

The transformation $x' = x + y^2 + 1$ can be implemented by the program

$$P_1 = \texttt{for}\, y(\texttt{for}\, y(\texttt{inc}\, x));\ \texttt{inc}\, x$$

After executing P_1, the transformation $y' = y - 2(x + y^2 + 1)^2 = y - 2x'^2$, can be implemented by the program

$$P_2 = \texttt{for}\, x(\texttt{for}\, x(\texttt{dec}\, y;\ \texttt{dec}\, y))$$

(notice that when P_2 starts, the program variable x contains the value x'). Thus, the following program implements the transformation (3).

$$\begin{aligned} P &= P_1; P_2 \\ &= \texttt{for}\, y(\texttt{for}\, y(\texttt{inc}\, x));\ \texttt{inc}\, x;\ \texttt{for}\, x(\texttt{for}\, x(\texttt{dec}\, y;\ \texttt{dec}\, y)) \end{aligned}$$

Step 2

We get

$$\begin{aligned} P^{-1} &= {P_2}^{-1}; {P_1}^{-1} \\ &= \texttt{for}\, x(\texttt{for}\, x(\texttt{inc}\, y;\ \texttt{inc}\, y));\ \texttt{dec}\, x;\ \texttt{for}\, y(\texttt{for}\, y(\texttt{dec}\, x)) \end{aligned}$$

Step 3

We have

$$P^{-1} = \overbrace{\texttt{for}\, x(\texttt{for}\, x(\texttt{inc}\, y;\ \texttt{inc}\, y))}^{\text{defines } y};\ \overbrace{\texttt{dec}\, x;\ \texttt{for}\, y(\texttt{for}\, y(\texttt{dec}\, x))}^{\text{defines } x}$$

The transformation implemented by this program is

$$\begin{cases} y &= y' + 2{x'}^2 \\ x &= x' - y^2 - 1 = x' - (y' + 2{x'}^2)^2 - 1 \end{cases}$$

that is,

$$\begin{cases} x &= x' - {y'}^2 - 4{x'}^2 y' - 4{x'}^4 - 1 \\ y &= y' + 2{x'}^2 \end{cases}$$

Let us check this with the values $(x, y) = (1, 2)$. From the original example (3) (page 31) we obtain

$$(1, 2) \to (6, -70)$$

and from (5.4.1) we get $y = (-70) + 72 = 2$ and $x = 6 - (-70)^2 - 4 \times 6^2 \times (-70) - 4 \times 6^4 - 1 = 1$.

5.4.2 Two-variable programs: the general case

We describe the transformations that can be implemented by SRL programs with 2 variables. These are integer, invertible transformations, defining bijections

$\mathbb{Z}^2 \to \mathbb{Z}^2$, and have the form

$$\begin{cases} x' &= f(x,y) \\ y' &= g(x,y) \end{cases} \tag{4}$$

where f and g are polynomials. However, not all transformations of this form are implementable in SRL, see Theorem 2.

Every IP transformation can be implemented by a two-variable SRL program

We begin by defining a form of two-variable transformation of the form (4).

Definition 6 *A two-variable transformation* $(x,y) \to (x',y')$ *is* IP *(invertible and polynomial) if there is an integer* $n \geq 0$ *such that, for* $1 \leq i < n$*, we have*

$$\begin{aligned} x^{(i+1)} &= x^{(i)} + R^{(i)}(y^{(i)}) \\ y^{(i+1)} &= y^{(i)} + S^{(i)}(x^{(i+1)}) \end{aligned}$$

where $x = x^{(1)}$*,* $y = y^{(1)}$ $x' = x^{(n)}$*,* $y' = y^{(n)}$*, and, for* $1 \leq i < n$*,* $R^{(i)}$ *and* $S^{(i)}$ *are arbitrary polynomials.*

In particular, the polynomials $R^{(i)}$ and $S^{(i)}$ may be equal to 1.

Note that the transformation 3 (page 31) has this form.

We now prove that every IP transformation can be implemented by a SRL program. For that purpose, we define SRL programs $P^{(1)}$, $P'^{(1)}$,... $P^{(n-1)}$, $P'^{(n-1)}$, such that

$$\begin{array}{lll} P^{(i)} & \text{implements} & x^{(i+1)} = x^{(i)} + R^{(i)}(y^{(i)}) \\ P'^{(i)} & \text{implements} & y^{(i+1)} = y^{(i)} + S^{(i)}(x^{(i+1)}) \end{array}$$

The SRL program that implements the given IP transformation is then

$$P^{(1)};\ P'^{(1)};\ \ldots P^{(n-1)};\ P'^{(n-1)}$$

In order to complete the proof, we explain with an example how to implement in SRL a transformation of the form

$$x' = x + R(y)$$

The example is

$$x' = x - y^3 + 3y + 1$$

This transformation is implemented by the program

$$\texttt{for}\, y(\texttt{for}\, y(\texttt{for}\, y(\texttt{dec}\, x)));\ \texttt{for}\, y(\texttt{inc}\, x);\ \texttt{inc}\, x$$

The generalization for an arbitrary polynomial $R(y)$ is simple.

Lemma 1 *Every IP transformation (Definition 6, page 34) can be implemented by a SRL program.*

Every two-variable SRL program implements an IP transformation

Definition 7 *We say that a SRL program P with two variables x and y "only modifies x" if P does not contain the instruction "*`inc` *y" nor the instruction "*`dec` *y"; in other words, P never modifies y in the sense of Definition 1, page 10.*

A two-variable SRL programs is a sequence of programs

$$P_1;\ P_2;\ \ldots P_m$$

where the program P_1 only modifies x (Definition 7), the program P_2 only modify y, the program P_3 only modifies $x \ldots$ In particular, the program P_1 may be null (so that the first transformed variable is y) and the program P_m may be null (so that the last transformed variable is x).

We will now prove that a two-variable SRL program that only modifies x implements a transformation of the form $x' = x + Q(y)$, where Q is a polynomial.

First observe that a sequence of SRL programs implementing IP transformations also implements an IP transformation. So we have only to prove the result for a single instruction (that only modifies x).

An instruction that only modifies x (Definition 7, page 35) can have one of the forms: "`inc` x", "`dec` x", "`for` $y(P)$" where P is a sequence of instructions that only modifies x.

Consider a sequence of instructions that only modifies[2] x. We will prove that the sequence of instructions implement an IP transformation (Definition 6, page 34) of the form $x' = x + Q(y)$ (where Q is a polynomial). Let n be the maximum imbrication level of "`for` $y(\cdots)$" instructions. The proof is by induction on n.

1. If $n = 0$, P can contain only "`inc` x" and "`inc` x" instructions, so that the transformation implemented by P is $x' = x + c$ where c is a constant.

2. If $n > 0$, consider an instruction with imbrication level n, "`for` $y(P)$". By the induction hypothesis, the program P implements an IP transformation of the form $x' = x + Q(y)$ (where Q is a polynomial), so that the instruction "`for` $y(P)$" implements the IP transformation $x' = x + yQ(y)$, which is of the same form.

A similar reasoning shows that a sequence of instructions that only modifies y corresponds to a transformation of the form $y' = y + Q(x)$, where Q is a polynomial. We have just proved the following result

Lemma 2 *Every two-variable SRL program implements an IP transformation (Definition 6).*

Two-variable SRL programs implement exactly the IP transformations

Recalling Lemmas 1 and 2, we get the following result.

Theorem 2 *Two-variable SRL programs implement exactly the class of IP transformation (Definition 6, page 34).*

As a simple corollary, we conclude that an exponential transformation (for instance, the transformation associated with Example 9, page 25) can not be implemented with a program having only 2 variables.

[2] It is interesting to notice that all these instructions commute.

5.5 Comments and further study

Two variable transformations associated with the language ESRL

In this chapter we have only considered the language SRL. In two-variable programs written in the language ESRL, an instruction `swap`(x, y) can only occur at the outermost level, that is, not under the scope of any "`for`" instruction.

Examples

There are several SRL programs presented in Section 4.2 (page 22) that use two variables, and as such, correspond to IP transformations. For instance, the Example 7 (page 24) corresponds to the transformation $x' = -x$, $y' = -y$, which is an IP transformation, so that it can be expressed in a sequence of transformations as explained in Definition 6, page 34; using the corresponding SRL program this is straightforward.

Three variable SRL programs

As we have seen, two-variable SRL programs correspond to certain forms of polynomial transformations, more specifically to IP transformations. If three variable are allowed, the transformations can be exponential, as the Example 9 shows.

A mathematical application

We think that the usage of the language SRL to characterize and generate non linear integer bijective (uniquely invertible) transformations may be an interesting tool in Mathematics.

5.5.1 Further study: integer transformations hierarchies

There are several classifications of primitive recursive functions into hierarchies, such as the Grzegorczyk hierarchy (see for instance [Grz53]), and the Cleave hierarchy ([Cle63]).

A primitive recursive hierarchy of classes is a sequence of classes of functions $\mathcal{F}_1$, $\mathcal{F}_2, \ldots$ such that (i) $\mathcal{F}_i \subset \mathcal{F}_{i+1}$ for $i = 1, 2 \ldots$ (proper inclusion) and (ii) $\cup_i F_i$ is the class of primitive recursive functions. A discussion of these sub-recursive hierarchies can be found for instance in [Mol73]. As the class of primitive recursive functions can be defined as the class of functions that can be implemented in the Loop language ([MR67a]), it is natural to try to relate primitive recursive hierarchies to appropriate restrictions on LOOP programs; for instance, we may limit the depth of nesting of `for` instructions. Some works in this direction are [Mey65, MR67b, GN81, GN78].

Similarly, it would be interesting to relate restrictions on SRL programs, such as the maximum nesting of "`for`" instructions or the number of variables used by the program, to hierarchies of integer bijective transformations.

Chapter 6

Partitions of $\mathbb{Z}^n$

6.1 General concept

Every $\star$SRL program P induces a bijection $[\![P]\!] : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$.

In an $\star$SRL program P consider a partition of its n registers in two sets, $\overline{x} = (\overline{y}, \overline{z})$; let $n = |\overline{x}|$, $p = |\overline{y}|$, $q = |\overline{z}|$, with $n = p + q$. Fixing the registers $\overline{y}$ we define the set of output values

$$Z^n(\overline{y}) = \{P(\overline{y}, \overline{z}) : \overline{z} \in \mathbb{Z}^q\} \tag{5}$$

The family of sets $Z^n(\overline{y})$ forms a partition of $\mathbb{Z}^n$ as follows. Each initial value of the registers $\overline{y}$ corresponds to the set of the partition whose elements are the output of the computations $P(\overline{y}, \overline{z})$, for all $\overline{z} \in \mathbb{Z}^q$, see (5).

This may be described pictorially as follows

$$qqq(\boxed{\overline{y}}, \overline{z}) \xrightarrow{P} (\overline{y'}, \overline{z'})$$

where the box in "$\boxed{\overline{y}}$" means that the registers $\overline{y}$ are fixed.

Notice that $\bigcup_{\overline{y}} \mathbb{Z}^n(\overline{y}) = \mathbb{Z}^n$ and that $\overline{y_1} \neq \overline{y_2}$ (that is, the tuples $\overline{y_1}$ and $\overline{y_2}$ differ in at least one position) implies $\mathbb{Z}^n(\overline{y_1}) \cap \mathbb{Z}^n(\overline{y_2}) = \emptyset$.

To this partition of $\mathbb{Z}^n$ corresponds the equivalence relation "$\sim$": $z \sim z'$ if and only if for some $\overline{y}$ we have z, $z' \in \mathbb{Z}^n(\overline{y})$.

6.2 Application to example 9

Consider now the example 9 in page 25.

1. Fix the initial values (a, b).
2. For every $r \in \mathbb{Z}$ run $P(r, a, b)$.
3. Select a' and b', the final values of a and b, respectively.

Given that this program does not modify the value of r, the partition in $\mathbb{Z}^3$ induces a partition of $\mathbb{Z}^2$ which can be described as

$$\mathbb{Z}^2(a, b) = \{(a', b') \ : \ P(r, a, b) = (r, a', b') \text{ for some } r \in \mathbb{Z}\}$$

Each pair (a, b) determines an equivalence class $C(a, b)$. Two examples:

$$\begin{aligned} C(3, 5) &= \{\ldots(-1, 1), (1, 0), (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), \ldots\} \\ C(-1, 2) &= \{\ldots(-4, 3), (3, -1), (-1, 2), (2, 1), (1, 3), (3, 4), (4, 7), \ldots\} \end{aligned}$$

In Figure 5 some sets of the partition are represented. Each point (x, y) is represented by a colored square and all the points of the same set correspond to squares with the same colour. For instance, the set containing the pair $(-1, 2)$ is represented in turquoise and corresponds to the sequence

$$\ldots 18,\ -11,\ 7,\ -4,\ 3,\ -1,\ 2,\ 1,\ 3,\ 4,\ 7,\ 11, \ldots$$

The corresponding set is

$$\{\ldots(-11, 7), (7, -4), (-4, 3), (3, -1), (-1, 2), (2, 1), (1, 3), (3, 4), (4, 7), \ldots\}$$

It should be emphasized that these sets form a partition of the plane, so that a "completed" Figure 5 is completely coloured, with no overlap between different colours; figure 6 (generated with "`sage`") contains all the partitions that contain a point (x, y) with $0 \leq |x|,\ |y| \leq 5$.

Each set has as asymptotes three out of the following four possibilities

$$\begin{cases} y = \alpha x, & x \geq 0 \quad \text{(first quadrant)} \\ y = -(\alpha - 1)x, & x \leq 0 \quad \text{(second quadrant)} \\ y = \alpha x, & x \leq 0 \quad \text{(third quadrant)} \\ y = -(\alpha - 1)x, & x \geq 0 \quad \text{(fourth quadrant)} \end{cases}$$

where $\alpha = (1 + \sqrt{5})/2 = 1.618\ldots$ is the golden ratio.

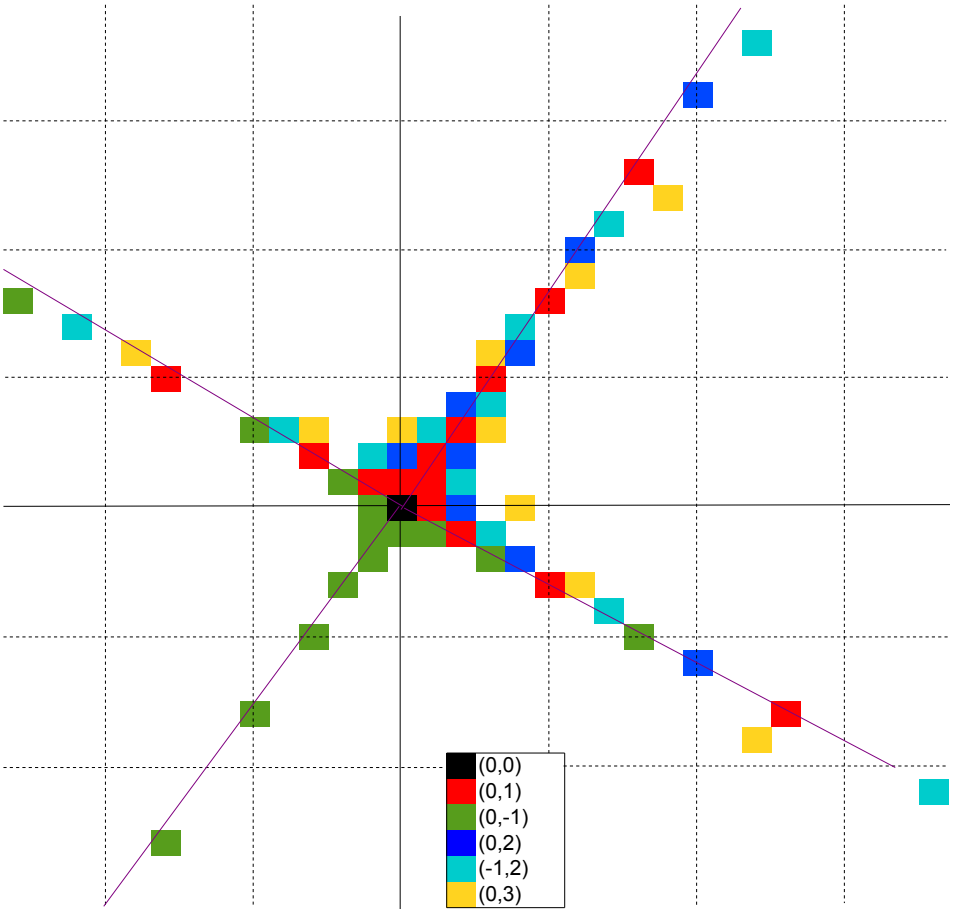


Figure 5: Some sets of the partition of $\mathbb{Z}^2$ induced by the program of Example 9. The points of each set are represented by colored squares. For instance, the sequence with the representative $(0,1)$ is $\ldots\ -3, 2, -1, 1, 0, 1, 1, 2, 3, 5\ \ldots$, and the corresponding set of the partition of $\mathbb{Z}^2$ is formed by the points represented as red squares, $\{\ldots(-3,2), (2,-1), (-1,1), (1,0), (0,1), (1,1), (1,2), (2,3), (3,5)\ldots\}$. The asymptotes are the magenta lines.

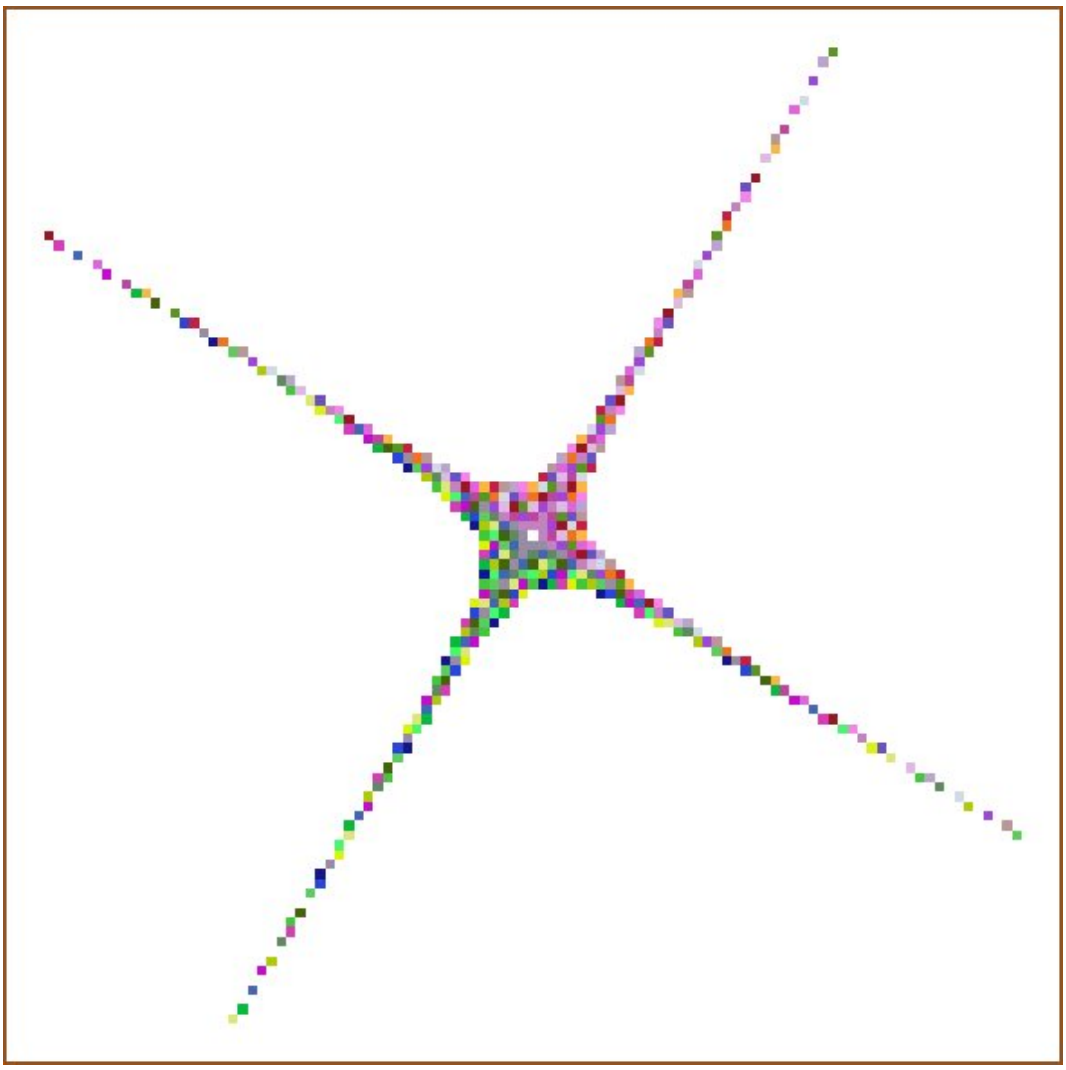


Figure 6: See Figure 5. Every set of the partition that contains a point (x, y) with $0 \leq |x|, |y| \leq 5$ (except $(0, 0)$) is represented as the set of the squares with the same color.

Chapter 7

Program composition

We describe some forms of combining two or more programs in a single program. Due to the reversibility, some common forms of program composition are not allowed; these include, for instance, the compositions that involve the "connection" of one output register of some program (or part of a circuit) to two or more inputs registers of other programs (or other parts of a circuit); this kind of connection would in general result in a program (or circuit) that does not implement a bijective transformation.

It is possible, and eventually it may be advantageous, to describe various forms of program composition (or of circuit composition) in terms of Category Theory, see for instance [GA06, YY09] and the citations therein.

Series and parallel composition

Let A and B be two programs that use respectively the register sets $R_a \cup R_c$ and $R_b \cup R_c$, where $R_a \cap R_b = \emptyset$. Figure 7 illustrates a form of composing A with B that includes both *parallel* and *series* composition.

$$\begin{cases} R_c = \emptyset & \text{parallel composition} \\ R_a = R_b = \emptyset & \text{series composition} \end{cases}$$

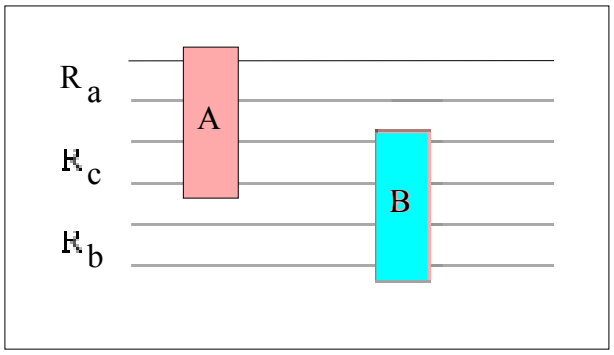


Figure 7: A form of composing two programs or circuits; the horizontal lines represent registers or "wires", respectively. It is assumed that $R_a \cap R_b = \emptyset$. It includes the parallel (when $R_c = \emptyset$) and the series (when $R_a = R_b = \emptyset$) forms of composition.

Extension and projection

The set of registers used by a program may be *extended*, as exemplified in Figure 8. The extension operation includes the *projection* which consists in removing some registers from the program; clearly, it is only possible to "remove a register from a program" when the program does not mention it.

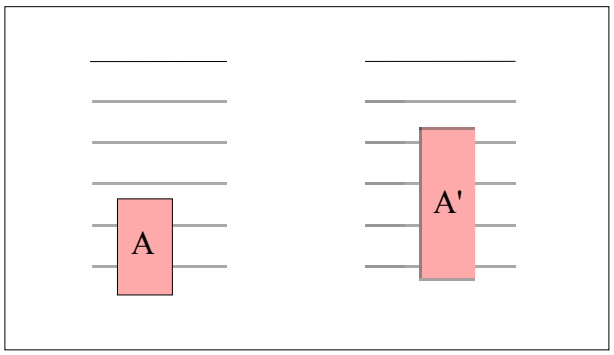


Figure 8: The program (or circuit) on the right was obtained from the program (or circuit) on the left by the "extend" transformation. The extension can be a projection, which happens when the program does not mention some registers and they are "removed"; for instance, A is a projection of A'.

Replacement of a part of a program by another program

Recall the definition of "registers modified" (at some step of a computation) by a program P, Definition 1, page 10.

We mention another form of composing the programs A and B which consists in *replacing* some sequence A' of instructions of A by B. This replacement is only possible when the following sets are disjoint

- The set of loop registers whose scope includes the replaced sequence of instructions A'.
- The set of registers modified (at some step of a computation) by B.

For instance, in "$A = \texttt{for}\, a(\texttt{inc}\, b; \texttt{dec}\, c)$" we can not replace the instruction "$A' = \texttt{dec}\, c$" by "$B = \texttt{for}\, b(\texttt{inc}\, a)$".

Register renaming

Given a set of programs arbitrarily composed as described above, we can *rename* the registers. This renaming must be done globally, that is, it should affect all the composed programs.

Chapter 8

Simulation results

8.1 Simulation of recursive Boolean circuits

In the 1980's, Fredkin, Toffoli, and others have studied reversible Boolean circuits, see for instance [FT82]. As a consequence of the fact that the number of possible inputs is finite, many of the decision problems associated with these circuits, such as "are two given circuits equivalent?", are trivially recursive. Also, by counting arguments, it follows that the length (number of bits or "wires) of the input must equal the length of the output.

With the languages ⋆SRL it is possible to simulate Boolean circuits, as explained below. Can we simulate any Boolean circuit? This is the question of *universality*. We will see that the answer to this question is affirmative, if we use the following modification of ⋆SRL programs: only some registers are considered in the transformation implemented by the program; the other registers have fixed values at the input stage and are discarded (as "garbage") after the computation; see also Chapter 12.

First, we have to decide how to represent a Boolean value by an integer. There are of course many possibilities, but we select the following mapping function τ.

$$\tau(x) = \begin{cases} \textsc{False} \text{ if } x \text{ is even} \\ \textsc{True} \text{ if } x \text{ is odd} \end{cases}$$

Now we present some programs for which the map τ is a homomorphism. According to our interpretation of integers as Boolean values, a condition like "if c" means "if c is odd".

⋆SRL program	Boolean transformation
`inc` x	$\neg x$
`dec` x	$\neg x$
$x' = -x$	identity
`swap`(x, y)	exchange the logical values of x and y
`swap`(x, y); $z' = -z$	exchange the logical values of x and y
`for` c(`swap`(x, y))	`if` c `then` exchange the logical values of x and y. This is the Fredkin gate, see [FT82]
`for` c(`inc` x)	CNOT(c,x): if c, `then` change the truth value of x
`for` c(`for` d(`inc` x))	CNOT(c,d,x): if c and d, `then` change the truth value of x. Notice that the product cd is odd exactly when both c and d are odd

It is well known that some gates or sets of gates are universal under the ordinary (non reversible) Boolean circuit compositional rules. One such universal gate is the two input NAND gate. In the theory of conservative logic, the Fredkin gate is universal ([FT82]), and we have just seen above that it can be simulated in the ESRL language. However, it should be noticed that, contrarily to what happens in the ⋆SRL languages, the usual formalization of reversible logical circuits allows gates that have pre-determined fixed values, and output gates that are not used ("garbage").

8.2 Simulation by primitive recursive functions

Each ⋆SRL program P can be simulated by a Loop program (whose registers are non-negative integers), representing each integer $x \in \mathbb{Z}$ by a pair of non-negative integers: the pair $(x, 0)$ if $x \geq 0$, or the pair $(0, -x)$ if $x < 0$. The simulation of loop instructions requires some care, because of the meaning of "`for` $x(P)$" when the value stored in x is negative; in this case, the inverse of the program P

is executed $-x$ times, so that the execution must somehow "compute" program inverses. The implementation of $\star$SRL in a low level machine, see Chapter 13 illustrates how this can be done.

The simulation shows that, in a sense, the transformations that can be implemented in $\star$SRL can also be implemented in Loop. However, the simulation itself not particularly interesting because the language in which the simulation is executed, the Loop language, is not reversible; only the simulated language is reversible.

Chapter 9

Equivalence and commutativity

9.1 The equivalence problem

To every program P corresponds a bijection $f : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$ such that only a finite number of registers can be modified. We say that the program P *implements* the bijection f and write $[\![P]\!] = f$.

The programs P and Q are *equivalent* and we write $P \equiv Q$, if $[\![P]\!] = [\![Q]\!]$; in detail:

1. the contents of registers in $\mathcal{R}(P) \setminus \mathcal{R}(Q)$ are not modified by P at the end of the computation,
2. the contents of the registers in $\mathcal{R}(Q) \setminus \mathcal{R}(P)$ are not modified by Q at the end of the computation, and
3. $P(\overline{r}) = Q(\overline{r})$ for every tuple $\overline{r} \in \mathcal{R}(P) \cap \mathcal{R}(Q)$.

In particular, the empty program $P = \varepsilon$ corresponds to the identity bijection ι, $[\![\varepsilon]\!] = \iota$.

Definition 8 *The* (program) equivalence problem *is: "Given two programs P and Q, is $P \equiv Q$?". It is assumed that $\mathcal{R}(P) = \mathcal{R}(Q)$, which can be obtained by extending the registers of P and/or Q.*

Definition 9 *The* kernel problem *is: "Given a program P, is $P \equiv \varepsilon$?".*

For total reversible languages we have "$P \equiv \varepsilon$" if and only if "$P;Q \equiv \varepsilon;Q \equiv Q$"; and "$P \equiv Q$" if and only if "$P;Q^{-1} \equiv \varepsilon$". Thus, for these languages, we have the following result.

Lemma 3 *The equivalence problem and the kernel problem are either both decidable or both undecidable. They both belong to Π_1^0, the class of problems that are complements of semi-decidable problems.*

However, it is unknown if these problems are decidable.

The kernel problem is similar to the group *word problem* [LS77, MKS76] which is the following: given a group defined by its generators and relators, and a product x of generators and its formal inverses, is x equal to the group identity? This problem was posed by Dehn [Deh12] in 1912, and it was shown in the 50's by Boone [Boo57] and Novikov [Nov55] that, in general, there is no algorithm that solves the word problem.

The kernel problem can be formulated as follows: given a ⋆SRL program (sequence of instructions) $I_1,\ldots,I_n$ is the corresponding product of bijections $[\![I_n]\!] \cdot [\![I_{n-1}]\!] \cdot \ldots \cdot [\![I_1]\!]$ (defined in the group G_{impl}, see page 56) equal to the identity bijection? The kernel problem is different from the word problem because: (i) ⋆SRL programs have a tree-like structure, being more complex than simple sequences of generators and its inverses; (ii) the "relators" associated with the languages ⋆SRL are fixed.

Equivalence of primitive recursive functions

For the class of primitive recursive programs, PR, the equivalence program is undecidable. A proof can be based on the following facts: (i) it is undecidable if two context free languages specified by its parsers, are equal, see for instance the Chapters 4 and 5 of [Sip97]; (ii) a parser for a context free language may be implemented as a primitive recursive functions (with 0/1 output). Thus we have

Theorem 3 *Consider two primitive recursive functions $f(\overline{x})$ and $g(\overline{y})$, specified*

by Loop *programs. It is undecidable if f and g are the same function. The problem is also undecidable if the functions are specified by standard primitive recursive definitions, see for instance [Her69, Odi89].*

The equivalence problem for primitive recursive functions defined by Loop programs has been analyzed in greater detail in [Tsi70]. For each non-negative integer i let L_i be the set of Loop programs such the the nesting of "`for`" is at most i. In [Tsi70] it is proved that the equivalence problem is decidable for L_1, but undecidable for L_i for every $i \geq 2$. The same result applies for some other decidability problems studied in that paper.

As we said above, it is not known if the equivalence program is undecidable for the SRL language.

9.2 Commutativity

The order of the instructions in a program is of course important; in general, instructions do not commute. For instance, the programs "`inc` y; `for` y(`dec` x)" and "`for` y(`dec` x); `inc` y" are not equivalent. Similarly, the programs "`inc` x; `swap`(x, y)" and "`swap`(x, y); `inc` x" are not equivalent.

We say that the programs P and Q *commute* if $P; Q \equiv Q; P$.

Definition 10 *The* commutativity problem *is: "Given two programs P and Q, is $P; Q \equiv Q; P$?". Equivalently (for total reversible languages), "is $P \equiv Q; P; Q^{-1}$?".*

In the following result we present a sufficient (but not necessary) condition for commutativity.

Theorem 4 *Let P and Q be two SRL programs. If the following two conditions hold*

$$\mathcal{V}(P) \cap \mathcal{L}(Q) = \emptyset, \qquad \mathcal{V}(Q) \cap \mathcal{L}(P) = \emptyset$$

then P and Q commute. In words: no loop register of Q can be modified by P and no loop register of P can be modified by Q.

The condition stated in the previous result is not necessary, as the following program shows

$$\overbrace{\texttt{dec}\, y;\ \texttt{inc}\, y;}^{P} \quad \overbrace{\texttt{for}\, y(\texttt{inc}\, z)}^{Q}$$

In this example, we could replace $P =$"$\texttt{dec}\, y;\ \texttt{inc}\, y$" by any, possibly very complex, program which is equivalent to ε. Note that equivalence to ε may be, to our knowledge, an undecidable problem.

As examples of commuting programs, we have

1. P and Q, where neither P nor Q contain loop instructions.
2. "$\texttt{inc}\, x$", and "$\texttt{for}\, y(\texttt{inc}\, w);\ \texttt{for}\, z(\texttt{inc}\, x;\ \texttt{dec}\, w)$".

9.3 Equivalence and commutativity are the same problem

We now study the relationship between the equivalence problem and the commutativity problem.

We present an algorithm that decides the equivalence problem, given an algorithm for the commutativity problem. This implies that the equivalence problem and the commutativity problem are equivalent, that is, if one of them is recursive, the other is also recursive.

It is obvious that the existence of an algorithm for deciding the equivalence problem would imply that the commutativity problem is also recursive. The other direction is proved in the following result.

Theorem 5 *The commutativity problem is recursively solvable (decidable) if and only if the equivalence problem is recursively solvable.*

Proof. It is obvious that an algorithm for deciding the equivalence problem can be used to decide the commutativity problem.

Assuming that there is an algorithm that decides the commutativity problem, we define an algorithm that decides if a given program P is equivalent to the empty program ε. By Lemma 3, page 51, the existence of such algorithm, implies that there is also an algorithm for the equivalence problem.

Input: a program P.
Oracle: decision of the commutativity problem.
Output: YES if $P \equiv \varepsilon$, NO otherwise.

> Let $x_1, x_2, \ldots, x_n$ be the registers mentioned by P, and let $y_1, y_2, \ldots, y_n$ be new registers.
>
> Let $Q = \texttt{for}\, x_1(\texttt{inc}\, y_1);\ \texttt{for}\, x_2(\texttt{inc}\, y_2);\ \ldots \texttt{for}\, x_n(\texttt{inc}\, y_n)$.
>
> If P commutes with Q, then answer YES (P is equivalent to ε), otherwise answer NO.

To justify the last statement, notice that

1. If $P \not\equiv \varepsilon$, then $P(\overline{x}) \neq \overline{x}$ for some input tuple $\overline{x}$; then, for (at least) some particular input $\overline{x}$, P changes at least one register, say x_j; consider the corresponding input value, also denoted by x_j, and let x'_j be the value of the register x_j, after the execution of P; thus $x_j \neq x'_j$. We write

   $$P(x_1, \ldots, x_j, \ldots, x_n) = (x'_1, \ldots, x'_j, \ldots, x'_n)$$

   where $x_j \neq x'_j$. Denote by y'_j and y''_j the final values of y_j after the execution of $(P;Q)$ and $(Q;P)$, respectively.

   $$\begin{cases} (P;Q): & y'_j = y_j + x'_j \\ (Q;P): & y''_j = y_j + x_j \end{cases}$$

   Those two final values are different, $y'_j \neq y''_j$; P and Q do not commute.

2. If $P \equiv \varepsilon$, then, for *any* register y_j and *any* initial value of y_j, we have

   $$\begin{cases} (P;Q): & y'_j = y_j + x_j \\ (Q;P): & y''_j = y_j + x_j \end{cases}$$

Those two final values are equal, $y'_j = y''_j$; obviously, the registers x_j are also not modified, so that we have

$$\forall \overline{x}, \forall \overline{y},\ (P;Q)(\overline{x},\overline{y}) = (Q;P)(\overline{x},\overline{y})$$

that is, P and Q commute.

In other words, $P \equiv \varepsilon$ if and only if P and Q commute. □

Theorem 5 can be generalized to any reversible, total, register language. Note that its proof is constructive: it includes an algorithm for program equivalence, given an oracle for commutativity.

Chapter 10

Further formalization; normal forms

10.1 In the language of group theory

It is possible to use Group Theory to formalize some of the concepts and problems mentioned in this work. There are several groups related to the languages ⋆SRL and to its semantics.

- The group $G_{\star\mathrm{SRL}}$ of ⋆SRL programs, where the inverse of a program is its formal inverse, see Section 3.4.
- The group G_{nf} of the normal forms of ⋆SRL programs, see Section 10.3. Again, the inverse of a normal form is its formal inverse (which is a normal form).
- The group G_{impl} of bijections $\mathbb{Z}^\infty \to \mathbb{Z}^\infty$ that can be implemented in ⋆SRL.
- The group G_{fin} of bijections $\mathbb{Z}^\infty \to \mathbb{Z}^\infty$ such that in every bijection

  $$(x_1, x_2, \ldots) \to (x'_1, x'_2, \ldots)$$

  we have $x'_i = x_i$, except for a finite number of integers i.
- The group $G_{\mathbb{Z}^\infty \to \mathbb{Z}^\infty}$ of bijections $\mathbb{Z}^\infty \to \mathbb{Z}^\infty$.

Clearly, $G_{\text{nf}} \leq G_{\star\text{SRL}}$, $G_{\text{impl}} \leq G_{\text{fin}} \leq G_{\mathbb{Z}^\infty \to \mathbb{Z}^\infty}$, and there are homomorphisms $G_{\star\text{SRL}} \to G_{\text{impl}}$ and $G_{\text{nf}} \to G_{\text{impl}}$. Here, "$\leq$" and "$G \to G'$" denote respectively "is a sub-group of" and "there is an homomorphism from G to G'.

The set of programs P in the languages $\star$SRL forms the group $G_{\star\text{SRL}}$, where the group operation is program concatenation ";":

- The composition of P with Q is denoted by "$P;Q$" (Q after P).
- The identity program is ε, the empty (or unit) program.
- The inverse of P is P^{-1} as defined in Section 3.4, page 15. It should be emphasised that P^{-1}, the inverse of P, must be exactly the program defined by those rules. It is called the "formal inverse of P". For instance, `for` x(`dec` y) is not the (formal) inverse of `for` x(`inc` x; `dec` x; `inc` y).

The properties defining a group are easy to verify. The group G_{nf} is the group of the reduced (normal form) programs. To obtain the reduced program corresponding to P, successively delete every sequence of the form "$Q;Q^{-1}$" that occurs in the program (a tree-like structure), until no further deletion is possible; see Section 10.3, where it is shown that, using by any order the rules (6), (7), (8), and (9), the same "normal form" (reduced program) is always obtained.

The set of bijections $f : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$ where only a finite number of registers can be modified is the group G_{fin}. The identity of this group is ι, the identity bijection.

The meaning of the programs, as described above, corresponds to an homomorphism $h : \mathcal{G}(\text{SRL}) \to \mathcal{G}(\mathbb{Z}^\infty \to \mathbb{Z}^\infty)$; we thus have $[\![P]\!] = h(P)$. If two programs have the same normal form, their meaning is the same.

We can also say that each program *acts* on $\mathbb{Z}$, in the sense that each program $P \in \mathcal{G}(\text{SRL})$ "acts" as an element of the permutation group in $\mathbb{Z}^\infty$, see for instance the Chapter 4 of [Ros94].

There are analogies between the mapping $\mathcal{G}(SRL) \to \mathcal{G}(\mathbb{Z}^\infty \to \mathbb{Z}^\infty)$ and the mapping between a monoid and a group that is classically used to characterize group presentations, see for instance the Chapter 1 of [MKS76]. However, SRL programs have a structure that is more complex than the words of a monoid, and it is not clear how to extend the presentation theory to our mapping.

The equivalence problem can now be rephrased as: given the programs P and Q, is $P; Q^{-1} \in \ker(h)$?

10.2 Order of a program

In the groups $G_{\star\text{SRL}}$ or G_{nf} the *order of a program*[1] P, is the smallest integer n such that $P^n \equiv \varepsilon$ (or, equivalently $[\![P^n]\!] = \iota$), or ∞ if there is no such n. This definition was inspired by the concept of order of a group element; however, the order of P is not a property of P as a member of $G_{\star\text{SRL}}$ or G_{nf}, but is defined in terms of the semantics of P.

It is not difficult to see that for every positive integer n there are $\star$SRL programs with order n, and that there are programs of infinite order. To show this, consider the ESRL program that swaps the values of two registers; obviously, its order is 2. It is well known that with this program we can implement any permutation of the registers[2], so that for every positive integer n, it is possible to implement the circular register permutation $x_i \to x_{[(i+1) \mod n]}$. If follows that for every $n \in \mathbb{N}$ there are ESRL programs (and SRL programs; use the program of Example 4, page 23) with order n. Moreover, the program "`inc` x" has infinite order.

As far as we know, finding the order of a program may be an unsoluble problem, because in terms of decidability it is a problem related to the kernel problem (Definition 9, page 51). More precisely consider the problem

> OPr, ORDER OF A PROGRAM
> Input $\langle P, n\rangle$ where P is a $\star$SRL program whose order is known to be finite, and n is a positive integer.
> Question is n the order of P?

[1] This concept is similar to the order of an element of a group, not to be confused with the order of the group (its cardinality).

[2] Which, in the circuit terminology is sometimes called a "wire" permutation; it should not to be confused with a bijection $\mathbb{Z}^\infty \to \mathbb{Z}^\infty$.

This question can be expressed as

$$\overbrace{P \not\equiv \varepsilon,\, P^2 \not\equiv \varepsilon, \ldots,\, P^{n-1} \not\equiv \varepsilon}^{A} \wedge \overbrace{P^n \equiv \varepsilon}^{B} \;\; ?$$

In terms of the arithmetical hierarchy (see for instance [Odi89]), the sub-problem A belongs to the class Σ_1^0, while the sub-problem B belongs to the class Π_1^0. It follows that the problem OPr belongs to the class Δ_2^0.

However, it may be not decidable if an input $\langle P, n\rangle$ to the problem OPr is appropriate, more specifically, it may be undecidable if "a given $\star$SRL program has finite order". So we consider the "order 1" problem instead, "is the order of a given $\star$SRL program equal to 1?".

Theorem 6 *If the order 1 problem is recursively solvable, the kernel and the equivalence problems are recursively solvable.*

Proof. (trivial) The question associated with the order problem is "$P \equiv \varepsilon$?", which is the kernel problem. □

10.3 Normal forms

Inspired by the concept of reduced words of free groups, we define *reduction* from one program to another and *normal form* of a program. However, the normal form of a program does not fully characterize its semantics, in the sense that two programs with different normal forms may have the same semantics, that is, induce the same bijection of $\mathbb{Z}^\infty$ in $\mathbb{Z}^\infty$.

Consider a program P. A "sub-sequence" of P is an arbitrary sequence of contiguous instructions of P, possibly under the scope of one or more loop instructions. For instance, "$\texttt{inc}\, z$; $\texttt{dec}\, x$; $\texttt{dec}\, w$" is a sub-sequence of

$$\texttt{inc}\, x;\ \texttt{for}\, y(\texttt{inc}\, x;\ \texttt{inc}\, z;\ \texttt{dec}\, x;\ \texttt{dec}\, w)$$

A reduction step of P consists of one of the following transformations[3]

$$\texttt{inc}\,x;\ \texttt{dec}\,x \ \rightarrow\ \varepsilon \tag{6}$$

$$\texttt{dec}\,x;\ \texttt{inc}\,x \ \rightarrow\ \varepsilon \tag{7}$$

$$\texttt{for}\,x(\varepsilon) \ \rightarrow\ \varepsilon \tag{8}$$

$$\texttt{for}\,x(P);\ \texttt{for}\,x(P^{-1}) \ \rightarrow\ \varepsilon \tag{9}$$

where the left sides are sub-sequences of P. In (9), P^{-1} denotes the formal inverse of P. A program P is *normal* or *irreducible* (*freely reduced* in the Group Theory) if no reduction step can be applied to P.

If, by a sequence of reductions, the program P reduces to Q, and Q is irreducible (that is, Q is a normal form of P), we write

$$P \mapsto Q \quad \text{or} \quad Q = \rho(P)$$

where the unicity of the normal form (proved in Theorem 7) justifies the usage of the function"ρ".

A reduction step strictly decreases the length of P; it follows that every program has at least one normal form.

Two examples of reductions follow.

$$\texttt{inc}\,x;\ \texttt{dec}\,x;\ \texttt{for}\,x(\underline{\texttt{inc}\,y;\ \texttt{dec}\,y;}\,\texttt{inc}\,z) \ \xrightarrow{(6)}\ \texttt{inc}\,x;\ \texttt{dec}\,x;\ \texttt{for}\,x(\texttt{inc}\,z)$$

$$\texttt{for}\,x(\texttt{inc}\,y);\ \underline{\texttt{for}\,x(\texttt{dec}\,y);\ \texttt{for}\,x(\texttt{inc}\,y)} \ \xrightarrow{(9)}\ \texttt{for}\,x(\texttt{inc}\,y)$$

where the adjacent instructions that were chosen for the reduction are underlined and the rule used in the reduction is indicated over the arrow.

The reflexive, symmetric, and transitive closure of the "reduction step" relation (steps (6), (7), (8), or (9)) is denoted by "$\approx$". Thus $P \approx Q$ if Q can be obtained from P by insertions or deletions of elements of the set

$$\{\text{"}\texttt{inc}\,x;\ \texttt{dec}\,x\text{"},\ \text{"}\texttt{dec}\,x;\ \texttt{inc}\,x\text{"},\ \text{"}\texttt{for}\,x(\varepsilon)\text{"},\ \text{"}\texttt{for}\,x(R);\ \texttt{for}\,x(R^{-1})\text{"}\} \tag{10}$$

[3]For the language ESRL, we also need: "$\texttt{swap}(x,y);\ \texttt{swap}(x,y) \rightarrow \varepsilon$" and "$\texttt{swap}(x,y);\ \texttt{swap}(y,x) \rightarrow \varepsilon$".

where R is any appropriate $\star$SRL program. If $P \equiv Q$, we say that program P is *related* to program Q; in the Group Theory analogy, they are called *freely equal*. Two equivalent programs (Definition 3, page 16) are necessarily related, but the converse is not true.

Theorem 7 shows that a normal form of a program P is unique. It should be emphasized that two programs which do not have the same normal form may be (semantically) equivalent. For instance,

$$\begin{aligned} \texttt{inc}\, x;\, \texttt{dec}\, y;\, \texttt{dec}\, x &\equiv \texttt{dec}\, y \\ \rho(\texttt{inc}\, x;\, \texttt{dec}\, y;\, \texttt{dec}\, x) = \texttt{inc}\, x;\, \texttt{dec}\, y;\, \texttt{dec}\, x &\neq \rho(\texttt{dec}\, y) = \texttt{dec}\, y \end{aligned}$$

However, it is obvious that, two programs with the same normal form are equivalent.

The following theorem shows that, independently of the reduction steps chosen, the normal form obtained is always the same, that is,

$$(P \mapsto Q,\ P \mapsto R) \ \Rightarrow Q = R$$

Theorem 7 (Normal form theorem) *(i) Every* $\star$SRL *program* P *has a unique normal form that will be denoted by* $\rho(P)$*. (ii) If* Q *is the (formal) inverse of* P*, the normal form of* Q *is the (formal) inverse of the normal form of* P*,* $Q = P^{-1} \Rightarrow \rho(Q) = \rho(P)^{-1}$*.*

Proof. We consider only the SRL language. The inclusion of "swap" instructions causes no special difficulty.

Let $P = I_1; I_2; \ldots; I_n$. The proof is by induction on the depth (Definition 4, page 17) of the program.

Base case. If there are no loop instructions, we are essentially dealing with the problem of proving that the reduced words of a free group are unique, see the proof for instance in [Mil96] or [LS77]. The property (ii) is simple to prove in this case. It is a consequence of the following facts: the normal forms of P and of the formal inverse P^{-1} are unique; every possible reduction in P has a "mirror" reduction in P^{-1}, and vice-versa.

General case. Given a program P, the following algorithm results in a particular normal form, which will be called $\rho(P)$. Later we show that it is unique.

Input: program $P = I_1; I_2; \ldots; I_n$.
Output: $\rho(P)$, a normal form of P.

1. For every I_i of the form "$\texttt{for}\, x(Q)$", reduce Q to the normal form $\rho(Q)$.

2. Compute $\rho(P)$, by reading the instructions I_1, I_2,..., I_n from the left to the right, and using the following pattern matching between I_i and the last instruction of the program already "processed".

	$\rho(P)$	I_i	$\Rightarrow$	$\rho(P;\, I_i)$
(1)	$Q; \texttt{inc}\, x$	$\texttt{dec}\, x$		Q
(2)	$Q; \texttt{dec}\, x$	$\texttt{inc}\, x$		Q
(3)	Q	$\texttt{for}\, x(\varepsilon)$		Q
(4)	$Q; \texttt{for}\, x(P)$	$\texttt{for}\, x(P^{-1})$		Q
(5)	Q	I_i		$Q; I_i$

   Rule (5) is applied only when none of the previous rules match the pair $(\rho(P), I_i)$. In the end, $\rho(P)$ is obtained.

Regarding rule (4) and *using the induction hypothesis*, we have that

- If, in the sequence "$\texttt{for}\, x(R); \texttt{for}\, x(S)$", we apply some reduction rules to R and S such that we get "$\texttt{for}\, x(T); \texttt{for}\, x(T^{-1})$",
- then we can also get a sequence of the form "$\texttt{for}\, x(U); \texttt{for}\, x(U^{-1})$", by starting by reducing R and S to the normal form.

In other words, if R and S can be reduced to programs that are inverses of each other, then $\rho(R) = \rho(S)^{-1}$.

Thus, if we begin by reducing the programs R and S to their normal forms (step 1 of the algorithm!), no opportunities of applying Rule (4) are missed.

From now on, the proof may proceed along the same lines as the proof of Theorem 1.2 of [MKS76]. The main difference is that more equivalence steps are

allowed (insertion or deletion of elements of the set (10)). First, the following results are proved by induction, where $\rho(P)$ is the result of the previous algorithm.

(a) If P is irreducible, $\rho(P) = P$. The proof is by induction on $|P|$.

(b) For any programs P and Q and for any instruction sequence ω with one of the forms

$$\text{“}\texttt{inc}\,x;\ \texttt{dec}\,x\text{”},\ \text{“}\texttt{dec}\,x;\ \texttt{inc}\,x\text{”},\ \text{“}\texttt{for}\,x(\varepsilon)\text{”},\ \text{“}\texttt{for}\,x(P);\ \texttt{for}\,x(P^{-1})\text{”}$$

(see page 60), $\rho(P\omega Q) = \rho(PQ)$. The proof is by induction on $|Q|$.

If two programs are related, $P \approx Q$, they have the same normal form $\rho(P) = \rho(Q)$. In fact, if P can be obtained from Q by insertions or deletions of sequences of the form ω, we have by property (b), that the algorithm described above, applied to P or Q, gives the same result.

We have still to prove that, if by some other method, we get a normal form of P, say $\sigma(P)$, then $\sigma(P) = \rho(P)$. By definition of normal form applied to σ, the programs P and $\sigma(P)$ are necessarily related, and $\sigma(P)$ is irreducible. Then, by property (a), $\rho(P) = \rho(\sigma(P)) = \sigma(P)$; it follows that the algorithm described above has the following properties

- The normal form $\rho(P)$ obtained by the algorithm is “well defined”: $P \approx Q \ \Rightarrow\ \rho(P) = \rho(Q)$.
- The normal form $\rho(P)$ is unique: $P \approx V$ and V is irreducible (that is, V is a normal form of P) implies that $\rho(P) = V$.

□

Thus, in order to obtain the normal form, the reductions may be done in any order; in particular, it is possible to start by reducing all those programs to the normal form. Thus, we have directly

$$\texttt{for}\,x(\texttt{inc}\,n;\ \texttt{dec}\,n;\ \texttt{inc}\,p);\ \texttt{for}\,x(\texttt{dec}\,p;\ \texttt{inc}\,n;\ \texttt{dec}\,n) \to \varepsilon$$

or

$$\texttt{for}\, x(\texttt{inc}\, n;\ \texttt{dec}\, n;\ \texttt{inc}\, p);\ \texttt{for}\, x(\texttt{dec}\, p;\ \texttt{inc}\, n;\ \texttt{dec}\, n) \to$$
$$\texttt{for}\, x(\texttt{inc}\, p);\ \texttt{for}\, x(\texttt{dec}\, p;\ \texttt{inc}\, n;\ \texttt{dec}\, n) \to$$
$$\texttt{for}\, x(\texttt{inc}\, p);\ \texttt{for}\, x(\texttt{dec}\, p) \to \varepsilon$$

10.4 More equivalence transformations

When trying to prove that $P \equiv Q$, we may begin by computing the normal forms $\rho(P)$ and $\rho(Q)$. If they are equal, then $P \equiv Q$, but, as we said above, the converse does not hold. We describe some other program transformations that, in some cases, are sufficient to prove the equivalence of two programs.

After presenting some introductory examples in 10.4.1, we consider a more general transformation in (10.4.2).

In this section it is convenient to use a generalization of the concept "never modifies" (Definition 1, page 10).

Definition 11 *If the program P never modifies x, then, for any program Q (which can modify x), we will also say that $Q;\ P;\ Q^{-1}$ never modifies x. Notice that the variable x is not modified by the execution of $Q;\ P;\ Q^{-1}$.*

Example. The instruction "$\texttt{for}\, x(\texttt{inc}\, x;\ \texttt{inc}\, y;\ \texttt{dec}\, x)$" is now allowed. □

Comment. Definition 1 can of course be applied repeatedly so that, if P never modifies x, then, for any programs Q and R (which can modify x), we will also say that $R;\ Q;\ P;\ Q^{-1};\ R^{-1}$ never modifies x. □

10.4.1 Introductory examples

Commutativity. Assuming that the conditions of Theorem 4 are verified, we can use the transformation "$P;\ Q \to Q;\ P$".

Change the order of "inc" and "for". Suppose that P is a program that does not mention the variable x. By this we mean that the variable x does not

occur in P. This is a stronger condition than "P never modifies x" (Definition 1, page 10). Then,

$$\texttt{inc}\, x;\ \texttt{for}\, x(P) \equiv \texttt{for}\, x(P);\ P;\ \texttt{inc}\, x \tag{11}$$

Equivalence (11) is illustrated in Figure 9.

Figure 9: An equivalence between two programs: "$\texttt{inc}\, x;\ \texttt{for}\, x(P(\overline{y}))$" (left) and "$\texttt{for}\, x(P(\overline{y}));\ P(\overline{y});\ \texttt{inc}\, x$" (right).

This transformation is not valid if only the condition "P never modifies x" is assumed, as the following example shows:

$$\texttt{inc}\, x;\ \texttt{for}\, x(\texttt{for}\, x(Q)) \ \text{ where } Q \text{ does not mention } x \tag{12}$$

is not equivalent to

$$\texttt{for}\, x(\texttt{for}\, x(Q));\ \texttt{for}\, x(Q);\ \texttt{inc}\, x \tag{13}$$

In (12) the program Q is executed $(x+1)^2$ times, while in (13) it is executed $x(x+1)$ times, where x also denotes the initial value of x.

Changing the order of two loops. Suppose that the program P does not mention neither x nor y. Then,

$$\texttt{for}\, x(\texttt{inc}\, y);\ \texttt{for}\, y(P) \equiv \texttt{for}\, y(P);\ \texttt{for}\, x(P);\ \texttt{for}\, x(\texttt{inc}\, y) \tag{14}$$

Note. In (11) the instructions "$\texttt{inc}\, x$" and "P" commute; similarly, in (14) the instructions "$\texttt{for}\, x(P)$" and "$\texttt{for}\, x(\texttt{inc}\, n)$" commute.

10.4.2 Transformation $[P;\ \texttt{for}(\ldots)] \to [\texttt{for}(\ldots);\ P]$

Let P and Q be arbitrary $\star$SRL programs. We have[4] the equivalence

$$P;\ \texttt{for}\,x(Q) \ \equiv\ \texttt{for}\,x(P;\ Q;\ P^{-1});\ P \tag{15}$$

In fact,

$$\begin{aligned} & \texttt{for}\,x(P;\ Q;\ P^{-1});\ P \\ \equiv\ & \underline{P;\ Q;\ P^{-1}}\ \underline{P;\ Q;\ P^{-1}} \ldots \underline{P;\ Q;\ P^{-1}}\ P \\ \equiv\ & P;\ \overbrace{Q;Q;\ldots Q}^{x} \\ \equiv\ & P;\ \texttt{for}\,x(Q) \end{aligned}$$

The expressions in the second and third lines above are not programs; they are used to prove the (semantic) equivalence between two programs.

Example. In transformation (15) consider the case $P = \texttt{inc}\,y$, $Q = \texttt{for}\,y(\texttt{inc}\,z)$. We get

$$\texttt{inc}\,y;\ \texttt{for}\,x(\texttt{for}\,y(\texttt{inc}\,z)) \ \equiv\ \texttt{for}\,x(\texttt{inc}\,y;\ \texttt{for}\,y(\texttt{inc}\,z);\ \texttt{dec}\,y);\ \texttt{inc}\,y$$

The reader can verify that the tuple transformations associated to the LHS and to the RHS programs are the same, namely

$$\begin{cases} x' &= x \\ y' &= y+1 \\ z' &= z + x(y+1) \end{cases}$$

A transformation similar to (15) is

$$A;\ B \ \equiv\ (A;\ B;\ A^{-1});\ A \tag{16}$$

This trivial transformation holds unconditionally.

[4] TO DO: check if there are limitations for the application of equivalence (15). If the "proofs" below are right, they are unconditionally valid.

Analysis of a particular case

In (15) replace P by "$\texttt{for}\, x(P)$" and rename $x \to y$; we get

$$\texttt{for}\, x(P);\, \texttt{for}\, y(Q) \;\equiv\; \texttt{for}\, y(\texttt{for}\, x(P);\, Q;\, \texttt{for}\, x(P^{-1}));\, \texttt{for}\, x(P) \tag{17}$$

Notice that P may modify y. This equivalence holds even if $x = y$,

$$\texttt{for}\, x(P);\, \texttt{for}\, x(Q) \;\equiv\; \texttt{for}\, x(\texttt{for}\, x(P);\, Q;\, \texttt{for}\, x(P^{-1}));\, \texttt{for}\, x(P) \tag{18}$$

Comment. In general, it is not true that

$$\texttt{for}\, x(P);\, \texttt{for}\, x(Q) \equiv \texttt{for}\, x(P;\, Q)$$

However, if P and Q commute, the equivalence holds.

10.4.3 When is "$\texttt{for}\, x(P)$" equivalent to "$\texttt{for}\, y(Q)$?

Suppose that

$$\texttt{for}\, x(P) \equiv \texttt{for}\, y(Q) \tag{19}$$

- If $x = y$, the equivalence holds iff $P \equiv Q$. To show this it is enough to consider the case in which $x = y = 1$. Suppose now that $x \neq y$, and assume that the equivalence (19) holds.
- The LHS of (19) when $x = 0$ is equivalent to the identity program ε. The same must happen for the RHS of (19). Considering also the symmetrical reasoning, we have

$$\texttt{for}\, x(P)|_{[x=0]} \equiv \texttt{for}\, x(P)|_{[y=0]} \equiv \texttt{for}\, y(Q)|_{[y=0]} \equiv \texttt{for}\, y(Q)|_{[y=0]} \equiv \varepsilon \tag{20}$$

- Also "$\texttt{for}\, x(P)|_{[y=0]} \equiv \varepsilon$" iff "$P|_{[x=1,\, y=0]} \equiv \varepsilon$", and "$\texttt{for}\, y(Q)|_{[x=0]} \equiv \varepsilon$" iff "$Q|_{[x=0,\, y=1]} \equiv \varepsilon$". Thus, a condition equivalent to (20) is

$$P|_{[x=1,\, y=0]} \equiv \varepsilon,\; Q|_{[x=0,\, y=1]} \equiv \varepsilon \tag{21}$$

– On the other hand, if, for some values of the registers, P modifies y, the equivalence (19) does not hold. To see this, consider $x = 1$ and notice that Q never modifies y. Similarly, if, for some values of the registers, Q modifies x, the equivalence (19) does not hold.

STUDY SEQUENCES OF `inc`'S, `dec`'S, AND `swap`'S. IF `swap`'S. ARE NOT INCLUDED, THIS IS VERY SIMPLE, DUE TO THE COMMUTATIVITY OF THE `inc`'S, `dec`'S.

Final comment. Using the equivalences (15), (17), and (18) in a systematic way, it is possible it is possible to transform an arbitrary program in a sequence of instructions such that, at the top level,

– All "`inc` $-$", "`dec` $-$", and "`swap`$(-,-)$" instructions occur at the end, by lexicographic order.

– For each loop variable there is only one loop.

– The sequence of loops is lexicographically ordered (by the loop variable).

None of the transformations described in this section (together with the normal form reduction) is sufficient to prove in general the equivalence of two arbitrary programs.

Moreover, even if if it is possible to characterize a finite set of transformations that is sufficient to prove the equivalence of two programs, that only constitutes an algorithm when the order of their application is specified.

Chapter 11

Execution time

The execution time of a computation can be defined as the number of individual (elementary) instructions executed. Similarly to what is done in [MR67a], we ignore the overhead associated with loop instructions, and count 1 for each execution of an "increment", "decrement", or "swap" instruction.

Let an execution step, or simply a "step" be the execution of an increment, decrement, or swap instruction. The language SRL does not contain swap instructions, so that in an execution step the value stored in any register changes by at most by one. However, the execution of the swap instruction "$\texttt{swap}(x, y)$" causes arbitrarily large changes in the values stored in the registers x and y.

An upper bound for the execution time

Given a program, we can define an upper bound of the change of the values stored in the registers. Let P be a $\star$SRL program that uses the registers $x_1, \ldots, x_n$.

Let the largest and smallest value stored in a register, as a funtion of time, be respectively

$$\begin{aligned} \max\{\overline{x}\} &\stackrel{\text{def}}{=} \max_i\{x_i : i = 1, 2, \ldots, n\} \\ \min\{\overline{x}\} &\stackrel{\text{def}}{=} \min_i\{x_i : i = 1, 2, \ldots, n\} \end{aligned}$$

and define $f(t) \stackrel{\text{def}}{=} \max\{\overline{x}\} - \min\{\overline{x}\}$. An execution step (possibly a swap instruction), causes $f(t)$ to change by at most 1, so that

$$\Delta(P, \overline{x}) \stackrel{\text{def}}{=} |[\max_i(x'_i) - \min_i(x'_i)] - [\max_i(x_i) - \min_i(x_i)]| \leq T(P(\overline{x})) \tag{22}$$

where T is the execution time of the computation, and the x_i and x'_i are respectively the initial and final values of the registers. When dealing with $\star$SRL languages, all the registers used by a program must be considered when measuring the "change" caused by its execution. An appropriate measure of the growth of the values stored in the registers is $\Delta(P, \overline{x})$ as defined in (22).

Nesting of loops and the size of the registers

It is easy to show that for linear programs (programs in the class S_1) the execution time is linear in terms of $\Delta(P, \overline{x})$. However, for programs in S_2, the execution time may be not quadratic, it may not even be bounded by any polynomial (in terms of $\Delta(P, \overline{x})$). This is shown by the program of Example 9 (page 25), whose execution time is exponential.

This phenomenon – the appearance of exponential sized values (accompanied by an exponential execution time) with a nesting of loop instructions of only 2 has been studied before in the realm of primitive recursive functions; see for instance [Rob65], which, in particular, presents a sufficient condition for avoiding this behaviour. In the Example 9 the exponential size is due to repetition of two linked loops: "`for` b(`inc` a)" which increments a (assuming that $b > 0$) and "`for` a(`inc` b)" which increments b (assuming that $a > 0$). By "linked loops" we mean that the loop variable of each loop is changed by the execution of the other.

Normal forms and execution time

The transformation of a program P into its normal form $\rho(P)$ is based on the reduction steps (6), (7), (8), and (9); it follows that the normal form of a program is at least as fast as the program itself,

$$\forall \overline{x} : \; T(\rho(P(\overline{x}))) \leq T(P(\overline{x}))$$

However, the normal form of a program is not necessarily the fastest equivalent program, as shown by the simple program "`inc` x; `dec` y; `dec` x".

Clearly, the problem of finding the optimal program which is equivalent to a given program P is at least as difficult as the kernel problem (Definition 9, page 51), which, to our knowledge, may be undecidable; in fact the fastest program equivalent to P has execution time 0 if and only if the shortest program equivalent to P is ε.

Chapter 12

⋆SRL with register initialization

All the registers used in a ⋆SRL program must be included in the transformation that characterizes the semantics of the program. In this Chapter we relax this rule and study the transformations that can be implemented when some of the input registers are preset (before the program execution) and some registers are discarded when the execution finishes.

This technique of presetting some input registers (or wires) and discarding some output registers (or wires) has been widely used in some other models of computation, such as the reversible logical gates, see for instance [FT82].

The main question that we try to answer is: what is the class of unary functions that can be implemented using this technique in ⋆SRL? There is probably no easy, "closed form" answer, but in order to have at least a feel for that class of functions, we present some examples.

We will only study *unary functions*, that is, we consider only the case in which all registers except one are preset in the beginning of the computation.

Moreover only the language SRL will be used, because the "`swap`(x, y)" instruction can be implemented in SRL (if fixing/discarding registers is allowed) using an new additional register u and the transformation mentioned in Example 4, page 23; the register u is inverted, that is, $u \to -u$ and discarded. Whenever it occurs, the instruction "`swap`(x, y)" is assumed to be converted in this way to SRL.

Notice also that presetting the registers with 0 is sufficient.

Notation.

1. If a is an integer *constant*, the concatenation of a identical programs P will be denoted by P^a. For $a < 0$, P^a is defined as $(P^{-1})^{-a}$.

2. If a is an integer, we use "$\texttt{for}^a\, x(P)$" to denote the loop

$$\texttt{for}\, x(\ldots(\texttt{for}\, x(P))\ldots)$$

   with imbrication level a; for instance, "$\texttt{for}^2\, x(P)$", "$\texttt{for}^0\, x(P)$" and "$\texttt{for}^{-1}\, x(P)$" denote "$\texttt{for}\, x(\texttt{for}\, x(P))$", "$\varepsilon$" and "$\texttt{for}\, x(P^{-1})$", respectively. Notice the difference between "$\texttt{for}^a\, x(P)$" and "$(\texttt{for}\, x(P))^n$".

3. The input registers, the output registers and the register presets are specified as follows

   [input regs] [initializations] [PROGRAM CODE] [output regs]

   At the end of the computation, the non output registers are discarded.

4. If it is known that $f(x)$ can be implemented, we may use in a SRL program the notation "$\{f(x) \to u\}$" to represent the computation of $f(x)$ in the register u.

12.1 $y = ax + b$

For any $a, b \in \mathbb{Z}$, the function $y = ax + b$ can be implemented with the program

$$[x]\ [y = 0]\ (\texttt{for}\, x(\texttt{inc}\, y))^a;\ (\texttt{inc}\, y)^b\ [y]$$

Obviously this class of functions includes the constant functions.

12.2 $y = x \bmod 2$

The function $y = x \bmod 2$ can be implemented in the following way: (i) initialize the registers y and z with 0 and 1 respectively; (ii) exchange x times the values of y and z

$$[x]\ [y = 0,\ z = 1]\ \texttt{for}\, x(\texttt{swap}(y, z))\ [y]$$

12.3 $y = x \bmod m$ for fixed $m \geq 2$

Suppose that y, z, and w have initial values 0, 0, and 1, respectively. Consider the program $P =$ "$\texttt{swap}(z, w)$; $\texttt{swap}(y, z)$" with output $y+2z$; running it x times, we get successively

x	y	z	w	$y + 2z$
0	0	0	1	0
1	1	0	0	1
2	0	1	0	2
3	0	0	1	0
4	1	0	0	1
5	0	1	0	2
6	0	0	1	0
...	...	...	...	...

We see that the program "$\texttt{for}\, x(P)$" implements the function $y = x \bmod 3$. This is easily generated for any $m \geq 2$.

12.4 $y = (ax + b) \bmod m$ for fixed $m \geq 2$

Composition of 12.1 with 12.3.

12.5 $y = x^2 + b$

Use the program: $[x]\ [y = b]\ \texttt{for}\, x(\texttt{for}\, x(\texttt{inc}\, y))\ [y]$.

12.6 Polynomials

We present an example. To implement the polynomial $y = 2x^3 - 3x + 4$ the following program can be used

$$[x]\ [y = 4]\ (\texttt{for}^3\, x(\texttt{inc}\, y))^2;\ (\texttt{for}\, x(\texttt{dec}\, y))^3;\ [y]$$

It should now be easy to see how to implement an arbitrary polynomial.

12.7 Sums and products of functions

If $f(x)$ and $g(x)$ are implementable, then $f(x) + g(x)$ and $f(x) \times g(x)$ are also implementable. In fact,

$$[x]\ [y = 0]; \{f(x) \to w\};\ \{g(x) \to z\};\ \texttt{for}\, w(\texttt{inc}\, y);\ \texttt{for}\, z(\texttt{inc}\, y)\ [y]$$

implements $f(x) + g(x)$, while

$$[x]\ [y = 0]\ \{f(x) \to w\};\ \{g(x) \to z\};\ \texttt{for}\, w(\texttt{for}\, z(\texttt{inc}\, y))\ [y]$$

implements $f(x) \times g(x)$.

12.8 $y = \texttt{fib}(2x)$

As explained in Example 9, page 25, the Fibonacci sequence $\texttt{fib}(n)$ can be naturally extended for negative values of n, in the sense that there is an unique, doubly infinite sequence $\texttt{fib}(x)$ that satisfies for every $x \in \mathbb{Z}$ the three conditions: $\texttt{fib}(0) = 0$, $\texttt{fib}(1) = 1$, $\texttt{fib}(x+2) = \texttt{fib}(x) + \texttt{fib}(x+1)$. See Example 9.

The following program implements the function $y = f(x) = \texttt{fib}(2x)$

$$[x]\ [y = 0,\ z = 1]\ \texttt{for}\, x(\texttt{for}\, z(\texttt{inc}\, y);\ \texttt{for}\, y(\texttt{inc}\, z))\ [y]$$

For instance: $f(-2) = -3$, $f(4) = 21$. Note that y is an exponential function of x; in fact we have $y \geq 2^x$ for $x \geq 3$.

Comment. It goes without saying that

- Many other functions can be obtained by composing the ones we have defined. For instance, if g and h are implementable, the following functions are also implementable

$$\begin{aligned} f(x) &= 6((\texttt{fib}((2x)^5 + 9)) \bmod 10)^3 + 2x \\ u(x) &= \texttt{if } x \text{ is even } \texttt{then } g(x) \texttt{ else } h(x) \end{aligned}$$

- Many functions are not implementable (even using the fix/discard technique).

- There are many functions for which it is not yet known if they are implementable.

Chapter 13

A machine that runs ⋆SRL programs

13.1 Assembly language

Programs written in the languages ⋆SRL can be translated into a simple assembly language. The purpose of translating a ⋆SRL program into a very low level machine is twofold

- To prove that a finite memory device is enough to implement ⋆SRL. The amount of memory depends only on the program, and not on its execution; in particular, there are no stacks.
- To show that there is no need to "evaluate" the inverse of parts of a program, as it would seem from the semantics of the instruction "`for` $x(P)$" when the value stored in x is negative, see Section 3.5.3, page 18. Furthermore, during the execution of a program, an invert switch can be actuated at any time by an external operator, causing the program to change its execution "direction" (from "forward" to "backward").

At each instant of the execution of the assembly code, the value δ is either 1 (forward execution) or -1 (backward execution), and this value changes at the beginning of the execution of the loop instruction "`for` $x(P)$" when $x < 0$. More-

over if, at any time during the execution, the value of δ is externally changed "by the user" (from 1 to -1, or from -1 to 1), the program changes the execution direction. If the execution was proceeding forward and δ is changed to -1, all the instructions done so far are undoed and the program finally halts at the begin of the code, with the initial values stored in the registers. We call this behaviour "instant reversibility", a strong and simple form of reversibility.

We illustrate the translation to assembly language with the program of Example 9, namely

$$\texttt{for}\, r(\texttt{for}\, b(\texttt{inc}\, a);\, \texttt{for}\, a(\texttt{inc}\, b))$$

See Figure 10.

⋆SRL	label	assembly language
`for` r {	A :	`FOR` r, c, A, B, C
	B :	`BOL` r, c, A, B, C
`for` b {	D :	`FOR` b, c', D, E, F
	E :	`BOL` b, c', D, E, F
`inc` a		`INC` a
}	F :	`EOL` b, c', D, E, F
`for` a {	G :	`FOR` a, c'', G, H, I
	H :	`BOL` a, c'', G, H, I
`inc` b		`INC` b
}	I :	`EOL` a, c'', G, H, I
}	C :	`EOL` r, c, A, B, C
		`HLT`

Figure 10: Translation of the program $\texttt{for}\, r(\texttt{for}\, b(\texttt{inc}\, a);\, \texttt{for}\, a(\texttt{inc}\, b))$ in ASRL.

There are two variables associated with the execution of a program: `pc`, the program counter, and δ which was explained above.

The assembly language is called ASRL (assembly for ⋆SRL) and the operation mnemonics are `INC`, `DEC`, `SWAP`, `FOR`, `BOL` (begin of loop), `EOL` (end of loop), and `HLT`. As exemplified above, each "$\texttt{for}\, x(P)$" is translated into the sequence

A `FOR` x, c, A, B, C
B `BOL` x, c, A, B, C
... translation of P ...
C `EOL` x, c, A, B, C
The variable c is a loop counter.

We describe in Figure 11 how each ASRL instruction is executed by the corresponding low level machine. In the Appendices A.1, A.2 and A.3 this execution is detailed.

13.2 Assembly language: inverting the execution direction

A fixed amount of memory is sufficient to implement the inversion of the execution direction. Appendices A.2 and A.2 shows implementations of this inversion mechanism in the languages Haskell and Prolog respectively.

If the inversion of direction occurs, the program runs backwards undoing all the instructions executed so far, and finally halting at the beginning of the code (at the first "halt" instruction mentioned below), with every register containing the corresponding initial value.

Modified assembly language

In our implementation of the inversion mechanism, the user specifies

- The program to be executed.
- The input data.
- The number of elementary steps after which there is an inversion.

It is convenient to change slightly the form of the Assembly program, as follows.

1. $\delta = 1$ `// go forward`

2. `pc` $= 1$ `// first instruction`

3. Execute the instruction I corresponding to the label `pc`.

   (a) $I =$ HLT: stop the execution.

   (b) $I =$ `INC` x: $x \leftarrow x + \delta$;
   `pc` $\leftarrow$ `pc` $+ \delta$; `goto` instruction with label `pc`.

   (c) $I =$ `DEC` x: $x \leftarrow x - \delta$;
   `pc` $\leftarrow$ `pc` $+ \delta$; `goto` instruction with label `pc`.

   (d) $I =$ `SWAP`(x, y): $(x, y) \leftarrow (y, x)$;
   `pc` $\leftarrow$ `pc` $+ \delta$; `goto` instruction with label `pc`.

   (e) $I =$ `FOR` x, c, A, B, C:
   $c \leftarrow -1$;
   `if` $x < 0$ `then` $\delta = -\delta$; `// change direction`
   `pc` $\leftarrow B$; `goto` instruction with label `pc`.

   (f) I=`BOL` x, c, A, B, C
   $c \leftarrow c + 1$;
   `if` $c \geq |x|$ `then` { `// the loop instruction ended`
   `if` $x < 0$ `then` $\delta = -\delta$;
   `// set` δ `to its previous value`
   `if` $\delta = 1$ `then` `pc` $\leftarrow C + 1$;
   `// ↓, go to the end of loop+1`
   `else pc` $\leftarrow A - 1$; `// ↑, go to the begin of loop-1`
   `goto` instruction with label `pc`
   }
   `else` { `// continue the loop`
   `if` $\delta = 1$ `then goto` instruction with label $B + 1$;
   `else goto` instruction with label $C - 1$
   }

   (g) I=`EOL` x, c, A, B, C:
   `if` $\delta = 1$ `then goto` B
   `// ↓, end of a loop,` $\delta = 1$`, back to BOL`
   `else goto` A
   `// ↑, go to FOR inst. (entering from below)`

Figure 11: Execution of ASRL instructions.

– There is an extra "halt" instruction at the beginning. This allows the halting of programs running backwards.

– Associated with every loop instruction there is an extra bit to indicate if the inversion "processing" has already occurred. That processing is essentially associated with the manipulation of the loop counter: step and finishing values.

Some examples of these transformed assembly programs can be seen in Appendix A.4.

Inversion bits

The names of the inversion bits must satisfy the following rule: nested loop instructions must have different names. This is because, in a program like

$$\texttt{for}\, x(P;\, \texttt{for}\, y(Q \star R);\, S);\, \texttt{for}\, z(T)$$

where the inversion takes place at the point marked "$\star$", it is necessary to

(i) "Invert" the $\texttt{for}\, y(\cdot)$ loop

(ii) Then, and after running P backwards once, "invert" the $\texttt{for}\, x(\cdot)$ loop.

On the other hand, the name of the inversion bits can be the same for loop instructions in sequence, possibly intermixed with other ("inc", "dec", or "swap") instructions.

In the example above, the names of the inversion bits could be as indicated.

$$\texttt{for}\, [x/\texttt{i1}](P;\, \texttt{for}\, [y/\texttt{i2}](Q \star R);\, S);\, \texttt{for}\, [z/\texttt{i1}](T)$$

where $[x/\texttt{i1}$ means that the name `i1` is associated with the loop with counter x.

Note. In this example, where the inversion occurs after the execution of Q, the program S is never executed; however, if the initial value of y is negative, the program S is executed.

Appendix A

⋆SRL implementations

This appendix is divided in the following sections

- Section A.1: the programming languages Haskell and Prolog are used to represent and test some of the ⋆SRL programs used in this work.
- Section A.2: listing of a Haskell program that interprets ⋆SRL programs translated to Prolog terms, "intermediate language".
- Section A.3: listing of a Prolog program that interprets ⋆SRL programs translated to Prolog terms, "intermediate language".
- Section A.4: listing of a Prolog program that interprets the assembly language (available from the author).
- Section A.6: listing of a Prolog program that interprets a slightly modified assembly language in which it is possible, at any instant, to invert the direction of the program (available from the author).
- Section A.7: listing of the "`sage`" program that generated Figure 6 (page 43) (available from the author).

We first list the representation of three of the programs presented in this work, Then, we present the interpreter code.

A.1 Representation of some of programs presented in this work

A.1.1 Examples in Haskell

Included below are the programs `sign2` and `fibo` that correspond to the examples 7 (page 24) and 9 (page 25) respectively. The program syntax is defined in A.2.

```
-- Execute srl programs
sign2 = [FOR 0 [INC 1],
         FOR 1 [DEC 0, DEC 0],
         FOR 0 [INC 1],
         FOR 1 [DEC 0, DEC 0]
        ]

test_sign2 = exec pt [2,5]
----------------
fibo   = [ FOR 2 [FOR 0 [INC 1], FOR 1 [INC 0]] ]
test_1 = exec p8 [0,1,10]
test_2 = [exec p8 [0,1,n] | n <- [-5..5]]
test_3 = map (!!0) [exec p8 [0,1,n] | n <- [-10..10]]
```

A.1.2 Examples in Prolog

The program syntax is defined in A.3.

A.1.3 Example 3, page 22

```
%   (Swap two pairs of registers)
prog(e4,[
        for(a,[i(c)]),  for(c,[d(a)]),  for(b,[i(d)]),
        for(d,[d(b)]),  for(c,[i(b)]),  for(d,[i(a)]),   % state M1
%
        for(a,[i(c)]),  for(c,[d(a)]),  for(b,[i(d)]),
        for(d,[d(b)]),  for(c,[i(b)]),  for(d,[i(a)]),   % state M2
%
        for(a,[d(c)]),  for(b,[d(c)]),
        for(a,[d(d)]),  for(b,[d(d)])]).                 % "corrections"
% Example:
mem(e4, [(a,-11),(b,-27),(c,-3),(d,44)]).
%   ->  [(a,-27),(b,-11),(c,44),(d,-3)]
```

A.1.4 Example 4, page 23

```
%   (Swap two registers and change the sign of anoher)
prog(e5,[
        for(a,[i(c)]),  for(c,[d(a)]),  for(b,[i(a)]),
        for(a,[d(b)]),  for(c,[i(b)]),  for(b,[d(c)]),
        for(c,[d(a)]),  for(c,[i(b),i(b)])               ]).
% Example:
mem(e5,[(a,30),(b,17),(c,12 )]).
%   -> [(a,17),(b,30),(c,-12)]
```

A.1.5 Example 9, page 25

```
%   (The Fibonacci example)
prog(e6,[for(n,[
 for(a,[i(b)]),
 for(b,[i(a)]),
                ]
            )
        ]
```

```
    ).
% Example:
mem(e6,[(a,0),   (b,1),   (n,10)]).
%   -> [(a,6765),(b,4181),(n,10)]
```

A.2 Intermediate language interpreter (Haskell)

```
-- define programs and instructions
data Inst = INC Int | DEC Int | SWAP Int Int | FOR Int Prog
  deriving (Show)
type Prog = [Inst]

-- invert a program
inv :: Prog -> Prog
inv [] = []
inv (inst:insts) = inv insts ++ [inv_i inst]

inv_i :: Inst -> Inst
inv_i (INC r) = (DEC r)
inv_i (DEC r) = (INC r)
inv_i (SWAP r s) = (SWAP r s)
inv_i (FOR r p) = FOR r (inv p)

-- execute: exec program registers
exec []    rs = rs
exec (i:p) rs = exec p (execi i rs)

-- execute one instruction
execi (INC r) rs = take r rs ++ [(rs!!r) + 1] ++ drop (r+1) rs
execi (DEC r) rs = take r rs ++ [(rs!!r) - 1] ++ drop (r+1) rs
execi (SWAP r s) rs = set s (val r rs) (set r (val s rs) rs)
execi (FOR r p) rs =
    execfor (val r rs) p rs

execfor vr p rs
  | vr==0 = rs
  | vr>0  = exec p (execfor (vr-1) p rs)
  | vr<0  = execfor (-vr) (inv p) rs

val r rs   = rs!!r
set r v rs = take r rs ++ [v] ++ drop (r+1) rs
```

A.3 Intermediate language interpreter (Prolog)

```
%-----------------------------------------------
% STRUCTURE: examples
%            program
% an example:
% prog(1,[i(a),i(a),d(b)]). 1: name of the example
% mem(1,[(a,1),(b,10)]).   initial var values, a=1, b=10
%   this example: a'=3, b'=9
%-----------------------------------------------

test(N) :-
    prog(N,P),
    mem(N,M),
    execute(P,M,Mf),
    write('Example '),   write(N), write(': '), nl,
    write('  Input:  '), write(M),  nl,
    write('  Output: '), write(Mf), nl.

rtest(N) :-
    prog(N,P),
    invert(P,Pr),
    write(Pr).

%-----------------------------------------------
% execute(P,M,M1)
%    The execution of program "P" with initial "memory M",
%    results in memory M1
%-----------------------------------------------
execute([],M,M).
execute([I|P],M,Mf) :-
    exec1(I,M,Mi),
    execute(P,Mi,Mf).

%-----------------------------------------------
% increment i(R)
exec1(i(R),M,M1) :-
    !,
    value(R,M,V),
    V1 is V+1,
    set(R,V1,M,M1).

% decrement i(R)
exec1(d(R),M,M1) :-
    !,
    value(R,M,V),
    V1 is V-1,
    set(R,V1,M,M1).

% change sign inv(R)
exec1(inv(R),M,M1) :-
    value(R,M,V),
```

```
    V1 is -V,
    set(R,V1,M,M1).

% "swap" instruction s(R,S) instruction only legal on the ESRL language
exec1(s(R,S),M,M1) :-
    !,
    value(R,M,U),
    value(S,M,V),
    set(R,V,M, Mi),
    set(S,U,Mi,M1).

% "for" instruction: execute "R" times the program "P"
exec1(for(R,P),M,M1) :-
    value(R,M,V),
    execfor(V,P,M,M1).
% Display or not (according to "trace") "L"
exec1(disp(_),M,M) :-
    not(trace(on)),
    !.
exec1(disp(L),M,M) :-
    !,
    values(L,M,Lv),
    write(Lv),
    nl.

%  "for" instruction
%  If V<0, invert P
execfor(V,P,M,M1) :-
    V>=0,
    !,
    execloop(V,P,M,M1).
execfor(V,P,M,M1) :-
    !,
    Vn is -V,
    invert(P,Pr),
    execloop(Vn,Pr,M,M1).

execloop(0,_,M,M) :-
    !.
execloop(C,P,M,Mf) :-
    execute(P,M,M1),
    C1 is C-1,
    execloop(C1,P,M1,Mf).

%-----------------------------------------------
% Invert a program, invert an instruction
%-----------------------------------------------
invert([],[]).
invert([I|P],Pr) :-
    invert_one(I,Ir),
    invert(P,P1),
    append(P1,[Ir],Pr).
```

```
invert_one(i(R),d(R)) :-
    !.
invert_one(d(R),i(R)) :-
    !.
invert_one(inv(R),inv(R)) :-
    !.
invert_one(s(R,S),s(R,S)) :-
    !.
invert_one(for(R,P),for(R,Pr)) :-
    invert(P,Pr).

%-----------------------------------------------
% Memory access
%-----------------------------------------------
% value(R,V): V is the value of R
value(R,[(R,V)|_],V) :-
    !.
value(R,[_|M],V) :-
    value(R,M,V).

set(R,V,[(R,_)|M],[(R,V)|M]) :-
    !.
set(R,V,[A|M],[A|M1]) :-
    set(R,V,M,M1).

values([],_,[]) :-
    !.
values([A|L],M,[V|Lv]) :-
    !,
    value(A,M,V),
    values(L,M,Lv).
values(A-B,M,Va-Vb) :-
    !,
    values(A,M,Va),
    values(B,M,Vb).
values(A,M,V) :-
    atomic(A),
    !,
    value(A,M,V).

%-----------------------------------------------
append([],L,L).
append([A|L],L1,[A|L2]) :-
    append(L,L1,L2).

len([],0).
len([_|A],L) :-
    len(A,L1),
    L is L1+1.
```

A.4 Assembly language interpreter and examples (Haskell)

A.4.1 Examples

We present three assembly language programs written in Haskell, see also A.4.2.

```
p0 = [HLT  0,           --          x <-> 0  y <-> 1
      INC  1 1,         -- 1 INC y
      DEC  2 0,         -- 1 DEC x
      SWAP 3 0 1,       -- 1 SWAP x y
      HLT  4
     ]
t0 = exec p0 [10, 20]
------------------------
p2 = [HLT 7,            --      x <-> 0  y <-> 1
      FOR 3 0 2 3 4 5,  --  3 FOR x
      BOL 4 0 2 3 4 5,  --  4 BOL
      INC 6 1,          --  6   INC y
      EOL 5 0 2 3 4 5,  --  5 EOL
      HLT 9             --  9 HLT
     ]
t2 = exec p2 [-5, 1]
------------------------
p8 = [HLT 0 ,               -- 0  r=15,   x=16,  y=17
      FOR 1  15 21 1 2 11,  -- 1  c1=21, c2=22, c3=23
      BOL 2  15 21 1 2 11,  -- 2
--
      FOR 3  16 22 3 4 6 ,  -- 3
      BOL 4  16 22 3 4 6 ,  -- 4
      INC 5  17 ,           -- 5
      EOL 6  16 22 3 4 6 ,  -- 6
--
      FOR 7  17 23 7 8 10,  -- 7
      BOL 8  17 23 7 8 10,  -- 8
      INC 9  16 ,           -- 9
      EOL 10 17 23 7 8 10,  -- 10
--
      EOL 11 15 21 1 2 11,  -- 11
      HLT 12                -- 12
    ]

--     index: 0 1 2 3 4 5 6 7 8 9  0 1 2 3 4   5 6 7  8 9 0  1 2 3
t8 = exec p8 [0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0, -4,0,1, 0,0,0, 0,0,0]
```

A.4.2 Haskell interpreter

Listing of the assembly interpreter, written in Haskell.

```
-- define programs and instructions
type Label = Int
data Dir = FORW | BACK

--               Lab   Reg Cnt Lab Lab Lab
data Inst = HLT  Label |
            INC  Label Int |
            DEC  Label Int |
            SWAP Label Int Int |
            FOR  Label Int Int Int Int Int |
            BOL  Label Int Int Int Int Int |
            EOL  Label Int Int Int Int Int
  deriving (Show)
type Prog = [Inst]

-- execute: exec program registers
exec p m = execute FORW (first_ins p) p m

execute _    (HLT _)   p m = m
execute FORW (INC  l r) p m
  = execute FORW (next_ins p l) p (increment m r)
execute FORW (DEC  l r) p m
  = execute FORW (next_ins p l) p (decrement m r)
execute FORW (SWAP l r r') p m
  = execute FORW (next_ins p l) p (replace m r r')
execute FORW (FOR l r i a b c) p m
  | val m r >= 0 = execute FORW (inst_lab p b) p (set m i 0)
  | otherwise    = execute BACK (inst_lab p b) p (set m i 0)
execute FORW (BOL l r i a b c) p m
  | val m i < abs(val m r) = execute FORW (next_ins p b) p mi
  | otherwise              = outFOR  FORW r a b c p m
  where mi = increment m i
execute FORW (EOL l r _ _ b _) p m =
     execute FORW (inst_lab p b) p m
--
execute BACK (INC  l r) p m
  = execute BACK (prev_ins p l) p (decrement m r)
execute BACK (DEC  l r) p m
  = execute BACK (prev_ins p l) p (increment m r)
execute BACK (SWAP l r r') p m
  = execute BACK (prev_ins p l) p (replace m r r')
execute BACK (FOR l r i a b c) p m
  | val m r >= 0 = execute BACK (inst_lab p b) p (set m i 0)
  | otherwise    = execute FORW (inst_lab p b) p (set m i 0)
execute BACK (BOL l r i a b c) p m
  | val m i < abs(val m r) = execute BACK (prev_ins p c) p mi
  | otherwise              = outFOR  BACK r a b c p m
```

```
  where mi = increment m i
execute BACK (EOL _ _ _ a _ _) p m =  -- EOL reached from below
     execute BACK (inst_lab p a) p m

-- when a FOR loop finishes...
outFOR FORW r a b c p m
          | val m r >= 0 = execute FORW (next_ins p c) p m
          | otherwise    = execute BACK (prev_ins p a) p m
outFOR BACK r a b c p m
          | val m r >= 0 = execute BACK (prev_ins p a) p m
          | otherwise    = execute FORW (next_ins p c) p m
--
inst_lab (i:p) l
  | label i == l   = i
  | otherwise      = inst_lab p l

next_ins (i:p) l
  | label i == l   = head p
  | otherwise      = next_ins p l

prev_ins (ia:i:p) l
  | label i == l   = ia
  | otherwise      = prev_ins (i:p) l

first_ins p = p!!1 -- first HALT ignored
--
label (HLT  l)     = l
label (INC  l _)   = l
label (DEC  l _)   = l
label (SWAP l _ _) = l
label (FOR  l _ _ _ _ _) = l
label (BOL  l _ _ _ _ _) = l
label (EOL  l _ _ _ _ _) = l

--
increment m r =
    take r m ++ [(m!!r) + 1] ++ drop (r+1) m

decrement m r =
    take r m ++ [(m!!r) - 1] ++ drop (r+1) m

replace m r r' = set (set m r (val m r')) r' (val m r)

val  m r   = m!!r
set  m r v = take r m ++ [v] ++ drop (r+1) m
```

A.5 Assembly language interpreter (Prolog)

Program available from the author (email address armandobcm@yahoo.com).

A.6 Interpreter with inversion (Prolog)

Program available from the author (email address armandobcm@yahoo.com).

A.7 Program in "sage" that generated Figure 6 (page 43)

Program available from the author (email address armandobcm@yahoo.com).

Bibliography

[Abr01] Samson Abramsky. A structural approach to reversible computation. Technical Report PRG-RR-01-09, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, February 2001.

[Art91] Michael Artin. *Algebra*. Prentice-Hall, 1991.

[BB96] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[Ben73] Charles H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 6:525–532, 1973.

[Ben07] Charles H. Bennett. Notes on Landauer's principle, reversible computation, and Maxwell's demon. *IBM Research Division, Yorktown Heights, NY 10598, USA*, 2007.

[Boo57] William W. Boone. Certain simple unsolvable problems in Group Theory. *Nederl. Akad. Wetensch Proc. Ser. A.*, 57 pp 231–236 (1954), 57 pp 231–236, 492–497 (1954), 58 pp 252–256, 571–577 (1955), 60 pp 22–26, 227–232 (1957), 1954-1957.

[Cle63] J. P. Cleave. A hierarchy of primitive recursive functions. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 9:331–345, 1963.

[CLR01] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Second edition. MIT Press and McGraw-Hill, 2001.

[Dav85] Martin Davis. *Computability and Unsolvability*. Dover, 1985.

[Deh12] Max Dehn. Über unendliche diskontinuierliche gruppen. *Math. Ann.*, 71:116–144, 1912. Translated by J. Stilwell, *On infinite discontinuous groups*, in *Papers on Group Theory and Topology*, Springer Verlag.

[FT82] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.

[GA06] Alexander Green and Thorsten Altenkirch. From reversible to irreversible computations. *QPL06, Workshop on Quantum Physics and Logic*, 2006.

[GN78] Bernhard Goetze and Werner Nehrlich. Loop programs and classes of primitive recursive functions. In *MFCS*, pages 232–238, 1978.

[GN81] Bernhard Goetze and Werner Nehrlich. The number of loops necessary and sufficient for computing simple functions. *Elektronische Informationsverarbeitung und Kybernetik*, 17(7):363–376, 1981.

[Grz53] A. Grzegorczyk. Some classes of recursive functions. *Rozprawy Matematyczne*, 4:1–45, 1953. Introduction. In this paper an increasing sequence $\mathcal{E}^0$, $\mathcal{E}^1$,... of classes of recursive functions is examined. Each class $\mathcal{E}^n$ is closed under the operations of substitution and under the operation of limited recursion. The initial functions are primitive recursive ones. Therefore $\mathcal{E}^n \subset \mathcal{R}$, where $\mathcal{R}$ is the class of primitive recursive functions. Strictly speaking $\mathcal{R} = \cup_n \mathcal{E}^n$. Hence in the definition of the class $\mathcal{R}$ the operation of recursion cannot be eliminated or exchanged into the operation of limited recursion. The classes $\mathcal{E}^0$ and $\mathcal{E}^3$ will be examined in particular. For each function $f \in \mathcal{E}^0$ there exists a number k_0 such that $f(n) < n + k_0$. However, each recursive enumerable set is enumerable by some function of the class $\mathcal{E}^0$. We start with the investigation of the class $\mathcal{E}^3$. It is the class of elementary computable functions of Kalmar.

[Her69] Hans Hermes. *Enumerability, Decidability, Computability.* Springer-Verlag, 1969.

[Hua82] Loo Keng Hua. *Introduction to Number Theory.* Springer-Verlag, 1982.

[Lan61] Ralf Landauer. Dissipation and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[Lec63] Yves Lecerf. Machines de Turing réversibles. Récursive insolubilité en $n \in \mathbb{N}$ de l'équation $u = \theta^n$ ou θ est un isomorphisme de codes. *Comptes Rendus*, 257:2597–2600, 1963.

[LS77] Roger Lyndon and Paul Schupp. *Combinatorial Group Theory*. Springer, 1977.

[LTV98] Ming Li, John Tromp, and Paul Vitányi. Reversible simulation of irreversible computation. *Physica D*, 120:168–176, 1998.

[Mat03] Armando B. Matos. Analysis of a simple reversible language. *Theoretical Computer Science*, 290(3):2063–2074, 2003.

[Mat12] Armando B. Matos. The efficiency of primitive recursive functions – a programmer's view, 2012. Not yet available.

[Mey65] A. R. Meyer. Depth of nesting and the Grzegorczyk hierarchy. *Notices of the American Mathematical Society*, 12:342, 1965. ABSTRACT. Loop programs have the property that an upper bound on the running time of a program is determined by its structure. Each program consists only of assignment and iteration (loop) statements, but all the arithmetic functions commonly encountered in digital computation can be computed by Loop programs. A simple procedure for bounding the running time is shown to be best possible; some programs actually achieve this bound, and it is effectively undecidable whether a program runs faster than the bound. The complexity of functions can be measured by the loop structure of the programs which compute them. The functions computable by Loop programs are precisely the primitive recursive functions.

[Mil96] J. S. Milne. Group theory, 1996. Notes written for a first-year graduate algebra course.

[MKS76] Wilhelm Magnus, Abraham Karrass, and Donald Solitar. *Combinatorial Group Theory, Presentations of Groups in Terms of Generators and Relations*. Dover, 1976. Second revised edition.

[Mol73] Robert Moll. *Complexity classes of recursive functions*. PhD in Mathematics, Massachusetts Institute of technology, 1973. Contents of Chapter 1, "A survey of work on subrecursive hierarchies and subrecursive degrees": (i) ω-hierarchies of primitive recursive functions. (ii) ω-hierarchies

of elementary functions. (iii) Transfinite hierarchies. (iv) Subrecursive degrees.

[Mor98] Bernard Moret. *The Theory of Computation*. Addison-Wesley, 1998.

[MP11] Armando B. Matos and António Porto. Ackermann and the superpowers (revised in 2011). *ACM SIGACT*, 12(Fall 1980), 1980/1991/2011. url: `http://www.dcc.fc.up.pt/~acm/ack.pdf`.

[MR67a] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. *Proceedings of 22nd National Conference of the ACM*, pages 465–469, 1967. (From the Intruduction) Although Loop [a language described in this paper]programs cannot compute all the computable functions, they can compute all the primitive recursive functions. The functions computable by Loop programs are, in fact, precisely the primitive recursive functions. Several of our results can be regarded as an attempt to make precise the notion that the complexity of a primitive recursive function is apparent from its definition or program. This property is one of the reasons that the primitive recursive functions are used throughout the theory of computability, for [. . .] knowing that a function is computable is not very useful unless one can tell how difficult the function is to compute. A bound on the running time of a Loop program provides a rough estimate of the degree of difficulty of the computation defined by the program. Loop programs are so powerful that our bounds on running time cannot be of practical value-for functions computable by Loop programs are almost wholly beyond the computational capacity of any real device. Nevertheless they provide a good illustration of the theoretical issues involved in estimating the running time of programs, and we believe that readers with a practical orientation may find some of the results provocative.

[MR67b] A. R. Meyer and D. M. Ritchie. Computational complexity and program structure. *IBM Research Report RC 1817*, 1967.

[Nov55] P. S. Novikov. On the algorithmic unsolvability of the word problem in group theory. *Proceedings of the Steklov Institute of Mathematics*, 44:1–143, 1955.

[Odi89] Piergiorgio Odifreddi. *The Theory of Functions and Sets of Natural*

Numbers. Studies in Logic and the Foundations of Mathematics. Elsevier North Holland, 1989.

[RK66] B. Rotman and G. T. Kneebone. *The Theory of Sets and Transfinite Numbers*. Elsevier, 1966.

[Rob47] Raphael Robinson. Primitive recursive functions. *Bull. Amer. Math. Soc.*, 53(10):925–942, 1947.

[Rob65] Joel Robbin. *Subrecursive Hierarchies*. PhD in Mathematics, Princeton University, 1965. ABSTRACT. The classification problem for recursive functions is the problem of assigning ordinals to recursive functions as a measure of their complexity. In this paper we consider three approaches to this problem: the ordinal recursion hierarchy, the extended Grzegorczyk hierarchy, and the Kleene subrecursive hierarchy. We obtain characterizations of the nested n-fold recursive functions in terms of each of these hierarchies. In the last section of the paper we show some of the problems that arise when we try to generalize these hierarchies. A characterization of the nested n-fold recursive functions in terms of computational complexity on a Turing machine is also given in the paper.

[Ros94] John S. Rose. *A Course on Group Theory*. Dover, 1994.

[Sch98] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS, 1997.

[Tsi70] D. Tsichritzis. The equivalence problem of simple programs. *Journal of the ACM*, 17(4):729–738, 1970. ABSTRACT. Many problems, some of them quite meaningful, have been proved to be recursively unsolvable for programs in general. The paper is directed towards a class of programs where many decision problems are solvable. The equivalence problem has been proved to be unsolvable for the class L_2 of Loop programs defining the class of elementary functions. A solution is given for the class L_1 defining the class of simple functions. Further, a set of other decision problems not directly connected with the equivalence problem is investigated. These problems are found again to be unsolvable for the class L_2; but, as before,

a solution is given for the class L_1. It is concluded, therefore, that there is a barrier of unsolvability between the classes L_1 and L_2.

[YY09] Tomoo Yokoyama and Tetsuo Yokoyama. Functoriality in reversible circuits (work in progress). *ENTCS. Elsevier*, 2009.