Reversible Programming Languages Capturing Complexity Classes

Lars Kristiansen[1,2(✉)]

[1] Department of Informatics, University of Oslo, Oslo, Norway
[2] Department of Mathematics, University of Oslo, Oslo, Norway
larsk@math.uio.no

Abstract. We argue that there is a link between implicit computational complexity theory and the theory of reversible computation. We show that the complexity classes ETIME and P can be captured by inherently reversible programming languages.

1 Introduction

The title above is inspired by the title of a paper I co-authored with Paul Voda more than 15 years ago: *Programming languages capturing complexity classes* [10]. In that paper we related the computational power of fragments of programming languages to complexity classes defined by imposing time and space constraints on Turing machines. Around that time, I authored and co-authored a number of related papers, e.g. [8,9,11], all of which were clearly inspired by work in *implicit computational complexity theory* from the 1990s, e.g., Bellatoni and Cook [2], Leivant [12,13] and, particularly, Jones [5,6].

Complexity classes like P, FP, NP, LOGSPACE, EXPTIME, and so on, are defined by imposing explicit resource bounds on a particular machine model, namely the Turing machine. E.g., FP is defined as the class of functions computable in polynomial time on a deterministic Turing machine. The definition puts constraints on the resources available to the Turing machines, but no constraints on the algorithms available to them. A Turing machine may compute a function in the class by any imaginable algorithm as long as it works in polynomial time. Implicit computational complexity theory studies classes of functions (problems, languages) that are defined without imposing explicit resource bounds on machine models, but rather by imposing linguistic constraints on the way algorithms can be formulated. When we explicitly restrict our language for formulating algorithms, that is, our programming language, then we may implicitly restrict the computational resources needed to execute algorithms. If we manage to find a restricted programming language that captures a complexity class, then we will have a so-called implicit characterization. A seminal example is Bellatoni and Cook's [2] characterization of FP. They give a functional programming language (which they call a function algebra). This language consists of a few initial functions and two definition schemes (safe composition and safe primitive recursion) which allow us to define new functions. These schemes put

I. Lanese and M. Rawski (Eds.): RC 2020, LNCS 12227, pp. 111–127, 2020.
https://doi.org/10.1007/978-3-030-52482-1_6

rather severe syntactical restrictions on how we can define functions, but they do not refer to polynomially bounded Turing machines or any other kind of resource bounded computing machinery. It is not easy to write programs when we have to stick to these schemes, even experienced programmers might find it hard to multiply two numbers but, be that as it may, this is a programming language that yields an implicit characterization of a complexity class. It turns out that a function can be computed by a program written in Bellantoni & Cook's language if and only if it belongs to the complexity class FP.

There is an obvious link between implicit computational complexity and reversible computing. A programming language based on natural reversible operations will impose restrictions on the way algorithms can be formulated, and thus, also restrictions on the computational resources needed to execute algorithms. Hence, the following question knocks at the door: Will it be possible find reversible programming languages that capture some of the standard complexity classes? The answer turns out to be YES. We will present a reversible language that captures, or if you like, gives an implicit characterization of, the (maybe not very well-known) complexity class ETIME. A few small modifications of this language yield a reversible language that captures the very well-known complexity class P.

Our languages are based on a couple of naturally reversible operations. To increase, or decrease, a natural number by 1 modulo a base b is such an operation: $\ldots 0, 1, 2, \ldots, b-2, b-1, 0, 1, 2 \ldots$. The successor of $b-1$ becomes 0, and then $b-1$ becomes the predecessor of 0. Thus, "increase" and "decrease" are the reverse of each other. To move an element from the top of one stack to the top of another stack is another such operation as we can simply move the element back to the stack it came from.

This paper addresses students and researchers interested in programming languages, reversible computations and computer science in general, they will not necessarily be experts in computability or complexity theory. We will give priority to readability over technical accuracy, but still this is a fairly technical paper, and we will assume that the reader is faintly acquainted with Turing machines and basic complexity theory (standard textbooks are Arora and Barac [1], Jones [7] and Sipser [16]).

Implicit computational complexity theory is definitely a broader and richer research area than our short discussion above may indicate. More on the subject can be found in Dal Lago [3].

2 Reversible Bottomless Stack (RBS) Programs

An infinite sequence of natural numbers $s_1, s_2, s_3, \ldots$ is a *bottomless stack* if there exists k such that $s_i = 0$ for all $i > k$. We use $\langle x_1, \ldots, x_n, 0^*]$ to denote the bottomless stack $s_1, s_2, s_3, \ldots$ where $s_i = x_i$ when $i \leq n$, and $s_i = 0$ when $i > n$. We say that x_1 is the *top element* of $\langle x_1, \ldots, x_n, 0^*]$. Observe that 0 is the top element of the stack $\langle 0^*]$. Furthermore, observe that $\langle 0, 0^*]$ is the same stack as $\langle 0^*]$ (since $\langle 0, 0^*]$ and $\langle 0^*]$ denote the same sequence of natural numbers). We will refer to $\langle 0^*]$ as the *zero stack*.

THE SYNTAX OF RBS

$$
\begin{array}{rcl}
X \in \textbf{Variable} & ::= & \mathtt{X}_1 \mid \mathtt{X}_2 \mid \mathtt{X}_3 \mid \ldots \\
com \in \textbf{Command} & ::= & X^+ \mid X^- \mid (X\,\mathtt{to}\,X) \mid com\,;\,com \\
& \mid & \mathtt{loop}\;X\;\{\,com\,\}
\end{array}
$$

Fig. 1. The syntax of the language RBS. The variable X in the loop command is not allowed to occur in the loop's body.

The syntax of the imperative programming language RBS is given in Fig. 1. Any element in the syntactic category **Command** will be called a *program*, and we will use the word *command* and the word *program* interchangeably throughout the paper. We will now explain the semantics of RBS.

An RBS program manipulates bottomless stacks, and each program variable holds such a stack. The input to a program is a single natural number m. When the execution of the program starts, the input m will be stored at the top of the stack hold by $\mathtt{X}_1$, that is, we have $\mathtt{X}_1 = \langle m, 0^*]$. All other variables occurring in the program hold the zero stack when the execution starts. A program is executed in a *base* b which is determined by the input: we have $b = \max(m+1, 2)$ if $\mathtt{X}_1 = \langle m, 0^*]$ when the execution starts. The execution base b is kept fixed during the entire execution.

Let X and Y be program variables. We will now explain how the primitive commands work. The command $(\mathtt{X}\,\mathtt{to}\,\mathtt{Y})$ pops off the top element of the stack held by X and pushes it onto the stack held by Y, that is

$$
\begin{aligned}
\{\mathtt{X} = \langle x_1, \ldots, x_n, 0^*] \;\wedge\; \mathtt{Y} = \langle y_1, \ldots, y_m, 0^*]\}\ (\mathtt{X}\,\mathtt{to}\,\mathtt{Y}) \\
\{\mathtt{X} = \langle x_2 \ldots, x_n, 0^*] \;\wedge\; \mathtt{Y} = \langle x_1, y_1, \ldots, y_m, 0^*]\}.
\end{aligned}
$$

The command $\mathtt{X}^+$ increases the the top element of the stack held by X by 1 (mod b), that is

$$\{\mathtt{X} = \langle x_1, \ldots, x_n, 0^*]\}\ \mathtt{X}^+\ \{\mathtt{X} = \langle x_1 + 1 \pmod b, x_2 \ldots, x_n, 0^*]\}.$$

The command $\mathtt{X}^-$ decreases the the top element of the stack held by X by 1 (mod b), that is

$$\{\mathtt{X} = \langle x_1, \ldots, x_n, 0^*]\}\ \mathtt{X}^-\ \{\mathtt{X} = \langle x_1 - 1 \pmod b, x_2 \ldots, x_n, 0^*]\}\,.$$

Observe that we have

$$\{\mathtt{X} = \langle b-1, x_2 \ldots, x_n, 0^*]\}\ \mathtt{X}^+\ \{\mathtt{X} = \langle 0, x_2 \ldots, x_n, 0^*]\}$$

and

$$\{\mathtt{X} = \langle 0, x_2 \ldots, x_n, 0^*]\}\ \mathtt{X}^-\ \{\mathtt{X} = \langle b-1, x_2 \ldots, x_n, 0^*]\}$$

when b is the base of the execution.

The semantics of the command $\mathtt{C}_1\,;\,\mathtt{C}_2$ is as expected. This is the standard composition of the commands $\mathtt{C}_1$ and $\mathtt{C}_2$, that is, first $\mathtt{C}_1$ is executed, then $\mathtt{C}_2$ is executed. The command $\mathtt{loop\ X}\ \{\,\mathtt{C}\,\}$ executes the command $\mathtt{C}$ repeatedly k times in a row where k is the top element of the stack held by $\mathtt{X}$. Note that the variable $\mathtt{X}$ is not allowed to occur in $\mathtt{C}$ and, moreover, the command $\mathtt{loop\ X}\ \{\,\mathtt{C}\,\}$ will not modify the stack held by $\mathtt{X}$.

Example 1. Let $\mathtt{C}_1$ be the program $\mathtt{loop\ X}_1\ \{\ \mathtt{X}_2^+\ \}\,;\,(\mathtt{X}_2\,\mathtt{to}\,\mathtt{X}_1)$. We have

$$\{\mathtt{X}_1 = \langle 17, 0^*]\ \wedge\ \mathtt{X}_2 = \langle 0^*]\}\ \mathtt{C}_1\ \{\mathtt{X}_1 = \langle 17, 17, 0^*]\ \wedge\ \mathtt{X}_2 = \langle 0^*]\}.$$

Let $\mathtt{C}_2$ be the program $\mathtt{loop\ X}_1\ \{\ \mathtt{X}_2^+\ \}\,;\,\mathtt{X}_2^+\,;\,(\mathtt{X}_2\,\mathtt{to}\,\mathtt{X}_1)$. We have

$$\{\mathtt{X}_1 = \langle 17, 0^*]\ \wedge\ \mathtt{X}_2 = \langle 0^*]\}\ \mathtt{C}_2\ \{\mathtt{X}_1 = \langle 0, 17, 0^*]\ \wedge\ \mathtt{X}_2 = \langle 0^*]\}$$

since the execution base is 18. All numbers stored on stacks during an execution will be strictly less than the execution base, and thus, less than or equal to $\max(m, 1)$ where m is the input. □

Intuitively, it should be clear that RBS programs are reversible in a very strong sense. RBS is an *inherently reversible* programming language in the terminology of Matos [14]. If we like, we can of course state this insight more formally. The next definition and the following theorem will be a step in that direction.

Definition 2. We define *reverse command* of $\mathtt{C}$, written $\mathtt{C}^R$, inductively over the structure $\mathtt{C}$:

- $(\mathtt{X}_i^+)^R = \mathtt{X}_i^-$
- $(\mathtt{X}_i^-)^R = \mathtt{X}_i^+$
- $(\mathtt{X}_i\,\mathtt{to}\,\mathtt{X}_j)^R = (\mathtt{X}_j\,\mathtt{to}\,\mathtt{X}_i)$
- $(\mathtt{C}_1\,;\,\mathtt{C}_2)^R = \mathtt{C}_2^R\,;\,\mathtt{C}_1^R$
- $(\mathtt{loop\ X}_i\ \{\,\mathtt{C}\,\})^R = \mathtt{loop\ X}_i\ \{\,\mathtt{C}^R\,\}$.

□

Theorem 3. *Let $\mathtt{C}$ be a program, and let $X_1, \ldots, X_n$ be the variables occurring in $\mathtt{C}$. Furthermore, let m be any natural number. We have*

$$\{X_1 = \langle m, 0^*]\ \wedge\ \bigwedge_{i=2}^{n} X_i = \langle 0^*]\}\ \mathtt{C}\,;\,\mathtt{C}^R\ \{X_1 = \langle m, 0^*]\ \wedge\ \bigwedge_{i=2}^{n} X_i = \langle 0^*]\}.$$

It is a nice, and maybe even challenging, exercise to write up a decent proof Theorem 3, even if it should be pretty clear that the theorems holds. We will offer a proof in the next section. The reader not interested in the details of the proof, may skip that section.

We will now define the set of problems that can be decided by an RBS programs. To that end, we need to determine how an RBS program should accept, and how an RBS program should reject, its input. Any reasonable convention will do, and we will just pick a simple and convenient one.

EXAMPLE

Program:	*Comments:*
	(* $\mathtt{X}_1 = \langle m, 0^* \rangle$ *)
$\mathtt{X}_1$ to $\mathtt{X}_9$;	(* the top elements of $\mathtt{X}_9$ is m *)
$\mathtt{X}_2^+$;	(* $\mathtt{X}_1 = \langle 0^* \rangle$ and $\mathtt{X}_2 = \langle 1, 0^* \rangle$ *)
loop $\mathtt{X}_9$ {	(* repeat m times *)
$\mathtt{X}_1$ to $\mathtt{X}_3$;	
$\mathtt{X}_2$ to $\mathtt{X}_1$;	(* swap the top elements of $\mathtt{X}_1$ and $\mathtt{X}_2$ *)
$\mathtt{X}_3$ to $\mathtt{X}_2$ }	

Fig. 2. The program accepts every even number and rejects every odd number.

Definition 4. An RBS program C accepts the natural number m if C executed with input m terminates with 0 at the top of the stack hold by $\mathtt{X}_1$, otherwise, C rejects m.

A problem is a set of natural numbers.[1] An RBS program C *decides the problem* A if C accepts all m that belong to A and rejects all m that do not belong to A. Let $\mathcal{S}$ denote class of problems decidable by an RBS program. □

Let A be the set of even numbers. Then A is a problem. Figure 2 shows an RBS program that decides A.

Now, any RBS program decides a problem, and $\mathcal{S}$ is obviously a well-defined class of computable (decidable) problems. We have defined $\mathcal{S}$ by a reversible programming language. We have not defined $\mathcal{S}$ by imposing resource bounds on Turing machines or any other machine models. What can we say about the computational complexity of the problems we find in $\mathcal{S}$? May it be the case that $\mathcal{S}$ equals a complexity class?

3 The Proof of Theorem 3

This section is dedicated to a detailed proof of Theorem 3 (readers not interested may jump ahead to Sect. 4). First, we need some terminology and notation: We will say that a (bottomless) stack is a *b-stack* if every number stored on the stack is strictly smaller than b. Furthermore, we will use $\mathcal{V}(\mathtt{C})$ to denote the set of program variables occurring in the command C, and for any positive integer m and any command C, we define the command $\mathtt{C}^m$ by $\mathtt{C}^1 \equiv \mathtt{C}$ and $\mathtt{C}^{m+1} \equiv \mathtt{C}^m\,;\,\mathtt{C}$.

Now, assume that C is an RBS command with $\mathcal{V}(\mathtt{C}) \subseteq \{\mathtt{X}_1, \ldots, \mathtt{X}_n\}$. Furthermore, assume that C is executed in base b and that $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$ are b-stacks. With these assumptions in mind, we make the following claim:

[1] It is pretty standard in computability and complexity theory to define a problem as a set of natural numbers.

$$\text{If} \quad \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ \mathtt{C}\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\,, \ \text{then} \quad \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\ \mathtt{C}^R\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}. \tag{claim}$$

Theorem 3 follows straightforwardly from this claim. So all we need to do is to prove the claim.

We will of course carry out induction on the structure of the command $\mathtt{C}$, and our proof will split into the tree base cases (i) $\mathtt{C} \equiv \mathtt{X}_i^+$, (ii) $\mathtt{C} \equiv \mathtt{X}_i^-$ and (iii) $\mathtt{C} \equiv (\mathtt{X}_j\,\mathtt{to}\,\mathtt{X}_i)$ and the two inductive cases (iv) $\mathtt{C} \equiv \mathtt{C}_1\,;\,\mathtt{C}_2$ and $\mathtt{C} \equiv \mathtt{loop}\,\mathtt{X}_i\,\{\mathtt{C}_0\}$ (see Fig. 1).

Case (i). Assume

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ \mathtt{X}_i^+\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}.$$

Then we also have $\{\mathtt{X}_i = \alpha_i\}\ \mathtt{X}_i^+\ \{\mathtt{X}_i = \beta_i\}$ where

$$\alpha_i = \langle m_1, m_2, \ldots, m_k, 0^*] \quad \text{and} \quad \beta_i = \langle m_1 + 1 \pmod b, m_2, \ldots, m_k, 0^*]$$

for some $m_1, \ldots, m_k < b$. We have $(m_1 + 1 \pmod b) - 1 \pmod b = m_1$ when $m_1 < b$. Thus we have $\{\mathtt{X}_i = \beta_i\}\ \mathtt{X}_i^-\ \{\mathtt{X}_i = \alpha_i\}$. By Definition 2, we have $\{\mathtt{X}_i = \beta_i\}\ (\mathtt{X}_i^+)^R\ \{\mathtt{X}_i = \alpha_i\}$. Now, since neither $\mathtt{X}_i^+$ nor $(\mathtt{X}_i^+)^R$ will modify any stack held by a variable $\mathtt{X}_j$ where $j \neq i$, we also have

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\ (\mathtt{X}_i^+)^R\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}.$$

This concludes the proof of case (i). The proofs of the cases (ii) and (iii) are very similar to the proof of case (i). We leave the details to the reader and proceed with the inductive cases.

Case (iv). Assume

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ \mathtt{C}_1\,;\,\mathtt{C}_2\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}.$$

Then there exist b-stacks $\gamma_1, \ldots, \gamma_n$ such that

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ \mathtt{C}_1\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \gamma_\ell\} \quad \text{and} \quad \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \gamma_\ell\}\ \mathtt{C}_2\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}.$$

We apply our induction hypothesis both to $\mathtt{C}_1$ and to $\mathtt{C}_2$ and conclude

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \gamma_\ell\}\ \mathtt{C}_1^R\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\} \quad \text{and} \quad \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\ \mathtt{C}_2^R\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \gamma_\ell\}.$$

It follows that

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\ \mathtt{C}_2^R\,;\,\mathtt{C}_1^R\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}.$$

Finally, as Definition 2 states that $(\mathtt{C}_1\,;\,\mathtt{C}_2)^R = \mathtt{C}_2^R\,;\,\mathtt{C}_1^R$, we have

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\ (\mathtt{C}_1\,;\,\mathtt{C}_2)^R\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}.$$

This completes the proof of case (iv).

Case (v). Assume

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ \mathtt{loop}\ \mathtt{X}_i\ \{\,\mathtt{C}_0\,\}\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\} \tag{*}$$

and let m be the top element of the stack α_i.

If $m = 0$, we have

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ \mathtt{loop}\ \mathtt{X}_i\ \{\,\mathtt{C}_0\,\}\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}.$$

as the command $\mathtt{C}_0$ will not be executed at all. Thus, we also have

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ \mathtt{loop}\ \mathtt{X}_i\ \{\,\mathtt{C}_0^R\,\}\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}.$$

and by Definition 2, we have

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ (\mathtt{loop}\ \mathtt{X}_i\ \{\,\mathtt{C}_0\,\})^R\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}.$$

This proves that the claim holds when $m = 0$. We are left to prove that the claim holds when $m > 0$. Thus, in the remainder of this proof we assume that $m > 0$.

First we prove

$$\text{If}\quad \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ \mathtt{C}_0^m\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\},\ \text{then}\quad \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\ (\mathtt{C}_0^R)^m\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}. \tag{†}$$

by a secondary induction on m.

Let $m = 1$. Then we have $C_0^m \equiv \mathtt{C}_0$, and an application of our main induction hypothesis to $\mathtt{C}_0$ yields (†). Let $m > 1$. Then we have

$$C_0^m \equiv C_0^{m-1}\,;\,\mathtt{C}_0 \quad \text{and} \quad (\mathtt{C}_0^R)^m \equiv \mathtt{C}_0^R\,;\,(\mathtt{C}_0^R)^{m-1}$$

and (†) holds by our induction hypothesis on m and case (iv) above. This concludes the proof of (†).

We are now ready to complete our proof the claim. By (*), we have

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\ \mathtt{C}_0^m\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}.$$

By (†), we have

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\ (\mathtt{C}_0^R)^m\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}.$$

Since $\mathtt{X}_i \notin \mathcal{V}(\mathtt{C}_0)$, we have $\beta_i = \alpha_i$, and thus, the top element of β_i is the same as the top element of α_i, namely m. It follows that

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\ \mathtt{loop}\ \mathtt{X}_i\ \{\ \mathtt{C}_0^R\ \}\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}.$$

Finally, as Definition 2 states that $\mathtt{loop}\ \mathtt{X}_i\ \{\ \mathtt{C}_0^R\ \} = (\mathtt{loop}\ \mathtt{X}_i\ \{\ \mathtt{C}_0\ \})^R$, we have

$$\{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\ (\mathtt{loop}\ \mathtt{X}_i\ \{\ \mathtt{C}_0\ \}\ )^R\ \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}.$$

This completes the proof of case (v).

4 Simulation of Turing Machines

4.1 A General Strategy

Let us first see how we can simulate a Turing machine in a standard way in a standard high-level language. Thereafter we will discuss how we can simulate a Turing machine in our rudimentary reversible language. In the standard language we will of course be able to simulate any Turing machine, no matter how much time and space resources the machine requires. In the reversible language we will only be able to simulate those Turing machines that run in time $O(2^{kn})$ (where k is a constant and n is the length of the input).

We assume some familiarity with Turing machines. The reader is expected to know that a Turing machine computes by writing symbols from a finite alphabet $a_1, \ldots, a_A$ on an infinite tape which is divided into cells; know that one of the cells is scanned by the machine's head; know a there is a finite number of states $q_1, \ldots, q_Q$; and so on.

The input w will be available on the tape when a Turing machine M starts, and the actions taken by M will be governed by a finite transition table. Each entry of the table is a 5-tuple

$$a_i, q_k, a_j, D, q_\ell \qquad (*)$$

where a_i, a_j are alphabet symbols; q_k, q_ℓ are states; and D is ether "left" or "right". Such a tuple is called a transition and tells M what to do when it scans the symbol a_j in state q_k: in that case M should write the symbol a_j, move its head one position in the direction given by D, and then proceed in state q_ℓ. We restrict our attention to deterministic Turing machines, and for each alphabet symbol a_i and each non-halting state q_k, there will be one, and only one, transition that starts with a_i, q_k. So a Turing machine knows exactly what to do until it reaches one its halting states, and then it simply halts (if it halts in a dedicated state q_{accept}, it accepts its input; if it halts in a dedicated state q_{reject}, it rejects its input). This entails that we can simulate a Turing machine by a sequence of if-then statements embedded into a while-loop. We need one if-then statement for each transition:

⟨initiate the tape with the input w⟩
`while` ⟨M is not in a halting state⟩ `do`
`if` ⟨a_1 is scanned in state q_1⟩ `then` ⟨do what should be done⟩`;`
`if` ⟨a_2 is scanned in state q_1⟩ `then` ⟨do what should be done ⟩`;`
⋮ ⋮
`if` ⟨a_A is scanned in state q_Q⟩ `then` ⟨do what should be done ⟩
`end-while`.

Minimum one transition will be executed each time the loop's body is executed, and the running time of M (on input w) will more or less be the number of times the body is executed. (It might happen that more than one transition is executed when the loop's body is executed once, but that will not cause any trouble.) In order to simulate the actions taken by the transitions, we need a representation of the computing machinery. We need to keep track of the current state, we need to keep track of the symbols on the tape, and we need to identify the scanned cell. The current state can simply be stored in a register `STATE`, but how should we deal with the tape? The tape is divided into an infinite sequence of cells

$$C_1, C_2, C_2, \ldots, C_{s-1}, C_s, C_{s+1}, \ldots$$

where one of the cells C_s is scanned by the head. Only finitely many of these cells will contain anything else than the blank symbol. Let us say that C_i contains blank when $i > B_0$. In order to simulate the machine it will obviously be sufficient to store the symbols in the cells $C_1, C_2, \ldots, C_B$ where $B = \max(B_0, s) + 1$. In addition we need to keep track of the scanned cell C_s. A convenient way to deal with the situation will be to use a stack $\texttt{STACK}_L$, a register `SCAN`, another stack $\texttt{STACK}_R$, and store the tape content in the following way:

C_{s-1}		C_{s+1}
⋮		⋮
C_1	C_s	C_B
$\texttt{STACK}_L$	`SCAN`	$\texttt{STACK}_R$

Now we can mimic the movements of the head by pushing and popping alphabet symbols in the obvious way, and the transition (*) can be implemented by a program of the form

$$\texttt{if SCAN} = a_i \texttt{ and STATE} = q_k \texttt{ then}$$
$$\{\, \texttt{SCAN:=}\, a_j \texttt{;} \ \ldots \text{ push and pop } \ldots \ \texttt{;STATE:=}\, q_\ell \,\}.$$

4.2 Can RBS Programs Simulate Turing Machines?

The input to an RBS program is a natural number, and we will thus discuss to what extent an RBS program can simulate a Turing machine that takes a single natural number as input.

We have seen that a program with only one while-loop can simulate a Turing machine (and we will for sure need at least one while-loop in order to simulate an arbitrary Turing machine). Now, while-loops are not available in RBS, and the best we can do in order to simulate a Turing machine is to use a fixed number of nested for-loops:

$$\texttt{loop}\, \texttt{Y}_1 \,\{\, \texttt{loop}\, \texttt{Y}_2 \,\{\, \ldots \texttt{loop}\, \texttt{Y}_k \,\{\, \langle \text{sequence of if-then statements} \rangle \,\} \ldots \,\}\}.$$

Since an RBS program cannot increase the numerical value of its input, the body of each of these loops will be executed maximum $\max(m, 1)$ times where m is the input to the RBS program (and to the Turing machine the program simulates). Thus it is pretty clear that we cannot simulate a Turing machine if its running time is not bounded by m^k for some constant k. This corresponds to a bound $2^{k|m|}$ where k is a constant and $|m|$ is the length of the input m, that is, $|m|$ equals the number of symbols needed to represent the natural number m in binary notation. In the following we will see that any Turing machine that uses such an amount of computation time can be simulated by an RBS program.

It turns out that an RBS program can simulate the transitions of a Turing machine M in essentially the same way as the high-level program sketched above, given that the input to M is sufficiently large (on small inputs the simulation might fail). Stacks are directly available in RBS, and thus an RBS program can easily represent the tape and mimic the movements of the head. On the other hand, assignment statements and if-then statements are not directly available. This makes things a bit tricky. Let us first see how RBS programs to a certain extent can simulate programs written in a non-reversible programming language called $\texttt{LOOP}^-$.

4.3 $\texttt{LOOP}^-$ Programs

The syntax of $\texttt{LOOP}^-$ is given in Fig. 3. Any element in the syntactic category **Command** will be called a program. A $\texttt{LOOP}^-$ program manipulates natural numbers, and each program variable holds a single natural number. The command $\texttt{X} := k$ assigns the fixed number k to the variable X. The command $\texttt{X} := \texttt{Y}$ assigns the number hold by the variable Y to the variable X. The command

THE SYNTAX OF $\texttt{LOOP}^-$

$$
\begin{array}{rcl}
X \in \textbf{Variable} & ::= & \texttt{X}_1 \mid \texttt{X}_2 \mid \texttt{X}_3 \mid \ldots \\
k \in \textbf{Constant} & ::= & 0 \mid 1 \mid 2 \mid 3 \mid \ldots \\
com \in \textbf{Command} & ::= & X\texttt{:=}k \mid X\texttt{:=}X \mid \texttt{pred(}X\texttt{)} \mid com\texttt{;}\ com \\
& \mid & \texttt{loop}\ X\ \{\, com \,\}
\end{array}
$$

Fig. 3. The syntax of the language $\texttt{LOOP}^-$. The variable X in the loop command is not allowed to occur in the loop's body.

`pred(X)` decreases the value hold by the variable `X` by 1 if the value is strictly greater than 0; and leave the value hold by `X` unchanged if the value is 0. Furthermore, the command $\texttt{C}_1\texttt{;}\ \texttt{C}_2$ is the standard composition of the commands $\texttt{C}_1$ and $\texttt{C}_2$, and the command $\texttt{loop X}\ \{\,\texttt{C}\,\}$ executes the command `C` repeatedly k times in a row where k in the number hold by `X`. Note that the variable `X` is not allowed to occur in `C` and that the command $\texttt{loop X}\ \{\,\texttt{C}\,\}$ does not modify the value held by `X`.

An `RBS` program can represent a $\texttt{LOOP}^-$ variable `X` holding natural number k by a variable `X` (we use the same variable name) holding the stack $\langle k, 0^*]$. The command $\texttt{X := }k$ can then be simulated by the program

$$(\texttt{X to Z});\ \underbrace{\texttt{X}^+\texttt{;}\ \texttt{X}^+\texttt{;}\ \ldots \texttt{X}^+}_{\text{increase } k \text{ times}}$$

where `Z` is an auxiliary variable (`Z` works as a trash bin). Now, observe that this will only work if the base of execution is strictly greater than k, but that will good enough to us. The command $\texttt{X} := \texttt{Y}$ can be simulated by the program

$$(\texttt{X to Z})\texttt{;}\ \texttt{loop Y}\ \{\,\texttt{X}^+\,\}$$

where `Z` is an auxiliary variable (`Z` works as a trash bin). Furthermore, the command `pred(X)` can be simulated by a program that uses auxiliary variables `Y` and `Z` (which represent natural numbers) and the simulations of the assignment statements given above:

$$\texttt{Z :=}0\texttt{;}\ \texttt{Y := X;}\ \texttt{loop Y}\ \{\,\texttt{X := Z;}\ \texttt{Z}^+\,\}.$$

This shows how `RBS` programs can simulate all the primitive $\texttt{LOOP}^-$ commands. It is easy to see that

- the `RBS` command $\texttt{C}_1'\texttt{;}\ \texttt{C}_2'$ simulates the $\texttt{LOOP}^-$ command $\texttt{C}_1\texttt{;}\ \texttt{C}_2$ if $\texttt{C}_1'$ simulates $\texttt{C}_1$ and $\texttt{C}_2'$ simulates $\texttt{C}_2$
- the `RBS` command $\texttt{loop X}\ \{\,\texttt{C}'\,\}$ simulates the $\texttt{LOOP}^-$ command $\texttt{loop X}\ \{\,\texttt{C}\,\}$ if $\texttt{C}'$ simulates `C`.

Hence, any $\texttt{LOOP}^-$ program can be simulated by an `RBS` program given that the input is sufficiently large. On small inputs simulations might fail since the

simulation of the assignment $\mathtt{X} := k$ only works if the execution base is strictly greater than k.

The $\mathtt{LOOP}^-$ language turns out to be more expressive than one might expect at a first glance, and all sorts of conditional statements and if-then constructions are available in the language. As an example, let us see how we can implement the construction

$$\mathtt{if\ X = Y\ then\ C_1\ else\ C_2}.$$

We will need some axillary variables $\mathtt{X}'$, $\mathtt{Y}'$, $\mathtt{Z}$, $\mathtt{U}$ which do not occur in any of the commands $\mathtt{C}_1$ and $\mathtt{C}_2$. First we execute the program

$$\mathtt{X}' := \mathtt{X};\ \mathtt{Y}' := \mathtt{Y};\ \mathtt{loop\ X}\ \{\ \mathtt{pred(Y')}\ \};\ \mathtt{loop\ Y}\ \{\ \mathtt{pred(X')}\ \}.$$

This program sets both $\mathtt{X}'$ and $\mathtt{Y}'$ to 0 if $\mathtt{X}$ and $\mathtt{Y}$ hold the same number. If $\mathtt{X}$ and $\mathtt{Y}$ hold different numbers, one of the two variables $\mathtt{X}', \mathtt{Y}'$ will be set to a number strictly greater than 0. Then we execute the program

$$\begin{array}{l}\mathtt{Z := 1;\ U := 1;}\\ \mathtt{loop\ X'}\ \{\ \mathtt{Z := 0}\ \};\ \mathtt{loop\ Y'}\ \{\ \mathtt{Z := 0}\ \};\\ \mathtt{loop\ Z}\ \{\ \mathtt{C_1;\ U := 0}\ \};\ \mathtt{loop\ U}\ \{\ \mathtt{C_2}\ \}.\end{array}$$

The composition of these two programs executes the program $\mathtt{C}_1$ exactly once (and $\mathtt{C}_2$ will not be executed at all) if $\mathtt{X}$ and $\mathtt{Y}$ hold the same number. If $\mathtt{X}$ and $\mathtt{Y}$ hold different numbers, $\mathtt{C}_2$ will be executed exactly once (and $\mathtt{C}_1$ will not be executed at all). The reader should note that this implementation of if-then-else construction does not contain any assignments of the form $\mathtt{X} := k$ where $k > 1$.

It is proved in Kristiansen [8] that $\mathtt{LOOP}^-$ captures the complexity class LINSPACE, that is, the set of problems decidable in space $O(n)$ on a deterministic Turing machine (n is the length of the input). Hence, the considerations above indicate that LINSPACE $\subseteq \mathcal{S}$. However, we are on our way to proving a stronger result, namely that LINSPACE $\subseteq \mathcal{S} =$ ETIME. The equality LINSPACE $\stackrel{?}{=}$ ETIME is one of the many notorious open problems of complexity theory. The general opinion is that the equality does not hold.

4.4 RBS Programs that Simulates Time-Bounded Turing Machines

We have seen that RBS programs (nearly) can simulate $\mathtt{LOOP}^-$ programs. $\mathtt{LOOP}^-$ can assign constants to registers and perform if-then-else constructions. This helps us to see how to an RBS program can simulate an arbitrary $2^{k|m|}$ time Turing machine M. Such a program may be of the form

$$\begin{array}{l}\langle\text{initiate the tape with the input } m\rangle\,;\\ \mathtt{Y}_1 := \langle\text{the input } m\rangle\,;\ \mathtt{Y}_2 := \langle\text{the input m}\rangle\,;\ \ldots\,;\ \mathtt{Y}_k := \langle\text{the input m}\rangle\,;\\ \mathtt{loop\ Y}_1\ \{\ \mathtt{loop\ Y}_2\ \{\ \ldots \mathtt{loop\ Y}_k\ \{\ \mathtt{T}_1\,;\ \mathtt{T}_2\,;\ \ldots\,;\ \mathtt{T}_r\ \}\ldots\}\}.\end{array}$$

We represent the symbols in M's alphabet $a_1, \ldots a_A$ by the numbers $1, \ldots, A$ and M's states $q_1, \ldots q_Q$ by the numbers $1, \ldots, Q$. We use two stacks to hold the

content of the tape, and we use registers `STATE` and `SCAN` to hold respectively the current state and the scanned cell. Each T_s will take care of a transition a_i, q_k, a_j, D, q_ℓ and be of the form

`if SCAN =` i `and STATE =` k `then {SCAN :=` j`; ...` push and pop `...; STATE :=` ℓ`}`.

We are left with a minor problem: This will not work for small inputs. This will only work if the base of execution $b = \max(m+1, 2)$ is strictly greater than $\max(A, Q)$. Only then will the simulating program be able to perform the necessary assignments of constants to variables. In some sense we cannot deal with this problem. An `RBS` program will not be able to simulate (in any reasonable sense of the word) an arbitrary $2^{k|m|}$ time Turing machine M on small inputs, but still there will be an `RBS` program that decides the same problem as M.

We have seen that it suffices to assign the constants 0 and 1 to variables in order to implement the if-then-else construction in `LOOP`$^-$. This entails that the if-then-else construction will work on small inputs as the base of execution always will be strictly greater than 1. Hence, if the problem A is decided by a $2^{k|m|}$ time Turing machine M, there will also be an `RBS` program that decides A. This program will be of the form

```
X := ⟨the input m⟩;
if X = 0 then ⟨give correct output for m = 0⟩
else { pred(X);
if X = 0 then ⟨give correct output for m = 1⟩
else { pred(X);
if X = 0 then ⟨give correct output for m = 2⟩
⋮
else {⟨the input is big enough, ...
      ... simulate M, accept if M accepts, reject if M rejects ⟩} ... } }.
```

5 Main Results

5.1 A Characterization of ETIME

Definition 5. Let $|m|$ denote the number of digits required to write the natural number m in binary notation. For any natural number k, let $ETIME_k$ be the class of problems decidable in time $O(2^{k|m|})$ on a deterministic Turing machine. Let $ETIME = \bigcup_{i \in \mathbb{N}} ETIME_i$. □

Theorem 6. $\mathcal{S} = ETIME$.

Proof. The proof of the inclusion $\mathcal{S} \subseteq$ ETIME should be straightforward to anyone experienced with Turing machines. Assume $A \in \mathcal{S}$ (we will argue that $A \in$ ETIME). Then there is an `RBS` program `C` that decides A. Let m be the input to `C`. Each loop in `C` will be executed maximum $m+1$ times since the base of

execution will be $\max(m+1, 2)$. Thus, there exist constants k_0, k_1 (not depending on m) such that $k_0(m+1)^{k_1}$ bounds the number of primitive commands executed by `C` on input m. A Turing machine can simulate the execution of `C` on input m with polynomial overhead. Thus there exist constants k_2, k_3 such that $k_2(m+1)^{k_3}$ bounds the number of steps a Turing machine needs to decide if m is in A. There exists k such that $k_2(m+1)^{k_3} < 2^{k|m|}$. Hence, $A \in$ ETIME. This proves the inclusion $\mathcal{S} \subseteq$ ETIME.

We turn to proof of the inclusion ETIME $\subseteq \mathcal{S}$. Assume $A \in$ ETIME (we will argue that $A \in \mathcal{S}$). Then there is a $O(2^{k|m|})$ time Turing machine M that decides A. Now, M will run in time $2^{k_0|m|}$ when k_0 is sufficiently large. In the previous section we saw that there will be an `RBS` program that decides the same problem as M. Hence, $A \in \mathcal{S}$. This proves the inclusion ETIME $\subseteq \mathcal{S}$. □

5.2 A Characterization of P

Would it not be nice if we could find a reversible language that captures a complexity class that is a bit more attractive than ETIME? Now, P is for a number of reasons, which the reader might be aware of, one of most popular and important complexity classes. Luckily, it turns out that a few modifications of `RBS` yield a characterization of P.

First we modify the way `RBS` programs receive input. The input will now be a string over some alphabet. Any alphabet that contains at least two symbols will do and, for convenience, we will stick to the alphabet $\{a, b\}$. The base of execution will at program start be set to the length of the input. Otherwise, nearly everything is kept as before: Every variable will still hold a bottomless stack storing natural numbers. All commands available in the original version of `RBS` will be available in the new version. A program will still accept its input by terminating with 0 at the top of the stack held by $\mathtt{X}_1$, otherwise, the program rejects its input. Moreover, all variables including $\mathtt{X}_1$, the variable that used to import the input, hold the zero stack when the execution of a program starts.

Next we extend `RBS` by two commands with the syntax

$$\texttt{case inp[}X\texttt{]=a: \{ }com\texttt{ \}} \quad \text{and} \quad \texttt{case inp[}X\texttt{]=b: \{ }com\texttt{ \}}$$

where X is a variable and *com* is a command which does not contain X. These commands make it possible for a program to access its input. The input is a string $\alpha_0\alpha_1, \ldots, \alpha_{b-1}$ where b is the execution base and $\alpha_i \in \{a, b\}$. Assume that $\mathtt{X}_j$ holds a stack where top element is k. The command

$$\texttt{case inp[X}_j\texttt{]=a: \{ C \}}$$

executes the command `C` if $\alpha_k = a$, otherwise, the command does nothing. The command

$$\texttt{case inp[X}_j\texttt{]=b: \{ C \}}$$

executes the command `C` if $\alpha_k = b$, otherwise, the command does nothing.

EXAMPLE

Program:	*Comments:*
	(* all stacks hold the zero stack *)
$\mathtt{X}_2^-$	(* the top element of $\mathtt{X}_2$ is $b-1$ *)
loop $\mathtt{X}_2$ {	(* repeat $b-1$ times *)
case inp[$\mathtt{X}_3$]=b:	(* $\mathtt{X}_3$ is a pointer into the input *)
{ $\mathtt{X}_1$ to $\mathtt{X}_9$;	(* $\mathtt{X}_1$ holds the zero stack *)
$\mathtt{X}_1^+$	(* top element of $\mathtt{X}_1$ is 1 *)
};	
$\mathtt{X}_3^+$	(* move pointer to the right *)
};	(* end of loop *)
case inp[$\mathtt{X}_3$]=a:	(* top element of $\mathtt{X}_3$ is $b-1$ *)
{ $\mathtt{X}_1$ to $\mathtt{X}_9$;$\mathtt{X}_1^+$ }	

Fig. 4. The program accepts any string that starts with a nonempty sequence of a's and ends with a single b (the input to a program should at least contain two symbols). The program rejects any string that is not of this form. The program accepts by terminating with $\mathtt{X}_1 = \langle 0^*]$ and rejects by terminating with $\mathtt{X}_1 = \langle 1, 0^*]$.

We still have a reversible language. The two new commands are reversible. The variable $\mathtt{X}_j$ is not allowed to occur in C and will consequently not be modified by C. Thus, for $\mathtt{x} \in \{\mathtt{a}, \mathtt{b}\}$, we may extend Definition 2 by

$$\big(\texttt{case}\quad \texttt{inp[X}_j\texttt{]=x: \{ C \}}\big)^R \;=\; \texttt{case}\quad \texttt{inp[X}_j\texttt{]=x: \{ C}^R\texttt{ \}}.$$

and Theorem 3 will still hold.

To avoid confusion we will use RBS′ to denote our new version of RBS. We require that the input to an RBS′ program is of length at least 2 (so we exclude the empty string and the one-symbol strings a and b). This is of course a bit artificial, but it seems to be the most convenient way to deal with a few annoying problems of technical nature. Accordingly, we also require that every string in a language (see the definition below) is of length at least 2.

Definition 7. A language L is a set of strings over the alphabet $\{a, b\}$, moreover, every string in L is of length at least 2.

An *RBS′* program C decides the language L if C accepts every string that belongs to L and rejects every string that does not belong to L. Let $\mathcal{S}'$ be class of languages decidable by an RBS′ program.

Let $|w|$ denote the length of the string w. For any natural number k, let P_k be the class of languages decidable in time $O(|w|^k)$ on a deterministic Turing machine. Let $P = \bigcup_{i \in \mathbb{N}} P_i$. □

Figure 4 shows an RBS′ program which decides the language given by the regular expression a^*ab.

The proof of the next theorem is very similar to the proof of Theorem 6, and the reader should be able to provide the details. Just recall that the execution base of an RBS′ program is set to the length of the input. Hence, the number of primitive instructions executed by an RBS′ program will be bounded by $|w|^k$ where $|w|$ is the length of the input w and k is a sufficiently large constant, and moreover, an RBS′ program of the form

$$\texttt{loop}\,\mathtt{Y}_1\,\{\,\texttt{loop}\,\mathtt{Y}_2\,\{\,\ldots\texttt{loop}\,\mathtt{Y}_k\,\{\,\langle\ldots\text{ a list of transitions }\ldots\rangle\,\}\ldots\}\}.$$

will execute $\langle\ldots$ a list of transitions $\ldots\rangle$ exactly $|w|^k$ times if each and one of the variables $\mathtt{Y}_1,\ldots\mathtt{Y}_k$ holds a stack where the top element is $|w|$.

Theorem 8. $\mathcal{S}' = P$.

6 Some Final Remarks

We have argued that there is a link between implicit computational complexity theory and the theory of reversible computation, and we have showed that both ETIME and P can be captured by inherently reversible programming languages. In general, implicit characterizations are meant to shed light on the nature of complexity classes and the many notoriously hard open problems involving such classes. Implicit characterizations by reversible formalisms might yield some new insights in this respect. It is beyond the scope of this paper to discuss or interpret the theorems proved above any further, but one might start to wonder how different aspects of reversibility relate to time complexity, space complexity and nondeterminism.

The author is not aware of any work in reversible computing that is closely related to the work presented above, but some work of Matos [14] is at least faintly related. Matos characterizes the primitive recursive functions by an inherently reversible loop-language.[2] Paolini et al. [15] do also characterize the primitive recursive functions by a reversible formalism. Their work is of a recursion-theoretic nature and has a different flavor than ours, but it is possible that such studies might lead to interesting characterizations of complexity classes.

We finish off this paper by suggesting a small research project. It should be possible to extend RBS to an inherently reversible higher-order language. First-order programs will be like the ones defined and explained above. Second-order programs will manipulate stacks of stacks, third-order programs will manipulate stacks of stacks of stacks, and so on. This will induce a hierarchy: the class of problems decidable by a first-order RBS program, the class of problems decidable by a second-order RBS program, ... by a third-order RBS program, and so on. By the same token, RBS′ will induce a hierarchy: the class of languages decidable by a first-order RBS′ program, the class of languages decidable by a second-order RBS′ program, and so on. These two hierarchies should be compared to the alternating time-space hierarchies studied in Goerdt [4], Jones [6], Kristiansen and Voda [10] and many other papers.

[2] The result is not stated very clearly in the paper. See the footnote at page 2066.

References

1. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press, Cambridge (2009)
2. Bellantoni, S.J., Cook, S.: A new recursion-theoretic characterizations of the polytime functions. Comput. Complex. **2**, 97–110 (1992)
3. Dal Lago, U.: A short introduction to implicit computational complexity. In: Bezhanishvili, N., Goranko, V. (eds.) ESSLLI 2010-2011. LNCS, vol. 7388, pp. 89–109. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31485-8_3
4. Goerdt, A.: Characterizing complexity classes by higher type primitive recursive definitions. Theoret. Comput. Sci. **100**, 45–66 (1992)
5. Jones, N.D.: LOGSPACE and PTIME characterized by programming languages. Theoret. Comput. Sci. **228**, 151–174 (1999)
6. Jones, N.D.: The expressive power of higher-order types or, life without CONS. J. Funct. Program. **11**, 55–94 (2001)
7. Jones, N.D.: Computability and Complexity from a Programming Perspective. The MIT Press, Cambridge (1997)
8. Kristiansen, L.: Neat function algebraic characterizations of LOGSPACE and LINSPACE. Comput. Complex. **14**, 72–88 (2005)
9. Kristiansen, L., Niggl, K.-H.: On the computational complexity of imperative programming languages. Theoret. Comput. Sci. **318**, 139–161 (2004)
10. Kristiansen, L., Voda, P.J.: Programming languages capturing complexity classes. Nord. J. Comput. **12**, 89–115 (2005)
11. Kristiansen, L., Voda, P.J.: Complexity classes and fragments of C. Inf. Process. Lett. **88**, 213–218 (2003)
12. Leivant, D.: A foundational delineation of computational feasibility. In: Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science, pp. 39–47. IEEE (1991)
13. Leivant, D.: Stratified functional programs and computational complexity. In: POPL 1993: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 325–333. ACM, New York (1993)
14. Matos, A.B.: Linear programs in a simple reversible language. Theoret. Comput. Sci. **290**, 2063–2074 (2003)
15. Paolini, L., Piccolo, M., Roversi, L.: On a class of reversible primitive recursive functions and its Turing-complete extensions. New Gener. Comput. **36**, 233–256 (2018)
16. Sipser, M.: Introduction to the Theory of Computation. PWS Publishing Company, Boston (1997)