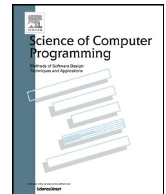
Science of Computer Programming 213 (2022) 102723

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico



Reversible computing and implicit computational complexity

Lars Kristiansen [a,b]

[a] *Department of Informatics, University of Oslo, Norway*
[b] *Department of Mathematics, University of Oslo, Norway*

A R T I C L E I N F O

Article history:
Received 19 November 2020
Received in revised form 21 July 2021
Accepted 10 September 2021
Available online 15 September 2021

Keywords:
Reversible computing
Invertible programming languages
Implicit computational complexity
Complexity classes
Turing machines

A B S T R A C T

We argue that there is a link between implicit computational complexity theory and reversible computation. We introduce inherently reversible programming languages which capture the complexity classes ETIME and P. Furthermore, we discuss and analyze higher-order versions of our reversible programming languages.

© 2021 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Around 15 years ago, I authored and co-authored a number of papers which related the computational power of fragments of programming languages to complexity classes defined by imposing time and space constraints on Turing machines, e.g. [12,14,13]. These papers were clearly inspired by work in *implicit computational complexity theory* from the 1990s, e.g., Bellantoni & Cook [3], Leivant [15,16] and, particularly, Jones [5,6]. Back then I had hardly heard of reversible computing, and it never occurred to me that reversible computing and implicit computational complexity should mix very well.

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 Bellantoni & Cook's [3] 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 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

E-mail address: larsk@math.uio.no.

https://doi.org/10.1016/j.scico.2021.102723
0167-6423/© 2021 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

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 to 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. We will also discuss and analyze the computational power of higher-order versions of our reversible languages.

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 computing 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 & Barak [1], Jones [7] and Sipser [25]).

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 [4] and Pechoux [21]. The latter article gives a survey of the developments in the field over the past 30 years.

The first part of this paper is extended and improved version of [11]. The material in Section 6 has not been published elsewhere. The author wants to thank Romain Pechoux, Geir Overskeid and an anonymous referee for corrections and helpful advice.

A few remarks before we move on

This paper addresses a broader audience than a standard research paper, and the author has intentionally toned down discussions and considerations that will be inaccessible or boring to non-experts. Thus, this is partly a dissemination paper, but it is not only a dissemination paper: Our characterizations of ETIME and P are new, and they should have an interest in their own right. The characterizations are neat, simple and based on a handful of very natural operations. That alone should make them interesting. In addition they are based on inherently reversible operations. That does certainly not make them less interesting, and as far as the author knows, these characterizations are the first of its kind.

The significance of the reversible features of our characterizations is a subject for further research. Perhaps these features can help us gain a better understanding of the nature of ETIME and P? Maybe they can help us gain a better understanding of the relationship between the time complexity classes ETIME and P and the space complexity classes LINSPACE, ESPACE, LOGSPACE and PSPACE? They might even bring us closer to solving some of the notoriously hard open problems involving these complexity classes? Maybe, or maybe not, but these are matters that should be investigated further, and the theory and methodology of reversible computing might very well play an important role in that respect: Our characterizations of ETIME and P were designed from scratch. At the outset they were not inspired by, or in any way related to, the theory and methodology of reversible computing (e.g., we did not achieve these characterizations by using known techniques for transforming a non-reversible language into a reversible one). Perhaps insights already won by research in reversible computing can shed some light on the significance of our characterizations? Maybe the theoretical framework developed to study reversible computing can help us gain a better understanding of the different nature of time complexity classes, like ETIME and P, and space complexity classes, like LINSPACE, ESPACE, LOGSPACE and PSPACE? Maybe the theory of reversible computing and (implicit) computational complexity theory has much to offer each other, in both directions?

2. Reversible bottomless stack (RBS) programs

2.1. The programming language RBS

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 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.

THE SYNTAX OF RBS

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

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.

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 held 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

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

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

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

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

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

Observe that we have

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

and

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

when b is the execution base.

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 $\mathtt{loop}\,\mathtt{X}\,\{\,\mathtt{C}\,\}$ executes the command C repeatedly k times in a row where k is the top element of the stack held by X. Note that the variable X is not allowed to occur in C and, moreover, the command $\mathtt{loop}\,\mathtt{X}\,\{\,\mathtt{C}\,\}$ will not modify the stack held by X.

A couple of examples might be helpful to the reader. Let $\mathtt{C}_1$ be the program $\mathtt{loop}\,\mathtt{X}_1\,\{\,\mathtt{X}_2^+\,\}\,;\,(\mathtt{X}_2\,\mathtt{to}\,\mathtt{X}_1)$. 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 $\mathtt{loop}\,\mathtt{X}_1\,\{\,\mathtt{X}_2^+\,\}\,;\,\mathtt{X}_2^+\,;\,(\mathtt{X}_2\,\mathtt{to}\,\mathtt{X}_1)$. 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. This leads to an important observation:

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

2.2. The problems decidable by RBS programs

We will now define the class of problems that can be decided by an RBS program. 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.

Definition 2.1. An RBS program C *accepts* the natural number m if C executed with input m terminates with 0 at the top of the stack held by $\mathtt{X}_1$, otherwise, C *rejects* m.

A *problem* is nothing but a (not necessarily strict) subset of the natural numbers.[1] An RBS program C *decides* the problem $A_{\mathtt{C}}$ where

$$A_{\mathtt{C}} = \{\, m \mid \mathtt{C}\ \text{accepts}\ m \,\}\ . \quad \square$$

[1] It is pretty standard in computability and complexity theory to define a problem as a set of natural numbers. Problems are often referred to as *decision problems*.

EXAMPLE

```
Program:                 Comments:
                         (* X1 = ⟨m, 0*]  *)
(X1 to X9);              (* the top elements of X9 is m *)
X2+;                     (* X1 = ⟨0*] and X2 = ⟨1, 0*]  *)
loop X9 {                (* repeat m times *)
  (X1 to X3);
  (X2 to X1);            (* swap the top elements of X1 and X2 *)
  (X3 to X2) }
```

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

Before we go on and give one of our main definitions, we will study a few examples: The set of all natural numbers $\mathbb{N}$ is a problem. The program $(\mathtt{X}_1\,\mathtt{to}\,\mathtt{X}_2)$ decides the problem $\mathbb{N}$. No matter what the input might be, the program $(\mathtt{X}_1\,\mathtt{to}\,\mathtt{X}_2)$ will terminate with 0 at the top of the stack held by $\mathtt{X}_1$, and hence, the program accepts all inputs. The empty set $\emptyset$ is also a problem according to our definition as $\emptyset$ is a subset of the natural numbers. The program $(\mathtt{X}_1\,\mathtt{to}\,\mathtt{X}_2)\,;\,\mathtt{X}_1^+$ decides $\emptyset$ as the program always, that is, no matter what the input might be, terminates with 1 at the top of the stack held by $\mathtt{X}_1$, and hence, rejects all inputs. Let A be the set of even numbers. Fig. 2 shows an RBS program which decides the problem A. All right, enough examples, here comes the definition:

Definition 2.2. Let $\mathcal{S}$ denote class of problems decidable by an RBS program, that is, let

$$\mathcal{S} \;=\; \{\, A_{\mathtt{C}} \mid \mathtt{C} \text{ is an } \mathtt{RBS} \text{ program} \,\}\,. \quad \square$$

Now, $\mathcal{S}$ 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?

2.3. RBS programs are reversible

Intuitively, it should be evident that RBS programs are reversible in a very strong sense. RBS is an *inherently reversible* programming language in the terminology of Matos [17] (see also the remarks in Section 7). Each command C has an obvious inverse command $(\mathtt{C})^{-1}$ which will undo the actions of C, and we can straightforwardly define the inversion operator $(\cdot)^{-1}$ inductively over the structure of a command. The details follow.

Definition 2.3. We define *inverse command* of C, written $(\mathtt{C})^{-1}$, inductively over the structure of C:

- $(\mathtt{X}_i^+)^{-1} = \mathtt{X}_i^-$
- $(\mathtt{X}_i^-)^{-1} = \mathtt{X}_i^+$
- $((\mathtt{X}_i\,\mathtt{to}\,\mathtt{X}_j))^{-1} = (\mathtt{X}_j\,\mathtt{to}\,\mathtt{X}_i)$
- $(\mathtt{C}_1\,;\,\mathtt{C}_2)^{-1} = (\mathtt{C}_2)^{-1}\,;\,(\mathtt{C}_1)^{-1}$
- $(\mathtt{loop}\,\mathtt{X}_i\,\{\,\mathtt{C}\,\})^{-1} = \mathtt{loop}\,\mathtt{X}_i\,\{\,(\mathtt{C})^{-1}\,\}$.

In order to improve the readability we may skip parenthesis and, e.g., write $\mathtt{C}^{-1}$ in place of $(\mathtt{C})^{-1}$. $\square$

Theorem 2.4. *Let* C *be a program, and let every variable occurring in* C *be in the list* $\mathtt{X}_1,\ldots,\mathtt{X}_n$*. Furthermore, let* m *be any natural number. We have*

$$\{\mathtt{X}_1 = \langle m, 0^*] \;\wedge\; \bigwedge_{i=2}^{n} \mathtt{X}_i = \langle 0^*]\}\;\mathtt{C}\,;\,\mathtt{C}^{-1}\;\{\mathtt{X}_1 = \langle m, 0^*] \;\wedge\; \bigwedge_{i=2}^{n} \mathtt{X}_i = \langle 0^*]\}\,.$$

It is a nice, and maybe even challenging, exercise to write up a decent proof Theorem 2.4, even if it should be pretty clear that the theorem holds. We offer a proof below.

2.4. The proof of Theorem 2.4

This subsection is dedicated to a detailed proof of Theorem 2.4 (readers not interested may jump ahead to Section 3). 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 $\mathtt{C}^{m+1} \equiv \mathtt{C}^m\,;\,\mathtt{C}$.

Obviously, if a program $\mathtt{C}$ is executed in base b, all the variables in $\mathcal{V}(\mathtt{C})$ will hold b-stacks during the entire execution (see the observation at page 3).

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

$$\text{If } \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}\, \mathtt{C}\, \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}, \text{ then } \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \beta_\ell\}\, \mathtt{C}^{-1}\, \{\bigwedge_{\ell=1}^{n} \mathtt{X}_\ell = \alpha_\ell\}. \qquad \text{(claim)}$$

Theorem 2.4 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 three 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

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

for some $m_1, \dots, m_k < b$. 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 2.3, we have $\{\mathtt{X}_i = \beta_i\}\, (\mathtt{X}_i^+)^{-1}\, \{\mathtt{X}_i = \alpha_i\}$. Now, since neither $\mathtt{X}_i^+$ nor $(\mathtt{X}_i^+)^{-1}$ will modify any stack held by a variable $\mathtt{X}_j$ where $j \neq i$, we also have

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

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 $\gamma_1, \dots, \gamma_n$ 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

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

It follows that

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

Finally, as Definition 2.3 states that $(\mathtt{C}_1\, ;\, \mathtt{C}_2)^{-1} = \mathtt{C}_2^{-1}\, ;\, \mathtt{C}_1^{-1}$, we have

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

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

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

and by Definition 2.3, we have

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

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

First we prove

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

by a secondary induction on m.

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

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

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

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

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

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

Finally, as Definition 2.3 states that $\mathtt{loop}\,\mathtt{X}_i \; \{ \; \mathtt{C}_0^{-1} \; \} = (\mathtt{loop}\,\mathtt{X}_i \; \{ \; \mathtt{C}_0 \; \})^{-1}$, we have

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

This completes the proof of case (v).

3. Simulation of Turing machines

3.1. 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 $O(2^{k|m|})$ (where k is a constant and $|m|$ 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 that 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

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

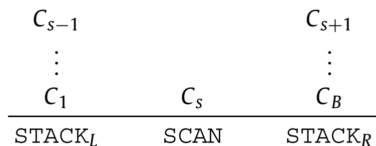
where a_i, a_j are alphabet symbols; q_k, q_ℓ are states; and D is either "left" or "right". Such a tuple is called a transition and tells M what to do when it scans the symbol a_j in state q_k: in that case M will write the symbol a_j, move its head one position in the direction given by D, and then 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 of 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. (Since we have used if-then statements without else-branches, 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. Well, for every given configuration of a Turing machine, 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 STACK_L, a register SCAN, another stack STACK_R, and store the tape content in the following way:



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

```
if SCAN = a_i and STATE = q_k then
    { SCAN := a_j ; ...push and pop according to D ... ; STATE := q_ℓ } .
```

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

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

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.

3.2. 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, except on the rare occasions when the input is 0, the body of each of these loops can 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. 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. 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 the irreversible programming language $\texttt{LOOP}^-$.

3.3. $\texttt{LOOP}^-$ programs

The programming language $\texttt{LOOP}^-$ was introduced in Kristiansen [9]. The type of programs that we normally, and perhaps somewhat colloquially, refer to as "loop programs" can be traced back to a classic paper by Meyer & Ritchie [19]. $\texttt{LOOP}^-$ is very similar to the programming language studied in Meyer & Ricthie's paper, but there is one crucial difference: The classic loop language has an instruction for incrementing the value of a variable by 1. That instruction is not available in $\texttt{LOOP}^-$. That is why it makes sense to dub the language $\texttt{LOOP}^-$. It is more or less the classic loop language *minus* the increment instruction.

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 X:=Y assigns the number held by the variable Y to the variable X. The command pred(X) decreases the value held by the variable X by 1 if the value is strictly greater than 0; and leaves the value held by X untouched 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 loop X { C } executes the command C repeatedly k times in a row where k in the number held by X. Note that the variable X is not allowed to occur in C and that the command loop X { C } does not modify the value held by X.

Let us see how an RBS program can simulate a $\texttt{LOOP}^-$ program. An RBS program can represent a $\texttt{LOOP}^-$ variable X holding natural number m by a single dedicated variable X (we use the same variable name) holding the stack $\langle m, 0^*]$. During the entire simulation the dedicated variable will store a stack of the form $\langle m, 0^*]$. The command $\texttt{X:=}k$ can then be simulated by the program

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

where the auxiliary variable Y works as a trash bin. We can do with only one such trash bin, and we dedicate a particular RBS variable TRASH to work as our trash bin. Thus, the command $\texttt{X:=}k$ is simulated by the program

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

Now, observe that this will only work if the execution base is strictly greater than k, but that will good enough to us. The command $\texttt{X} := \texttt{Y}$ can be simulated by the program

$$(\texttt{X to TRASH})\texttt{;}\ \texttt{loop}\ \texttt{Y}\ \{\ \texttt{X}^+\ \}\ .$$

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:

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

To verify that this simulation indeed works, observe that

- after each execution of the loop's body the value held by X is the predecessor of the value held by Z
- when the loop terminates, the value held by Z equals the value held by Y (which again equals the initial value of X).

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

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

Hence, we arrive at the following conclusion which will be very important to us:

Conclusion. *A $\mathtt{LOOP}^-$ 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 auxiliary variables $\mathtt{X}'$, $\mathtt{Y}'$, Z, U which do not occur in any of the commands $\mathtt{C}_1$ and $\mathtt{C}_2$. First we execute the program

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

This program sets both $\mathtt{X}'$ and $\mathtt{Y}'$ to 0 if X and Y hold the same number. If X and 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

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

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 X and Y hold the same number. If X and 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 [9] 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 $\text{LINSPACE} \subseteq \mathcal{S}$. However, we are on our way to proving a stronger result, namely that $\text{LINSPACE} \subseteq \mathcal{S} = \text{ETIME}$. The equality $\text{LINSPACE} \stackrel{?}{=} \text{ETIME}$ is one of the many notorious open problems of complexity theory. The general opinion is that the equality does not hold.

3.4. *RBS programs that simulates time-bounded Turing machines*

We have seen that RBS programs nearly[2] can simulate $\mathtt{LOOP}^-$ programs, and $\mathtt{LOOP}^-$ programs can assign constants to variables and perform if-then-else constructions. This helps us to see how an RBS program $\mathtt{P}_M$ nearly can simulate an $2^{k_0|m|}$ time Turing machine M.

First, let us clarify what a $2^{k_0|m|}$ time Turing machine is. Our Turing machines take a single natural number m as input. The length of that input is denoted $|m|$, and $|m|$ equals the number of symbols needed to represent m in binary notation. Regard k_0 as a fixed number, e.g., 17 or 314. A $2^{k_0|m|}$ time Turing machine is, by definition, a Turing machine that executes no more than $2^{k_0|m|}$ transitions.

Observe that we have

$$2^{k_0|m|} \;=\; (2^{|m|})^{k_0} \;\le\; (2m)^{k_0} \;\le\; m^{k_0+1}$$

[2] Why do we say "nearly"? We say "nearly" because the simulation of the assignment $\mathtt{X}:=k$ will fail when the execution base is less than or equal to k.

when $m \geq 2$. Now, let $k = k_0 + 1$ and look away from the small inputs 0 and 1, that is, let $m \geq 2$. Then a $2^{k_0|m|}$ time Turing machine M can be simulated by an RBS program $\mathtt{P}_M$ of the form

$\langle$initiate the tape with the input $m\rangle$;
$\mathtt{Y}_1 := \langle$the input $m\rangle$; $\mathtt{Y}_2 := \langle$the input $m\rangle$; ... ; $\mathtt{Y}_k := \langle$the input $m\rangle$;
$\mathtt{loop}\ \mathtt{Y}_1 \{ \mathtt{loop}\ \mathtt{Y}_2 \{ \dots \mathtt{loop}\ \mathtt{Y}_k \{ \mathtt{T}_1 ; \mathtt{T}_2 ; \dots ; \mathtt{T}_r \} \dots \} \}$.

Furthermore, the program $\mathtt{P}_M$ represents

- M's alphabet $a_1, \dots a_A$ by the numbers $1, \dots, A$
- M's states $q_1, \dots q_Q$ by the numbers $1, \dots, Q$
- the current content of the tape by two stacks
- the current state by a variable STATE
- the scanned cell by a variable SCAN

and emulates a transition a_i, q_k, a_j, D, q_ℓ with the command

$\mathtt{T}_s :\equiv$ if SCAN = i and STATE = k then { SCAN := j ; ...push and pop according to D ... ; STATE := ℓ }

for $s = 1, \dots, r$.

This completes our description of the RBS program $\mathtt{P}_M$. The program performs (simulated) assignments of constants to (simulated) variables holding natural numbers. All these constants will be less than or equal to $\max(A, Q)$. If $\mathtt{P}_M$ is executed in a base that is strictly greater than $\max(A, Q)$, then $\mathtt{P}_M$ will for sure simulate the Turing machine M. If $\mathtt{P}_M$ is executed in a base that is less than or equal to $\max(A, Q)$, then $\mathtt{P}_M$ will probably fail to simulate M (see our conclusion at page 9). So we have a problem, and in some sense we cannot deal with that problem. An RBS program will not be able to simulate (in any reasonable sense of the word) an arbitrary $2^{k_0|m|}$ time Turing machine M on small inputs, but fortunately, there will still 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 execution base always will be strictly greater than 1. Hence, if the problem A is decided by a $2^{k_0|m|}$ time Turing machine M, there will also be an RBS program that decides A. Such a program will be of the form

X := $\langle$the input $m\rangle$;
if X = 0 then $\langle$ give correct output for $m = 0$ $\rangle$
else { pred(X) ;
if X = 0 then $\langle$ give correct output for $m = 1$ $\rangle$
else { pred(X) ;
if X = 0 then $\langle$ give correct output for $m = 2$ $\rangle$
⋮
else { $\langle$ the input is big enough, that is, bigger than $\max(A, Q)$ ...
...use $\mathtt{P}_M$ to simulate M ...
...accept if M accepts, reject if M rejects $\rangle$ } ... } } .

4. Implicit characterizations of ETIME and P

4.1. A characterization of ETIME

Definition 4.1. 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 $\mathrm{ETIME} = \bigcup_{i \in \mathbb{N}} \mathrm{ETIME}_i$.

Theorem 4.2. $\mathcal{S} = \mathrm{ETIME}$.

Proof. First we prove the inclusion $\mathcal{S} \subseteq \mathrm{ETIME}$. Assume $A \in \mathcal{S}$ (we will argue that $A \in \mathrm{ETIME}$). Then there is an RBS program C that decides A. Assume that C is executed with input m. By the observation at page 3, all the numbers stored on stacks during the execution will be less than or equal to $\max(m, 1)$, and thus, no loop will iterate its body more than $\max(m, 1)$ times in a row. Thus, it is easy to see that there exists a polynomial p_0 such that $p_0(m)$ is an upper bound for the number of primitive instruction executed by C. A Turing machine can simulate the execution of C at the expense of a polynomial-time overhead. Thus there exists a polynomial p, such that A can be decided by a Turing machine working in time $p(m)$. Let k_0 be a sufficiently large number. Then we have

$$p(m) < (m+2)^{k_0} \leq (2^{|m|} + 2)^{k_0} < (2^{|m|})^{(k_0+1)} = 2^{(k_0+1)|m|} .$$

Thus, there exists k such that A can be decided by a Turing machine working in time $2^{k|m|}$. Thus, we have $A \in$ 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 $\subseteq \mathcal{S}$. □

4.2. 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 $\{\mathbf{a}, \mathbf{b}\}$. Every program has a dedicated read-only variable inp which holds the input string when the execution of the program starts, and the execution base will be set to the length of the string stored in inp, that is, to the number of symbols in the input. Otherwise, nearly everything is kept as before: Every variable will still hold a bottomless stack of 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 X_1, otherwise, the program rejects its input. Moreover, all variables, including X_1, the variable that was 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

case inp[X]=a: { *com* } and case inp[X]=b: { *com* }

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 $\alpha_0\alpha_1, \dots, \alpha_{b-1}$ where b is the execution base and $\alpha_i \in \{\mathbf{a}, \mathbf{b}\}$. Assume that X_j holds a stack where top element is k. The command

case inp[X_j]=a: { C }

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

case inp[X_j]=b: { C }

executes the command C if $\alpha_k = \mathbf{b}$, otherwise, the command does nothing. We still have a reversible language. We can straightforwardly define the inverse commands of our two new commands. The variable X_j is not allowed to occur in C and will consequently not be modified by C. Thus, we may extend Definition 2.3 by

- (case inp[X_j]=a: { C })$^{-1}$ = case inp[X_j]=a: { (C)$^{-1}$ }
- (case inp[X_j]=b: { C })$^{-1}$ = case inp[X_j]=b: { (C)$^{-1}$ }

and Theorem 2.4 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. At this stage, we should make an observation which is analogous to the one we made at page 3:

Observation. *All numbers stored on stacks during an execution of an RBS′ program will be strictly less than the execution base, and thus, less than or equal to $|w| - 1$ where $|w|$ is the number of symbols in the input w.*

Definition 4.3. A *language* L is a set of strings over the alphabet $\{\mathbf{a}, \mathbf{b}\}$, moreover, every string in L is of length at least 2.

An RBS′ program C decides the language L_C where

$$L_C = \{ w \mid \text{C accepts } w \} .$$

Let $\mathcal{S}'$ denote the class of languages decidable by an RBS′ program, that is

$$\mathcal{S}' = \{ L_C \mid \text{C is an RBS}' \text{ program} \} .$$

Let $|w|$ denote the length of the string w. A Turing machine M is a *polynomial time Turing machine* if there exists a polynomial p such that $p(|w|)$ is an upper bound for the number of steps M will execute on input w. A language L is a *polynomial time language* if there exists a polynomial time Turing machine that decides L. The complexity class P is the class of all polynomial time languages. □

EXAMPLE

```
Program:                 Comments:
                         (* all stacks hold the zero stack *)
X_2^-                    (* the top element of X_2 is b − 1 *)
loop X_2 {               (* repeat b − 1 times *)
case inp[X_3]=b:         (* X_3 is a pointer into the input *)
  { (X_1 to X_9);        (* X_1 holds the zero stack *)
    X_1^+                (* top element of X_1 is 1 *)
  };
X_3^+                    (* move pointer to the right *)
};                       (* end of loop *)
case inp[X_3]=a:         (* top element of X_3 is b − 1 *)
  { (X_1 to X_9);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 $\mathtt{X}_1 = \langle 1, 0^*]$.

Fig. 4 shows an $\mathtt{RBS}'$ program which decides the language given by the regular expression $\mathbf{a}^*\mathbf{ab}$.

Theorem 4.4. $\mathcal{S}' = \mathrm{P}$.

Proof. Assume $L \in \mathcal{S}'$ (we will prove $L \in \mathrm{P}$). Then we have an $\mathtt{RBS}'$ program $\mathtt{C}$ that decides L. There exists a polynomial p such that the number of primitive instructions executed by $\mathtt{C}$ on input w is bounded by $p(|w|)$. It is easy to see that such a p exists: By the observation above, all numbers stored on stacks during an execution of $\mathtt{C}$ will be strictly less than or equal to $|w|-1$, and thus, a loop body in $\mathtt{C}$ will be executed maximum $|w|-1$ times. A Turing machine can simulate the execution of $\mathtt{C}$ on input w at the expense of a polynomial-time overhead. Thus, there exists a polynomial time Turing machine that decides L. Thus, L is a polynomial time language, and we have $L \in \mathrm{P}$. This proves the inclusion $\mathcal{S}' \subseteq \mathrm{P}$.

Assume $L \in \mathrm{P}$ (we will prove $L \in \mathcal{S}'$). Then we have polynomial time Turing machine M that decides L. Let $p(|w|)$ be a polynomial upper bound on the number of steps M will execute on input w. When k is sufficiently big and $|w| > 2$, an $\mathtt{RBS}'$ program of the form

$$\mathtt{Y}_1 := |w|-1\,;\ \mathtt{Y}_2 := |w|-1\,;\ \ldots\,;\ \mathtt{Y}_k := |w|-1\,;$$
$$\mathtt{loop}\ \mathtt{Y}_1\ \{\ \mathtt{loop}\ \mathtt{Y}_2\ \{\ \ldots \mathtt{loop}\ \mathtt{Y}_k\ \{\ \langle \ldots \text{a list of transitions} \ldots \rangle\ \}\ \ldots\ \}\ \}$$

will iterate $\langle \ldots$ a list of transitions $\ldots\rangle$ more than $p(|w|)$ times in a row. Now we can proceed as in Subsection 3.4 and construct an $\mathtt{RBS}'$ program that decides the same language as M. Thus we have $L \in \mathcal{S}'$. This proves the inclusion $\mathrm{P} \subseteq \mathcal{S}'$. □

5. A brief interlude

Our proofs of the two equalities $\mathcal{S} = \mathrm{ETIME}$ and $\mathcal{S}' = \mathrm{P}$ follow a kind of standard recipe which is used over and over again in implicit computational complexity theory. Let us consider the proof of $\mathcal{S}' = \mathrm{P}$. We prove the equality by proving that $\mathcal{S}'$ is included in P and that P is included in $\mathcal{S}'$. In order to prove the inclusion $\mathcal{S}' \subseteq \mathrm{P}$ we argue that

(I) The number of primitive instructions carried out by an $\mathtt{RBS}'$ program is bounded by $p(|w|)$ where $|w|$ is the length of the input and p is a polynomial.
(II) A Turing machine can simulate an $\mathtt{RBS}'$ program at the expense of a polynomial-time overhead.

In order to prove the inclusion $\mathrm{P} \subseteq \mathcal{S}'$ we argue that

(III) For any polynomial p there exists an $\mathtt{RBS}'$ program that is able to iterate an arbitrary command $\mathtt{C}$ more than $p(|w|)$ times where $|w|$ is the length of the input.
(IV) $\mathtt{RBS}'$ is expressive enough to represent Turing machine configurations and emulate a transition from one configuration to another.

These are the four main insights which our proof of $\mathcal{S}' = \mathrm{P}$ is based on. The two former ones yield the inclusion $\mathcal{S}' \subseteq \mathrm{P}$, and the two latter ones yield the inclusion $\mathrm{P} \subseteq \mathcal{S}'$. Our proof of $\mathcal{S} = \mathrm{ETIME}$ is based on four similar insights: Now, (I) becomes

> the number of primitive instructions carried out by an $\mathtt{RBS}$ program is bounded by $2^{k|m|}$ where $|m|$ is the length of the input and k is a constant

and (III) becomes

THE SYNTAX OF 2RBS

$$\begin{array}{rcl} X \in \textbf{Variable} & ::= & \mathtt{X}_1 \mid \mathtt{X}_2 \mid \mathtt{X}_3 \mid \ldots \\ com \in \textbf{Command} & ::= & X^+ \mid X^- \mid (X \,\mathtt{to}\, X) \mid com \,;\, com \\ & \mid & \mathtt{loop}\, X \,\{\, com \,\} \mid \mathtt{2loop}\, X \,\{\, com \,\} \end{array}$$

Fig. 5. The syntax of the language 2RBS. The variable X in the loop-command is not allowed to occur in the loop's body. The same goes for variable X in the 2loop-command.

for any fixed k there exists an RBS program that is able to iterate an arbitrary command C more than $2^{k|m|}$ where $|m|$ is the length of the input

whereas (II) and (IV) more or less remain as above.

Proofs of implicit characterizations of complexity classes tend to follow the recipe indicated above. Maybe it would be too categorical to say that all such proofs stick to the pattern of the recipe, but it should not be controversial to say that insights corresponding to (I), (II), (III) and (IV) to a certain extent will be present in any such proof. We will for sure stick to the pattern in the next section where we give an implicit characterization of the class of Kalmar elementary problems.

6. Reversible bottomless stack programs of higher orders

6.1. Some considerations

What's next? Well, it seems like it will be hard to find a natural and interesting reversible language that is weaker than RBS or RBS′. It might be possible, but the author's hunch is that research along that line might easily turn into a waste of time. It seems more fruitful to look for extensions of RBS and RBS′. For convenience, let us pay attention to RBS, and just forget RBS′. Can we extend RBS with natural reversible commands which make the language more powerful?

There are plenty of examples of restricted programming languages that become more powerful when they are extended to higher orders, see e.g. Jones [6], Kristiansen [10] or Kop & Simonsen [8]. Higher-order programs of such languages can decide problems no first-order program of the language can decide. It seems like a reasonable research project to extend RBS with higher types, and then study classes of decision problems captured by higher-order fragments of RBS. We will give this project a modest try.

At the outset it is absolutely not clear what a higher-order RBS program should look like. Our first-order objects are indeed not stacks, they are natural numbers. Stacks are second-order objects. Thus, stack of stacks will be third-order objects. What will the forth-order objects be? stacks of "stacks of stacks" or, perhaps, "stacks of stacks" of "stacks of stacks"? And what kind of invertible commands and program constructions do we need to deal with higher-order stacks? The overall picture is confusing, but we will at least be able to present a decent second-order version of RBS.

6.2. The programming language 2RBS

The programming language 2RBS is RBS extended with a command of the form

$$\mathtt{2loop}\ \mathtt{X}\ \{\ \mathtt{C}\ \}\ . \qquad (*)$$

This command is a loop construction where the number of times the loop's body is executed is determined by several of the numbers stored by the stack X and not only the top element of the stack.

The loop command of RBS, which also will be available in 2RBS, is a *first-order loop* as the number of times the loop's body is executed is determined by a first-order object (a natural number). The number of times the body of (*) is executed is determined by a collection of first-order objects (a stack). Collections of first-order objects might be viewed as *second-order objects*. Accordingly, we will say that 2RBS is second-order programming languages and that (*) is a *second-order command* or a *second-order loop*. Commands that do not contain second-order loops will be referred to as first-order commands.

We will now explain the semantics of the second-order loop. Let C be a command, and let $\langle x_1, x_2, x_3, \ldots]$ be a bottomless stack. Furthermore, let

$$\textbf{it}(\ \langle x_1, x_2, x_3, \ldots]\ ,\ \mathtt{C}\ ) = \begin{cases} \textbf{it}(\ \langle x_2, x_3, \ldots]\ ,\ \mathtt{C}\ )^{x_1} & \text{if } x_1 > 0 \\ \mathtt{C} & \text{otherwise.} \end{cases}$$

The command (*) is executed by executing the command $\textbf{it}(\alpha, \mathtt{C})$ where α is the stack held by X.

A small example might be helpful. We have

$$\begin{aligned} \textbf{it}(\ \langle 3,2,7,0,0^*]\ ,\ \mathtt{C}\ ) &= \textbf{it}(\ \langle 2,7,0,0^*]\ ,\ \mathtt{C}\ )^3 = (\textbf{it}(\ \langle 7,0,0^*]\ ,\ \mathtt{C}\ )^2)^3 \\ &= (\textbf{it}(\ \langle 7,0,0^*]\ ,\ \mathtt{C}\ )^6 = ((\textbf{it}(\ \langle 0,0^*]\ ,\ \mathtt{C}\ )^7)^6 \\ &= ((\textbf{it}(\ \langle 0,0^*]\ ,\ \mathtt{C}\ )^{42} = \mathtt{C}^{42} \end{aligned}$$

Thus, when X holds the stack $\langle 3, 2, 7, 0, 0^*]$, the command (*) will execute the command $\mathtt{C}^{42}$, that is, C iterated 42 times in a row.

This semantics should be considered as natural: Observe that our second-order loop is nothing but a sequence of nested first-order loops. Let us for a while permit constants in our programs, and let $x_1, \dots, x_n$ be natural numbers strictly greater than 0. Then, executing the second-order loop

$$\mathtt{2loop}\ \langle x_1, x_2 \dots, x_n, 0, \dots]\ \{\ \mathtt{C}\ \}$$

amounts to executing the sequence of nested first-order loops

$$\mathtt{loop}\ x_1\ \{\ \mathtt{loop}\ x_2\ \{\ \dots \mathtt{loop}\ x_n\ \{\ \mathtt{C}\ \} \dots\ \}\ \}\ .$$

This explains the semantics of the second-order loop. The remaining 2RBS commands are all RBS commands and work the way we are used to (see Fig. 5). Indeed, the programming language 2RBS is RBS with second-order loops, nothing more, nothing less: a 2RBS program imports its input, sets the execution base and accepts, or rejects, the input in the same way as an RBS program does. The stack held by X will not be modified by a command of the form 2loop X { C }, indeed, X is not even allowed to occur in C, and Theorem 2.4 will still hold when we extend the definition of our inversion operator (Definition 2.3) by the clause

$$(\mathtt{2loop}\ \mathtt{X}_i\ \{\ \mathtt{C}\ \})^{-1} = \mathtt{2loop}\ \mathtt{X}_i\ \{\ (\mathtt{C})^{-1}\ \}\ .$$

Thus, 2RBS is a reversible language. It is equally as reversible as RBS.

6.3. *The power of 2RBS*

So now we have a second-order version of RBS. It might happen that it is possible find more attractive versions. Maybe we could, or should, have done things differently, but we have at least cooked up a version that makes sense. What will a third-order version look like? Well, third-order programs should be able to work with stacks of stacks, and thus variables might hold objects of the form

$$\Big\langle\ \langle x_1^1, x_2^1, x_3^1, \dots]\ ,\ \langle x_1^2, x_2^2, x_3^2, \dots]\ ,\ \langle x_1^3, x_2^3, x_3^3, \dots]\ ,\ \langle x_1^4, x_2^4, x_3^4, \dots]\ ,\ \dots\ \Big]$$

where each x_j^i is a natural number. A third-order loop

$$\mathtt{3loop}\ \mathtt{X}\ \{\ \mathtt{C}\ \}$$

might then be executed as sequence of nested second-order loops

$$\mathtt{2loop}\ \langle x_1^1, x_2^1, x_3^1, \dots]\ \{\ \mathtt{2loop}\ \langle x_1^2, x_2^2, x_3^2, \dots]\ \{\ \dots\ \{\ \mathtt{C}\ \} \dots\ \}\ \}$$

...something along these lines should work ...In such a language we will of course also need operators that can build and modify stacks of stacks.

We will not discuss or speculate any further on what third or fourth order versions RBS might look like. It is beyond the scope of the current paper to study such languages. We will go on and study the modest higher-order version we already have defined, namely 2RBS. This is a version with just a slight touch of higher-orderness. Still it turns out that 2RBS is very powerful compared to RBS when it comes to decision problems. It seems like the class of all 2RBS programs captures the class of primitive recursive problems. A small fragment of 2RBS (the programs of rank of 1) captures the class of all elementary problems which—from a complexity-theoretic point of view—is a huge class of problems.

Intuitively, a program is of rank 1 if no second-order loop in the program occurs inside another loop. The formal definition follows.

Definition 6.1. Let C be a 2RBS command. The natural number $R(\mathtt{C})$ is given by

- if C is a first-order command, let $R(\mathtt{C}) = 0$
- if $R(\mathtt{C}_1) > 0$ or $R(\mathtt{C}_2) > 0$, let $R(\mathtt{C}_1\ ;\ \mathtt{C}_2) = \max(R(\mathtt{C}_1), R(\mathtt{C}_2))$
- if $R(\mathtt{C}) > 0$, let $R\left(\mathtt{loop}\ \{\ \mathtt{C}\ \}\right) = R(\mathtt{C}) + 1$
- let $R\left(\mathtt{2loop}\ \{\ \mathtt{C}\ \}\right) = R(\mathtt{C}) + 1$ for any command C.

This completes the definition of $R(n)$. A command C is *of rank n* if $R(\mathtt{C}) \leq n$. We will use $\mathcal{S}_n$ to denote the class of problems decidable by a 2RBS program of rank n. □

The notion of being elementary[3] is important in computability and complexity theory. A function might be elementary, and so might a relation or a problem. The reader should be aware that our definition below of the elementary problems is not a standard one. For convenience we give a definition based on Turing machines. Our definition is of course equivalent to the standard one, but it might look pretty different form definitions you might find elsewhere, e.g. in Odifreddi [20] (see also Theorem VIII.7.6 at page 272 in [20]).

Definition 6.2. Let $2^y_0 = y$ and $2^y_{x+1} = 2^{2^y_x}$.

A problem is *elementary* if it can be decided by a 2^n_k time Turing machine for some constant k.

We will use $\mathcal{E}_*$ to denote the class of elementary problems. □

It is a standard convention to use n to denote the length of the input and measure the running time of a Turing machine as a function of n. We stick to that convention. So in the definition above n is the length of the input whereas k is a fixed number which does not depend on n.

Theorem 6.3. $\mathcal{E}_* = \mathcal{S}_1$.

We will give a detailed proof of Theorem 6.3. In Section 6.4, we prove that any elementary problem can be decided by a rank-1 `2RBS` program. In Section 6.5, we prove that any problem decided by a rank-1 `2RBS` program is elementary. The theorem follows.

6.4. The proof of the inclusion $\mathcal{E}_* \subseteq \mathcal{S}_1$

Lemma 6.4. *For any natural number k there exist a rank-1 `2RBS` program $\mathtt{TowerOfTwo}_k$ and a program variable[4] $\mathtt{Y}_k$ such that*

$$\{\mathtt{X}_1 = \langle m, 0^*]\}\ \mathtt{TowerOfTwo}_k\ \{\mathtt{Y}_k = \langle x_1, \ldots, x_n, 0^*]\}\ ,$$

where $n = 2^m_k$ and $x_i = 2$ (for all $i \le n$).

Proof. We prove the lemma by induction on k. Throughout this proof we use `ZERO` to denote a dedicated program variable holding the zero stack.

Assume $k = 0$. Let $\mathtt{TowerOfTwo}_0$ be the program

$$\mathtt{loop}\ \mathtt{X}_1\ \{\ (\mathtt{ZERO}\ \mathtt{to}\ \mathtt{Y}_0)\ ;\ \mathtt{Y}_0^+\ ;\ \mathtt{Y}_0^+\ \} \qquad (*)$$

The variable $\mathtt{Y}_0$ holds the zero stack when the execution of (*) starts. The one and only loop in (*) will be executed m times since m is the top element of the stack held by $\mathtt{X}_1$. Each time the loop is executed an additional copy of 2 is created at the top of the stack held $\mathtt{Y}_0$. Thus the m top elements of the stack will all equal 2 when the program terminates, and element number $m+1$ will be a 0. We have $m = 2^m_0 = 2^m_k$. Thus we have

$$\{\mathtt{X}_1 = \langle m, 0^*]\}\ \mathtt{TowerOfTwo}_0\ \{\mathtt{Y}_0 = \langle x_1, \ldots, x_n, 0^*]\}$$

where $n = 2^m_k$ and $x_i = 2$ (for all $i \le n$). Moreover, it is easy to see that $\mathtt{TowerOfTwo}_0$ is of rank 1. The program is even of rank 0. This proves that the lemma holds when $k = 0$.

We turn to the induction step. Our induction hypothesis yields a rank-1 program $\mathtt{TowerOfTwo}_{k-1}$ and a variable $\mathtt{Y}_{k-1}$ with the properties given in the lemma. Let $\mathtt{Y}_k$ be a fresh variable, and let $\mathtt{TowerOfTwo}_k$ be the program

$$\mathtt{TowerOfTwo}_{k-1}\ ;\ \mathtt{2loop}\ \mathtt{Y}_{k-1}\ \{\ (\mathtt{ZERO}\ \mathtt{to}\ \mathtt{Y}_k)\ ;\ \mathtt{Y}_k^+\ ;\ \mathtt{Y}_k^+\ \} \qquad (**)$$

Assume that $\mathtt{X}_1 = \langle m, 0^*]$ when the execution of (**) starts. Our induction hypothesis assures that we have $\mathtt{Y}_{k-1} = \langle x_1, \ldots x_n, 0^*]$, with $n = 2^m_{k-1}$ and $x_i = 2$ (for all $i \le n$), when the execution of the final loop of (**) starts. Moreover, as $\mathtt{Y}_k$ is fresh, $\mathtt{Y}_k$ holds the zero stack when the execution of the final loop of (**) starts. The final loops body will obvious be iterated 2^n times, and we have $2^n = 2^{2^m_{k-1}} = 2^m_k$. Each iteration creates a new 2 at top of the stack held by $\mathtt{Y}_k$. When (**) terminates, we have $\mathtt{Y}_k = \langle x_1, \ldots, x_n, 0^*]$ where $n = 2^m_k$ and $x_i = 2$ (for all $i \le n$). It is easy to see that $\mathtt{TowerOfTwo}_k$ will be of rank 1. This concludes the induction step. □

[3] Also known as *Kalmar elementary* and as *Kalmar-Csillag elementary*.

[4] The variables in our programming language are named $\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3, \ldots$ (see Fig. 5). All these variables are different, e.g., $\mathtt{X}_4$ is not the same variable as $\mathtt{X}_9$. We will be a bit sloppy and use the same typewriter font to denote metavariables, that is, variables that range over the variables $\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3, \ldots$. If $\mathtt{Y}_4$ and $\mathtt{Y}_9$ are metavariables, then it might very well happen that $\mathtt{Y}_4$ and $\mathtt{Y}_9$ are the same variable, e.g., they might both be the variable $\mathtt{X}_{17}$.

We are now ready to prove the inclusion $\mathcal{E}_* \subseteq \mathcal{S}_1$. Let A be an arbitrary problem in $\mathcal{E}_*$. By Definition 6.2, there exist a natural number k and a Turing machine M that decides if m is in A in less than $2_k^{|m|}$ steps. We will see that a 2RBS program of rank 1 also can decide if m is in A. Thus we have $A \in \mathcal{S}_1$, and our theorem holds.

So what does this rank-1 program look like? Well, it looks quite like the program we discussed in Section 3.4. It is based on the same ideas. Recall the command $\mathtt{T}_1\,;\,\mathtt{T}_2\,;\,\dots\,;\,\mathtt{T}_r$ explained in Subsection 3.4. Each $\mathtt{T}_s$ (for $i = 1,\dots,r$) is a command

$$\mathtt{if\ SCAN} = i \ \mathtt{and\ STATE} = k \ \mathtt{then}\ \{\ \mathtt{SCAN} := j\,;\ \dots\text{push and pop}\dots\ ;\ \mathtt{STATE} := \ell\ \}\ ,$$

which takes care of one of M's transitions a_i, q_k, a_j, D, q_ℓ. A program that iterates the command $\mathtt{T}_1\,;\,\mathtt{T}_2\,;\,\dots\,;\,\mathtt{T}_r$ sufficiently many times, that is, at least $2_k^{|m|}$ times, can be easily turned into a program that decides if m is in A. Now, a rank-1 program can indeed iterate the command sufficiently many times. By Lemma 6.4, we have a rank-1 program $\mathtt{TowerOfTwo}_k$ and a variable Y such that

$$\{\mathtt{X}_1 = \langle m, 0^*]\}\ \mathtt{TowerOfTwo}_k\ \{\mathtt{Y} = \langle x_1,\dots,x_n,0^*]\}$$

where $n = 2_k^m$ and $x_i = 2$ (for all $i \le n$). We have $2_k^m \ge 2_k^{|m|}$. Hence, the program

$$\mathtt{TowerOfTwo}_k\,;\,\mathtt{2loop\ Y}\ \{\ \mathtt{T}_1\,;\,\mathtt{T}_2\,;\,\dots\,;\,\mathtt{T}_r\ \} \tag{*}$$

does the job. It is easy to see that (*) is a program of rank 1 (the subprogram $\mathtt{T}_1\,;\,\mathtt{T}_2\,;\,\dots\,;\,\mathtt{T}_r$ is of rank 0 as it is a first-order program).

6.5. The proof of the inclusion $\mathcal{S}_1 \subseteq \mathcal{E}_*$

Definition 6.5. We define the *height of the (bottomless) stack* $\langle x_1, x_2,\dots]$, written $\|\langle x_1, x_2,\dots]\|$, by

$$\|\langle x_1,x_2,x_3,\dots]\| = \begin{cases} 1 + \|\langle x_2,x_3,\dots]\| & \text{if } x_1 > 0 \\ 0 & \text{otherwise.}\end{cases}$$

A stack α is *stored by the program* C *on input* m if some variable occurring in C holds α during the execution of C on input m. A *stack bound* for a program C is a function $S:\mathbb{N}\to\mathbb{N}$ such that any stack α stored by C on input m satisfies $\|\alpha\| \le S(m)$.

A *running-time bound of the program* C is a function $T:\mathbb{N}\to\mathbb{N}$ such that $T(b)$ is an upper bound for the number of primitive commands (commands of the form $\mathtt{X}_i^+$, $\mathtt{X}_i^-$ and $(\mathtt{X}_i\ \mathtt{to}\ \mathtt{X}_j)$) carried out when C is executed in base b (recall that $b = \max(m+1, 2)$ where m is the input to C). □

Let us study a small example before we proceed with our proofs. Assume $\alpha = \langle 2,5,2,0,17,0^*]$. Then we have $\|\alpha\| = 3$. Furthermore, assume that Y holds the stack α. Then the loop $\mathtt{2loop\ Y}\ \{\ \dots\ \}$ will execute its body 20 times because 20 is the number we get when we multiply the 3 first element of α.

Lemma 6.6. *If T is running-time bound T for the program* C*, then T is also a stack bound for* C*.*

Proof. The height of the input stack is 1 and any program has to execute at least one primitive command. If the program executes exactly one command, it cannot make a stack of height 2. Furthermore, when a program increases the height of a stack by 1, it has to execute at least one primitive instruction. This proves the lemma. □

Lemma 6.7. *Let* C *be a program of rank 1, and let b be the execution base of* C*. Then, there exists k such that 2_k^b is a running-time bound for* C*.*

Proof. A second-order loop cannot occur inside another loop in a command of rank 1, see Definition 6.1. Thus, C will be of the form

$$\begin{aligned}&\mathtt{C}_0\,;\,\mathtt{2loop\ Y}_1\ \{\ \mathtt{B}_1\ \}\,;\,\mathtt{C}_1\,;\,\mathtt{2loop\ Y}_2\ \{\ \mathtt{B}_2\ \}\,;\,\mathtt{C}_2\,;\,\dots\\ &\quad \dots\,;\,\mathtt{2loop\ Y}_{n-1}\ \{\ \mathtt{B}_{n-1}\ \}\,;\,\mathtt{C}_{n-1}\,;\,\mathtt{2loop\ Y}_n\ \{\ \mathtt{B}_n\ \}\,;\,\mathtt{C}_n\end{aligned} \tag{*}$$

where $n \ge 0$ and $\mathtt{C}_0,\mathtt{C}_1,\dots,\mathtt{C}_n,\mathtt{B}_1,\dots\mathtt{B}_n$ are first-order programs. Maybe some of the commands $\mathtt{C}_0,\mathtt{C}_1,\dots,\mathtt{C}_n$ are absent, but we can without loss of generality assume that they all are there.

At this stage it is convenient to introduce the notation $\exp(x,y,n)$ defined by

$$\exp(x,y,0) = y \quad \text{and} \quad \exp(x,y,n+1) = x^{\exp(x,y,n)}$$

and make the following claim:

(Claim) There exists k such that $\exp(b, b^k, n)$ is a running-time bound for the program (*).

Note that (*) is executed in base b and that n and k are constants which do not depend on b. The claim states that the function $T : \mathbb{N} \to \mathbb{N}$, where $T(b) = \exp(b, b^k, n)$, is a running-time bound for (*). We prove the claim by induction on n. Note we can make the number b^k arbitrary large by picking a large k since the execution base b always is strictly greater than 1.

Let $n = 0$. Then (*) is a first-order program. Any loop in the program will be executed maximum $b - 1$ times. Thus it is easy to see that b^k will be a running-time bound for (*) when k is a sufficiently large constant. We have $\exp(b, b^k, 0) = b^k$ and thus the claim holds when $n = 0$.

Let $n > 0$. Our induction hypothesis yields ℓ such that $\exp(b, b^\ell, n - 1)$ is a running-time bound for the command

$$\mathtt{C}_0\,\mathtt{;}\,\mathtt{2loop}\,\mathtt{Y}_1\,\{\,\mathtt{B}_1\,\}\,\mathtt{;}\,\mathtt{C}_1\,\mathtt{;}\,\mathtt{2loop}\,\mathtt{Y}_2\,\{\,\mathtt{B}_2\,\}\,\mathtt{;}\,\mathtt{C}_2\,\mathtt{;}\,\ldots\ldots\,\mathtt{;}\,\mathtt{2loop}\,\mathtt{Y}_{n-1}\,\{\,\mathtt{B}_{n-1}\,\}\,\mathtt{;}\,\mathtt{C}_{n-1} \qquad (**)$$

By Lemma 6.6, $\exp(b, b^\ell, n - 1)$ is also a stack bound for (**). Hence we have $\|\alpha\| \leq \exp(b, b^\ell, n - 1)$ where α denotes the stack held by $\mathtt{Y}_n$ when the execution of (**) terminates. All elements stored on α will be strictly less than b. Hence, the loop $\mathtt{2loop}\,\mathtt{Y}_n\,\{\,\mathtt{B}_n\,\}$ will iterate its body $\mathtt{B}_n$ less than $b^{\exp(b,b^\ell,n-1)}$ times. Now, $\mathtt{B}_n$ and $\mathtt{C}_n$ are first-order commands, and thus we have ℓ_1, ℓ_2 such that b^{ℓ_1} and b^{ℓ_3}, respectively, are upper bounds for the number of primitive instructions carried out by $\mathtt{B}_n$ and $\mathtt{C}_n$. Hence the number of primitive instructions carried out by the command $\mathtt{2loop}\,\mathtt{Y}_n\,\{\,\mathtt{B}_n\,\}\,\mathtt{;}\,\mathtt{C}_n$ will be bounded by

$$(b^{\exp(b,b^\ell,n-1)} \times b^{\ell_1}) + b^{\ell_2}\,.$$

The number of primitive commands carried out by the program (*) will be bounded by

$$\exp(b, b^\ell, n-1) + (b^{\exp(b,b^\ell,n-1)} \times b^{\ell_1}) + b^{\ell_2}$$

and thus the claim holds as we have

$$\exp(b, b^\ell, n-1) + (b^{\exp(b,b^\ell,n-1)} \times b^{\ell_1}) + b^{\ell_2} \leq b^{\exp(b,b^k,n-1)} = \exp(b, b^k, n)$$

when we pick a sufficiently large k, e.g., let $k = 2\ell + \ell_1 + \ell_2$. This completes the proof of our claim.

The lemma follows straightforwardly from the claim. Just observe that for any n_0, k_0 there exists k such that $\exp(b, b^{k_0}, n_0) \leq 2^b_k$. □

We are now ready to prove the inclusion $\mathcal{S}_1 \subseteq \mathcal{E}_*$. Assume $A \in \mathcal{S}_1$ (we will argue that $A \in \mathcal{E}_*$). Thus, a rank-1 program $\mathtt{C}$ can decide if m is in A. By Lemma 6.7, we have a constant ℓ such that $2_\ell^{\max(m+1,2)}$ is a running-time bound for $\mathtt{C}$ (recall that $\mathtt{C}$ with input m will be executed in base $\max(m + 1, 2)$). Turing machines can of course simulate 2RBS programs. A Turing machine simulating $\mathtt{C}$ on input m will not execute more than $2_\ell^{\max(m+1,2)}$ primitive 2RBS instructions. Thus, it is not very hard to see that there will be a polynomial $p(x)$ such that a Turing machine working in time $p(2_\ell^{\max(m+1,2)})$ can decide if m is in A (this polynomial is due to the overhead the Turing machine introduces when simulating the primitives of 2RBS). For some sufficiently large constant k, we have $p(2_\ell^{\max(m+1,2)}) < 2^n_k$ where n is the length of m, that is, $n = |m|$. Thus, a 2^n_k time Turing machine can decide A. By Definition 6.2, we have $A \in \mathcal{E}_*$.

6.6. A conjecture

Recall that $\mathcal{S}_1$ denotes the set of problems decidable by a 2RBS program of rank 1, $\mathcal{S}_2$ denotes the set of problems decidable by a 2RBS program of rank 2, and so on.

The author will not be surprised if it turns out that $\mathcal{E}_*^{i+2} = \mathcal{S}_i$, for all $i \geq 1$, where $\mathcal{E}_*^{i+2}$ is the set of problems that belong to the Grzegorczyk class $\mathcal{E}^{i+2}$ (a definition of the Grzegorczyk classes can be found in Odifreddi [20] or Rose [24]). This is in some sense a natural and rather obvious conjecture, but still it will require a good deal of non-trivial technical work to write up a decent proof.

It is well known that $\mathcal{E}_*^3 = \mathcal{E}_*$. Thus, Theorem 6.3 is a special case of our more general conjecture. Moreover, a problem is primitive recursive if and only if it belongs to one of the Grzegorczyk classes $\mathcal{E}_*^3, \mathcal{E}_*^4, \mathcal{E}_*^5, \ldots$. Thus, if our conjecture holds, 2RBS will capture the class of primitive recursive problems.

7. Some remarks and references

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 notori-

ously 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. One might also ask if our characterizations are a hundred percent implicit. Why should they not be so? Well, the execution base is there, and the execution base is given explicitly, isn't it? So to what extent are we dealing with purely implicit characterizations here? These questions may call for a long and involved, and perhaps inconclusive, discussion amongst feinschmeckers. Right now, just let us say that it is beyond the scope of the paper to take that discussion any further. Just let us all implicitly agree that our characterizations are implicit enough to be explicitly characterized as implicit characterizations.

Reversible computation models have been studied for many different purposes and in widely different areas of computer science. Historically, much work has been devoted to what might be characterized as *reversible simulations* of *irreversible computations*. Throughout the literature, the use of words and phrases, like "reverse", "inverse", "reversibility" and "reversible computation", varies significantly. The later years there has been some work of foundation nature which aims at clarifying notions and establishing a more standard terminology, e.g., Axelsen & Glück [2] and Yokoama et al. [26]. Such a terminology is certainly needed. Anyway, it should be clear that we in this paper have used the word "reversible" in a very strict sense and, moreover, that our programming languages are reversible in a very strong sense: Every primitive command has an inverse primitive command, every subcommand of a command has an inverse command, the command $\texttt{C}$ contains exactly the same variables as its inverse $(\texttt{C})^{-1}$, and so on. Indeed, every command $\texttt{C}$ has a natural and obvious inverse $(\texttt{C})^{-1}$, moreover, $((\texttt{C})^{-1})^{-1}$ is syntactically equal to $\texttt{C}$. Following Matos [17], we may call such languages *inherently reversible* languages.

The author is not aware of any work in reversible computing that is closely related to the work presented above, but the work of Matos [17] is at least faintly related. Matos introduces the inherently reversible programming language SRL and makes some informal observations regarding the close relationship between primitive recursive functions and functions implementable in SRL. This relationship is investigated further in recent work Paolini et al. [23] and Matos et al. [18], and a formal characterization the tight relationship between primitive recursion and SRL programs is given in [18]. The work on reversible computing published in Paolini et al. [22] is also faintly related to the work presented above, but this is work of a recursion-theoretic nature which has a different flavor than ours. Still, it is possible that studies along the lines of [22] might lead to interesting characterizations of complexity classes.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] S. Arora, B. Barak, Computational Complexity: A Modern Approach, Cambridge University Press, 2009.
[2] H.B. Axelsen, R. Glück, What do reversible programs compute?, in: M. Hofmann (Ed.), FOSSACS 2011, in: LNCS, vol. 6604, Springer, 2011, pp. 42–56.
[3] S.J. Bellantoni, S. Cook, A new recursion-theoretic characterizations of the polytime functions, Comput. Complex. 2 (1992) 97–110.
[4] U. Dal Lago, A short introduction to implicit computational complexity, in: N. Bezhanishvili, V. Goranko (Eds.), Lectures on Logic and Computation, in: LNCS, vol. 7388, Springer, 2011, pp. 89–109.
[5] N.D. Jones, LOGSPACE and PTIME characterized by programming languages, Theor. Comput. Sci. 228 (1999) 151–174.
[6] N.D. Jones, The expressive power of higher-order types or, life without CONS, J. Funct. Program. 11 (2001) 55–94.
[7] N.D. Jones, Computability and Complexity from a Programming Perspective, The MIT Press, 1997.
[8] C. Kop, J.G. Simonsen, Complexity hierarchies and higher-order Cons-free term rewriting, Log. Methods Comput. Sci. 13 (2017), https://lmcs.episciences.org/3847.
[9] L. Kristiansen, Neat function algebraic characterizations of LOGSPACE and LINSPACE, Comput. Complex. 14 (2005) 72–88.
[10] L. Kristiansen, Higher types, finite domains and resource-bounded Turing machines, J. Log. Comput. 22 (2012) 281–304.
[11] L. Kristiansen, Reversible programming languages capturing complexity classes, in: I. Lanese, M. Rawski (Eds.), Reversible Computation, RC 2020, in: LNCS, vol. 12227, Springer, 2020, pp. 111–127.
[12] L. Kristiansen, K-H. Niggl, On the computational complexity of imperative programming languages, Theor. Comput. Sci. 318 (2004) 139–161.
[13] L. Kristiansen, P.J. Voda, Programming languages capturing complexity classes, Nord. J. Comput. 12 (2005) 89–115.
[14] L. Kristiansen, P.J. Voda, Complexity classes and fragments of C, Inf. Process. Lett. 88 (2003) 213–218.
[15] D. Leivant, A foundational delineation of computational feasibility, in: Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science, IEEE, 1991, pp. 39–47.
[16] D. Leivant, Stratified functional programs and computational complexity, in: POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages, ACM, New York, 1993, pp. 325–333.
[17] A.B. Matos, Linear programs in a simple reversible language, Theor. Comput. Sci. 290 (2003) 2063–2074.
[18] A.B. Matos, L. Paolini, L. Roversi, On the expressivity of total reversible programming languages, in: I. Lanese, M. Rawski (Eds.), Reversible Computation, RC 2020, in: LNCS, vol. 12227, Springer, 2020, pp. 128–143.
[19] A.R. Meyer, D.M. Ritchie, The complexity of loop programs, in: ACM '67: Proceedings of the 1967 22nd National Conference, January 1967, pp. 465–469.
[20] P. Odifreddi, Classical Recursion Theory, vol. II, North Holland, 1999.
[21] R. Pechoux, Implicit computational complexity: past and future, Complexite [cs.CC], Universite de Lorraine, 2020, tel-02978986, https://hal.univ-lorraine.fr/tel-02978986.
[22] L. Paolini, M. Piccolo, L. Roversi, On a class of reversible primitive recursive functions and its Turing-complete extensions, New Gener. Comput. 36 (2018) 233–256.
[23] L. Paolini, M. Piccolo, L. Roversi, A class of recursive permutations which is primitive recursive complete, Theor. Comput. Sci. 813 (2020) 218–233.

[24] H.E. Rose, Subrecursion. Functions and Hierarchies, Oxford Logic Guides, vol. 9, Clarendon Press, 1984.
[25] M. Sipser, Introduction to the Theory of Computation, PWS Publishing Company, 1997.
[26] T. Yokoyama, H.B. Axelsen, R. Glück, Fundamentals of reversible flowchart languages, Theor. Comput. Sci. 611 (2016) 87–115.