Instruction sequences and non-uniform complexity theory

Bergstra, J.A.; Middelburg, C.A.

Publication date
2008

Document Version
Submitted manuscript

Instruction Sequences and Non-uniform Complexity Theory*

J.A. Bergstra and C.A. Middelburg

Programming Research Group, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, the Netherlands
J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl

Abstract. We develop theory concerning non-uniform complexity in a setting in which the notion of single-pass instruction sequence considered in program algebra is the central notion. We define counterparts of the complexity classes P/poly and NP/poly and formulate a counterpart of the complexity theoretic conjecture that NP $\not\subseteq$ P/poly. In addition, we define a notion of completeness for the counterpart of NP/poly using a non-uniform reducibility relation and formulate complexity hypotheses which concern restrictions on the instruction sequences used for computation. We think that the theory developed opens up an additional way of investigating issues concerning non-uniform complexity.

Keywords: single-pass instruction sequence, non-uniform complexity, non-uniform super-polynomial complexity hypothesis, super-polynomial feature elimination complexity.

1998 ACM Computing Classification: F.1.1, F.1.3.

1 Introduction

In this paper, we develop theory about non-uniform complexity in a setting in which the notion of single-pass instruction sequence considered in program algebra is the central notion.

In the first place, we define a counterpart of the classical non-uniform complexity class P/poly and formulate a counterpart of a well-known complexity theoretic conjecture. The conjecture in question is the conjecture that NP $\not\subseteq$ P/poly. Some evidence for this conjecture is the Karp-Lipton theorem [11], which says that the polynomial time hierarchy collapses to the second level if NP $\subseteq$ P/poly. If the conjecture is right, then the conjecture that P $\neq$ NP is right as well. The counterpart of the former conjecture introduced in this paper is called the non-uniform super-polynomial complexity hypothesis. It is called a hypothesis instead of a conjecture because we are primarily interested in its consequences.

Over and above that, we define a counterpart of the non-uniform complexity class NP/poly, introduce a notion of completeness for this complexity class

* This research was partly carried out in the framework of the Jacquard-project Symbiosis, which is funded by the Netherlands Organisation for Scientific Research (NWO).

using a non-uniform reducibility relation, and formulate three complexity hypotheses which concern restrictions on the instruction sequences used for computation. These three hypotheses are called super-polynomial feature elimination complexity hypotheses. The first of them is equivalent to the hypothesis that NP/poly $\not\subseteq$ P/poly. We do not know whether there are equivalent hypotheses for the other two hypotheses in well-known settings such as Turing machines with advice and Boolean circuits. All three hypotheses are intuitively appealing in the setting of single-pass instruction sequences.

We show among other things that P/poly and NP/poly coincide with their counterparts in the setting of single-pass instruction sequences as defined in this paper and that a problem closely related to 3SAT is NP-complete as well as complete for the counterpart of NP/poly. An interesting corollary of the mentioned results is that NP $\not\subseteq$ P/poly is equivalent to NP/poly $\not\subseteq$ P/poly.

With the work presented in this paper, we carry on the line of research with which a start was made in [4]. The working hypothesis of this line of research is that single-pass instruction sequence is a central notion of computer science. The object pursued with this line of research is the development of theory from this working hypothesis. It is clear that by this working hypothesis we leave the lines that have been taken in the development of most existing theory.

Program algebra, which is intended as a setting suited for developing theory from the above-mentioned working hypothesis, is taken for the basis of the development aimed at. Program algebra is not intended to provide a notation for programs that is suited for actual programming. With program algebra we have in view contemplation on programs rather than construction of programs.

The starting-point of program algebra is the perception of a program as a single-pass instruction sequence, i.e. a finite or infinite sequence of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over. This perception is simple, appealing, and links up with practice. The concepts underlying the primitives of program algebra are common in programming, but the particular form of the primitives is not common. The predominant concern in the design of program algebra has been to achieve simple syntax and semantics, while maintaining the expressive power of arbitrary finite control.

A single-pass instruction sequence under execution is considered to produce a behaviour to be controlled by some execution environment. Threads as considered in basic thread algebra model such behaviours: upon each action performed by a thread, a reply from the execution environment determines how the thread proceeds. A thread may make use of services, i.e. components of the execution environment. Each Turing machine can be simulated by means of a thread that makes use of a service. The thread and service correspond to the finite control and tape of the Turing machine. Simulation by means of a thread that makes use of a service is also possible for other machines that have been proposed as a computational model, such as register machines or multi-stack machines.

The threads that correspond to the finite controls of Turing machines are examples of regular threads, i.e. threads that can only be in a finite number of states. The behaviours of all instruction sequences considered in program algebra are regular threads and each regular thread is produced by some instruction sequence. This implies, for instance, that program algebra can be used to program the finite control of any Turing machine.

In our study of non-uniform computational complexity, we are concerned with functions that can be computed by finite instruction sequences whose behaviours make use of services that make up Boolean registers. The instruction sequences considered in program algebra are sufficient to define a counterpart of P/poly, but not to define a counterpart of NP/poly. For a counterpart of NP/poly, we introduce an extension of program algebra that allows for single-pass instruction sequences to split and an extension of basic thread algebra with a behavioural counterpart of instruction sequence splitting that is reminiscent of thread forking.

The approach to complexity followed in this paper is not suited to uniform complexity. This is not considered a great drawback. Non-uniform complexity is the relevant notion of complexity when studying what looks to be the major complexity issue in practice: the scale-dependence of efficient solutions for computational problems.

This paper is organized as follows. First, we review basic thread algebra and program algebra (Sections 2 and 3). Next, we present mechanisms for interaction of threads with services and and give a description of Boolean register services (Sections 4 and 5). Then, we introduce the complexity class corresponding to P/poly and formulate the non-uniform super-polynomial complexity hypothesis (Sections 6 and 7). After that, we present extensions of program algebra and basic thread algebra needed in the subsequent sections (Section 8). Following this, we introduce the complexity class corresponding to NP/poly and formulate the super-polynomial feature elimination complexity hypotheses (Sections 9 and 10). Finally, we make some concluding remarks (Section 11).

Some familiarity with complexity theory is assumed. The definitions of the complexity theoretic notions that are assumed known can be found in many textbooks on computational complexity. We mention [1, 3, 9] as examples of textbooks in which all the notions in question are introduced.

2 Basic Thread Algebra

In this section, we review BTA (Basic Thread Algebra), a form of process algebra which is tailored to the description and analysis of the behaviours of sequential programs under execution. The behaviours concerned are called *threads*.

In BTA, it is assumed that a fixed but arbitrary set $\mathcal{A}$ of *basic actions*, with $\mathsf{tau} \notin \mathcal{A}$, has been given. We write $\mathcal{A}_{\mathsf{tau}}$ for $\mathcal{A} \cup \{\mathsf{tau}\}$. The members of $\mathcal{A}_{\mathsf{tau}}$ are referred to as *actions*.

Threads proceed by performing actions in a sequential fashion. Each basic action performed by a thread is taken as a command to be processed by some service provided by the execution environment of the thread. The processing of a command may involve a change of state of the service concerned. At completion

Table 1. Axiom of BTA

$x \unlhd \mathsf{tau} \unrhd y = x \unlhd \mathsf{tau} \unrhd x$	T1

of the processing of the command, the service produces a reply value. This reply is either T or F and is returned to the thread concerned. Performing the action tau will never lead to a state change and always lead to the reply T, but notwithstanding that its presence matters.

BTA has one sort: the sort $\mathbf{T}$ of *threads*. To build terms of sort $\mathbf{T}$, BTA has the following constants and operators:

- the *deadlock* constant $\mathsf{D} : \mathbf{T}$;
- the *termination* constant $\mathsf{S} : \mathbf{T}$;
- for each $a \in \mathcal{A}_{\mathsf{tau}}$, the binary *postconditional composition* operator $\_ \unlhd a \unrhd \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

Terms of sort $\mathbf{T}$ are built as usual (see e.g. [13, 15]). Throughout the paper, we assume that there is a countably infinite set of variables of sort $\mathbf{T}$ which includes x, y, z.

We use infix notation for postconditional composition. We introduce *action prefixing* as an abbreviation: $a \circ p$, where p is a term of sort $\mathbf{T}$, abbreviates $p \unlhd a \unrhd p$.

Let p and q be closed terms of sort $\mathbf{T}$ and $a \in \mathcal{A}_{\mathsf{tau}}$. Then $p \unlhd a \unrhd q$ will perform action a, and after that proceed as p if the processing of a leads to the reply T (called a positive reply), and proceed as q if the processing of a leads to the reply F (called a negative reply).

BTA has only one axiom. This axiom is given in Table 1. Using the abbreviation introduced above, axiom T1 can be written as follows: $x \unlhd \mathsf{tau} \unrhd y = \mathsf{tau} \circ x$.

Each closed BTA term of sort $\mathbf{T}$ denotes a finite thread, i.e. a thread of which the length of the sequences of actions that it can perform is bounded. Infinite threads can be defined by means of a set of recursion equations (see e.g. [6, 7]). Regular threads, i.e. threads that can only be in a finite number of states, can be defined by means of a finite set of recursion equations.

3 Program Algebra

In this section, we review PGA (ProGram Algebra). The starting-point of PGA is the perception of a program as a single-pass instruction sequence, i.e. a finite or infinite sequence of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over.

In PGA, it is assumed that there is a fixed but arbitrary set $\mathfrak{A}$ of *basic instructions*. PGA has the following *primitive instructions*:

- for each $a \in \mathfrak{A}$, a *plain basic instruction* a;
- for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$;

Table 2. Axioms of PGA

$(x \,;\, y) \,;\, z = x \,;\, (y \,;\, z)$	PGA1
$(x^n)^\omega = x^\omega$	PGA2
$x^\omega \,;\, y = x^\omega$	PGA3
$(x \,;\, y)^\omega = x \,;\, (y \,;\, x)^\omega$	PGA4

– for each $l \in \mathbb{N}$, a *forward jump instruction* $\#l$;
– a *termination instruction* !.

We write $\mathfrak{I}$ for the set of all primitive instructions.

The intuition is that the execution of a basic instruction a may modify a state and produces T or F at its completion. In the case of a positive test instruction $+a$, basic instruction a is executed and execution proceeds with the next primitive instruction if T is produced and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one. In the case where T is produced and there is not at least one subsequent primitive instruction and in the case where F is produced and there are not at least two subsequent primitive instructions, deadlock occurs. In the case of a negative test instruction $-a$, the role of the value produced is reversed. In the case of a plain basic instruction a, the value produced is disregarded: execution always proceeds as if T is produced. The effect of a forward jump instruction $\#l$ is that execution proceeds with the l-th next instruction of the instruction sequence concerned. If l equals 0 or the l-th next instruction does not exist, then $\#l$ results in deadlock. The effect of the termination instruction ! is that execution terminates.

PGA has the following constants and operators:

– for each $u \in \mathfrak{I}$, an *instruction* constant u ;
– the binary *concatenation* operator $\_ \,;\, \_$;
– the unary *repetition* operator $\_^\omega$.

Terms are built as usual. Throughout the paper, we assume that there is a countably infinite set of variables which includes x, y, z.

We use infix notation for concatenation and postfix notation for repetition.

A closed PGA term is considered to denote a non-empty, finite or periodic infinite sequence of primitive instructions.[1] Closed PGA terms are considered equal if they represent the same instruction sequence. The axioms for instruction sequence equivalence are given in Table 2. In this table, n stands for an arbitrary natural number greater than 0. For each $n > 0$, the term x^n is defined by induction on n as follows: $x^1 = x$ and $x^{n+1} = x \,;\, x^n$. The *unfolding* equation $x^\omega = x \,;\, x^\omega$ is derivable. Each closed PGA term is derivably equal to a term in *canonical form*, i.e. a term of the form P or $P \,;\, Q^\omega$, where P and Q are closed PGA terms in which the repetition operator does not occur.

[1] A periodic infinite sequence is an infinite sequence with only finitely many subsequences.

Table 3. Defining equations for thread extraction operation

$\lvert a \rvert = a \circ \mathsf{D}$	$\lvert \#l \rvert = \mathsf{D}$
$\lvert a \,;\, x \rvert = a \circ \lvert x \rvert$	$\lvert \#0 \,;\, x \rvert = \mathsf{D}$
$\lvert +a \rvert = a \circ \mathsf{D}$	$\lvert \#1 \,;\, x \rvert = \lvert x \rvert$
$\lvert +a \,;\, x \rvert = \lvert x \rvert \trianglelefteq a \trianglerighteq \lvert \#2 \,;\, x \rvert$	$\lvert \#l+2 \,;\, u \rvert = \mathsf{D}$
$\lvert -a \rvert = a \circ \mathsf{D}$	$\lvert \#l+2 \,;\, u \,;\, x \rvert = \lvert \#l+1 \,;\, x \rvert$
$\lvert -a \,;\, x \rvert = \lvert \#2 \,;\, x \rvert \trianglelefteq a \trianglerighteq \lvert x \rvert$	$\lvert ! \rvert = \mathsf{S}$
	$\lvert ! \,;\, x \rvert = \mathsf{S}$

The members of the domain of an initial model of PGA are called *instruction sequences*. This is justified by the fact that one of the initial models of PGA is the model in which:

- the domain is the set of all finite and periodic infinite sequences over the set $\mathfrak{I}$ of primitive instructions;
- the operation associated with ; is concatenation;
- the operation associated with $^\omega$ is the operation $^{\underline{\omega}}$ defined as follows:
  - if X is a finite sequence, then $X^{\underline{\omega}}$ is the unique periodic infinite sequence Y such that X concatenated n times with itself is a proper prefix of Y for each $n \in \mathbb{N}$;
  - if X is a periodic infinite sequence, then $X^{\underline{\omega}}$ is X.

To simplify matters, we confine ourselves to this initial model of PGA for the interpretation of PGA terms.

The behaviours of the instruction sequences denoted by closed PGA terms are considered to be regular threads, with the basic instructions taken for basic actions. Moreover, all regular threads in which $\mathtt{tau}$ is absent are behaviours of instruction sequences that can be denoted by closed PGA terms. Closed PGA terms in which the repetition operator does not occur correspond to finite threads.

In the remainder of this paper, we consider instruction sequences that can be denoted by closed PGA terms in which the repetition operator does not occur. The *thread extraction* operation $\lvert \_ \rvert$ defined by the equations given in Table 3 (for $a \in \mathfrak{A}$, $l \in \mathbb{N}$ and $u \in \mathfrak{I}$) gives, for each closed PGA term P in which the repetition operator does not occur, a closed BTA term that denotes the behaviour of the instruction sequence denoted by P.

Henceforth, we will write $\mathrm{PGA}_{\mathrm{fin}}$ for PGA without the repetition operator and axioms PGA2–PGA4, and we will write $\mathcal{IS}_{\mathrm{fin}}$ for the set of all instruction sequences that can be denoted by closed $\mathrm{PGA}_{\mathrm{fin}}$ terms. Moreover, we will write *length*(X), where $X \in \mathcal{IS}_{\mathrm{fin}}$, for the length of X.

The introduction of an algebraic framework for the description and analysis of single-pass instruction sequences might appear somewhat far-fetched: if we consider only finite instruction sequences, only the concatenation operator and its associativity axiom are left. In Appendix A, we show that useful additional axioms can be devised.

The use of a closed $\mathrm{PGA}_{\mathrm{fin}}$ term is sometimes preferable to the use of the corresponding closed BTA term because thread extraction can give rise to a combinatorial explosion. For instance, suppose that p is a closed BTA term such that

$$p = |\overbrace{+a\,;+b\,;\ldots;+a\,;+b}^{k \text{ times}}\,;c\,;!|\ .$$

Then the size of p is greater than $2^{k/2}$. In Appendix B, we show that such combinatorial explosions can be eliminated if we add explicit substitution to thread algebra.

4 Interaction of Threads with Services

A thread may make use of services. That is, a thread may perform an action for the purpose of interacting with a service that takes the action as a command to be processed. The processing of an action may involve a change of state of the service and at completion of the processing of the action the service returns a reply value to the thread. In this section, we introduce the use mechanism and the apply mechanism, which are concerned with this kind of interaction between threads and services. The difference between the use mechanism and the apply mechanism is a matter of perspective: the former is concerned with the effect of services on threads and therefore produces threads, whereas the latter is concerned with the effect of threads on services and therefore produces services.

It is assumed that a fixed but arbitrary set $\mathcal{F}$ of *foci* and a fixed but arbitrary set $\mathcal{M}$ of *methods* have been given. Each focus plays the role of a name of some service provided by an execution environment that can be requested to process a command. Each method plays the role of a command proper. For the set $\mathcal{A}$ of actions, we take the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. Performing an action $f.m$ is taken as making a request to the service named f to process command m.

A *service* H consists of

- a set S of *states*;
- an *effect* function $\mathit{eff} : \mathcal{M} \times S \to S$;
- a *yield* function $\mathit{yld} : \mathcal{M} \times S \to \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$;
- an *initial state* $s_0 \in S$;

satisfying the following condition:

$$\forall m \in \mathcal{M}, s \in S \bullet (\mathit{yld}(m, s) = \mathsf{B} \Rightarrow \forall m' \in \mathcal{M} \bullet \mathit{yld}(m', \mathit{eff}(m, s)) = \mathsf{B})\ .$$

The set S contains the states in which the service may be, and the functions *eff* and *yld* give, for each method m and state s, the state and reply, respectively, that result from processing m in state s.

Given a service $H = (S, \mathit{eff}, \mathit{yld}, s_0)$ and a method $m \in \mathcal{M}$:

- the *derived service* of H after processing m, written $\frac{\partial}{\partial m}H$, is the service $(S, \mathit{eff}, \mathit{yld}, \mathit{eff}(m, s_0))$;
- the *reply* of H after processing m, written $H(m)$, is $\mathit{yld}(m, s_0)$.

A service H can be understood as follows:

- if a thread makes a request to the service to process m and $H(m) \neq \mathsf{B}$, then the request is accepted, the reply is $H(m)$, and the service proceeds as $\frac{\partial}{\partial m}H$;
- if a thread makes a request to the service to process m and $H(m) = \mathsf{B}$, then the request is rejected and the service proceeds as a service that rejects any request.

A service H is called *divergent* if $\frac{\partial}{\partial m}H = H$ and $H(m) = \mathsf{B}$ for all $m \in \mathcal{M}$. The effect of different divergent services on a thread is the same. Therefore, all divergent services are identified.

We introduce the additional sort $\mathbf{S}$ of *services* and the following additional constant and operators:

- the *divergent service* constant $\underline{\mathsf{D}} : \mathbf{S}$;
- for each $f \in \mathcal{F}$, the binary *use* operator $\_ /_f \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$;
- for each $f \in \mathcal{F}$, the binary *apply* operator $\_ \bullet_f \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{S}$.

We use infix notation for the use and apply operators.

$\underline{\mathsf{D}}$ is a fixed but arbitrary divergent service. The operators $\_ /_f \_$ and $\_ \bullet_f \_$ are complementary. Intuitively, $p /_f H$ is the thread that results from processing all actions performed by thread p that are of the form $f.m$ by service H. When an action of the form $f.m$ performed by thread p is processed by service H, that action is turned into the internal action tau and postconditional composition is removed in favour of action prefixing on the basis of the reply value produced. Intuitively, $p \bullet_f H$ is the service that results from processing all basic actions performed by thread p that are of the form $f.m$ by service H. When an action of the form $f.m$ performed by thread p is processed by service H, that service is changed into $\frac{\partial}{\partial m}H$.

The axioms for the use and apply operators are given in Tables 4 and 5. In these tables, f and g stand for arbitrary foci from $\mathcal{F}$, m stands for an arbitrary method from $\mathcal{M}$, and H is a variable of sort $\mathbf{S}$. Axioms TSU3 and TSU4 express that the action tau and actions of the form $g.m$, where $f \neq g$, are not processed. Axioms TSU5 and TSU6 express that a thread is affected by a service as described above when an action of the form $f.m$ performed by the thread is processed by the service. Axiom TSU7 expresses that deadlock takes place when an action to be processed is not accepted. Axiom TSU8 expresses that the divergent service does not accept any action. Axiom TSA3 expresses that a service is not affected by a thread when the action tau is performed by the thread and axiom TSA4 expresses that a service is turned into the divergent service when an action of the form $g.m$, where $f \neq g$, is performed by the thread. Axioms TSA5 and TSA6 express that a service is affected by a thread as described above when

Table 4. Axioms for use operators

$\mathsf{S} /_f H = \mathsf{S}$		TSU1
$\mathsf{D} /_f H = \mathsf{D}$		TSU2
$(\mathsf{tau} \circ x) /_f H = \mathsf{tau} \circ (x /_f H)$		TSU3
$(x \trianglelefteq g.m \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq g.m \trianglerighteq (y /_f H)$	if $f \neq g$	TSU4
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{tau} \circ (x /_f \frac{\partial}{\partial m} H)$	if $H(m) = \mathsf{T}$	TSU5
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{tau} \circ (y /_f \frac{\partial}{\partial m} H)$	if $H(m) = \mathsf{F}$	TSU6
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{D}$	if $H(m) = \mathsf{B}$	TSU7
$(x \trianglelefteq f.m \trianglerighteq y) /_f \underline{\mathsf{D}} = \mathsf{D}$		TSU8

Table 5. Axioms for apply operators

$\mathsf{S} \bullet_f H = H$		TSA1
$\mathsf{D} \bullet_f H = \underline{\mathsf{D}}$		TSA2
$(\mathsf{tau} \circ x) \bullet_f H = x \bullet_f H$		TSA3
$(x \trianglelefteq g.m \trianglerighteq y) \bullet_f H = \underline{\mathsf{D}}$	if $f \neq g$	TSA4
$(x \trianglelefteq f.m \trianglerighteq y) \bullet_f H = x \bullet_f \frac{\partial}{\partial m} H$	if $H(m) = \mathsf{T}$	TSA5
$(x \trianglelefteq f.m \trianglerighteq y) \bullet_f H = y \bullet_f \frac{\partial}{\partial m} H$	if $H(m) = \mathsf{F}$	TSA6
$(x \trianglelefteq f.m \trianglerighteq y) \bullet_f H = \underline{\mathsf{D}}$	if $H(m) = \mathsf{B}$	TSA7
$(x \trianglelefteq f.m \trianglerighteq y) \bullet_f \underline{\mathsf{D}} = \underline{\mathsf{D}}$		TSA8

an action of the form $f.m$ performed by the thread is processed by the service. Axiom TSA7 expresses that a service is turned into the divergent service when an action performed by the thread is not accepted. Axiom TSA8 expresses that the divergent service is not affected by a thread when an action of the form $f.m$ is performed by the thread.

5 Instruction Sequences Acting on Boolean Registers

Our study of computational complexity is concerned with instruction sequences that act on Boolean registers. In this section, we describe services that make up Boolean registers. We also introduce special foci that serve as names of Boolean register services.

The Boolean register services accept the following methods:

- a *set to true method* $\mathsf{set{:}T}$;
- a *set to false method* $\mathsf{set{:}F}$;
- a *get method* get.

We write $\mathcal{M}_{\mathsf{BR}}$ for the set $\{\mathsf{set{:}T}, \mathsf{set{:}F}, \mathsf{get}\}$. It is assumed that $\mathcal{M}_{\mathsf{BR}} \subseteq \mathcal{M}$.

The methods accepted by Boolean register services can be explained as follows:

– $\mathsf{set{:}T}$: the contents of the Boolean register becomes T and the reply is T;
– $\mathsf{set{:}F}$: the contents of the Boolean register becomes F and the reply is F;
– get : nothing changes and the reply is the contents of the Boolean register.

Let $s \in \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$. Then the *Boolean register service* with initial state s, written BR_s, is the service $(\{\mathsf{T}, \mathsf{F}, \mathsf{B}\}, \mathit{eff}, \mathit{eff}, s)$, where the functions eff and yld are defined as follows ($b \in \{\mathsf{T}, \mathsf{F}\}$):

$$\begin{aligned} &\mathit{eff}(\mathsf{set{:}T}, b) = \mathsf{T}\ , && \mathit{eff}(m, b) = \mathsf{B} \quad \text{if } m \notin \mathcal{M}_{\mathsf{BR}}\ , \\ &\mathit{eff}(\mathsf{set{:}F}, b) = \mathsf{F}\ , && \mathit{eff}(m, \mathsf{B}) = \mathsf{B}\ . \\ &\mathit{eff}(\mathsf{get}, b) = b\ , && \end{aligned}$$

The effect and yield functions of a Boolean register service are the same. This means that at completion of the processing of a method the state that results from the processing is returned as the reply.

In the instruction sequences which concern us in the remainder of this paper, a number of Boolean registers is used as input registers, a number of Boolean registers is used as auxiliary registers, and one Boolean register is used as output register.

It is assumed that $\mathsf{in{:}}1, \mathsf{in{:}}2, \ldots \in \mathcal{F}$, $\mathsf{aux{:}}1, \mathsf{aux{:}}2, \ldots \in \mathcal{F}$, and $\mathsf{out} \in \mathcal{F}$. These foci play special roles:

– for each $i \in \mathbb{N}^+$,[2] $\mathsf{in{:}}i$ serves as the name of the Boolean register that is used as i-th input register in instruction sequences;
– for each $i \in \mathbb{N}^+$, $\mathsf{aux{:}}i$ serves as the name of the Boolean register that is used as i-th auxiliary register in instruction sequences;
– out serves as the name of the Boolean register that is used as output register in instruction sequences.

Henceforth, we will write $\mathcal{F}_{\mathsf{in}}$ for $\{\mathsf{in{:}}i \mid i \in \mathbb{N}^+\}$ and $\mathcal{F}_{\mathsf{aux}}$ for $\{\mathsf{aux{:}}i \mid i \in \mathbb{N}^+\}$. Moreover, we will write $\mathcal{IS}_{\mathrm{P}^*}$ for the set of all instruction sequences from $\mathcal{IS}_{\mathrm{fin}}$ in which all primitive instructions, with the exception of jump instructions and the termination instruction, contain only basic instructions from the set

$$\{f.\mathsf{get} \mid f \in \mathcal{F}_{\mathsf{in}} \cup \mathcal{F}_{\mathsf{aux}}\} \cup \{f.\mathsf{set{:}}b \mid f \in \mathcal{F}_{\mathsf{aux}} \cup \{\mathsf{out}\} \wedge b \in \{\mathsf{T}, \mathsf{F}\}\}$$

and $\mathcal{IS}^{\mathrm{na}}_{\mathrm{P}^*}$ for the set of all instruction sequences from $\mathcal{IS}_{\mathrm{fin}}$ in which all primitive instructions, with the exception of jump instructions and the termination instruction, contain only basic instructions from the set

$$\{f.\mathsf{get} \mid f \in \mathcal{F}_{\mathsf{in}}\} \cup \{\mathsf{out.set{:}}b \mid b \in \{\mathsf{T}, \mathsf{F}\}\}\ .$$

$\mathcal{IS}^{\mathrm{na}}_{\mathrm{P}^*}$ is the set of all instruction sequences from $\mathcal{IS}_{\mathrm{P}^*}$ in which no auxiliary registers are used. $\mathcal{IS}_{\mathrm{P}^*}$ is the set of all instruction sequences from $\mathcal{IS}_{\mathrm{fin}}$ that matter to the complexity class P^* which will be introduced in Section 6.

[2] We write $\mathbb{N}^+$ for the set $\{n \in \mathbb{N} \mid n > 0\}$.

6 The Complexity Class P^*

In the field of computational complexity, it is quite common to study the complexity of computing functions on finite strings over a binary alphabet. Since strings over an alphabet of any fixed size can be efficiently encoded as strings over a binary alphabet, it is sufficient to consider only a binary alphabet. We adopt the set $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$ as preferred binary alphabet.

An important special case of functions on finite strings over a binary alphabet is the case where the value of functions is restricted to strings of length 1. Such a function is often identified with the set of strings of which it is the characteristic function. The set in question is usually called a language or a decision problem. The identification mentioned above allows of looking at the problem of computing a function $f : \mathbb{B}^* \to \mathbb{B}$ as the problem of deciding membership of the set $\{w \in \mathbb{B}^* \mid f(w) = \mathsf{T}\}$.

With each function $f : \mathbb{B}^* \to \mathbb{B}$, we can associate an infinite sequence $\langle f_n \rangle_{n \in \mathbb{N}}$ of functions, with $f_n : \mathbb{B}^n \to \mathbb{B}$ for every $n \in \mathbb{N}$, such that f_n is the restriction of f to $\mathbb{B}^n$ for each $n \in \mathbb{N}$. The complexity of computing such sequences of functions, which we call Boolean function families, is studied in the remainder of this paper. In the current section, we introduce the class P^* of all Boolean function families that can be computed by polynomial-length instruction sequences from $\mathcal{IS}_{\mathrm{P}^*}$.

An *n-ary Boolean function* is a function $f : \mathbb{B}^n \to \mathbb{B}$. Let ϕ be a Boolean formula containing the variables $v_1, \ldots, v_n$. Then ϕ induces an n-ary Boolean function f_n such that $f_n(b_1, \ldots, b_n) = \mathsf{T}$ iff ϕ is satisfied by the assignment σ to the variables $v_1, \ldots, v_n$ defined by $\sigma(v_1) = b_1, \ldots, \sigma(v_n) = b_n$. The Boolean function in question is called the Boolean function *induced* by ϕ.

A *Boolean function family* is an infinite sequence $\langle f_n \rangle_{n \in \mathbb{N}}$ of functions, where f_n is an n-ary Boolean function for each $n \in \mathbb{N}$. A Boolean function family $\langle f_n \rangle_{n \in \mathbb{N}}$ can be identified with the unique function $f : \mathbb{B}^* \to \mathbb{B}$ such that for each $n \in \mathbb{N}$, for each $w \in \mathbb{B}^n$, $f(w) = f_n(w)$. In this paper, we are concerned with non-uniform complexity. Considering sets of Boolean function families as complexity classes looks to be most natural when studying non-uniform complexity. We will make the identification mentioned above only where connections with well-known complexity classes are made.

Let $n \in \mathbb{N}$, let $f : \mathbb{B}^n \to \mathbb{B}$, and let $X \in \mathcal{IS}_{\mathrm{P}^*}$. Then X *computes* f if there exists an $l \in \mathbb{N}$ such that for all $b_1, \ldots, b_n \in \mathbb{B}$:

$$
\begin{aligned}
&(\ldots((\ldots(|X| /_{\mathsf{aux}:1} BR_{\mathsf{F}}) \ldots /_{\mathsf{aux}:l} BR_{\mathsf{F}}) /_{\mathsf{in}:1} BR_{b_1}) \ldots /_{\mathsf{in}:n} BR_{b_n}) \bullet_{\mathsf{out}} BR_{\mathsf{F}} \\
&\quad = BR_{f(b_1, \ldots, b_n)} .
\end{aligned}
$$

P^* is the class of all Boolean function families $\langle f_n \rangle_{n \in \mathbb{N}}$ that satisfy:

> there exists a polynomial function $h : \mathbb{N} \to \mathbb{N}$ such that for all $n \in \mathbb{N}$ there exists an $X \in \mathcal{IS}_{\mathrm{P}^*}$ such that X computes f_n and $\mathit{length}(X) \leq h(n)$.

The question arises whether all n-ary Boolean functions can be computed by an instruction sequence from $\mathcal{IS}_{\mathrm{P}^*}$. This question can answered in the affirmative.

Theorem 1. *For each $n \in \mathbb{N}$, for each n-ary Boolean function $f_n : \mathbb{B}^n \to \mathbb{B}$, there exists an $X \in \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ in which no other jump instruction than $\#2$ occurs such that X computes f_n and $\mathit{length}(X) = O(2^n)$.*

Proof. The following is well-known (see e.g. [1], Claim 2.14): for each n-ary Boolean function $f_n : \mathbb{B}^n \to \mathbb{B}$, there is a CNF-formula ϕ containing n variables such that $f_n : \mathbb{B}^n \to \mathbb{B}$ is the Boolean function induced by ϕ and the size of ϕ is $n \cdot 2^n$. Therefore, it is sufficient to show that, for each CNF-formula ϕ containing the variables $v_1, \ldots, v_n$, there exists an $X \in \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ in which no other jump instruction than $\#2$ occurs such that X computes the Boolean function induced by ϕ and $\mathit{length}(X)$ is linear in the size of ϕ.

Let $\mathit{inseq}_{\mathrm{CNF}}$ be the function from the set of all CNF-formulas containing the variables $v_1, \ldots, v_n$ to $\mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ as follows:

$$
\begin{aligned}
&\mathit{inseq}_{\mathrm{CNF}}(\textstyle\bigwedge_{i \in [1,m]} \bigvee_{j \in [1,n_i]} \xi_{ij}) = \\
&\quad \mathit{inseq}'_{\mathrm{CNF}}(\xi_{11}) \mathbin{;} \ldots \mathbin{;} \mathit{inseq}'_{\mathrm{CNF}}(\xi_{1n_1}) \mathbin{;} {+}\mathsf{out.set{:}F} \mathbin{;} \#2 \mathbin{;} ! ; \\
&\qquad \vdots \\
&\quad \mathit{inseq}'_{\mathrm{CNF}}(\xi_{m1}) \mathbin{;} \ldots \mathbin{;} \mathit{inseq}'_{\mathrm{CNF}}(\xi_{mn_m}) \mathbin{;} {+}\mathsf{out.set{:}F} \mathbin{;} \#2 \mathbin{;} ! \mathbin{;} {+}\mathsf{out.set{:}T} \mathbin{;} ! \;,
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{inseq}'_{\mathrm{CNF}}(v_k) &= {+}\mathsf{in{:}}k\mathsf{.get} \mathbin{;} \#2 \;, \\
\mathit{inseq}'_{\mathrm{CNF}}(\neg\, v_k) &= {-}\mathsf{in{:}}k\mathsf{.get} \mathbin{;} \#2 \;.
\end{aligned}
$$

Recall that a disjunction is satisfied if one of its disjuncts is satisfied and a conjunction is satisfied if each of its conjuncts is satisfied. Using these facts, it is easy to prove by induction on the number of clauses in a CNF-formula, and in the basis step by induction on the number of literals in a clause, that no other jump instruction than $\#2$ occurs in $\mathit{inseq}_{\mathrm{CNF}}(\phi)$ and that $\mathit{inseq}_{\mathrm{CNF}}(\phi)$ computes the Boolean function induced by ϕ. Moreover, it is easy to see that $\mathit{length}(\mathit{inseq}_{\mathrm{CNF}}(\phi))$ is linear in the size of ϕ. □

In the proof of Theorem 1, it is shown that the Boolean function induced by a CNF-formula can be computed, without using auxiliary Boolean registers, by an instruction sequence from $\mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ that contains no other jump instructions than $\#2$ and whose length is linear in the size of the formula. If we permit arbitrary jump instructions, this result generalizes from CNF-formulas to arbitrary Boolean formulas in which no other connectives than $\neg$, $\vee$ and $\wedge$ occur.

Theorem 2. *For each Boolean formula ϕ containing no other connectives than $\neg$, $\vee$ and $\wedge$, there exists an $X \in \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ such that X computes the Boolean function induced by ϕ and $\mathit{length}(X)$ is linear in the size of ϕ.*

Proof. Let inseq be the function from the set of all Boolean formulas containing the variables $v_1, \ldots, v_n$ and no other connectives than $\neg$, $\vee$ and $\wedge$ to $\mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ as follows:

$$
\mathit{inseq}(\phi) = \mathit{inseq}'(\phi) \mathbin{;} \#3 \mathbin{;} {+}\mathsf{out.set{:}F} \mathbin{;} \#2 \mathbin{;} {+}\mathsf{out.set{:}T} \mathbin{;} ! \;,
$$

where

$$\begin{aligned}
\mathit{inseq}'(v_k) &= +\mathsf{in}{:}k.\mathsf{get}\ ,\\
\mathit{inseq}'(\neg\ \phi) &= \mathit{inseq}'(\phi)\ ;\#2\ ,\\
\mathit{inseq}'(\phi \lor \psi) &= \mathit{inseq}'(\phi)\ ;\#\mathit{length}(\mathit{inseq}'(\psi)) + 1\ ;\mathit{inseq}'(\psi)\ ,\\
\mathit{inseq}'(\phi \land \psi) &= \mathit{inseq}'(\phi)\ ;\#2\ ;\#\mathit{length}(\mathit{inseq}'(\psi)) + 2\ ;\mathit{inseq}'(\psi)\ .
\end{aligned}$$

Using the same facts about disjunctions and conjunctions as in the proof of Theorem 1, it is easy to prove by induction on the structure of ϕ that $\mathit{inseq}(\phi)$ computes the Boolean function induced by ϕ. Moreover, it is easy to see that $\mathit{length}(\mathit{inseq}(\phi))$ is linear in the size of ϕ. □

The instruction sequences yielded by the function *inseq* used in the proof of Theorem 2 contain long jumps. From the fact that each Boolean formula is logically equivalent to a CNF-formula and the proof of Theorem 1, we draw the conclusion that these long jumps can be eliminated.

Corollary 1. *For each Boolean formula ϕ containing no other connectives than $\neg$, $\lor$ and $\land$, there exists an $X \in \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ in which no other jump instruction than $\#2$ occurs such that X computes the Boolean function induced by ϕ and $\mathit{length}(X)$ is exponential in the size of ϕ.*

We do not know whether the elimination of long jumps really forces an exponential blow-up.

We consider the proof of Theorem 1 once again. The instruction sequences yielded by the function $\mathit{inseq}_{\mathrm{CNF}}$ contain the jump instruction #2. Each occurrence of #2 belongs to a jump chain ending in the instruction sequence $+\mathsf{out.set{:}T}\ ;!$. Therefore, each occurrence of #2 can safely be replaced by the instruction $+\mathsf{out.set{:}F}$, which like #2 skips the next instruction. Moreover, the occurrences of the instruction sequence $+\mathsf{out.set{:}F}\,;\#2\,;!$ can be replaced by the instruction ! because the content of the Boolean register concerned is initially F. The former point gives rise to the following interesting corollary.

Corollary 2. *For each $n \in \mathbb{N}$, for each n-ary Boolean function $f_n : \mathbb{B}^n \to \mathbb{B}$, there exists an $X \in \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ in which jump instructions do not occur such that X computes f_n and $\mathit{length}(X) = O(2^n)$.*

In Corollary 2, the instruction sequences in question contain no jump instructions. However, they contain multiple termination instructions and both $\mathsf{out.set{:}T}$ and $\mathsf{out.set{:}F}$. This raises the question whether further restrictions are possible. We have a negative result.

Theorem 3. *Let ϕ be the Boolean formula $v_1 \land v_2 \land v_3$. Then there does not exist an $X \in \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ in which jump instructions do not occur, multiple termination instructions do not occur and the basic instruction* $\mathsf{out.set{:}F}$ *does not occur such that X computes the Boolean function induced by ϕ.*

Proof. Suppose that $X = u_1\ ; \ldots ;\ u_k$ is an instruction sequence from $\mathcal{IS}^{\mathrm{na}}_{\mathrm{P}*}$ satisfying the restrictions and computing the Boolean function induced by ϕ.

Consider the smallest $l \in [1,k]$ such that u_l is either $\mathsf{out.set{:}T}$, $+\mathsf{out.set{:}T}$ or $-\mathsf{out.set{:}T}$ (there must be such an l). Because ϕ is not satisfied by all assignments to the variables v_1, v_2, v_3, it cannot be the case that $l = 1$. In the case where $l > 1$, for each $i \in [1, l-1]$, u_i is either $\mathsf{in{:}}j\mathsf{.get}$, $+\mathsf{in{:}}j\mathsf{.get}$ or $-\mathsf{in{:}}j\mathsf{.get}$ for some $j \in \{1,2,3\}$. This implies that, for each $i \in [1, l-1]$, there exists a basic Boolean formula ψ_i over the variables v_1, v_2, v_3 that is unique up to logical equivalence such that, for each $b_1, b_2, b_3 \in \mathbb{B}$, if the initial states of the Boolean registers named $\mathsf{in{:}1}$, $\mathsf{in{:}2}$ and $\mathsf{in{:}3}$ are b_1, b_2 and b_3, respectively, then u_{i+1} will be executed iff ψ_i is satisfied by the assignment σ to the variables v_1, v_2, v_3 defined by $\sigma(v_1) = b_1$, $\sigma(v_2) = b_2$ and $\sigma(v_3) = b_2$. It is easy to prove by induction on i that, for each $i \in [1, l-1]$, $\psi_i \Rightarrow \phi$ is not satisfied by any assignment to the variables v_1, v_2, v_3. Hence, X cannot exist. □

Henceforth, a Boolean formula is called a *basic* Boolean formula if it contains no other connectives than $\neg$, $\vee$ and $\wedge$. Moreover, we write $\phi(b_1, \ldots, b_n)$, where ϕ is a Boolean formula containing the variables $v_1, \ldots, v_n$ and $b_1, \ldots, b_n \in \mathbb{B}$, to indicate that ϕ is satisfied by the assignment σ to the variables $v_1, \ldots, v_n$ defined by $\sigma(v_1) = b_1, \ldots, \sigma(v_n) = b_n$.

P^* includes Boolean function families that correspond to uncomputable functions from $\mathbb{B}^*$ to $\mathbb{B}$. Take an undecidable set $N \subseteq \mathbb{N}$ and consider the Boolean function family $\langle f_n \rangle_{n \in \mathbb{N}}$ with, for each $n \in \mathbb{N}$, $f_n : \mathbb{B}^n \to \mathbb{B}$ defined by

$$f_n(b_1, \ldots, b_n) = \mathsf{T} \text{ if } n \in N \;,$$
$$f_n(b_1, \ldots, b_n) = \mathsf{F} \text{ if } n \notin N \;.$$

For each $n \in N$, f_n is computed by the instruction sequence $\mathsf{out.set{:}T} \,;\, !$. For each $n \notin N$, f_n is computed by the instruction sequence $\mathsf{out.set{:}F} \,;\, !$. The length of these instruction sequences is linear in n. Hence, $\langle f_n \rangle_{n \in \mathbb{N}}$ is in P^*. However, the corresponding function $f : \mathbb{B}^* \to \mathbb{B}$ is clearly uncomputable. This reminds of the fact that P/poly includes uncomputable functions from $\mathbb{B}^*$ to $\mathbb{B}$.

It happens that P^* and P/poly coincide, provided that we identify each Boolean function family $\langle f_n \rangle_{n \in \mathbb{N}}$ with the unique function $f : \mathbb{B}^* \to \mathbb{B}$ such that for each $n \in \mathbb{N}$, for each $w \in \mathbb{B}^n$, $f(w) = f_n(w)$.

Theorem 4. $\mathrm{P}^* = \mathrm{P/poly}$.

Proof. We will prove the inclusion $\mathrm{P/poly} \subseteq \mathrm{P}^*$ using the definition of P/poly in terms of Boolean circuits and we will prove the inclusion $\mathrm{P}^* \subseteq \mathrm{P/poly}$ using the definition of P/poly in terms of Turing machines that take advice.

$\mathrm{P/poly} \subseteq \mathrm{P}^*$: Suppose that $\langle f_n \rangle_{n \in \mathbb{N}}$ in P/poly. Then, for all $n \in \mathbb{N}$, there exists a Boolean circuit C such that C computes f_n and the size of C is polynomial in n. For each $n \in \mathbb{N}$, let C_n be such a C. Looking upon basic Boolean formulas as Boolean circuits in which all gates have out-degree 1, it is easy to see that Theorem 2 generalizes from basic Boolean formulas to Boolean circuits: for each Boolean circuit C, there exists an $X \in \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}^*}$ such that X computes the Boolean function computed by C and $\mathit{length}(X)$ is linear in the size of C. From this and the fact that linear in the size of C_n implies polynomial in n, it follows that each Boolean function family in P/poly is also in P^*.

$\mathrm{P}^* \subseteq \mathrm{P/poly}$: Suppose that $\langle f_n \rangle_{n \in \mathbb{N}}$ in P^*. Then, for all $n \in \mathbb{N}$, there exists an $X \in \mathcal{IS}_{\mathrm{P}^{**}}$ such that X computes f_n and $\mathit{length}(X)$ is polynomial in n. For each $n \in \mathbb{N}$, let X_n be such an X. Then f can be computed by a Turing machine that, on an input of size n, takes a binary description of X_n as advice and then just simulates the execution of X_n. It is easy to see that, under the assumption that instructions $\mathsf{aux}{:}i.m$, $+\mathsf{aux}{:}i.m$, $-\mathsf{aux}{:}i.m$ and $\#i$ with $i > \mathit{length}(X_n)$ do not occur in X_n, the size of the description of X_n and the number of steps that it takes to simulate the execution of X_n are both polynomial in n. It is obvious that we can make the assumption without loss of generality. Hence, each Boolean function family in P^* is also in P/poly. □

We do not know whether there are restrictions on the number of auxiliary Boolean registers in the definition of P^* that lead to a class different from P^*. In particular, it is unknown to us whether the restriction to zero auxiliary Boolean registers leads to a class different from P^*.

7 The Non-uniform Super-polynomial Complexity Hypothesis

In this section, we introduce a complexity hypothesis which is a counterpart of the classical complexity theoretic conjecture that $\mathrm{NP} \not\subseteq \mathrm{P/poly}$ in the current setting. The counterpart in question corresponds to the conjecture that $3\mathrm{SAT} \notin \mathrm{P/poly}$. By the NP-completeness of 3SAT, these conjectures are equivalent. If they are right, then the conjecture that $\mathrm{NP} \neq \mathrm{P}$ is right as well. We talk here about a hypothesis instead of a conjecture because we are primarily interested in its consequences.

To formulate the hypothesis, we need a Boolean function family $\langle 3\mathrm{SAT}'_n \rangle_{n \in \mathbb{N}}$ that corresponds to 3SAT. We obtain this Boolean function family by encoding 3CNF-formulas as sequences of Boolean values.

We write $H(k)$ for $\binom{2k}{1} + \binom{2k}{2} + \binom{2k}{3}$. $H(k)$ is the number of combinations of at most 3 elements from a set with $2k$ elements. Notice that $H(k) = (4k^3 + 5k)/3$.

It is assumed that a countably infinite set $\{v_1, v_2, \ldots\}$ of propositional variables and a family of bijections

$$\langle \alpha_k : [1, H(k)] \to \{L \subseteq \{v_1, \neg\, v_1, \ldots, v_k, \neg\, v_k\} \mid 1 \leq \mathrm{card}(L) \leq 3\} \rangle_{k \in \mathbb{N}}$$

have been given.

The basic idea underlying the encoding of 3CNF-formulas as sequences of Boolean values is as follows:

- if $n = H(k)$ for some $k \in \mathbb{N}$, then the input of $3\mathrm{SAT}'_n$ consists of one Boolean value for each disjunction of at most three literals from the set $\{v_1, \neg\, v_1, \ldots, v_k, \neg\, v_k\}$;
- each Boolean value indicates whether the corresponding disjunction occurs in the encoded 3CNF-formula;

– if $H(k) < n < H(k+1)$ for some $k \in \mathbb{N}$, then only the first $H(k)$ Boolean values form part of the encoding.

For each $n \in \mathbb{N}$, $3\mathrm{SAT}'_n : \mathbb{B}^n \to \mathbb{B}$ is defined as follows:

– if $n = H(k)$ for some $k \in \mathbb{N}$:

$$3\mathrm{SAT}'_n\,(b_1,\ldots,b_n) = \mathsf{T} \quad \text{iff} \quad \bigwedge_{i\in[1,n] \text{ s.t. } b_i=\mathsf{T}} \bigvee \alpha_k(i) \ \text{ is satisfiable}\ ,$$

where k is such that $n = H(k)$;

– if $H(k) < n < H(k+1)$ for some $k \in \mathbb{N}$:

$$3\mathrm{SAT}'_n\,(b_1,\ldots,b_n) = 3\mathrm{SAT}'_{H(k)}\left(b_1,\ldots,b_{H(k)}\right)\ ,$$

where k is such that $H(k) < n < H(k+1)$.

$3\mathrm{SAT}'$ is meant to correspond to 3SAT. Therefore, the following theorem does not come as a surprise. Notice that we identify in this theorem the Boolean function family $3\mathrm{SAT}' = \left\langle 3\mathrm{SAT}'_n\right\rangle_{n\in\mathbb{N}}$ with the unique function $3\mathrm{SAT}' : \mathbb{B}^* \to \mathbb{B}$ such that for each $n \in \mathbb{N}$, for each $w \in \mathbb{B}^n$, $3\mathrm{SAT}'(w) = 3\mathrm{SAT}'_n(w)$.

Theorem 5. $3\mathrm{SAT}'$ *is* NP*-complete.*

Proof. It is clear that $3\mathrm{SAT}'$ is in NP: a satisfying assignment for the 3CNF-formula corresponding with the input can serve as a certificate. Thus, it remains to prove that $3\mathrm{SAT}'$ is NP-hard. We will do this by showing that 3SAT is polynomial-time Karp reducible to $3\mathrm{SAT}'$.

For each $k \in \mathbb{N}$, let f_k be a function from the set of all 3CNF-formulas containing the variables $v_1,\ldots,v_k$ to $\mathbb{B}^{H(k)}$ such that $3\mathrm{SAT}'_{H(k)}(f_k(\phi)) = \mathsf{T}$ iff ϕ is satisfiable (the existence of such a function follows immediately from the definition of $3\mathrm{SAT}'$). Take the unique function f from the set of all 3CNF-formulas containing the variables $v_1, v_2, \ldots$ to $\mathbb{B}^*$ such that for each $k \in \mathbb{N}$, for each 3CNF-formula ϕ containing the variables $v_1,\ldots,v_k$, $f(\phi) = f_k(\phi)$. We have that $3\mathrm{SAT}(\phi) = 3\mathrm{SAT}'(f(\phi))$. It remains to show that f is computable in a number of steps that is polynomial in the formula size. The function f restricted to the set of all 3CNF-formulas containing the variables $v_1,\ldots,v_k$, i.e. the function f_k, can be computed in a number of steps that is of order $H(k)\cdot l$, where l is the formula size. The size of 3CNF-formulas containing the variables $v_1,\ldots,v_k$ is not less than k. From these two facts and the fact that $H(k) = (4k^3+5k)/3$, it follows easily that f can be computed in a number of steps that is polynomial in the formula size. □

It is easy to see that the choice of the family of bijections in the definition of $3\mathrm{SAT}'$ is not essential. Let $3\mathrm{SAT}''$ be the same as $3\mathrm{SAT}'$, but based on another family of bijections, say $\langle \alpha'_n\rangle_{n\in\mathbb{N}}$, and let, for each $i \in \mathbb{N}$, for each $j \in [1, H(i)]$, $b'_j = b_{\alpha_i{}^{-1}(\alpha'_i(j))}$. Then:

– if $n = H(k)$ for some $k \in \mathbb{N}$:

$$3\mathrm{SAT}'_n\,(b_1,\ldots,b_n) = 3\mathrm{SAT}''_n\,(b'_1,\ldots,b'_n)\ ;$$

– if $H(k) < n < H(k+1)$ for some $k \in \mathbb{N}$:

$$3\mathrm{SAT}'_n(b_1, \ldots, b_n) = 3\mathrm{SAT}''_n\left(b'_1, \ldots, b'_{H(k)}, b_{H(k)+1}, \ldots, b_n\right) ,$$

where k is such that $H(k) < n < H(k+1)$.

This means that the only effect of another family of bijections is another order of the relevant arguments. Hence, we could have assumed without loss of generality that the family of bijections $\langle \alpha_n \rangle_{n\in\mathbb{N}}$ satisfies the restriction that ${\alpha_i}^{-1}(\alpha_{i+1}(j)) = j$ for all $i \in \mathbb{N}$ and $j \in [1, H(i)]$.

Before we turn to non-uniform super-polynomial complexity hypothesis, we touch lightly on the following property of 3SAT′: it is a Boolean function family for which a family of embedding functions exists.

A *family of embedding function* for a Boolean function family $\langle f_n \rangle_{n\in\mathbb{N}}$ is an infinite sequence $\langle \epsilon_n \rangle_{n\in\mathbb{N}}$ of functions $\epsilon_n : \mathbb{B}^n \to \mathbb{B}^{n+1}$ such that for all $n \in \mathbb{N}$, for all $b_1, \ldots, b_n \in \mathbb{B}$:

$$f_n(b_1, \ldots, b_n) = f_{n+1}(\epsilon_n(b_1, \ldots, b_n)) .$$

The point is that the existence of a family of embedding functions for a Boolean function family $\langle f_n \rangle_{n\in\mathbb{N}}$ indicates the convergence of that Boolean function family to the unique function $f : \mathbb{B}^* \to \mathbb{B}$ such that for each $n \in \mathbb{N}$, for each $w \in \mathbb{B}^n$, $f(w) = f_n(w)$. Moreover, it implies that an instruction sequence computing f_{n+1} can essentially handle all inputs that an instruction sequence computing f_n can handle. All this makes the existence of a family of embedding functions a relevant property of Boolean function families.

We can define a *family of embedding function* for 3SAT′ as follows:

– if $n + 1 = H(k+1)$ for some $k \in \mathbb{N}$:

$$\epsilon_n(b_1, \ldots, b_n) = \left(b'_1, \ldots, b'_{n+1}\right) ,$$

where for each $i \in [1, n+1]$:

$$\begin{array}{ll} b'_i = b_{{\alpha_k}^{-1}(\alpha_{k+1}(i))} & \mathsf{if}\ \alpha_{k+1}(i) \subseteq \{v_1, \neg\, v_1, \ldots, v_k, \neg\, v_k\} , \\ b'_i = \mathsf{F} & \mathsf{if}\ \alpha_{k+1}(i) \not\subseteq \{v_1, \neg\, v_1, \ldots, v_k, \neg\, v_k\} , \end{array}$$

where k is such that $n + 1 = H(k+1)$;

– if $H(k) < n + 1 < H(k+1)$ for some $k \in \mathbb{N}$:

$$\epsilon_n(b_1, \ldots, b_n) = (b_1, \ldots, b_n, \mathsf{F}) .$$

If the above-mentioned restriction on the family of bijections $\langle \alpha_n \rangle_{n\in\mathbb{N}}$ is satisfied, the definition could simply be $\epsilon_n(b_1, \ldots, b_n) = (b_1, \ldots, b_n, \mathsf{F})$.

The *non-uniform super-polynomial complexity hypothesis* is the following hypothesis:

Hypothesis 1. $3\mathrm{SAT}' \notin \mathrm{P}^*$.

$3SAT' \notin P^*$ expresses in short that there does not exist a polynomial function $h{:}\mathbb{N} \to \mathbb{N}$ such that for all $n \in \mathbb{N}$ there exists an $X \in \mathcal{IS}_{P^*}$ such that X computes $3SAT'_n$ and $\mathit{length}(X) \leq h(n)$. This corresponds with the following informal formulation of the non-uniform super-polynomial complexity hypothesis:

> the lengths of the shortest instruction sequences that compute the Boolean functions $3SAT'_n$ are not bounded by a polynomial in n.

The statement that Hypothesis 1 is a counterpart of the conjecture that $3SAT \notin P/poly$ is made rigorous in the following theorem.

Theorem 6. $3SAT' \notin P^*$ *is equivalent to* $3SAT \notin P/poly$.

Proof. This follows immediately from Theorems 4 and 5 and the fact that 3SAT is NP-complete. □

8 Splitting of Instruction Sequences and Threads

The instruction sequences considered in PGA are sufficient to define a counterpart of P/poly, but not to define a counterpart of NP/poly. For a counterpart of NP/poly, we introduce in this section an extension of PGA that allows for single-pass instruction sequences to split. We also introduce an extension of BTA with a behavioural counterpart of instruction sequence splitting that is reminiscent of thread forking. First, we extend PGA with instruction sequence splitting.

It is assumed that a fixed but arbitrary countably infinite set $\mathcal{BP}$ of *Boolean parameters* has been given. Boolean parameters are used to set up a simple form of parameterization for single-pass instruction sequences.

$\mathrm{PGA}_{\mathrm{split}}$ is PGA with built-in basic instructions for instruction sequence splitting. In $\mathrm{PGA}_{\mathrm{split}}$, the following basic instructions belong to $\mathfrak{A}$:

- for each $bp \in \mathcal{BP}$, a *splitting instruction* $\mathsf{split}(bp)$;
- for each $bp \in \mathcal{BP}$, a *direct replying instruction* $\mathsf{reply}(bp)$.

On execution of the instruction sequence $+\mathsf{split}(bp)\,;X$, the primitive instruction $+\mathsf{split}(bp)$ brings about concurrent execution of the instruction sequence X with the Boolean parameter bp instantiated to T and the instruction sequence $\#2\,;X$ with the Boolean parameter bp instantiated to F. The case where $+\mathsf{split}(bp)$ is replaced by $-\mathsf{split}(bp)$ differs in the obvious way, and likewise the case where $+\mathsf{split}(bp)$ is replaced by $\mathsf{split}(bp)$.

On execution of the instruction sequence $+\mathsf{reply}(bp)\ ;X$, the primitive instruction $+\mathsf{reply}(bp)$ brings about execution of the instruction sequence X if the value taken by the Boolean parameter bp is T and execution of the instruction sequence $\#2\,;X$ if the value taken by the Boolean parameter bp is F. The case where $+\mathsf{reply}(bp)$ is replaced by $-\mathsf{reply}(bp)$ differs in the obvious way, and likewise the case where $+\mathsf{reply}(bp)$ is replaced by $\mathsf{reply}(bp)$.

The axioms of $\mathrm{PGA}_{\mathrm{split}}$ are the same as the axioms of PGA. The thread extraction operation for closed $\mathrm{PGA}_{\mathrm{split}}$ terms in which the repetition operator does not occur is defined as for closed PGA terms in which the repetition operator does not occur. However, in the presence of the additional instructions of $\mathrm{PGA}_{\mathrm{split}}$, the intended behaviour of the instruction sequence denoted by a closed term P is not $|P|$. In the notation of the extension of BTA introduced below, the intended behaviour is described by $\|(\langle|P|\rangle)$.

Henceforth, we will write $\mathcal{IS}^{\mathrm{split}}_{\mathrm{fin}}$ for the set of all instruction sequences that can be denoted by closed $\mathrm{PGA}_{\mathrm{split}}$ terms in which the repetition operator does not occur. Moreover, we will write $\mathcal{IS}_{\mathrm{P}**}$ for the set of all instruction sequences from $\mathcal{IS}^{\mathrm{split}}_{\mathrm{fin}}$ in which all primitive instructions, with the exception of jump instructions and the termination instruction, contain only basic instructions from the set

$$\{f.\mathsf{get} \mid f \in \mathcal{F}_{\mathsf{in}}\} \cup \{\mathsf{out.set{:}T}\} \;.$$

In the remainder of this section, we extend BTA with a mechanism for multithreading that supports thread splitting, the behavioural counterpart of instruction sequence splitting. This extension is entirely tailored to the behaviours of the instruction sequences that can be denoted by closed $\mathrm{PGA}_{\mathrm{split}}$ terms.

It is assumed that the collection of threads to be interleaved takes the form of a sequence of threads, called a *thread vector*.

The interleaving of threads is based on the simplest deterministic interleaving strategy treated in [5], namely cyclic interleaving, but any other plausible deterministic interleaving strategy would be appropriate for our purpose.[3] Cyclic interleaving basically operates as follows: at each stage of the interleaving, the first thread in the thread vector gets a turn to perform a basic action and then the thread vector undergoes cyclic permutation. We mean by cyclic permutation of a thread vector that the first thread in the thread vector becomes the last one and all others move one position to the left. If one thread in the thread vector deadlocks, the whole does not deadlock till all others have terminated or deadlocked.

We introduce the additional sort $\mathbf{TV}$ of *thread vectors*. To build terms of sort $\mathbf{T}$, we introduce the following additional operators:

- the unary *cyclic interleaving* operator $\| : \mathbf{TV} \to \mathbf{T}$;
- the unary *deadlock at termination* operator $\mathsf{S_D} : \mathbf{T} \to \mathbf{T}$;
- for each $bp \in \mathcal{BP}$ and $b \in \{\mathsf{T}, \mathsf{F}\}$, the unary *parameter instantiation* operator $\mathsf{I}^{bp}_{b} : \mathbf{T} \to \mathbf{T}$;
- for each $bp \in \mathcal{BP}$, the two binary *postconditional composition* operators $\_ \trianglelefteq \mathsf{split}(bp) \trianglerighteq \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$ and $\_ \trianglelefteq \mathsf{reply}(bp) \trianglerighteq \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

To build terms of sort $\mathbf{TV}$, we introduce the following constants and operators:

- the *empty thread vector* constant $\langle\,\rangle : \mathbf{TV}$;
- the *singleton thread vector* operator $\langle \_ \rangle : \mathbf{T} \to \mathbf{TV}$;

[3] Fairness of the strategy is not an issue because the behaviours of the instruction sequences that can be denoted by closed $\mathrm{PGA}_{\mathrm{split}}$ terms are finite threads. However, deadlock of one thread in the thread vector should not prevent others to proceed.

Table 6. Axioms for cyclic interleaving with thread splitting

$\Vert(\langle\,\rangle) = \mathsf{S}$	CSI1
$\Vert(\langle \mathsf{S}\rangle \frown \alpha) = \Vert(\alpha)$	CSI2
$\Vert(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S_D}(\Vert(\alpha))$	CSI3
$\Vert(\langle \mathsf{tau} \circ x\rangle \frown \alpha) = \mathsf{tau} \circ \Vert(\alpha \frown \langle x\rangle)$	CSI4
$\Vert(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) = \Vert(\alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \Vert(\alpha \frown \langle y\rangle)$	CSI5
$\Vert(\langle x \trianglelefteq \mathsf{split}(bp) \trianglerighteq y\rangle \frown \alpha) = \mathsf{tau} \circ \Vert(\alpha \frown \langle \mathsf{I}^{bp}_{\mathsf{T}}(x)\rangle \frown \langle \mathsf{I}^{bp}_{\mathsf{F}}(y)\rangle)$	CSI6
$\Vert(\langle x \trianglelefteq \mathsf{reply}(bp) \trianglerighteq y\rangle \frown \alpha) = \mathsf{S_D}(\Vert(\alpha))$	CSI7

Table 7. Axioms for deadlock at termination

$\mathsf{S_D}(\mathsf{S}) = \mathsf{D}$	S2D1
$\mathsf{S_D}(\mathsf{D}) = \mathsf{D}$	S2D2
$\mathsf{S_D}(\mathsf{tau} \circ x) = \mathsf{tau} \circ \mathsf{S_D}(x)$	S2D3
$\mathsf{S_D}(x \trianglelefteq a \trianglerighteq y) = \mathsf{S_D}(x) \trianglelefteq a \trianglerighteq \mathsf{S_D}(y)$	S2D4
$\mathsf{S_D}(x \trianglelefteq \mathsf{split}(bp) \trianglerighteq y) = \mathsf{S_D}(x) \trianglelefteq \mathsf{split}(bp) \trianglerighteq \mathsf{S_D}(y)$	S2D5
$\mathsf{S_D}(x \trianglelefteq \mathsf{reply}(bp) \trianglerighteq y) = \mathsf{S_D}(x) \trianglelefteq \mathsf{split}(bp) \trianglerighteq \mathsf{S_D}(y)$	S2D6

– the *thread vector concatenation* operator $\_ \frown \_ : \mathbf{TV} \times \mathbf{TV} \to \mathbf{TV}$.

Throughout the paper, we assume that there are infinitely many variables of sort $\mathbf{TV}$, including α.

For an operational intuition, $\mathsf{split}(bp)$ can be considered a thread splitting action: when encountering $p \trianglelefteq \mathsf{split}(bp) \trianglerighteq q$ at some stage of interleaving, this thread is split into two threads, namely p with the Boolean parameter bp instantiated to T and q with the Boolean parameter bp instantiated to F. For an operational intuition, $\mathsf{reply}(bp)$ can be considered a direct replying action: on performing $\mathsf{reply}(bp)$ the value taken by the Boolean parameter bp is returned as reply value without any further processing.

Intuitively, $\Vert(\alpha)$ is the thread that results from cyclic interleaving of the threads in the thread vector α, covering the above-mentioned splitting of a thread in the thread vector into two threads. This splitting involves instantiation of Boolean parameters in threads. Intuitively, $\mathsf{I}^{bp}_{b}(p)$ is the thread that results from instantiating the Boolean parameter bp to b in thread p. In the event of deadlock of one thread in the thread vector, the whole deadlocks only after all others have terminated or deadlocked. The auxiliary operator $\mathsf{S_D}$ is introduced to describe this fully precise. Intuitively, $\mathsf{S_D}(p)$ is the thread that results from turning termination into deadlock in p.

The axioms for cyclic interleaving with thread splitting, deadlock at termination, and parameter instantiation are given in Tables 6, 7 and 8. In these tables, a stands for an arbitrary action from $\mathcal{A}$. With the exception of CSI7 and BPI6, the axioms simply formalize the informal explanations given above. Axiom CSI7 expresses that deadlock takes place when $\mathsf{reply}(bp)$ ought to be

Table 8. Axioms for parameter instantiation

$\mathsf{I}_b^{bp}(\mathsf{S}) = \mathsf{S}$	BPI1
$\mathsf{I}_b^{bp}(\mathsf{D}) = \mathsf{D}$	BPI2
$\mathsf{I}_b^{bp}(\mathsf{tau} \circ x) = \mathsf{tau} \circ \mathsf{I}_b^{bp}(x)$	BPI3
$\mathsf{I}_b^{bp}(x \trianglelefteq a \trianglerighteq y) = \mathsf{I}_b^{bp}(x) \trianglelefteq a \trianglerighteq \mathsf{I}_b^{bp}(y)$	BPI4
$\mathsf{I}_b^{bp}(x \trianglelefteq \mathsf{split}(bp') \trianglerighteq y) = \mathsf{I}_b^{bp}(x) \trianglelefteq \mathsf{split}(bp') \trianglerighteq \mathsf{I}_b^{bp}(y) \quad \text{if } bp \neq bp'$	BPI5
$\mathsf{I}_b^{bp}(x \trianglelefteq \mathsf{split}(bp) \trianglerighteq y) = \mathsf{D}$	BPI6
$\mathsf{I}_b^{bp}(x \trianglelefteq \mathsf{reply}(bp') \trianglerighteq y) = \mathsf{I}_b^{bp}(x) \trianglelefteq \mathsf{reply}(bp') \trianglerighteq \mathsf{I}_b^{bp}(y) \quad \text{if } bp \neq bp'$	BPI7
$\mathsf{I}_\mathsf{T}^{bp}(x \trianglelefteq \mathsf{reply}(bp) \trianglerighteq y) = \mathsf{tau} \circ \mathsf{I}_\mathsf{T}^{bp}(x)$	BPI8
$\mathsf{I}_\mathsf{F}^{bp}(x \trianglelefteq \mathsf{reply}(bp) \trianglerighteq y) = \mathsf{tau} \circ \mathsf{I}_\mathsf{F}^{bp}(y)$	BPI9

performed next but bp is an uninstantiated Boolean parameter. Axiom BPI6 expresses that deadlock takes place when $\mathsf{split}(bp)$ ought to be performed next but bp is an instantiated Boolean parameter. To be fully precise, we should give axioms concerning the constants and operators to build terms of the sort **TV** as well. We refrain from doing so because the constants and operators concerned are the usual ones for sequences.

To simplify matters, we will henceforth take the set $\{\mathsf{par}{:}i \mid i \in \mathbb{N}^+\}$ for the set $\mathcal{BP}$ of Boolean parameters.

9 The Complexity Class P**

In this section, we introduce the class P^{**} of all Boolean function families that can be computed by polynomial-length instruction sequences from $\mathcal{IS}_{\mathrm{P}^{**}}$.

Let $n \in \mathbb{N}$, let $f : \mathbb{B}^n \to \mathbb{B}$, and let $X \in \mathcal{IS}_{\mathrm{P}^{**}}$. Then X *splitting computes* f if for all $b_1, \ldots, b_n \in \mathbb{B}$:

$$(\ldots(\|(\langle|X|\rangle) /_{\mathsf{in}:1} BR_{b_1}) \ldots /_{\mathsf{in}:n} BR_{b_n}) \bullet_{\mathsf{out}} BR_\mathsf{F} = BR_{f(b_1,\ldots,b_n)} .$$

P^{**} is the class of all Boolean function families $\langle f_n \rangle_{n \in \mathbb{N}}$ that satisfy:

> there exists a polynomial function $h{:}\mathbb{N} \to \mathbb{N}$ such that for all $n \in \mathbb{N}$ there exists an $X \in \mathcal{IS}_{\mathrm{P}^{**}}$ such that X splitting computes f_n and $\mathit{length}(X) \leq h(n)$.

A question that arises is how P^* and P^{**} are related. It happens that P^* is included in P^{**}.

Theorem 7. *$\mathrm{P}^* \subseteq \mathrm{P}^{**}$.*

Proof. Suppose that $\langle f_n \rangle_{n \in \mathbb{N}}$ in P^*. Let $n \in \mathbb{N}$, and let $X \in \mathcal{IS}_{\mathrm{P}^*}$ be such that X computes f_n and $\mathit{length}(X)$ is polynomial in n. Then an $Y \in \mathcal{IS}_{\mathrm{P}^{**}}$ such that Y splitting computes f_n and $\mathit{length}(Y)$ is polynomial in n can be obtained from X as described below.

Suppose that $X = u_1 \,;\ldots;\, u_k$. Let Y be obtained from $u_1 \,;\ldots;\, u_k$ as follows:

1. stop if there exists no $i \in [1,k]$ such that $u_i \equiv \mathsf{aux}{:}j.\mathsf{set}{:}b$ or $u_i \equiv +\mathsf{aux}{:}j.\mathsf{set}{:}b$ or $u_i \equiv -\mathsf{aux}{:}j.\mathsf{set}{:}b$ for some $j \in \mathbb{N}$ and $b \in \mathbb{B}$;
2. find the greatest $i \in [1,k]$ such that $u_i \equiv \mathsf{aux}{:}j.\mathsf{set}{:}b$ or $u_i \equiv +\mathsf{aux}{:}j.\mathsf{set}{:}b$ or $u_i \equiv -\mathsf{aux}{:}j.\mathsf{set}{:}b$ for some $j \in \mathbb{N}$ and $b \in \mathbb{B}$;
3. find the unique $j \in \mathbb{N}$ such that focus $\mathsf{aux}{:}j$ occurs in u_i;
4. find the least $j' \in \mathbb{N}^+$ such that parameter $\mathsf{par}{:}j'$ does not occur in $u_i ; \ldots ; u_k$;
5. if $u_i \equiv \mathsf{aux}{:}j.\mathsf{set}{:}\mathsf{T}$ or $u_i \equiv +\mathsf{aux}{:}j.\mathsf{set}{:}\mathsf{T}$, then
   (a) replace u_i by $-\mathsf{split}(\mathsf{par}{:}j') \,;!$,
   (b) for each $i' \in [1, i-1]$, replace $u_{i'}$ by $\#l+1$ if $u_{i'} \equiv \#l$ and $i'+l \geq i+1$;
6. if $u_i \equiv -\mathsf{aux}{:}j.\mathsf{set}{:}\mathsf{T}$, then
   (a) replace u_i by $-\mathsf{split}(\mathsf{par}{:}j') \,;! \,; \#2$,
   (b) for each $i' \in [1, i-1]$, replace $u_{i'}$ by $\#l+2$ if $u_{i'} \equiv \#l$ and $i'+l \geq i+1$;
7. if $u_i \equiv \mathsf{aux}{:}j.\mathsf{set}{:}\mathsf{F}$ or $u_i \equiv -\mathsf{aux}{:}j.\mathsf{set}{:}\mathsf{F}$, then
   (a) replace u_i by $+\mathsf{split}(\mathsf{par}{:}j') \,;!$,
   (b) for each $i' \in [1, i-1]$, replace $u_{i'}$ by $\#l+1$ if $u_{i'} \equiv \#l$ and $i'+l \geq i+1$;
8. if $u_i \equiv +\mathsf{aux}{:}j.\mathsf{set}{:}\mathsf{F}$, then
   (a) replace u_i by $+\mathsf{split}(\mathsf{par}{:}j') \,;! \,; \#2$,
   (b) for each $i' \in [1, i-1]$, replace $u_{i'}$ by $\#l+2$ if $u_{i'} \equiv \#l$ and $i'+l \geq i+1$;
9. for each $i' \in [i+1, k]$:
   (a) if $u_{i'} \equiv \mathsf{aux}{:}j.\mathsf{get}$, then replace $u_{i'}$ by $\mathsf{reply}(\mathsf{par}{:}j')$,
   (b) if $u_{i'} \equiv +\mathsf{aux}{:}j.\mathsf{get}$, then replace $u_{i'}$ by $+\mathsf{reply}(\mathsf{par}{:}j')$,
   (c) if $u_{i'} \equiv -\mathsf{aux}{:}j.\mathsf{get}$, then replace $u_{i'}$ by $-\mathsf{reply}(\mathsf{par}{:}j')$;
10. repeat the preceding steps for the resulting instruction sequence.

Then it is easy to prove by induction on k that $Y \in \mathcal{IS}_{\mathrm{P}^{**}}$ and Y splitting computes f_n. Moreover, it is easy to see that $\mathit{length}(Y) < 3 \cdot \mathit{length}(X)$. Hence, Y is also polynomial in n. □

The chances are that $\mathrm{P}^{**} \not\subseteq \mathrm{P}^{*}$. In Section 10, we will hypothesize this.

In Section 7, we have hypothesized that $3\mathrm{SAT}' \notin \mathrm{P}^{*}$. The question arises whether $3\mathrm{SAT}' \in \mathrm{P}^{**}$. This question can be answered in the affirmative.

Theorem 8. $3\mathrm{SAT}' \in \mathrm{P}^{**}$.

Proof. Let $n \in \mathbb{N}$, let $k \in \mathbb{N}$ be the unique k such that $H(k) \leq n < H(k{+}1)$, and, for each $b_1, \ldots, b_n \in \mathbb{B}$, let $\phi_{b_1,\ldots,b_n}$ be the 3CNF-formula $\bigwedge_{i \in [1,n] \text{ s.t. } b_i = \mathsf{T}} \bigvee \alpha_k(i)$. We have that $3\mathrm{SAT}'_n(b_1, \ldots, b_n) = \mathsf{T}$ iff $\phi_{b_1,\ldots,b_n}$ is satisfiable. Let ψ be the basic Boolean formula $\bigwedge_{i \in [1,n]} (\neg\, v_{k+i} \lor \bigvee \alpha_k(i))$. We have that $\phi_{b_{k+1},\ldots,b_{k+n}}(b_1, \ldots, b_k)$ iff $\psi(b_1, \ldots, b_{k+n})$. Let $X \in \mathcal{IS}_{\mathrm{P}^*}$ be such that X computes the Boolean function induced by ψ and $\mathit{length}(X)$ is polynomial in n. It follows from Theorem 2 that such an X exists. Assume that instructions $\mathsf{in}{:}i.\mathsf{get}$, $+\mathsf{in}{:}i.\mathsf{get}$, and $-\mathsf{in}{:}i.\mathsf{get}$ with $i > k$ do not occur in X. It is obvious that this assumption can be made without loss of generality. Let $Y \in \mathcal{IS}_{\mathrm{P}^{**}}$ be the instruction sequence obtained from X by replacing, for each $i \in [1,k]$, all occurrences of the primitive instructions $\mathsf{in}{:}i.\mathsf{get}$, $+\mathsf{in}{:}i.\mathsf{get}$, and $-\mathsf{in}{:}i.\mathsf{get}$ by the primitive instructions $\mathsf{reply}(\mathsf{par}{:}i)$, $+\mathsf{reply}(\mathsf{par}{:}i)$, and $-\mathsf{reply}(\mathsf{par}{:}i)$, respectively, and let $Z = \mathsf{split}(\mathsf{par}{:}1) \,; \ldots ; \mathsf{split}(\mathsf{par}{:}k) \,; Y$. We have that $Z \in \mathcal{IS}_{\mathrm{P}^{**}}$, Z splitting computes $3\mathrm{SAT}'_n$, and $\mathit{length}(Z)$ is polynomial in n. Hence, $3\mathrm{SAT}' \in \mathrm{P}^{**}$. □

Below we will define P**-completeness, the counterpart of NP-completeness in the current setting. Like NP-completeness, P**-completeness will be defined in terms of a reducibility relation. Because 3SAT$'$ is closely related to 3SAT and 3SAT$' \in$ P**, we expect 3SAT$'$ to be P**-complete.

Let $l, m, n \in \mathbb{N}$, and let $f : \mathbb{B}^n \to \mathbb{B}$ and $g : \mathbb{B}^m \to \mathbb{B}$. Then f is *length l reducible* to g, written $f \leq^l_{\mathrm{P}^*} g$, if there exist $h_1, \ldots, h_m : \mathbb{B}^n \to \mathbb{B}$ such that:

– there exist $X_1, \ldots, X_m \in \mathcal{IS}_{\mathrm{P}^*}$ such that $X_1, \ldots, X_m$ compute $h_1, \ldots, h_m$ and $\mathit{length}(X_1), \ldots, \mathit{length}(X_m) \leq l$;
– for all $b_1, \ldots, b_n \in \mathbb{B}$, $f(b_1, \ldots, b_n) = g(h_1(b_1, \ldots, b_n), \ldots, h_m(b_1, \ldots, b_n))$.

Let $\langle f_n \rangle_{n \in \mathbb{N}}$ and $\langle g_n \rangle_{n \in \mathbb{N}}$ be Boolean function families. Then $\langle f_n \rangle_{n \in \mathbb{N}}$ is *non-uniform polynomial-length reducible* to $\langle g_n \rangle_{n \in \mathbb{N}}$, written $\langle f_n \rangle_{n \in \mathbb{N}} \leq_{\mathrm{P}^*} \langle g_n \rangle_{n \in \mathbb{N}}$, if there exists a polynomial function $q : \mathbb{N} \to \mathbb{N}$ such that:

– for all $n \in \mathbb{N}$, there exist $l, m \in \mathbb{N}$ with $l, m \leq q(n)$ such that $f_n \leq^l_{\mathrm{P}^*} g_m$.

Let $\langle f_n \rangle_{n \in \mathbb{N}}$ be a Boolean function family. Then $\langle f_n \rangle_{n \in \mathbb{N}}$ is P**-*complete* if:

– $\langle f_n \rangle_{n \in \mathbb{N}} \in \mathrm{P}^{**}$;
– for all $\langle g_n \rangle_{n \in \mathbb{N}} \in \mathrm{P}^{**}$, $\langle g_n \rangle_{n \in \mathbb{N}} \leq_{\mathrm{P}^*} \langle f_n \rangle_{n \in \mathbb{N}}$.

The most important properties of non-uniform polynomial-length reducibility and P**-completeness as defined above are stated in the following two propositions.

Proposition 1.

1. *if* $\langle f_n \rangle_{n \in \mathbb{N}} \leq_{\mathrm{P}^*} \langle g_n \rangle_{n \in \mathbb{N}}$ *and* $\langle g_n \rangle_{n \in \mathbb{N}} \in \mathrm{P}^*$, *then* $\langle f_n \rangle_{n \in \mathbb{N}} \in \mathrm{P}^*$;
2. $\leq_{\mathrm{P}^*}$ *is reflexive and transitive.*

Proposition 2.

1. *if* $\langle f_n \rangle_{n \in \mathbb{N}}$ *is* P**-*complete and* $\langle f_n \rangle_{n \in \mathbb{N}} \in \mathrm{P}^*$, *then* $\mathrm{P}^{**} = \mathrm{P}^*$;
2. *if* $\langle f_n \rangle_{n \in \mathbb{N}}$ *is* P**-*complete,* $\langle g_n \rangle_{n \in \mathbb{N}} \in \mathrm{P}^{**}$ *and* $\langle f_n \rangle_{n \in \mathbb{N}} \leq_{\mathrm{P}^*} \langle g_n \rangle_{n \in \mathbb{N}}$, *then* $\langle g_n \rangle_{n \in \mathbb{N}}$ *is* P**-*complete.*

These properties make P**-completeness as defined above adequate for our purposes. In the following proposition, non-uniform polynomial-length reducibility is related to polynomial-time Karp reducibility ($\leq_{\mathrm{P}}$).

Proposition 3. *Let* $\langle f_n \rangle_{n \in \mathbb{N}}$ *and* $\langle g_n \rangle_{n \in \mathbb{N}}$ *be the Boolean function families, and let* f *and* g *be the unique functions* $f, g : \mathbb{B}^* \to \mathbb{B}$ *such that for each* $n \in \mathbb{N}$, *for each* $w \in \mathbb{B}^n$, $f(w) = f_n(w)$ *and* $g(w) = g_n(w)$. *Then* $f \leq_{\mathrm{P}} g$ *only if* $\langle f_n \rangle_{n \in \mathbb{N}} \leq_{\mathrm{P}^*} \langle g_n \rangle_{n \in \mathbb{N}}$.

The property stated in this proposition allows for results concerning polynomial-time Karp reducibility to be reused in the current setting.

Now we turn to the anticipated P**-completeness of 3SAT$'$.

Theorem 9. 3SAT$'$ *is* P**-*complete.*

Proof. By Theorem 8, we have that $3\mathrm{SAT}' \in \mathrm{P}^{**}$. It remains to prove that for all $\langle f_n \rangle_{n\in\mathbb{N}} \in \mathrm{P}^{**}$, $\langle f_n \rangle_{n\in\mathbb{N}} \leq_{\mathrm{P}^*} 3\mathrm{SAT}'$.

Suppose that $\langle f_n \rangle_{n\in\mathbb{N}} \in \mathrm{P}^{**}$. Let $n \in \mathbb{N}$, and let $X \in \mathcal{IS}_{\mathrm{P}^{**}}$ be such that X splitting computes f_n and $\mathit{length}(X)$ is polynomial in n. Assume that $\mathsf{out.set{:}T}$ occurs only once in X. This assumption can be made without loss of generality: multiple occurrences can always be eliminated through replacement by jump instructions. Suppose that $X = u_1 \,;\, \dots \,;\, u_k$, and let $l \in [1,k]$ be such that u_l is either $\mathsf{out.set{:}T}$, $+\mathsf{out.set{:}T}$ or $-\mathsf{out.set{:}T}$.

We look for a transformation that gives, for each $b_1,\dots,b_n \in \mathbb{B}$, a Boolean formula $\phi_{b_1,\dots,b_n}$ such that $f_n(b_1,\dots,b_n) = \mathsf{T}$ iff $\phi_{b_1,\dots,b_n}$ is satisfiable. Notice that, for fixed initial states of the Boolean registers named $\mathsf{in{:}1},\dots,\mathsf{in{:}}n$, it is possible that there exist several execution paths through X because of the split instructions that may occur in X. We have that $f_n(b_1,\dots,b_n) = \mathsf{T}$ iff there exists an execution path through X that reaches u_l if the initial states of the Boolean registers named $\mathsf{in{:}1},\dots,\mathsf{in{:}}n$ are $b_1,\dots,b_n$, respectively. The existence of such an execution path corresponds to the satisfiability of the Boolean formula $v_1 \wedge v_l \wedge \bigwedge_{i\in[2,k]}(v_i \Leftrightarrow \bigvee_{j\in B(i)} v_j)$, where, for each $i \in [2,k]$, $B(i)$ is the set of all $j \in [1,i]$ for which execution may proceed with u_i after execution of u_j if the initial states of the Boolean registers named $\mathsf{in{:}1},\dots,\mathsf{in{:}}n$ are $b_1,\dots,b_n$, respectively. Let $\phi_{b_1,\dots,b_n}$ be this Boolean formula. Then $f_n(b_1,\dots,b_n) = \mathsf{T}$ iff $\phi_{b_1,\dots,b_n}$ is satisfiable.

For some $m \in \mathbb{N}$, $\phi_{b_1,\dots,b_n}$ still has to be transformed into a $w_{b_1,\dots,b_n} \in \mathbb{B}^m$ such that $\phi_{b_1,\dots,b_n}$ is satisfiable iff $3\mathrm{SAT}'_m(w_{b_1,\dots,b_n}) = \mathsf{T}$. We look upon this transformation as a composition of two transformations: first $\phi_{b_1,\dots,b_n}$ is transformed into a 3CNF-formula $\psi_{b_1,\dots,b_n}$ such that $\phi_{b_1,\dots,b_n}$ is satisfiable iff $\psi_{b_1,\dots,b_n}$ is satisfiable, and next, for some $m \in \mathbb{N}$, $\psi_{b_1,\dots,b_n}$ is transformed into a $w_{b_1,\dots,b_n} \in \mathbb{B}^m$ such that $\psi_{b_1,\dots,b_n}$ is satisfiable iff $3\mathrm{SAT}'_m(w_{b_1,\dots,b_n}) = \mathsf{T}$.

It is easy to see that the size of $\phi_{b_1,\dots,b_n}$ is polynomial in n and that $(b_1,\dots,b_n)$ can be transformed into $\phi_{b_1,\dots,b_n}$ in time polynomial in n. It is well-known that each Boolean formula ψ can be transformed in time polynomial in the size of ψ into a 3CNF-formula ψ', with size and number of variables linear in the size of ψ, such that ψ is satisfiable iff ψ' is satisfiable (see e.g. [3], Theorem 3.7). Moreover, it is known from the proof of Theorem 5 that each 3CNF-formula ϕ can be transformed in time polynomial in the size of ϕ into a $w \in \mathbb{B}^{H(l)}$, where l is the number of variables in ϕ, such that $3\mathrm{SAT}(\phi) = 3\mathrm{SAT}'(w)$. From these facts, and Proposition 3, it follows easily that $\langle f_n \rangle_{n\in\mathbb{N}}$ is non-uniform polynomial-length reducible to $3\mathrm{SAT}'$. ☐

It happens that P^{**} and NP/poly coincide.

Theorem 10. $\mathrm{P}^{**} = \mathrm{NP/poly}$.

Proof. It follows easily from the definitions concerned that $f \in \mathrm{NP/poly}$ iff there exist a $k \in \mathbb{N}$ and a $g \in \mathrm{P/poly}$ such that, for all $w \in \mathbb{B}^*$:

$$f(w) = \mathsf{T} \Leftrightarrow \exists c \in \mathbb{B}^* \bullet |c| \leq |w|^k \wedge g(w,c) = \mathsf{T} .$$

Below, we will refer to such a g as a *checking function* for f. We will first prove the inclusion NP/poly $\subseteq$ P** and then the inclusion P** $\subseteq$ NP/poly.

NP/poly $\subseteq$ P**: Suppose that $f \in$ NP/poly. Then there exists a checking function for f. Let g be a checking function for f, and let $\langle g_n \rangle_{n \in \mathbb{N}}$ be the Boolean function family corresponding to g. Because $g \in$ P/poly, we have by Theorem 4 that $\langle g_n \rangle_{n \in \mathbb{N}} \in$ P*. This implies that, for all $n \in \mathbb{N}$, there exists an $X \in \mathcal{IS}_{\mathrm{P}^*}$ such that X computes g_n and $\mathit{length}(X)$ is polynomial in n. For each $n \in \mathbb{N}$, let X_n be such an X. Moreover, let $\langle f_n \rangle_{n \in \mathbb{N}}$ be the Boolean function family corresponding to f. For each $n \in \mathbb{N}$, there exists an $m \in \mathbb{N}$ such that a $Z \in \mathcal{IS}_{\mathrm{P}^{**}}$ can be obtained from X_m in the way followed in the proof of Theorem 8 such that Z splitting computes f_n and $\mathit{length}(Z)$ is polynomial in n. Hence, each Boolean function family in NP/poly is also in P**.

P** $\subseteq$ NP/poly: Suppose that $\langle f_n \rangle_{n \in \mathbb{N}}$ in P**. Then, for all $n \in \mathbb{N}$, there exists an $X \in \mathcal{IS}_{\mathrm{P}^{**}}$ such that X splitting computes f_n and $\mathit{length}(X)$ is polynomial in n. For each $n \in \mathbb{N}$, let X_n be such an X. Moreover, let $f : \mathbb{B}^* \to \mathbb{B}$ be the function corresponding to $\langle f_n \rangle_{n \in \mathbb{N}}$, and let g be a checking function for f. Then g can also be computed by a Turing machine that, on an input of size n, takes a binary description of X_n as advice and then simulates the execution of X_n treating the additional input as a description of the choices to make at each split. It is easy to see that, under the assumption that instructions $\mathsf{split}(\mathsf{par}{:}i)$, $+\mathsf{split}(\mathsf{par}{:}i)$, $-\mathsf{split}(\mathsf{par}{:}i)$, $\mathsf{reply}(\mathsf{par}{:}i)$, $+\mathsf{reply}(\mathsf{par}{:}i)$, $-\mathsf{reply}(\mathsf{par}{:}i)$ and $\#i$ with $i > \mathit{length}(X_n)$ do not occur in X_n, the size of the description of X_n and the number of steps that it takes to simulate the execution of X_n are both polynomial in n. It is obvious that we can make the assumption without loss of generality. Hence, each Boolean function family in P** is also in NP/poly. $\square$

Theorems 4, 5, 9 and 10 give rise to an interesting corollary.

Corollary 3. NP $\not\subseteq$ P/poly *is equivalent to* NP/poly $\not\subseteq$ P/poly.

10 Super-polynomial Feature Elimination Complexity Hypotheses

In this section, we introduce three complexity hypotheses which concern restrictions on the instruction sequences with which Boolean functions are computed.

By Theorem 7, we have that P* $\subseteq$ P**. We hypothesize that P** $\not\subseteq$ P*. We can think of P* as roughly obtained from P** by restricting the computing instruction sequences to non-splitting instruction sequences. This motivates the formulation of the hypothesis that P** $\not\subseteq$ P* as a feature elimination complexity hypothesis.

The *first super-polynomial feature elimination complexity hypothesis* is the following hypothesis:

Hypothesis 2. *Let $\rho : \mathcal{IS}_{\mathrm{P}^{**}} \to \mathcal{IS}_{\mathrm{P}^*}$ be such that, for each $X \in \mathcal{IS}_{\mathrm{P}^{**}}$, $\rho(X)$ computes the same Boolean function as X. Then $\mathit{length}(\rho(X))$ is not polynomially bounded in $\mathit{length}(X)$.*

We can also think of complexity classes obtained from P^* by restricting the computing instruction sequences further. They can, for instance, be restricted to instruction sequences in which:

– the primitive instructions $f.m$, $+f.m$ and $-f.m$ with $f \in \mathcal{F}_{\mathsf{aux}}$ do not occur;
– for some fixed $k \in \mathbb{N}$, the jump instructions $\#l$ with $l > k$ do not occur;
– the primitive instructions $\mathsf{out.set{:}F}$, $+\mathsf{out.set{:}F}$ and $-\mathsf{out.set{:}F}$ do not occur;
– multiple termination instructions do not occur.

Below we introduce two hypotheses that concern the first two of these restrictions.

The *second super-polynomial feature elimination complexity hypothesis* is the following hypothesis:

Hypothesis 3. *Let $\rho : \mathcal{IS}_{\mathrm{P}^*} \to \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}^*}$ be such that, for each $X \in \mathcal{IS}_{\mathrm{P}^*}$, $\rho(X)$ computes the same Boolean function as X. Then $length(\rho(X))$ is not polynomially bounded in $length(X)$.*

The *third super-polynomial feature elimination complexity hypothesis* is the following hypothesis:

Hypothesis 4. *Let $k \in \mathbb{N}$, and let $\rho : \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}^*} \to \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}^*}$ be such that, for each $X \in \mathcal{IS}^{\mathrm{na}}_{\mathrm{P}^*}$, $\rho(X)$ computes the same Boolean function as X and, for each jump instruction $\#l$ occurring in $\rho(X)$, $l \leq k$. Then $length(\rho(X))$ is not polynomially bounded in $length(X)$.*

These hypotheses motivate the introduction of subclasses of P^*. For each $k, l \in \mathbb{N}$, P^k_l is the class of all Boolean function families $\langle f_n \rangle_{n \in \mathbb{N}}$ that satisfy:

> there exists a polynomial function $h : \mathbb{N} \to \mathbb{N}$ such that for all $n \in \mathbb{N}$ there exists an $X \in \mathcal{IS}_{\mathrm{P}^*}$ such that:
> – X computes f_n and $length(X) \leq h(n)$;
> – instructions $f.m$, $+f.m$ and $-f.m$ with $f = \mathsf{aux}{:}i$ for some $i > k$ do not occur in X;
> – instructions $\#i$ with $i > l$ do not occur in X.

Moreover, for each $k, l \in \mathbb{N}$, P^k_* is the class $\bigcup_{l \in \mathbb{N}} \mathrm{P}^k_l$, and P^*_l is the class $\bigcup_{k \in \mathbb{N}} \mathrm{P}^k_l$.

The hypotheses formulated above, can also be expresses in terms of these subclasses of P^*: Hypotheses 2, 3, and 4 are equivalent to $\mathrm{P}^{**} \not\subseteq \mathrm{P}^*$, $\mathrm{P}^* \not\subseteq \mathrm{P}^0_*$, and $\mathrm{P}^0_* \not\subseteq \mathrm{P}^0_k$ for all $k \in \mathbb{N}$, respectively.

11 Conclusions

We have developed theory concerning non-uniform complexity based on the simple idea that each n-ary Boolean function can be computed by a single-pass instruction sequence that contains only instructions to read and write the contents of Boolean registers, forward jump instructions, and a termination instruction.

We have defined the non-uniform complexity classes P^* and P^{**}, counterparts of the classical non-uniform complexity classes P/poly and NP/poly, and the notion of P^{**}-completeness using a non-uniform reducibility relation. We have shown that P^{*} and P^{**}coincide with P/poly and NP/poly. This makes it clear that there are close connections between non-uniform complexity theory based on single-pass instruction sequences and non-uniform complexity theory based on Turing machines with advice or Boolean circuits. We have also shown that $3\mathrm{SAT}'$, a problem closely related to 3SAT, is both NP-complete and P^{**}-complete.

Moreover, we have formulated a counterpart of the well-known complexity theoretic conjecture that $\mathrm{NP} \not\subseteq \mathrm{P/poly}$ and three complexity hypotheses which concern restrictions on the instruction sequences used for computation. The latter three hypotheses are intuitively appealing in the setting of single-pass instruction sequences. The first of these has a natural counterpart in the setting of Turing machines with advice, but not in the setting of Boolean circuits. The second and third of these appear to have no natural counterparts in the settings of both Turing machines with advice and Boolean circuits.

The approaches to computational complexity based on loop programs [12], straight-line programs [10], and branching programs [8] appear to be the closest related to the approach followed in this paper.

The notion of loop program is far from abstract or general: a loop program consists of assignment statements and possibly nested loop statements of a special kind. To our knowledge, this notion is only used in the work presented in [12]. That work is mainly concerned with upper bounds on the running time of loop programs that can be determined syntactically.

The notion of straight-line program is relatively close to the notion of single-pass instruction sequence: a straight-line program is a sequence of steps, where in each step a language is generated by selecting an element from an alphabet or by taking the union, intersection or concatenation of languages generated in previous steps. In other words, a straight-line program can be looked upon as a single-pass instruction sequence without test and jump instructions, with basic instructions which are rather distant from those usually found. In [10], a complexity measure for straight-line programs is introduced which is closely related to Boolean circuit size. To our knowledge, the notion of straight-line program is only used in the work presented in [10, 2].

The notion of branching program is actually a generalization of the notion of decision tree from trees to graphs, so the term branching program seems rather far-fetched. However, width two branching programs bear a slight resemblance to threads as considered in basic thread algebra. Branching programs are related to non-uniform space complexity like Boolean circuits are related to non-uniform time complexity. Like the notion of Boolean circuit, the notion of branching program looks to be lasting in complexity theory (see e.g. [14]).

A Beyond Instruction Sequence Congruence

Instruction sequence equivalence is a congruence and the axioms of PGA are complete for this congruence. In this appendix, we show that there are interesting coarser congruences for which additional axioms for can be devised.

Table 9. Axioms for structural congruence

$\#n+1 \,;\, u_1 \,;\, \ldots \,;\, u_n \,;\, \#0 = \#0 \,;\, u_1 \,;\, \ldots \,;\, u_n \,;\, \#0$
$\#n+1 \,;\, u_1 \,;\, \ldots \,;\, u_n \,;\, \#m = \#m+n+1 \,;\, u_1 \,;\, \ldots \,;\, u_n \,;\, \#m$

Table 10. Axioms for behavioural congruence

$+f.\mathsf{set}{:}\mathsf{T} = f.\mathsf{set}{:}\mathsf{T}$
$-f.\mathsf{set}{:}\mathsf{F} = f.\mathsf{set}{:}\mathsf{F}$
$-f.\mathsf{set}{:}\mathsf{T} \,;\, f.\mathsf{set}{:}\mathsf{T} = \#1 \,;\, f.\mathsf{set}{:}\mathsf{T}$
$+f.\mathsf{set}{:}\mathsf{F} \,;\, f.\mathsf{set}{:}\mathsf{F} = \#1 \,;\, f.\mathsf{set}{:}\mathsf{F}$
$-f.\mathsf{set}{:}\mathsf{T} \,;\, \#n+2 \,;\, \#n+2 \,;\, u_1 \,;\, \ldots \,;\, u_n \,;\, f.\mathsf{set}{:}\mathsf{T} =$
$\quad \#1 \,;\, \#n+2 \,;\, \#n+2 \,;\, u_1 \,;\, \ldots \,;\, u_n \,;\, f.\mathsf{set}{:}\mathsf{T}$
$+f.\mathsf{set}{:}\mathsf{F} \,;\, \#n+2 \,;\, \#n+2 \,;\, u_1 \,;\, \ldots \,;\, u_n \,;\, f.\mathsf{set}{:}\mathsf{F} =$
$\quad \#1 \,;\, \#n+2 \,;\, \#n+2 \,;\, u_1 \,;\, \ldots \,;\, u_n \,;\, f.\mathsf{set}{:}\mathsf{F}$

It follows from the defining equations of thread extraction that instruction sequences that are the same after removal of chains of forward jumps in favour of single jumps exhibit the same behaviour. Such instruction sequences are called structurally congruent. The additional axioms for structural congruence in the case of $\mathrm{PGA}_{\mathrm{fin}}$ are given in Table 9. In this table, n and m stand for arbitrary numbers from $\mathbb{N}$ and $u_1, \ldots, u_n$ stand for arbitrary primitive instructions from $\mathfrak{I}$.

If we take $\{f.\mathsf{get} \mid f \in \mathcal{F}_{\mathsf{in}} \cup \mathcal{F}_{\mathsf{aux}}\} \cup \{f.\mathsf{set}{:}b \mid f \in \mathcal{F}_{\mathsf{aux}} \cup \{\mathsf{out}\} \land b \in \{\mathsf{T},\mathsf{F}\}\}$ for the set $\mathfrak{A}$ of basic instructions, then certain instruction sequences can be identified because they exhibit the same behaviour on the intended interaction with Boolean register services. Such instruction sequences are called behaviourally congruent. The additional axioms for behavioural congruence in this case are given in Table 10. In this table, f stands for an arbitrary focus from $\mathcal{F}_{\mathsf{aux}} \cup \{\mathsf{out}\}$, n stands for an arbitrary number from $\mathbb{N}$, and $u_1, \ldots, u_n$ stand for arbitrary primitive instructions from $\mathfrak{I}$.

B Explicit Substitution for Linear-size Thread Extraction

In this appendix, we show that the combinatorial explosions mentioned at the end of Section 3 can be eliminated if we add explicit substitution to BTA. We write $\mathcal{V}$ for the countably infinite set of variables of sort $\mathbf{T}$.

The extension of BTA with explicit substitution has the constants and operators of BTA and in addition:

- for each $x \in \mathcal{V}$, the *substitution* operator $[\_/x] : \mathbf{T} \to \mathbf{T}$.

The additional axioms are given in Table 11. In this table, X and Y stand for arbitrary variables from $\mathcal{V}$, p, q and r stand for arbitrary terms of this extension of BTA with explicit substitution, and a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$.

Table 11. Axioms for substitution operators

$[p/X]\,X = p$		ES1
$[p/X]\,Y = Y$	if $X \not\equiv Y$	ES2
$[p/X]\,\mathsf{S} = \mathsf{S}$		ES3
$[p/X]\,\mathsf{D} = \mathsf{D}$		ES4
$[p/X](q \trianglelefteq a \trianglerighteq r) = ([p/X]\,q) \trianglelefteq a \trianglerighteq ([p/X]\,r)$		ES5

The *size* of a term of the extension of BTA with explicit substitution is defined as follows:

$$
\begin{aligned}
&\mathit{size}(X) = 1\;, && \mathit{size}(p \trianglelefteq a \trianglerighteq q) = \mathit{size}(p) + \mathit{size}(q) + 1\;,\\
&\mathit{size}(\mathsf{S}) = 1\;, && \mathit{size}([p/X]\,q) = \mathit{size}(p) + \mathit{size}(q) + 1\;.\\
&\mathit{size}(\mathsf{D}) = 1\;,
\end{aligned}
$$

The following theorem states that linear-size thread extraction is possible if explicit substitution is added to BTA.

Theorem 11. *There exists a function ρ from the set of all closed* $\mathrm{PGA}_{\mathrm{fin}}$ *terms to the set of all closed terms of the extension of* BTA *with explicit substitution such that for all closed* $\mathrm{PGA}_{\mathrm{fin}}$ *terms P, $|P| = \rho(P)$ and $\mathit{size}(\rho(P)) \leq 4 \cdot \mathit{length}(P) + 1$.*

Proof. For each $i \in \mathbb{N}$, let $x_i \in \mathcal{V}$. Take ρ as follows ($k, l > 0$):

$$
\begin{aligned}
&\rho(u_1) = |u_1|\;,\\
&\rho(u_1 \,;\ldots;\, u_{k+1}) = [|u_{k+1}|/x_{k+1}]([\rho'_k(u_k)/x_k]\ldots([\rho'_1(u_1)/x_1]\,x_1)\ldots)\;,
\end{aligned}
$$

where, for each $i \in [1, k]$:

$$
\begin{aligned}
&\rho'_i(!) \;\;\;= \mathsf{S}\;, && \rho'_i(a) \;\;\;= x_{i+1} \trianglelefteq a \trianglerighteq x_{i+1}\;,\\
&\rho'_i(\#0) = \mathsf{D}\;, && \rho'_i(+a) = x_{i+1} \trianglelefteq a \trianglerighteq x_{i+2}\;,\\
&\rho'_i(\#l) \;= x_{i+l}\;, && \rho'_i(-a) = x_{i+2} \trianglelefteq a \trianglerighteq x_{i+1}\;.
\end{aligned}
$$

It is easy to prove by induction on *length*(P) that for all closed $\mathrm{PGA}_{\mathrm{fin}}$ terms P, $|P| = \rho(P)$ and $\mathit{size}(\rho(P)) \leq 4 \cdot \mathit{length}(P) + 1$. $\square$

References

1. Arora, S., Barak, B.: Computational complexity: A modern approach (2007). Draft of a book to appear at Cambridge University Press in 2009. Available from `http://www.cs.princeton.edu/theory/complexity/`
2. Balcázar, J.L., Díaz, J., Gabarró, J.: Uniform characterizations of non-uniform complexity measures. Information and Control **67**(1), 53–69 (1985)
3. Balcázar, J.L., Díaz, J., Gabarró, J.: Structural Complexity I, *EATCS Monographs on Theoretical Computer Science*, vol. 11. Springer-Verlag, Berlin (1988)

4. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming **51**(2), 125–156 (2002)
5. Bergstra, J.A., Middelburg, C.A.: Thread algebra for strategic interleaving. Formal Aspects of Computing **19**(4), 445–474 (2007)
6. Bergstra, J.A., Middelburg, C.A.: A thread algebra with multi-level strategic interleaving. Theory of Computing Systems **41**(1), 3–32 (2007)
7. Bergstra, J.A., Middelburg, C.A.: Distributed strategic interleaving with load balancing. Future Generation Computer Systems **24**(6), 530–548 (2008)
8. Borodin, A., Dolev, D., Fich, F.E., Paul, W.: Bounds for width two branching programs. SIAM Journal of Computing **15**(2), 549–560 (1986)
9. Goldreich, O.: Computational Complexity: A Conceptual Perspective. Cambridge University Press, Cambridge (2008)
10. Goodrich, G.B., Ladner, R.E., Fischer, M.J.: Straight-line programs to compute finite languages. In: Conference on Theoretical Computer Science, Waterloo, pp. 221–229 (1977)
11. Karp, R.M., Lipton, R.J.: Some connections between nonuniform and uniform complexity classes. In: STOC '80, pp. 302–309. ACM Press (1980)
12. Meyer, A.R., Ritchie, D.M.: The complexity of loop programs. In: S. Rosenthal (ed.) 22nd ACM National Conference, pp. 465–469. ACM Press (1967)
13. Sannella, D., Tarlecki, A.: Algebraic preliminaries. In: E. Astesiano, H.J. Kreowski, B. Krieg-Brückner (eds.) Algebraic Foundations of Systems Specification, pp. 13–30. Springer-Verlag, Berlin (1999)
14. Wegener, I.: Branching Programs and Binary Decision Diagrams – Theory and Applications. SIAM Monographs on Discrete Mathematics and Applications. SIAM (2000)
15. Wirsing, M.: Algebraic specification. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 675–788. Elsevier, Amsterdam (1990)