# Program Algebra for
# Random Access Machine Programs

C.A. Middelburg[1]

## Abstract

This paper presents an algebraic theory of instruction sequences with instructions for a random access machine (RAM) as basic instructions, the behaviours produced by the instruction sequences concerned under execution, and the interaction between such behaviours and RAM memories. This theory provides a setting for the development of theory in areas such as computational complexity and analysis of algorithms that distinguishes itself by offering the possibility of equational reasoning to establish whether an instruction sequence computes a given function and being more general than the setting provided by any known version of the RAM model of computation. In this setting, a semi-realistic version of the RAM model of computation and a bit-oriented time complexity measure for this version are introduced. Under the time measure concerned, semi-realistic RAMs can be simulated by multi-tape Turing machines with quadratic time overhead.

**Keywords:** program algebra, thread algebra, random access machine, semi-realistic RAM program, bit-oriented time complexity.

---

[1]Informatics Institute, Faculty of Science, University of Amsterdam, Science Park 900, 1098 XH Amsterdam, the Netherlands, email: `C.A.Middelburg@uva.nl`.

# 1  Introduction

This paper introduces an algebraic theory which provides a setting for the development of theory in areas such as computational complexity and analysis of algorithms that elaborates on a version of the random access machine (RAM) model of computation. The setting in question distinguishes itself by offering the possibility of equational reasoning to establish whether an instruction sequence computes a given function, and by being more general than the setting provided by any known version of the RAM model of computation. Many known and unknown versions of this model of computation can be dealt with by imposing apposite restrictions. We expect that the generality is conducive to the investigation of novel issues in the areas of computational complexity and analysis of algorithms. This expectation is based on our experience with a comparable algebraic theory of instruction sequences, where instructions operate on Boolean registers, in previous work (see [8, 9, 10, 11, 12, 15]).

This paper belongs to a line of research that started with [4], and of which an enumeration is available at [25]. The first objective of this line of research is to understand the concept of a program. The notion of an instruction sequence appears in the work in question as a mathematical abstraction for which the rationale is based on this objective. The structure of the mathematical abstraction at issue has been determined in advance with the hope of applying it in diverse circumstances where in each case the fit may be less than perfect. Until now, this work has, among other things, yielded an approach to computational complexity where program size is used as complexity measure, a contribution to the conceptual analysis of the notion of an algorithm, and new insights into such diverse issues as the halting problem, garbage collection, program parallelization for the purpose of explicit multi-threading, and virus detection.

The basis of all the work in question (see [25]) is the combination of an algebraic theory of single-pass instruction sequences, called program algebra, and an algebraic theory of mathematical objects that represent the behaviours produced by such instruction sequences under execution, called basic thread algebra, extended to deal with the interaction between such behaviours and components of an execution environment for instruction sequences. This combination is parameterized by a set of basic instructions and a set of mathematical objects that represent the execution environment components.

The current paper contains a simplified presentation of the instantiation of this combination in which RAM memories are taken as the components

of an execution environment, instructions for a RAM are taken as basic instructions, and an execution environment consists of only one component. Because we opt for the most general instantiation, all instructions that do not read out or alter more than one register from the RAM memory are taken as basic instructions. Both known and unknown versions of the RAM model of computation can be dealt with by restriction on the set of basic instructions. We expect that by this set-up the presented instantiation can be useful to rigorous investigations of novel issues relating to computational complexity and analysis of algorithms.

Program algebra and basic thread algebra were first presented in [4].[2] The extension of basic thread algebra referred to above, an extension to deal with the interaction between the behaviours produced by instruction sequences under execution and components of an execution environment, was first presented in [6]. The presentation of the extension is rather involved because it is parameterized and owing to this covers a generic set of basic instructions and a generic set of objects that represent execution environment components. In the current paper, a much less involved presentation is obtained by considering only the case where execution environment components are RAM memories, basic instructions are instructions for a RAM, and an execution environment consists of only one component.

After the presentation in question, we make precise in the setting of the presented theory what it means that a given instruction sequence computes a given partial function on bit strings, show that a relatively unknown, but more or less realistic, version of the RAM model of computation can be dealt with in this setting by imposing apposite restrictions, and introduce for this model an alternative to the usual time measures for versions of the RAM model. Under the alternative time measure, RAMs from the version of the RAM model concerned can be simulated by multi-tape Turing machines with quadratic time overhead. Moreover, under a usual space measure for versions of the RAM model, RAMs from this version of the RAM model can be simulated by multi-tape Turing machines with constant-factor space overhead.

With the instruction set of the version of the RAM model of computation dealt with in this paper, a fairly realistic idealization of a real computer is obtained. The introduced alternative to the usual time measures for versions of the RAM model has its origin in the simple idea that the time that it takes

---

[2]In that paper and the first subsequent papers, basic thread algebra was introduced under the name basic polarized process algebra.

to execute an instruction from this instruction set should be based on the number of steps that a multi-tape Turing machine with input alphabet $\{0, 1\}$ needs to simulate the instruction. It is to be expected that its instruction set makes the version of the RAM model dealt with in this paper very practical to the expression and analysis of many algorithms.

This paper is organized as follows. First, a survey is given of program algebra, basic thread algebra, and an extension of their combination that makes precise which behaviours are produced by instruction sequences under execution (Sections 2, 3, and 4). Next, the surveyed theory is instantiated and extended to handle interaction between instruction sequences (with instructions for a RAM) under execution and the memory of a RAM (Sections 5 and 6). Then, in the setting of the resulting theory, it is made precise what it means that a given instruction sequence computes a given partial function (Section 7) and a more or less realistic version of the RAM model of computation is described (Sections 8, 9, and 10). After that, a new time measure and a known space measure for this model of computation are introduced (Sections 11 and 12) and the former measure is discussed (Section 13). Finally, some concluding remarks are made (Section 14).

In this paper, some familiarity with algebraic specification, computability, and computational complexity is assumed. The relevant notions are explained in many handbook chapters and textbooks, e.g. [21, 29, 33] for the relevant notions concerning algebraic specification and [1, 26, 28] for the relevant notions concerning computability and computational complexity.

Sections 2–4, i.e. the preliminary sections of this paper, are largely shortened versions of Sections 2–4 of [13], which, in turn, draw from the preliminary sections of several earlier papers.

## 2   Program Algebra

This section presents a survey of program algebra (PGA). A program is perceived in PGA as a single-pass instruction sequence, i.e. a possibly 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.

It is assumed that a fixed but arbitrary set $\mathcal{A}$ of *basic instructions* has been given. $\mathcal{A}$ is the basis for the set of instructions that may occur in instruction sequences. The intuition is that the execution of a basic instruction may modify a state and must produce the value 0 or 1 as reply at its completion. The produced reply may be state-dependent. In applications

of PGA, the instructions taken as basic instructions vary from instructions relating to Boolean registers to machine language instructions of actual computers.

The set of instructions of which the instruction sequences are composed is the set that consists of the following elements:

- for each $a \in \mathcal{A}$, a *plain basic instruction a*;

- for each $a \in \mathcal{A}$, a *positive test instruction +a*;

- for each $a \in \mathcal{A}$, a *negative test instruction −a*;

- for each $l \in \mathbb{N}$, a *forward jump instruction #l*;

- a *termination instruction* !.

We write $\mathcal{I}$ for this set. The elements from this set are called *primitive instructions*.

On execution of an instruction sequence, the primitive instructions of which it is composed have the following effects:

- the effect of a positive test instruction $+a$ is that basic instruction $a$ is executed and execution proceeds with the next primitive instruction if the produced reply is 1 and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one — inaction occurs if there is no primitive instruction to proceed with;

- the effect of a negative test instruction $-a$ is the same as the effect of $+a$, but with the role of the produced reply reversed;

- the effect of a plain basic instruction $a$ is the same as the effect of $+a$, but execution always proceeds as if the produced reply is 1;

- the effect of a forward jump instruction $\#l$ is that execution proceeds with the $l$th next primitive instruction — inaction occurs if $l$ equals 0 or there is no primitive instruction to proceed with;

- the effect of the termination instruction ! is that execution terminates.

The phrase "inaction occurs" indicates that no more basic instructions are executed, but execution does not terminate.

PGA has one sort: the sort **IS** of *instruction sequences*. To build terms of sort **IS**, PGA has the following constants and operators:

- for each $u \in \mathcal{I}$, the *instruction* constant $u : \to \textbf{IS}$;

- the binary *concatenation* operator $\_ \mathbin{;} \_ : \textbf{IS} \times \textbf{IS} \to \textbf{IS}$;

- the unary *repetition* operator $\_^\omega : \textbf{IS} \to \textbf{IS}$.

Terms of sort **IS** are built as usual in the one-sorted case. We assume that there are infinitely many variables of sort **IS**, including $X, Y, Z$. We use infix notation for concatenation and postfix notation for repetition.

A PGA term in which the repetition operator does not occur is called a *repetition-free* PGA term.

One way of thinking about closed PGA terms is that they represent non-empty, possibly infinite sequences of primitive instructions with finitely many distinct suffixes. Let $t$ and $t'$ be closed PGA terms representing instruction sequences $s$ and $s'$. Then the operators of PGA can be explained as follows:

- $t \mathbin{;} t'$ represents the concatenation of $s$ and $s'$;

- $t^\omega$ represents $s$ concatenated infinitely many times with itself.

The axioms of PGA are given in Table 1. In this table, $u, u_1, \ldots, u_k$ and $v_1, \ldots, v_{k'+1}$ stand for arbitrary primitive instructions from $\mathcal{I}$, $k$, $k'$, and $l$ stand for arbitrary natural numbers from $\mathbb{N}$, and $n$ stands for an arbitrary natural number from $\mathbb{N}_1$.[3] For each $n \in \mathbb{N}_1$, the term $t^n$, where $t$ is a PGA term, is defined by induction on $n$ as follows: $t^1 = t$, and $t^{n+1} = t \mathbin{;} t^n$.

Table 1: Axioms of PGA

| | |
|---|---|
| $(X \mathbin{;} Y) \mathbin{;} Z = X \mathbin{;} (Y \mathbin{;} Z)$ | PGA1 |
| $(X^n)^\omega = X^\omega$ | PGA2 |
| $X^\omega \mathbin{;} Y = X^\omega$ | PGA3 |
| $(X \mathbin{;} Y)^\omega = X \mathbin{;} (Y \mathbin{;} X)^\omega$ | PGA4 |
| | |
| $\#k{+}1 \mathbin{;} u_1 \mathbin{;} \ldots \mathbin{;} u_k \mathbin{;} \#0 = \#0 \mathbin{;} u_1 \mathbin{;} \ldots \mathbin{;} u_k \mathbin{;} \#0$ | PGA5 |
| $\#k{+}1 \mathbin{;} u_1 \mathbin{;} \ldots \mathbin{;} u_k \mathbin{;} \#l = \#l{+}k{+}1 \mathbin{;} u_1 \mathbin{;} \ldots \mathbin{;} u_k \mathbin{;} \#l$ | PGA6 |
| $(\#l{+}k{+}1 \mathbin{;} u_1 \mathbin{;} \ldots \mathbin{;} u_k)^\omega = (\#l \mathbin{;} u_1 \mathbin{;} \ldots \mathbin{;} u_k)^\omega$ | PGA7 |
| $\#l{+}k{+}k'{+}2 \mathbin{;} u_1 \mathbin{;} \ldots \mathbin{;} u_k \mathbin{;} (v_1 \mathbin{;} \ldots \mathbin{;} v_{k'+1})^\omega =$ | |
| $\quad \#l{+}k{+}1 \mathbin{;} u_1 \mathbin{;} \ldots \mathbin{;} u_k \mathbin{;} (v_1 \mathbin{;} \ldots \mathbin{;} v_{k'+1})^\omega$ | PGA8 |

---

[3]We write $\mathbb{N}_1$ for the set $\{n \in \mathbb{N} \mid n \geq 1\}$ of positive natural numbers.

Let $t$ and $t'$ be closed PGA terms. Then $t = t'$ is derivable from the axioms of PGA iff $t$ and $t'$ represent the same instruction sequence after changing all chained jumps into single jumps (which corresponds to applying axioms PGA5 and PGA6) and making all jumps as short as possible (which corresponds to applying axioms PGA7 and PGA8). Moreover, $t = t'$ is derivable from PGA1–PGA4 iff $t$ and $t'$ represent the same instruction sequence.

The informal explanation of closed PGA terms as sequences of primitive instructions given above can be looked upon as a sketch of the intended model of axioms PGA1–PGA4. This model, which is described in detail in, for example, [7], is an initial model of axioms PGA1–PGA4. Henceforth, the instruction sequences of the kind considered in PGA are called PGA instruction sequences.

## 3    Basic Thread Algebra for Finite and Infinite Threads

In this section, we introduce basic thread algebra (BTA) and an extension of BTA that reflects the idea that infinite threads are identical if their approximations up to any finite depth are identical.

BTA is concerned with mathematical objects that model in a direct way the behaviours produced by PGA instruction sequences under execution. The objects in question are called threads. A thread models a behaviour that consists of performing basic actions in a sequential fashion. Upon performing a basic action, a reply from an execution environment determines how the behaviour proceeds subsequently. The possible replies are the values 0 and 1.

The basic instructions from $\mathcal{A}$ are taken as basic actions. Besides, tau is taken as a special basic action. It is assumed that $\mathsf{tau} \notin \mathcal{A}$. We write $\mathcal{A}_{\mathsf{tau}}$ for $\mathcal{A} \cup \{\mathsf{tau}\}$.

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 *inaction* constant $\mathsf{D} :\, \to \mathbf{T}$;

- the *termination* constant $\mathsf{S} :\, \to \mathbf{T}$;

- for each $\alpha \in \mathcal{A}_{\mathsf{tau}}$, the binary *postconditional composition* operator $_{-} \triangleleft \alpha \trianglerighteq {_{-}} : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

Terms of sort $\mathbf{T}$ are built as usual in the one-sorted case. We assume that there are infinitely many variables of sort $\mathbf{T}$, including $x, y, z$. We use infix

notation for postconditional composition. We introduce *basic action prefixing* as an abbreviation: $\alpha \circ t$, where $\alpha \in \mathcal{A}_{\mathsf{tau}}$ and $t$ is a BTA term, abbreviates $t \trianglelefteq \alpha \trianglerighteq t$. We treat an expression of the form $\alpha \circ t$ and the BTA term that it abbreviates as syntactically the same.

Closed BTA terms are considered to represent threads. The constants of BTA can be explained as follows:

- $\mathsf{D}$ represents the thread that models inactive behaviour, i.e. the behaviour that performs no more basic actions and does not terminate either;

- $\mathsf{S}$ represents the thread that models the behaviour that does nothing else but terminate.

Let $t$ and $t'$ be closed BTA terms representing threads $r$ and $r'$. Then the operators of PGA can be explained as follows:

- $t \trianglelefteq \alpha \trianglerighteq t'$ represents the thread that models the behaviour that first performs $\alpha$ and then proceeds as the behaviour modeled by $r$ if the reply from the execution environment is 1 and otherwise proceeds as the behaviour modeled by $r'$.

BTA has only one axiom. This axiom is given in Table 2. It tells us that performing $\mathsf{tau}$, which is considered performing an internal action, always leads to the reply 1.

<div align="center">

Table 2: Axiom of BTA

$x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$   T1

</div>

Each closed BTA term represents a finite thread, i.e. a thread with a finite upper bound to the number of basic actions that it can perform. Infinite threads, i.e. threads without such a finite upper bound, can be defined by means of a set of *recursion equations*, i.e. a set $\{x_i = t_i \mid i \in I\}$, where $I$ is an index set, each $x_i$ is a variable of sort $\mathbf{T}$, each $t_i$ is a BTA term in which only variables from $\{x_i \mid i \in I\}$ occur, and $x_i \neq x_j$ for all $i, j \in I$ with $i \neq j$. A regular thread is a finite or infinite thread that can be defined by means of a finite set of recursion equations. The behaviours produced by PGA instruction sequences under execution are exactly the behaviours modeled by regular threads.

Two infinite threads are considered identical if their approximations up to any finite depth are identical. The approximation up to depth $n$ of a

thread models the behaviour that differs from the behaviour modeled by the thread in that it will become inactive after it has performed $n$ actions unless it would terminate at this point. The approximation induction principle (AIP) is a conditional equation that formalizes the above-mentioned view on infinite threads. In AIP, the approximation up to depth $n$ is phrased in terms of the unary *projection* operator $\pi_n : \mathbf{T} \to \mathbf{T}$.

The axioms for the projection operators and AIP are given in Table 3. In this table, $\alpha$ stands for an arbitrary basic action from $\mathcal{A}_{\mathsf{tau}}$ and $n$ stands for an arbitrary natural number from $\mathbb{N}$. We write $\mathrm{BTA}^\infty$ for BTA extended with the projection operators, the axioms for the projection operators, and AIP.

Table 3: Axioms for the projection operators and AIP

| | |
|---|---|
| $\pi_0(x) = \mathsf{D}$ | PR1 |
| $\pi_{n+1}(\mathsf{D}) = \mathsf{D}$ | PR2 |
| $\pi_{n+1}(\mathsf{S}) = \mathsf{S}$ | PR3 |
| $\pi_{n+1}(x \trianglelefteq \alpha \trianglerighteq y) = \pi_n(x) \trianglelefteq \alpha \trianglerighteq \pi_n(y)$ | PR4 |
| $\bigwedge_{n \geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$ | AIP |

Because we have to deal with conditional equational formulas with a countably infinite number of premises in $\mathrm{BTA}^\infty$, it is understood that infinitary conditional equational logic is used in deriving equations from the axioms of $\mathrm{BTA}^\infty$. A complete inference system for infinitary conditional equational logic can be found in [3, 32, 22].

The depth of a finite thread is the maximum number of basic actions that it can perform before it terminates or becomes inactive. We define the function *depth* that assigns to each closed BTA term the depth of the finite thread that it represents recursively as follows:

$$depth(\mathsf{S}) = 0 \ ,$$
$$depth(\mathsf{D}) = 0 \ ,$$
$$depth(t \trianglelefteq \alpha \trianglerighteq t') = \max\{depth(t), depth(t')\} + 1 \ .$$

## 4   Thread Extraction from Instruction Sequences

In this section, we make precise in the setting of $\mathrm{BTA}^\infty$ which behaviours are produced by PGA instruction sequences under execution.

To make precise which behaviours are produced by PGA instruction sequences under execution, we introduce an operator $|_-|$. For each closed PGA term $t$, $|t|$ represents the thread that models the behaviour produced by the instruction sequence represented by $t$ under execution.

Formally, we combine PGA with $\text{BTA}^\infty$ and extend the combination with the *thread extraction* operator $|_-| : \textbf{IS} \rightarrow \textbf{T}$ and the axioms given in Table 4. In this table, $a$ stands for an arbitrary basic instruction from $\mathcal{A}$, $u$ stands for an arbitrary primitive instruction from $\mathcal{I}$, and $l$ stands for an arbitrary natural number from $\mathbb{N}$. We write $\text{PGA}/\text{BTA}^\infty$ for the combination of PGA and $\text{BTA}^\infty$ extended with the thread extraction operator and the axioms for the thread extraction operator.

Table 4: Axioms for the thread extraction operator

| | | | |
|---|---|---|---|
| $\|a\| = a \circ \mathsf{D}$ | TE1 | $\|\#l\| = \mathsf{D}$ | TE7 |
| $\|a\,;X\| = a \circ \|X\|$ | TE2 | $\|\#0\,;X\| = \mathsf{D}$ | TE8 |
| $\|+a\| = a \circ \mathsf{D}$ | TE3 | $\|\#1\,;X\| = \|X\|$ | TE9 |
| $\|+a\,;X\| = \|X\| \trianglelefteq a \trianglerighteq \|\#2\,;X\|$ | TE4 | $\|\#l+2\,;u\| = \mathsf{D}$ | TE10 |
| $\|-a\| = a \circ \mathsf{D}$ | TE5 | $\|\#l+2\,;u\,;X\| = \|\#l+1\,;X\|$ | TE11 |
| $\|-a\,;X\| = \|\#2\,;X\| \trianglelefteq a \trianglerighteq \|X\|$ | TE6 | $\|!\| = \mathsf{S}$ | TE12 |
| | | $\|!\,;X\| = \mathsf{S}$ | TE13 |

As mentioned in Section 2, on execution of an instruction sequence, inaction occurs if there is no primitive instruction to proceed with. That is why $\mathsf{D}$ occurs in axioms TE1, TE3, TE5, TE7, and TE10.

If a closed PGA term $t$ represents an infinite instruction sequence, then we can extract the approximations of the thread modeling the behaviour produced by that instruction sequence under execution up to every finite depth: for each $n \in \mathbb{N}$, there exists a closed BTA term $t''$ such that $\pi_n(|t|) = t''$ is derivable from axioms PGA1–PGA8, PR1–PR4, and TE1–TE13. If closed PGA terms $t$ and $t'$ represent infinite instruction sequences that produce the same behaviour under execution, then this can be proved using the following instance of AIP: $\bigwedge_{n\geq 0} \pi_n(|t|) = \pi_n(|t'|) \;\Rightarrow\; |t| = |t'|$.

If a closed PGA term $t$ represents an instruction sequence that starts with an infinite chain of forward jumps, then TE9 and TE11 can be applied to $|t|$ infinitely often without ever showing that a basic action is performed. In this case, we have to do with inaction and, being consistent with that, $|t| = \mathsf{D}$ is derivable from axioms PGA1–PGA8, PR1–PR4, AIP, and TE1–TE13. By contrast, $|t| = \mathsf{D}$ is not derivable from axioms PGA1–PGA4,

PR1–PR4, AIP, and TE1–TE13. However, if closed PGA terms $t$ and $t'$ represent instruction sequences in which no infinite chains of forward jumps occur, then $t = t'$ is derivable from the axioms of PGA only if $|t| = |t'|$ is derivable from PGA1–PGA4, PR1–PR4, AIP, and TE1–TE13.

The following proposition, proved in [7], puts the expressiveness of PGA in terms of producible behaviours.

**Proposition 1** *Let $\mathcal{M}$ be a model of* $\mathrm{PGA/BTA}^\infty$. *Then, for each element $r$ from the domain associated with the sort* $\mathbf{T}$ *in $\mathcal{M}$, there exists a closed* PGA *term $t$ such that the interpretation of $|t|$ in $\mathcal{M}$ is $r$ iff $r$ can be defined by means of a finite set of recursion equations.*

PGA instruction sequences are behaviourally equivalent if they produce the same behaviour under execution. Instruction sequences are behaviourally congruent if they produce the same behaviour irrespective of the way they are entered and the way they are left during execution (see also [4, 7]).

## 5    Basic Instructions for Random Access Machines

PGA instruction sequences under execution may interact with components of their execution environment. The execution environment components vary from one application of PGA to another. In this section, we consider basic instruction for the case where the execution environment components are memories of RAMs.

The memory of a RAM consists of a countably infinite number of registers which are numbered by natural numbers. Each register is capable of containing a bit string of arbitrary length. The contents of the registers constitute the state of the memory.

A *RAM memory state* is a function $\sigma : \mathbb{N} \to \{0,1\}^*$ that satisfies the condition that there exists a $i \in \mathbb{N}$ such that, for all $j \in \mathbb{N}$, $\sigma(i+j) = \epsilon$.[4] We write $\Sigma_{\mathrm{rm}}$ for the set of all RAM memory states, and we write $\sigma_\epsilon$ for the unique $\sigma \in \Sigma_{\mathrm{rm}}$ such that $\sigma(i) = \epsilon$ for all $i \in \mathbb{N}$.

Let $\sigma$ be a RAM memory state. Then, for all $i \in \mathbb{N}$, $\sigma(i)$ is the content of the register with number $i$ in memory state $\sigma$. The condition expresses that the part of the memory that is actually in use remains finite.

Henceforth, we will use the notation $(\sigma : i_1 \mapsto w_1, \ldots, i_n \mapsto w_n)$. For each $\sigma : \mathbb{N} \to \{0,1\}^*$, $i_1, \ldots, i_n \in \mathbb{N}$, and $w_1, \ldots, w_n \in \{0,1\}^*$, $(\sigma : i_1 \mapsto w_1, \ldots, i_n \mapsto w_n)$

---

[4]We write $\epsilon$ for the empty bit string.

is the function $\sigma':\mathbb{N}\to\{0,1\}^*$ defined as follows: $\sigma'(i_1) = w_1, \ldots, \sigma'(i_n) = w_n$, and, for all $j \in \mathbb{N}$ with $j \notin \{i_1,\ldots,i_n\}$, $\sigma'(j) = \sigma(j)$.

The execution of an instruction by a RAM may change the memory state of the RAM and must produce the value 0 or 1 as reply.

The set of basic instructions used in this case consists of a *basic RAM instruction* $p/q$ for each $p : \Sigma_{\mathrm{rm}} \to \{0,1\}$ and $q : \Sigma_{\mathrm{rm}} \to \Sigma_{\mathrm{rm}}$ that satisfy the following conditions (which are explained below) for all $\sigma \in \Sigma_{\mathrm{rm}}$:

(a) there exists at most one $i \in \mathbb{N}$ for which there exists a $w \in \{0,1\}^*$ such that $\sigma(i) \neq w$ and $p(\sigma) \neq p((\sigma : i \mapsto w))$,

(b) there exists at most one $i \in \mathbb{N}$ for which $\sigma(i) \neq q(\sigma)(i)$,

(c) if there exists an $i \in \mathbb{N}$ for which there exists a $w \in \{0,1\}^*$ such that $\sigma(i) \neq w$ and $p(\sigma) \neq p((\sigma : i \mapsto w))$ and there exists an $i \in \mathbb{N}$ for which $\sigma(i) \neq q(\sigma)(i)$, then there exists an $i \in \mathbb{N}$ for which there exists a $w \in \{0,1\}^*$ such that $\sigma(i) \neq w$ and $p(\sigma) \neq p((\sigma : i \mapsto w))$ and $\sigma(i) \neq q(\sigma)(i)$.

We write $\mathcal{A}_{\mathrm{ram}}$ for this set.

Each basic RAM instruction leads to carrying out an operation on a RAM memory when the instruction is executed. The intuition is basically that carrying out the operation modifies the content of a single register of the RAM memory and produces the value 0 or 1 as reply depending on the content of this register. More precisely, the execution of a basic RAM instruction $p/q$ has the following effects:

- if the RAM memory state is $\sigma$ when the execution of $p/q$ starts, then the reply produced on termination of the execution of $p/q$ is $p(\sigma)$;

- if the RAM memory state is $\sigma$ when the execution of $p/q$ starts, then the RAM memory state is $q(\sigma)$ when the execution of $p/q$ terminates.

Condition (a) expresses that a basic RAM instruction does not produce a reply that depends on the content of more than one register. Condition (b) expresses that a basic RAM instruction does not modify the content of more than one register. Condition (c) expresses that a basic RAM instruction produces a reply that depends on the content of a register and modifies the content of a register only if the former register is the same as the latter register.

A function from $\Sigma_{\mathrm{rm}}$ to $\{0,1\}$ for which condition (a) trivially holds is the function 1 defined by $1(\sigma) = 1$. A function from $\Sigma_{\mathrm{rm}}$ to $\Sigma_{\mathrm{rm}}$ for

which condition (b) trivially holds is the function $i$ defined by $i(\sigma) = \sigma$. From Section 9, only basic RAM instruction of the forms $1/q$ and $p/i$ are considered.

We write $[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{ram}})$ for $\mathrm{PGA/BTA}^\infty$ with $\mathcal{A}$ instantiated by $\mathcal{A}_{\mathrm{ram}}$.

# 6    Interaction of Threads with RAM Memories

If instructions from $\mathcal{A}_{\mathrm{ram}}$ are taken as basic instructions, a PGA instruction sequence under execution may interact with the memory of a RAM. In line with this kind of interaction, a thread may perform a basic action basically for the purpose of changing the memory state of a RAM or receiving a reply that depends on the memory state of a RAM. In this section, we introduce related constants and operators.

We extend $\mathrm{PGA/BTA}^\infty(\mathcal{A}_{\mathrm{ram}})$ with the sort **RM** of *RAM memories*, the following operators:

- for each $\sigma \in \Sigma_{\mathrm{rm}} \cup \{*\}$, the *RAM memory* constant $\mathsf{rm}(\sigma) : \to \mathbf{RM}$;

- the binary *use* operator $\_ / \_ : \mathbf{T} \times \mathbf{RM} \to \mathbf{T}$;

- the binary *apply* operator $\_ \bullet \_ : \mathbf{T} \times \mathbf{RM} \to \mathbf{RM}$;

and the axioms given in Tables 5.[5] In these tables, $p$ stands for an arbitrary function from $\Sigma_{\mathrm{rm}}$ to $\{0, 1\}$, $q$ stands for an arbitrary function from $\Sigma_{\mathrm{rm}}$ to $\Sigma_{\mathrm{rm}}$, $\sigma$ stands for an arbitrary RAM memory state from $\Sigma_{\mathrm{rm}}$, $n$ stands for an arbitrary natural number from $\mathbb{N}$, and $t$ and $t'$ stand for arbitrary terms of sort **RM**. Moreover, $u$ is assumed to be a variable of sort **RM**. We use infix notation for the use and apply operators. We write $\mathrm{PGA/BTA}^\infty/\mathrm{RAM}$ for $[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{ram}})$ extended with the sort **RM**, the RAM memory constants, the use operator, the apply operator, and the axioms for these operators.

Axioms U1–U6 and A1–A6 formalize the informal explanation of the use operator and the apply operator given below and in addition stipulate what is the result of use and apply if an inoperative RAM memory is involved (U6 and A6). Axioms U7 and A7 allow for reasoning about infinite threads, and therefore about the behaviour produced by infinite instruction sequences under execution, in the context of use and apply, respectively.

---

[5]We write $t[t'/x]$ for the result of substituting term $t'$ for variable $x$ in term $t$.

Table 5: Axioms for the use and apply operator

| | | |
|---|---|---|
| $\mathsf{S} \mathbin{/} u = \mathsf{S}$ | | U1 |
| $\mathsf{D} \mathbin{/} u = \mathsf{D}$ | | U2 |
| $(\mathsf{tau} \circ x) \mathbin{/} u = \mathsf{tau} \circ (x \mathbin{/} u)$ | | U3 |
| $(x \trianglelefteq p/q \trianglerighteq y) \mathbin{/} \mathsf{rm}(\sigma) = \mathsf{tau} \circ (x \mathbin{/} \mathsf{rm}(q(\sigma)))$ | if $p(\sigma) = 1$ | U4 |
| $(x \trianglelefteq p/q \trianglerighteq y) \mathbin{/} \mathsf{rm}(\sigma) = \mathsf{tau} \circ (y \mathbin{/} \mathsf{rm}(q(\sigma)))$ | if $p(\sigma) = 0$ | U5 |
| $(x \trianglelefteq p/q \trianglerighteq y) \mathbin{/} \mathsf{rm}(*) = \mathsf{tau} \circ \mathsf{D}$ | | U6 |
| $\pi_n(x \mathbin{/} u) = \pi_n(x) \mathbin{/} u$ | | U7 |
| $\mathsf{S} \bullet u = u$ | | A1 |
| $\mathsf{D} \bullet u = \mathsf{rm}(*)$ | | A2 |
| $(\mathsf{tau} \circ x) \bullet u = \mathsf{tau} \circ (x \bullet u)$ | | A3 |
| $(x \trianglelefteq p/q \trianglerighteq y) \bullet \mathsf{rm}(\sigma) = x \bullet \mathsf{rm}(q(\sigma))$ | if $p(\sigma) = 1$ | A4 |
| $(x \trianglelefteq p/q \trianglerighteq y) \bullet \mathsf{rm}(\sigma) = y \bullet \mathsf{rm}(q(\sigma))$ | if $p(\sigma) = 0$ | A5 |
| $(x \trianglelefteq p/q \trianglerighteq y) \bullet \mathsf{rm}(*) = \mathsf{rm}(*)$ | | A6 |
| $\bigwedge_{k \geq n} t[\pi_k(x)/z] = t'[\pi_k(y)/z] \;\Rightarrow\; t[x/z] = t'[y/z]$ | | A7 |

The RAM memory denoted by a closed term of the form $\mathsf{rm}(\sigma)$, where $\sigma \in \Sigma_{\mathrm{rm}}$, is an operative RAM memory whose state is $\sigma$. The RAM memory denoted by a closed term of the form $\mathsf{rm}(*)$ is an inoperative RAM memory. An inoperative RAM memory can be viewed as a RAM memory whose state is unavailable. Carrying out an operation on an inoperative RAM memory is impossible.

On interaction between a thread and a RAM memory, the thread affects the RAM memory and the RAM memory affects the thread. The use operator concerns the effects of a RAM memory on a thread and the apply operator concerns the effects of a thread on a RAM memory. The thread denoted by a closed term of the form $t \mathbin{/} t'$ and the RAM memory denoted by a closed term of the form $t \bullet t'$ are the thread and RAM memory, respectively, that result from carrying out the operations that go with the basic actions performed by the thread denoted by $t$ on the RAM memory denoted by $t'$. When the operation that goes with a basic action performed by a thread is carried out on a RAM memory, the state of the RAM memory is changed according to the operation concerned and the thread is affected as follows: the basic action turns into the internal action $\mathsf{tau}$ and the two ways to proceed reduce to one on the basis of the reply produced according to the operation concerned. Thus, the internal action $\mathsf{tau}$ is left as a trace of each

basic action that has led to carrying out an operation on the RAM memory.

The following two elimination results for closed PGA/BTA$^\infty$/RAM terms are proved similarly to Theorems 1 and 2 from [14].

**Proposition 2** *For all closed* PGA/BTA$^\infty$/RAM *terms $t$ of sort* **T** *in which all subterms of sort* **IS** *are repetition-free, there exists a closed* [PGA/BTA$^\infty$]$(\mathcal{A}_{\mathrm{ram}})$ *term $t'$ of sort* **T** *such that $t = t'$ is derivable from the axioms of* PGA/BTA$^\infty$/RAM.

**Proposition 3** *For all closed* PGA/BTA$^\infty$/RAM *terms $t$ of sort* **RM** *in which all subterms of sort* **IS** *are repetition-free, there exists a closed* [PGA/BTA$^\infty$]$(\mathcal{A}_{\mathrm{ram}})$ *term $t'$ of sort* **RM** *such that $t = t'$ is derivable from the axioms of* PGA/BTA$^\infty$/RAM.

## 7   Computing Partial Functions from $(\{0,1\}^*)^n$ to $\{0,1\}^*$

In this section, we make precise in the setting of the algebraic theory PGA/BTA$^\infty$/RAM what it means that a given instruction sequence computes a given partial function from $(\{0,1\}^*)^n$ to $\{0,1\}^*$ ($n \in \mathbb{N}$).

We use the notation $f : A \twoheadrightarrow B$ to indicate that $f$ is a partial function from $A$ to $B$. We write $\ell(w)$, where $w \in \{0,1\}^*$, for the length of $w$.

Let $t$ be a closed PGA/BTA$^\infty$/RAM term of sort **IS**, let $n \in \mathbb{N}$, let $F : (\{0,1\}^*)^n \twoheadrightarrow \{0,1\}^*$, and let $T : \mathbb{N} \to \mathbb{N}$. Then $t$ *computes $F$ in time $T$ under the uniform time measure* if:

- for all $w_1, \ldots, w_n \in \{0,1\}^*$ such that $F(w_1, \ldots, w_n)$ is defined, there exists a $\sigma \in \Sigma_{\mathrm{rm}}$ such that:

$$|t| \bullet \mathsf{rm}((\sigma_\epsilon : 1 \mapsto w_1, \ldots, n \mapsto w_n)) = \mathsf{rm}((\sigma : 0 \mapsto F(w_1, \ldots, w_n))) \,,$$

$$depth(|t| \,/\, \mathsf{rm}((\sigma_\epsilon : 1 \mapsto w_1, \ldots, n \mapsto w_n))) \leq T(\ell(w_1) + \ldots + \ell(w_n)) \,;$$

- for all $w_1, \ldots, w_n \in \{0,1\}^*$ such that $F(w_1, \ldots, w_n)$ is undefined:

$$|t| \bullet \mathsf{rm}((\sigma_\epsilon : 1 \mapsto w_1, \ldots, n \mapsto w_n)) = \mathsf{rm}(*) \,.$$

We say that $t$ *computes $F$* if there exists a $T : \mathbb{N} \to \mathbb{N}$ such that $t$ computes $F$ in time $T$ under the uniform time measure.

With the above definition, we can establish whether an instruction sequence of the kind considered in PGA/BTA$^\infty$/RAM computes a given partial function from $(\{0,1\}^*)^n$ to $\{0,1\}^*$ ($n \in \mathbb{N}$) mainly by equational reasoning using the axioms of PGA/BTA$^\infty$/RAM. The axioms for the apply operator given in Table 5, i.e. axioms A1–A7, are instrumental in that.

The setting provided by PGA/BTA$^\infty$/RAM is more general than the setting provided by any known version of the RAM model of computation. PGA/BTA$^\infty$/RAM is not suitable as a model of computation itself, but virtually all known versions of the RAM model of computation can be dealt with by imposing restrictions on the set of basic RAM instructions ($\mathcal{A}_{\mathrm{ram}}$). Investigations of issues in areas such as computational complexity and analysis of algorithms require restriction to instructions that are found to be sufficiently primitive. Without any restriction on $\mathcal{A}_{\mathrm{ram}}$, we even have that, for each computable $F : (\{0,1\}^*)^n \rightarrow \{0,1\}^*$, there exists a closed PGA/BTA$^\infty$/RAM term $t$ of sort **IS** such that $t$ computes $F$ in one step.

Restriction of the set of basic RAM instructions to instructions, with both direct and indirect addressing of registers, to carry out addition by one on natural numbers, to carry out comparisons of natural numbers on equal to and greater than, and to copy natural numbers (identifying bit strings with the natural numbers that they represent) gives rise to the version of the RAM model of computation known as the successor RAM model. The basic instructions of a successor RAM are clearly very primitive, but as a consequence of that a successor RAM is not a very realistic idealization of a real computer. In Section 8, we present a set of basic RAM instructions that yields a much more realistic idealization of a real computer.

Whatever version of the RAM model of computation is obtained by restriction of the set of basic RAM instructions considered in PGA/BTA$^\infty$/RAM, it is an idealization of a real computer in the sense that its memory offers an unbounded number of registers that can contain a bit string of arbitrary length instead of a bounded number of registers that can only contain a bit string of a fixed length.

# 8   Basic Instructions for More or Less Realistic RAMs

In this section, we introduce a set of basic RAM instructions that give rise to a version of the RAM model of computation that is a fairly realistic idealization of a real computer.

In general, the execution of an instruction by a real computer changes the memory state of the computer by carrying out a certain operation on the contents of certain registers and changing the content of a certain register into the result of this. We use a special notation reflecting this for the restricted set of basic RAM instructions with which a fairly realistic idealization of a real computer is obtained. This restricted set of basic RAM instructions consists of all basic RAM instructions that have one of the following forms in the special notation:

$$binop{:}s_1{:}s_2{:}d \quad \text{or} \quad unop{:}s_1{:}d \quad \text{or} \quad cmpop{:}s_1{:}s_2 \, ,$$

where

$$
\begin{aligned}
binop &\in \{\mathsf{add}, \mathsf{sub}, \mathsf{mul}, \mathsf{div}, \mathsf{and}, \mathsf{or}, \mathsf{xor}\} \, , \\
unop &\in \{\mathsf{not}, \mathsf{shl}, \mathsf{shr}, \mathsf{rol}, \mathsf{ror}, \mathsf{mov}\} \, , \\
cmpop &\in \{\mathsf{eq}, \mathsf{gt}\} \, .
\end{aligned}
$$

and

$s_1$ has one of the following forms: $\#i$ or $i$ or $@i$, where $i \in \mathbb{N}$,

$s_2$ has one of the following forms: $\#i$ or $i$ or $@i$, where $i \in \mathbb{N}$,

$d$  has one of the following forms:      $i$ or $@i$, where $i \in \mathbb{N}$,

We write $\mathcal{A}^{\mathrm{sr}}_{\mathrm{ram}}$ for this set of basic RAM instructions. Moreover, we write $Src$ for the set $\{\#i \mid i \in \mathbb{N}\} \cup \mathbb{N} \cup \{@i \mid i \in \mathbb{N}\}$, $Dst$ for the set $\mathbb{N} \cup \{@i \mid i \in \mathbb{N}\}$, and $\mathcal{C}^{\mathrm{sr}}_{\mathrm{ram}}$ for the set $\{cmpop{:}s_1{:}s_2 \mid cmpop \in \{\mathsf{eq}, \mathsf{gt}\} \wedge s_1, s_2 \in Src\}$.

The following is a preliminary explanation of basic RAM instructions of the different forms:

- on execution of an instruction of the form $binop{:}s_1{:}s_2{:}d$, the binary operation named $binop$ is carried out on the values given by $s_1$ and $s_2$ and the content of the register given by $d$ is changed into the result of this;

- on execution of an instruction of the form $unop{:}s_1{:}d$, the unary operation named $unop$ is carried out on the value given by $s_1$ and the content of the register given by $d$ is changed into the result of this;

- on execution of an instruction of the form $cmpop{:}s_1{:}s_2$, the comparison operation named $cmpop$ is carried out on the values given by $s_1$ and $s_2$ and the result of this is produced as reply.

For each of the basic RAM instructions from $\mathcal{A}_{\mathrm{ram}}^{\mathrm{sr}}$, each operand of the operation to be carried out on its execution is given in one the following three ways:

- *immediate*: it is the shortest bit string representing the natural number $i$ if $s$ is of the form $\#i$;

- *direct addressing*: it is the content of the register with number $i$ if $s$ is of the form $i$;

- *indirect addressing*: it is the content of the register whose number is represented by the content of the register with number $i$ if $s$ is of the form $@i$.

Except for the comparison instructions, the result of the operation concerned becomes the content of a register in one the following two ways:

- *direct addressing*: it becomes the content of the register with number $i$ if $d$ is of the form $i$;

- *indirect addressing*: it becomes the content of the register whose number is represented by the content of the register with number $i$ if $d$ is of the form $@i$.

As mentioned above, in the case of comparison instructions, the result of the operation concerned becomes the reply produced.

The following kinds of instructions are included in $\mathcal{A}_{\mathrm{ram}}^{\mathrm{sr}}$:

- *arithmetic* instructions ($\mathsf{add}, \mathsf{sub}, \mathsf{mul}, \mathsf{div}$) for carrying out operations that model arithmetic operations on natural numbers with respect to their binary representation by bit strings;

- *logical* instructions ($\mathsf{and}, \mathsf{or}, \mathsf{xor}, \mathsf{not}$) for carrying out bitwise logical operations on bit strings;

- *shift/rotate* instructions ($\mathsf{shl}, \mathsf{shr}, \mathsf{rol}, \mathsf{ror}$) for carrying out bit shift and rotate operations on bit strings;

- *data transfer* instructions ($\mathsf{mov}$) for copying bit strings;

- *comparison* instructions ($\mathsf{eq}, \mathsf{gt}$) for carrying out comparison operations on bit strings.

Data transfer instructions can be interpreted as instructions for carrying out the identity operation on bit strings.

Virtually all common general-purpose instructions of real computers are essentially variants of the basic RAM instructions from $\mathcal{A}_{\mathrm{ram}}^{\mathrm{sr}}$. Therefore, we believe that $\mathcal{A}_{\mathrm{ram}}^{\mathrm{sr}}$ yields a version of the RAM model of computation that is a fairly realistic idealization of a real computer.

Above, a special notation is used for the basic RAM instructions from the set $\mathcal{A}_{\mathrm{ram}}^{\mathrm{sr}}$. In order to use the version of the RAM model of computation with this set of basic RAM instructions in the setting of PGA/BTA$^\infty$/RAM, the special notation must be related to the notation used in that setting.

## 9    More or Less Realistic RAM Instructions and PGA/BTA$^\infty$/RAM

In this section, we relate the special notation for basic RAM instructions used in Section 8 to the notation used in the setting of PGA/BTA$^\infty$/RAM.

We start with defining auxiliary functions for conversion between natural numbers and bit strings and evaluation of the elements of *Src* and *Dst*.

We write $\dotminus$ for proper subtraction of natural numbers. We write $\div$ for zero-totalized Euclidean division of natural numbers, i.e. Euclidean division made total by imposing that division by zero yields zero (like in meadows, see e.g. [16, 5]). We use juxtaposition for concatenation of bit strings.

The *natural to bit string* function $\mathsf{b} : \mathbb{N} \to \{0,1\}^*$ is recursively defined as follows:

$$\mathsf{b}(b) = b \text{ and } \mathsf{b}(n) = (n \bmod 2)\,\mathsf{b}(n \div 2) \text{ if } n > 1$$

and the *bit string to natural* function $\mathsf{n} : \{0,1\}^* \to \mathbb{N}$ is recursively defined as follows:

$$\mathsf{n}(\epsilon) = 0 \text{ and } \mathsf{n}(bw) = 2 \cdot \mathsf{n}(w) + b.$$

These definitions tell us that, when viewed as the binary representation of a natural number, the first bit of a bit string is considered the least significant bit. Results of applying $\mathsf{b}$ have no leading zeros, but the operand of $\mathsf{n}$ may have leading zeros. Thus, we have that $\mathsf{n}(\mathsf{b}(n)) = n$ and $\mathsf{b}(\mathsf{n}(w)) = w'$, where $w'$ is $w$ without leading zeros.

For each $\sigma \in \Sigma_{\mathrm{rm}}$, the *src-valuation in $\sigma$* function $\mathsf{v}_\sigma : Src \to \{0,1\}^*$ is defined as follows:

$$\mathsf{v}_\sigma(\#i) = \mathsf{b}(i), \ \mathsf{v}_\sigma(i) = \sigma(i), \ \text{and} \ \mathsf{v}_\sigma(@i) = \sigma(\mathsf{n}(\sigma(i)))$$

and, for each $\sigma \in \Sigma_{\mathrm{rm}}$, the *dst-valuation in $\sigma$* function $\mathsf{r}_\sigma : Dst \to \mathbb{N}$ is defined as follows:

$$\mathsf{r}_\sigma(i) = i \ \text{and} \ \mathsf{r}_\sigma(@i) = \mathsf{n}(\sigma(i)).$$

We define the operations on bit strings that the operation names add, sub, mul, and div refer to as follows:

$$+ : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* : w_1 + w_2 = \mathsf{b}(\mathsf{n}(w_1) + \mathsf{n}(w_2));$$
$$\dot{-} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* : w_1 \dot{-} w_2 = \mathsf{b}(\mathsf{n}(w_1) \dot{-} \mathsf{n}(w_2));$$
$$\cdot : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* : w_1 \cdot w_2 = \mathsf{b}(\mathsf{n}(w_1) \cdot \mathsf{n}(w_2));$$
$$\div : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* : w_1 \div w_2 = \mathsf{b}(\mathsf{n}(w_1) \div \mathsf{n}(w_2)).$$

These definitions tell us that, although the operands of the operations $+$, $\dot{-}$, $\cdot$, and $\div$ may have leading zeros, results of applying these operations have no leading zeros.

We define the operations on bit strings that the operation names and, or, xor, and not refer to recursively as follows:

$$\land : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* : \ \epsilon \land \epsilon = \epsilon, \ \ \epsilon \land (bw) = 0\,(\epsilon \land w),$$
$$(bw) \land \epsilon = 0\,(w \land \epsilon), \ \ (b_1\,w_1) \land (b_2\,w_2) = (b_1 \land b_2)\,(w_1 \land w_2);$$
$$\lor : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* : \ \epsilon \lor \epsilon = \epsilon, \ \ \epsilon \lor (bw) = b\,(\epsilon \lor w),$$
$$(bw) \lor \epsilon = b\,(w \lor \epsilon), \ \ (b_1\,w_1) \lor (b_2\,w_2) = (b_1 \lor b_2)\,(w_1 \lor w_2);$$
$$\oplus : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* : \ \epsilon \oplus \epsilon = \epsilon, \ \ \epsilon \oplus (bw) = b\,(\epsilon \oplus w),$$
$$(bw) \oplus \epsilon = b\,(w \oplus \epsilon), \ \ (b_1\,w_1) \oplus (b_2\,w_2) = (b_1 \oplus b_2)\,(w_1 \oplus w_2);$$
$$\neg : \{0,1\}^* \to \{0,1\}^* : \ \neg\epsilon = \epsilon, \ \ \neg(bw) = (\neg b)\,(\neg w).$$

These definitions tell us that, if the operands of the operations $\land$, $\lor$, and $\oplus$ do not have the same length, sufficient leading zeros are assumed to exist. Moreover, results of applying these operations and results of applying $\neg$ can have leading zeros.

We define the operations on bit strings that the operation names shl, shr, rol, and ror refer to as follows:

$$\ll : \{0,1\}^* \to \{0,1\}^* : \ \ll\epsilon = \epsilon, \ \ \ll(bw) = 0bw;$$
$$\gg : \{0,1\}^* \to \{0,1\}^* : \ \gg\epsilon = \epsilon, \ \ \gg(bw) = w;$$
$$\oslash : \{0,1\}^* \to \{0,1\}^* : \ \oslash\epsilon = \epsilon, \ \ \oslash(wb) = bw;$$
$$\obslash : \{0,1\}^* \to \{0,1\}^* : \ \obslash\epsilon = \epsilon, \ \ \obslash(bw) = wb.$$

These definitions tell us that results of applying the operations $\ll$, $\gg$, $\diamond\!\!\!<$, and $\diamond\!\!\!>$ can have leading zeros. We have that $\mathsf{n}(\ll w) = \mathsf{n}(w) \cdot 2$ and $\mathsf{n}(\gg w) = \mathsf{n}(w) \div 2$.

Now, we are ready to relate the special notation for basic RAM instructions used in Section 8 to the notation used in the setting of PGA/BTA$^\infty$/RAM:

$\mathsf{add}{:}s_1{:}s_2{:}d$ stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto \mathsf{v}_\sigma(s_1) + \mathsf{v}_\sigma(s_2))$;

$\mathsf{sub}{:}s_1{:}s_2{:}d$ stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto \mathsf{v}_\sigma(s_1) \mathbin{\dot{-}} \mathsf{v}_\sigma(s_2))$;

$\mathsf{mul}{:}s_1{:}s_2{:}d$ stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto \mathsf{v}_\sigma(s_1) \cdot \mathsf{v}_\sigma(s_2))$;

$\mathsf{div}{:}s_1{:}s_2{:}d$  stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto \mathsf{v}_\sigma(s_1) \div \mathsf{v}_\sigma(s_2))$;

$\mathsf{and}{:}s_1{:}s_2{:}d$ stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto \mathsf{v}_\sigma(s_1) \wedge \mathsf{v}_\sigma(s_2))$;

$\mathsf{or}{:}s_1{:}s_2{:}d$   stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto \mathsf{v}_\sigma(s_1) \vee \mathsf{v}_\sigma(s_2))$;

$\mathsf{xor}{:}s_1{:}s_2{:}d$  stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto \mathsf{v}_\sigma(s_1) \oplus \mathsf{v}_\sigma(s_2))$;

$\mathsf{not}{:}s_1{:}d$     stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto \neg \mathsf{v}_\sigma(s_1))$;

$\mathsf{shl}{:}s_1{:}d$     stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto {\ll} \mathsf{v}_\sigma(s_1))$;

$\mathsf{shr}{:}s_1{:}d$     stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto {\gg} \mathsf{v}_\sigma(s_1))$;

$\mathsf{rol}{:}s_1{:}d$     stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto {\diamond\!\!\!<} \mathsf{v}_\sigma(s_1))$;

$\mathsf{ror}{:}s_1{:}d$     stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto {\diamond\!\!\!>} \mathsf{v}_\sigma(s_1))$;

$\mathsf{mov}{:}s_1{:}d$     stands for $1/q$ where $q(\sigma) = (\sigma : \mathsf{r}_\sigma(d) \mapsto \mathsf{v}_\sigma(s_1))$;

$\mathsf{eq}{:}s_1{:}s_2$      stands for $p/\mathsf{i}$  where $p(\sigma) = 1$ iff $\mathsf{n}(\mathsf{v}_\sigma(s_1)) = \mathsf{n}(\mathsf{v}_\sigma(s_2))$;

$\mathsf{gt}{:}s_1{:}s_2$      stands for $p/\mathsf{i}$  where $p(\sigma) = 1$ iff $\mathsf{n}(\mathsf{v}_\sigma(s_1)) > \mathsf{n}(\mathsf{v}_\sigma(s_2))$.

## 10   Semi-Realistic RAM Programs

In this section, we introduce a version of the RAM model of computation that is intended to be a more or less realistic idealization of a real computer. This version is obtained by restriction of the set of basic RAM instructions considered in PGA/BTA$^\infty$/RAM.

A *semi-realistic RAM program*, called an *SRRAM program* for short, is a closed PGA/BTA$^\infty$/RAM term of sort **IS** that is of the form $(t_1 \mathbin{;} \ldots \mathbin{;} t_n)^\omega$,

where each $t_i$ has one of the following forms:

$$
\begin{aligned}
&a && \text{where } \ a \in \mathcal{A}^{\mathrm{sr}}_{\mathrm{ram}} \setminus \mathcal{C}^{\mathrm{sr}}_{\mathrm{ram}}; \\
&+a \,;\, \#l && \text{where } \ a \in \mathcal{C}^{\mathrm{sr}}_{\mathrm{ram}} \ \text{and} \ l \in \mathbb{N}; \\
&\#l && \text{where } \ l \in \mathbb{N}; \\
&! .
\end{aligned}
$$

In the *SRRAM model of computation*, machines, called SRRAMs, consist of an SRRAM program together with a RAM memory on which it operates during execution.

A *standard RAM program* is an SRRAM program in which only addition instructions, subtraction instructions, data transfer instructions, and comparison instructions occur (cf. [19]). A *successor RAM program* is an SRRAM program in which only addition instructions of the form add:$s_1$:#1:$d$, data transfer instructions, and comparison instructions occur (cf. [30]).

The following theorem is a result concerning the computational power of SRRAM programs.

**Theorem 1** *For each $F : (\{0,1\}^*)^n \nrightarrow \{0,1\}^*$, there exists an SRRAM program $P$ such that $P$ computes $F$ iff $F$ is Turing-computable.*

**Proof:**    The SRRAM model of computation is essentially the same as the MBRAM model of computation from [31] extended with shift/rotate instructions. It follows directly from simulation results mentioned in [31] (part (5) of Theorem 2.4, part (1) of Theorem 2.5, and part (3) of Theorem 2.6) that each MBRAM can be simulated by a Turing machine and vice versa. Because each Turing machine can be simulated by a MBRAM, we immediate have that each Turing machine can be simulated by an SRRAM. It is easy to see that the shift/rotate instructions can be simulated by a Turing machine. From this and the fact that each MBRAM can be simulated by a Turing machine, it follows that each SRRAM can be simulated by a Turing machine as well. Hence, each SRRAM is Turing equivalent to a Turing machine. From this, the theorem follows immediately.    □

Below, we write *POLY* for $\{T \mid T : \mathbb{N} \to \mathbb{N} \wedge T \text{ is a polynomial function}\}$.

The following theorem is a result relating the complexity class **PSPACE** to the functions from $\{0,1\}^*$ to $\{0,1\}$ that can be computed by an SRRAM program in polynomial time.

**Theorem 2** *For each $F:\{0,1\}^* \to \{0,1\}$, there exist an SRRAM program $P$ and a $T \in POLY$ such that $P$ computes $F$ in time $T$ under the uniform time measure iff $F \in$* **PSPACE**.

**Proof:**     The SRRAM model of computation is essentially the same as the MRAM model of computation from [24] extended with division and shift/rotate instructions. We know from the main result of that paper that, for each $F : \{0,1\}^* \to \{0,1\}$, there even exists an SRRAM program $P$ in which division and shift/rotate instructions do not occur and a $T \in POLY$ such that $P$ computes $F$ in time $T$ under the uniform time measure iff $F \in$ **PSPACE**. □

Theorem 2 tell us that all decision problems that belong to **PSPACE** can be solved by means of a SRRAM program in polynomial time. This means that it is highly questionable whether the SRRAM model of computation is a reasonable model of computation. However, it can be made a reasonable model by switching from the uniform time measure to another time measure. Such a time measure is introduced in Section 11.

The proof of Theorem 2 reveals that the theorem still holds if division and shift/rotate instructions are excluded from the SRRAM programs. It turns out that we get another result if multiplication instructions are excluded as well.

**Theorem 3** *For each $F:\{0,1\}^* \to \{0,1\}$, there exist an SRRAM program $P$ in which multiplication, division, and shift/rotate instructions do not occur and a $T \in POLY$ such that $P$ computes $F$ in time $T$ under the uniform time measure iff $F \in$* **P**.

**Proof:**     The model of computation obtained by excluding multiplication, division, and shift/rotate instructions from the SRRAM programs is the standard RAM model of computation extended with logical instructions. From Theorem 2 in [19], we know that time complexity on standard RAMs under the uniform time measure and time complexity on multi-tape Turing machines are polynomially related. It is easy to see that the logical instructions can be simulated by a multi-tape Turing machine in linear time. Hence, the time complexities remain polynomially related if the standard RAM model is extended with logical instructions. From this, the theorem follows immediately. □

# 11  A Bit-Oriented Time Measure
for SRRAM Programs

In this section, we introduce a time measure for the SRRAM model of computation that has it origin in the idea that the time that it takes to execute an instruction on an SRRAM should be based on the number of steps that a multi-tape Turing machine with input alphabet $\{0, 1\}$ needs to simulate the instruction. The choice have been made to make use of well-known upper bounds, but lesser upper bounds could have been used instead.

We write $\mathcal{C}_{\mathbf{T}}$ for the set of all closed PGA/BTA$^\infty$/RAM terms of sort $\mathbf{T}$.

We define a family $\mathsf{c}$ of partial *non-uniform cost* functions $\mathsf{c}_\sigma : \mathcal{C}_{\mathbf{T}} \nrightarrow \mathbb{N}$, one for each $\sigma \in \Sigma_{\mathrm{rm}}$, recursively as follows:

$$\mathsf{c}_\sigma(\mathsf{S}) = 0 \ ,$$
$$\mathsf{c}_\sigma(t \trianglelefteq p/q \trianglerighteq t') = \mathsf{c}_\sigma(p/q) + \mathsf{c}_{q(\sigma)}(t) \quad \text{if } p(\sigma) = 1 \ ,$$
$$\mathsf{c}_\sigma(t \trianglelefteq p/q \trianglerighteq t') = \mathsf{c}_\sigma(p/q) + \mathsf{c}_{q(\sigma)}(t') \quad \text{if } p(\sigma) = 0 \ ,$$

where the family of partial functions $\mathsf{c}_\sigma : \mathcal{A}_{\mathrm{ram}} \nrightarrow \mathbb{N}$ (defined for all basic RAM instructions from $\mathcal{A}_{\mathrm{ram}}^{\mathrm{sr}}$), one for each $\sigma \in \Sigma_{\mathrm{rm}}$, is defined as follows:

$$\mathsf{c}_\sigma(binop{:}s_1{:}s_2{:}d) = \max\{\mathsf{c}_\sigma(s_1), \mathsf{c}_\sigma(s_2)\} + \mathsf{c}'_\sigma(d) \quad \text{if } binop \notin \{\mathsf{mul}, \mathsf{div}\} \ ,$$
$$\mathsf{c}_\sigma(binop{:}s_1{:}s_2{:}d) = \mathsf{c}_\sigma(s_1) \cdot \mathsf{c}_\sigma(s_2) + \mathsf{c}'_\sigma(d) \qquad\quad \text{if } binop \in \{\mathsf{mul}, \mathsf{div}\} \ ,$$
$$\mathsf{c}_\sigma(unop{:}s_1{:}d) = \mathsf{c}_\sigma(s_1) + \mathsf{c}'_\sigma(d) \ ,$$
$$\mathsf{c}_\sigma(cmpop{:}s_1{:}s_2) = \max\{\mathsf{c}_\sigma(s_1), \mathsf{c}_\sigma(s_2)\} \ ,$$

where the family of total functions $\mathsf{c}_\sigma : Src \to \mathbb{N}$, one for each $\sigma \in \Sigma_{\mathrm{rm}}$, is defined as follows:[6]

$$\mathsf{c}_\sigma(\#i) = \ell(\mathsf{b}(i)) \ ,$$
$$\mathsf{c}_\sigma(i) = \ell(\mathsf{b}(i)) + \ell(\mathsf{v}_\sigma(i)) \ ,$$
$$\mathsf{c}_\sigma(@i) = \ell(\mathsf{b}(i)) + \ell(\mathsf{v}_\sigma(i)) + \ell(\mathsf{v}_\sigma(\mathsf{n}(\mathsf{v}_\sigma(i))))$$

and the family of total functions $\mathsf{c}'_\sigma : Dst \to \mathbb{N}$, one for each $\sigma \in \Sigma_{\mathrm{rm}}$, is defined as follows:

$$\mathsf{c}'_\sigma(i) = \ell(\mathsf{b}(i)) \ ,$$
$$\mathsf{c}'_\sigma(@i) = \ell(\mathsf{b}(i)) + \ell(\mathsf{v}_\sigma(i)) \ .$$

---

[6] Recall that $\ell(w)$, where $w \in \{0, 1\}^*$, stands for the length of $w$.

Let $t$ be a closed PGA/BTA$^\infty$/RAM term of sort **IS**, let $n \in \mathbb{N}$, let $F : (\{0,1\}^*)^n \nrightarrow \{0,1\}^*$, and let $T : \mathbb{N} \to \mathbb{N}$. Then $t$ *computes $F$ in time $T$ under the bit-oriented time measure* if:

- for all $w_1, \ldots, w_n \in \{0,1\}^*$ such that $F(w_1, \ldots, w_n)$ is defined, there exist a $\sigma \in \Sigma_{\mathrm{rm}}$ such that:

$$|t| \bullet \mathsf{rm}((\sigma_\epsilon : 1 \mapsto w_1, \ldots, n \mapsto w_n)) = \mathsf{rm}((\sigma : 0 \mapsto F(w_1, \ldots, w_n))) \,,$$

$$\mathsf{c}_{(\sigma_\epsilon : 1 \mapsto w_1, \ldots, n \mapsto w_n)}(|t|) \leq T(\ell(w_1) + \ldots + \ell(w_n)) \,;$$

- for all $w_1, \ldots, w_n \in \{0,1\}^*$ such that $F(w_1, \ldots, w_n)$ is undefined:

$$|t| \bullet \mathsf{rm}((\sigma_\epsilon : 1 \mapsto w_1, \ldots, n \mapsto w_n)) = \mathsf{rm}(*) \,.$$

Fine-tuning this definition boils down to adapting the definition of the family of partial functions $\mathsf{c}_\sigma : \mathcal{A}_{\mathrm{ram}} \nrightarrow \mathbb{N}$.

The parts of Theorem 2 from [19] that concern standard RAMs under the logarithmic time measure hold also for SRRAMs under the bit-oriented time measure.

**Theorem 4** *For each $F : (\{0,1\}^*)^n \nrightarrow \{0,1\}^*$:*

(a) *if there exist an SRRAM program $P$ and a $T : \mathbb{N} \to \mathbb{N}$ such that $P$ computes $F$ in time $T$ under the bit-oriented time measure, then there exists a multi-tape Turing machine $M$ such that $M$ computes $F$ in time $O(T^2)$;*

(b) *if there exist a multi-tape Turing machine $M$ and a $T : \mathbb{N} \to \mathbb{N}$ such that $M$ computes $F$ in time $T$, then there exists an SRRAM program $P$ such that $P$ computes $F$ in time $O(T \cdot \log_2(T))$ under the bit-oriented time measure.*

**Proof:**     In the proof of (a), one of the working tapes of $M$ is considered to hold a representation of the state of the RAM memory on which $P$ operates during execution. A RAM memory state is represented on this working tape by a string over the alphabet $\{0, 1, \sqcup\}$ that belongs to the set defined by the regular expression $(\sqcup\sqcup(0+1)(0+1)^*\sqcup(0+1)(0+1)^*)^*$. The working tape content $\sqcup\sqcup w_1 \sqcup w_1' \ldots \sqcup\sqcup w_n \sqcup w_n' \sqcup\sqcup\sqcup \ldots$ represents the RAM memory state $(\sigma_\epsilon : \mathsf{n}(w_1) \mapsto w_1', \ldots, \mathsf{n}(w_n) \mapsto w_n')$.

Take arbitrary $\sigma \in \Sigma_{\mathrm{rm}}$ and $i \in \mathbb{N}$, and let $w = \mathsf{b}(i)$ and $w' = \mathsf{v}_\sigma(i)$. Then $\sqcup\sqcup w \sqcup w'$ occurs in the representation of $\sigma$ iff $\sigma(i) \neq \epsilon$. Moreover,

$\ell(\sqcup\sqcup w \sqcup w') = \ell(\mathsf{b}(i)) + \ell(\mathsf{v}_\sigma(i))$. From this and the definition of the function $\mathsf{c}_\sigma : \mathcal{A}_{\mathrm{ram}} \nrightarrow \mathbb{N}$, it follows readily that, if $u \in \mathcal{A}^{\mathrm{sr}}_{\mathrm{ram}}$ and $u$ is of the form $binop{:}s_1{:}s_2{:}i$, $unop{:}s_1{:}i$, $binop{:}s_1{:}s_2{:}@j$ or $unop{:}s_1{:}@j$, where $\mathsf{v}_\sigma(j) = i$, then $\ell(\sqcup\sqcup w \sqcup w')$ is bounded by a constant times $\mathsf{c}_\sigma(u)$. From this and the fact that $P$ computes $F$ in time $T$ under the bit-oriented time measure, it follows immediately that the length of the representation of $\sigma$ on the work tape is bounded by $O(T)$. This means that searching the working tape for the entry of a register takes at most $O(T)$ steps. Since $\mathsf{c}_\sigma(u) \geq 1$ for all instructions $u \in \mathcal{A}^{\mathrm{sr}}_{\mathrm{ram}}$, at most $O(T)$ instructions are executed. Because each instruction $u \in \mathcal{A}^{\mathrm{sr}}_{\mathrm{ram}}$ involves a constant number of searches for register entries on the working tape, this means that the total number of steps spent on searching the working tape for the entries of registers is bounded by $O(T^2)$.

The functions $\mathsf{c}_\sigma : \mathcal{A}_{\mathrm{ram}} \nrightarrow \mathbb{N}$ are defined such that, in the case that $P$ computes $F$ in time $T$ under the bit-oriented time measure, the total number of steps that a multi-tape Turing machine needs to compute $F$, not counting the steps spent on searching the working tape for the entries of registers, is bounded by $O(T)$. Because the total number of steps spent on searching the working tape for the entries of registers is bounded by $O(T^2)$, this means that the total number of steps that a multi-tape Turing machine needs to compute $F$ is bounded by $O(T^2)$.

In the proof of (b), $M$ is assumed to have $k$ tapes. The state of the RAM memory on which $P$ operates during execution is considered to represent the contents of the $k$ tapes of $M$ as follows: the content of the $i$th cell on the $j$th tape is the content of the register with number $k \cdot i + j + c$, where $c$ is the number of auxiliary registers that $P$ needs to simulate Turing machine steps. The auxiliary registers include $k$ registers for the positions of the $k$ tape heads. $P$ can read out or alter the cells under the $k$ tape heads by means of indirect addressing through these position-holding registers.

It follows immediately from the definition of the functions $\mathsf{c}_\sigma : \mathcal{A}_{\mathrm{ram}} \nrightarrow \mathbb{N}$ that, under the bit oriented time measure, the time that an SRRAM program needs per Turing machine step is a constant plus the time spent on accessing the registers that contain the contents of the cells under the tape heads. In the case that $M$ computes $F$ in time $T$, the number of cells $M$ can move tape heads away from the starting position is bounded by $T$. From this, the fact that an SRRAM program uses indirect addressing to access the registers that contain the contents of the cells under the tape heads, and the fact that $\ell(\mathsf{b}(i)) = \lfloor \log_2(i) \rfloor + 1$ if $i > 0$, it follows immediately that the time it takes an SRRAM program computing $F$ to access these farthest tape cells

is bounded by $O(\log_2(T))$. Because each step of $M$ involves accessing the cells under its tape heads, this means that the time that a SRRAM program needs to compute $F$ is bounded by $O(T \cdot \log_2(T))$.                    □

The approaches to the proofs of the two parts of Theorem 4 have been inspired by the proofs of the corresponding parts of Theorem 2 from [19].
     The following corollary of Theorem 4 is a counterpart of Theorem 3.

**Corollary 1** *For each $F : (\{0,1\}^*)^n \to \{0,1\}$, there exist an SRRAM program $P$ and a $T \in POLY$ such that $P$ computes $F$ in time $T$ under the bit-oriented time measure iff $F \in \mathbf{P}$.*

# 12   A Bit-Oriented Space Measure
#       for SRRAM Programs

In this section, we introduce for the sake of completeness a bit-oriented space measure for the SRRAM model of computation. This space measure originates from [31].
     Let $t$ be a closed PGA/BTA$^\infty$/RAM term of sort **IS**, let $n \in \mathbb{N}$, let $F : (\{0,1\}^*)^n \twoheadrightarrow \{0,1\}^*$, and let $S : \mathbb{N} \to \mathbb{N}$. Then $t$ *computes $F$ in space $S$* if $t$ computes $F$ and, for all $w_1, \ldots, w_n \in \{0,1\}^*$ such that $F(w_1, \ldots, w_n)$ is defined, for some $m \in \mathbb{N}_1$, there exist closed PGA/BTA$^\infty$/RAM terms $t_1, \ldots, t_m$ of sort **T** and RAM memory states $\sigma_1, \ldots, \sigma_m$ such that:

- $t_1 = |t|$;

- $t_m = \mathsf{S}$;

- $\sigma_1(i) = \epsilon$ for all $i \in \mathbb{N}$ with $i \notin \{1, \ldots, n\}$;

- $\sigma_j(i) = w_i$ for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$;

- $\sigma_m(0) = F(w_1, \ldots, w_n)$;

- for all $j \in \{1, \ldots, m\}$, $t_j \bullet \mathsf{rm}(\sigma_j) = t_{j+1} \bullet \mathsf{rm}(\sigma_{j+1})$ is a closed substitution instance of an instance of axiom schema A4 or A5;

- $\max\{\sum_{i \in \mathbb{N} \setminus \{1, \ldots, n\}} (\ell(i) + \ell(\sigma_j(i))) \mid j \in \{1, \ldots, m\}\} \leq S(\ell(w_1) + \ldots + \ell(w_n))$.

The pairs $(t_j, \sigma_j)$, for $j \in \{1, \ldots, m\}$, can be looked upon as SRRAM configurations and the sequence $(t_1, \sigma_1) \ldots (t_m, \sigma_m)$ can be looked upon as a SRRAM computation. Instead of introducing off-line SRRAMs, we require that during computations the contents of the input registers are never changed.

In the above definition space is essentially measured following the third method mentioned in [31], using the function $size_b$ from that paper as size function. By this space measure, it is guaranteed that space complexity on SRRAMs and space complexity on multi-tape Turing machines are related by a constant factor.

## 13    Discussion on the Bit-Oriented Time Measure

In the field of computational complexity, a model of computation is considered a reasonable sequential model of computation if time complexity on its machines and time complexity on multi-tape Turing machines are polynomially related and space complexity on its machines and space complexity on multi-tape Turing machines are related by a constant factor (cf. the Invariance Thesis in [31]). This makes the complexity classes that represent the fundamental concepts of computational complexity, i.e. L, NL, P, NP, PSPACE, NPSPACE, EXP, NEXP, EXPSPACE, NEXPSPACE, machine-independent insofar as reasonable sequential models of computation are concerned.

The logarithmic time measure has been introduced in all but the simplest known versions of the RAM model of computation to obtain a reasonable model. However, it is questionable whether the logarithmic time measure is the most natural time measure. It takes the lengths of the bit strings involved in the execution of an instruction into account, but not the operation involved. The logarithmic time measure works in the case of the known versions of the RAM model of computation only because the operations involved can always be simulated by a multi-tape Turing machine in polynomial time.

The bit-oriented time measure introduced in this paper takes both the operation and the lengths of the bit strings involved in the execution of an instruction into account. Thereby, the bit-oriented time measure actually takes the total number of operations on bits involved in the execution of an instruction into account. This property is arguably the best justification of a time measure intended to make the time measures of different models of computation comparable.

With $\{0, 1\}$ as input alphabet, a version of the Turing machine model of computation supports operations on bits more directly than most other well-known models of computation. This has been an important reason to consider in this paper the times that it takes to carry out the operations on bits in the setting of a version of the Turing machine model. Another important reason has been that the complexity classes that represent the fundamental concepts of computational complexity were initially introduced and studied in the setting of the multi-tape Turing machine model.

The extended logarithmic time measure introduced in [20] also takes the total number of operations on bits involved in the execution of an instruction into account, but, there, the choice is made to consider the times that it takes to carry out the operations on bits in the setting of the successor RAM model. This is the most primitive version of the RAM model and supports operations on bits equally directly as multi-tape Turing machine model. The approach of [20] may be advantageous if one is interested in relating complexity results based on other versions of the RAM model to complexity results based on the successor RAM model, but is disadvantageous if one is interested in relating complexity results based on versions of the RAM model to complexity results based on the multi-tape Turing machine model.

The idea behind the bit-oriented time measure from this paper is that the time that it takes to execute an instruction on an SRRAM should be based on the number of steps that a multi-tape Turing machine with input alphabet $\{0, 1\}$ needs to simulate the instruction. Moreover, the choice has been made to use well-known polynomial upper bounds. By producing the bit-oriented measure in this way for programs of RAMs of a kind obtained by restricting the set of basic RAM instructions of PGA/BTA$^\infty$/RAM in another way than for SRRAM programs, it is guaranteed that Theorem 4 holds for these programs as well. Examination of the proof of that theorem teaches us that it depends only on the use of upper bounds for the number of steps that a multi-tape Turing machine with input alphabet $\{0, 1\}$ needs to simulate the instructions that may occur in the programs.

## 14  Concluding Remarks

We have presented an instantiation of a parameterized algebraic theory of single-pass instruction sequences, the behaviours produced by such instruction sequences under execution, and the interaction between such behaviours and components of an execution environment for instruction sequences. In

the instantiation concerned, RAM memories are taken as the components of an execution environment, instructions for a RAM are taken as basic instructions, and an execution environment consists of only one component. Because we have opted for the most general instantiation, all instructions that do not read out or alter more than one register from the RAM memory are taken as basic instructions.

The presentation of the instantiation has been set up in such a way that the introduction of services, the generic kind of execution-environment components from the parameterized theory, is circumvented. In [13], the presentation of another instantiation of the same parameterized theory has been set up in the same way. The distinguishing feature of this way of presenting an instantiation of the parameterized theory is that it yields a less involved presentation than the way adopted in earlier work based on an instantiation of this parameterized theory.

We have provided evidence for the claim that the presented algebraic theory provides a setting for the development of theory in areas such as computational complexity and analysis of algorithms that is more general than the setting provided by some known version of the RAM model of computation. We have among other things shown that a relatively unknown, but realistic, version of the RAM model can be dealt with in the setting concerned by imposing apposite restrictions. For this model, an alternative to the usual time measures for versions of the RAM model, called the bit-oriented time measure has been introduced.

Related to the introduction of the bit-oriented time measure is the choice for registers that contain bit strings instead of natural numbers. Whereas it is usual in versions of the RAM model of computation that bit strings are represented by natural numbers, here natural numbers are represented by bit strings. Moreover, the choice has been made to represent the natural number 0 by the bit string 0 and to adopt the empty bit string as the register content that indicates that a register is (as yet) unused. Therefore, we have, as in most other versions of the RAM model, $\ell(0) = 1$ and $\ell(i + 1) = \lfloor \log_2(i + 1) \rfloor + 1$ if $\ell$ on natural numbers is simply defined by $\ell(i) = \ell(\mathsf{b}(i))$ (as before $\ell(w)$, where $w \in \{0, 1\}^*$, stands for the length of $w$).

The closed terms of the presented algebraic theory that are used as RAM programs can be considered to constitute a programming language of which the syntax and semantics is defined following an algebraic approach. However, this approach is more operational than the usual algebraic approach, which is among other things followed in [17, 18, 23]. The more operational

approach is advantageous in the case of a language that is used to investigate issues in the areas of computational complexity and analysis of algorithms.

The work presented in this paper is among other things concerned with formalization in the areas of computational complexity and analysis of algorithms. To the best of my knowledge, very little work has been done in this area. Three notable exceptions are [27, 34, 2]. However, those papers are concerned with formalization in a theorem prover (HOL4, Isabelle/HOL, Matita) and focussed on some version of the Turing machine model of computation. This makes it impracticable to compare the work presented in those papers with the work presented here.

The contributions of this paper to the work on models of computation rely heavily on [19, 24]. A variant of the bit-oriented time measure has been proposed in [20].

This paper introduces a setting for the development of theory in areas such as computational complexity and analysis of algorithms using virtually any version of the RAM model of computation. This setting is an instantiation of a parametrized algebraic theory. Several other models of computation can be covered by other instantiations of this theory. The instantiation for the Turing machine model of computation is described in [14]. However, the theory concerned is not general enough to cover parallel models of computation. An interesting option for future work is to study how it can be extended to a theory that covers parallel models of computation.

## Acknowledgement

## References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Company, 1974.

[2] Andrea Asperti and Wilmer Ricciotti. A formalization of multi-tape Turing machines. *Theoretical Computer Science*, 603:23–42, 2015. `doi: 10.1016/j.tcs.2015.07.013`.

[3] Jos C. M. Baeten and W. Peter Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990. `doi:10.1017/CBO9780511624193`.

[4] Jan A. Bergstra and Marijke E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002. `doi:10.1016/S1567-8326(02)00018-8`.

[5] Jan A. Bergstra and Cornelis A. Middelburg. Inversive meadows and divisive meadows. *Journal of Applied Logic*, 9(3):203–220, 2011. `doi:10.1016/j.jal.2011.03.001`.

[6] Jan A. Bergstra and Cornelis A. Middelburg. Instruction sequence processing operators. *Acta Informatica*, 49(3):139–172, 2012. `doi:10.1007/s00236-012-0154-2`.

[7] Jan A. Bergstra and Cornelis A. Middelburg. *Instruction Sequences for Computer Science*, volume 2 of *Atlantis Studies in Computing*. Atlantis Press, 2012. `doi:10.2991/978-94-91216-65-7`.

[8] Jan A. Bergstra and Cornelis A. Middelburg. Instruction sequence expressions for the secure hash algorithm SHA-256. *CoRR*, abs/1308.0219, 2013. `arXiv:1308.0219`.

[9] Jan A. Bergstra and Cornelis A. Middelburg. Instruction sequence based non-uniform complexity classes. *Scientific Annals of Computer Science*, 24(1):47–89, 2014. `doi:10.7561/SACS.2014.1.47`.

[10] Jan A. Bergstra and Cornelis A. Middelburg. On algorithmic equivalence of instruction sequences for computing bit string functions. *Fundamenta Informaticae*, 138(4):411–434, 2015. `doi:10.3233/FI-2015-1219`.

[11] Jan A. Bergstra and Cornelis A. Middelburg. Instruction sequence size complexity of parity. *Fundamenta Informaticae*, 149(3):297–309, 2016. `doi:10.3233/FI-2016-1450`.

[12] Jan A. Bergstra and Cornelis A. Middelburg. Instruction sequences expressing multiplication algorithms. *Scientific Annals of Computer Science*, 28(1):39–66, 2018. `doi:10.7561/sacs.2018.1.39`.

[13] Jan A. Bergstra and Cornelis A. Middelburg. A short introduction to program algebra with instructions for Boolean registers. *Computer Science Journal of Moldova*, 26(3):199–232, 2018.

[14] Jan A. Bergstra and Cornelis A. Middelburg. Program algebra for Turing-machine programs. *Scientific Annals of Computer Science*, 29(2):113–139, 2019. `doi:10.7561/SACS.2019.2.113`.

[15] Jan A. Bergstra and Cornelis A. Middelburg. On the complexity of the correctness problem for non-zeroness test instruction sequences. *Theoretical Computer Science*, 802:1–18, 2020. `doi:10.1016/j.tcs.2019.03.040`.

[16] Jan A. Bergstra and John V. Tucker. The rational numbers as an abstract data type. *Journal of the ACM*, 54(2):7, 2007. `doi:10.1145/1219092.1219095`.

[17] Manfred Broy, Walter Dosch, Bernhard Möller, and Martin Wirsing. Gotos - A study in the algebraic specification of programming languages (extended abstract). In Wilfried Brauer, editor, *GI - 11. Jahrestagung in Verbindung mit Third Conference of the European Co-operation in Informatics (ECI)*, volume 50 of *Informatik-Fachberichte*, pages 109–121. Springer, 1981. `doi:10.1007/978-3-662-01089-1_13`.

[18] Manfred Broy, Martin Wirsing, and Peter Pepper. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, 9(1):54–99, 1987. `doi:10.1145/9758.10501`.

[19] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973. `doi:10.1016/S0022-0000(73)80029-7`.

[20] Claus Diem. On the notion of bit complexity. *Bulletin of the EATCS*, 103:35–52, 2011.

[21] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985. `doi:10.1007/978-3-642-69962-7`.

[22] Joseph A. Goguen. Theorem proving and algebra. *CoRR*, abs/2101.02690, 2021. `arXiv:2101.02690`.

[23] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Foundations of Computing Series. MIT Press, 1996. `doi:10.7551/mitpress/1188.001.0001`.

[24] Juris Hartmanis and Janos Simon. On the power of multiplication in random access machines. In *15th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 13–23. IEEE Computer Society, 1974. `doi:10.1109/SWAT.1974.20`.

[25] Cornelis A. Middelburg. Instruction sequences as a theme in computer science, 2021. URL: `https://instructionsequence.wordpress.com`

[26] Bernard M. E. Moret. *Theory of Computation*. Addison-Wesley-Longman, 1998.

[27] Michael Norrish. Mechanised computability theory. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Second International Conference on Interactive Theorem Proving, (ITP 2011)*, volume 6898 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011. `doi:10.1007/978-3-642-22863-6_22`.

[28] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[29] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2012. `doi:10.1007/978-3-642-17336-3`.

[30] Arnold Schönhage. On the power of random access machines. In Hermann A. Maurer, editor, *6th Colloquium on Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 520–529. Springer, 1979. `doi:10.1007/3-540-09510-1_42`.

[31] Peter van Emde Boas. Machine models and simulations. In Jan van Leeuwen, editor, *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 1–66. Elsevier, Amsterdam, 1990. `doi:10.1016/B978-0-444-88071-0.50006-0`.

[32] Rob J. van Glabbeek and Frits W. Vaandrager. Modular specification of process algebras. *Theoretical Computer Science*, 113(2):293–348, 1993. `doi:10.1016/0304-3975(93)90006-F`.

[33] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 675–788. Elsevier, Amsterdam, 1990. `doi:10.1016/B978-0-444-88074-1.50018-4`.

[34] Jian Xu, Xingyuan Zhang, and Christian Urban. Mechanising Turing machines and computability theory in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *4th International Conference on Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013. `doi:10.1007/978-3-642-39634-2_13`.