Program Algebra for Turing-Machine Programs

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

DOI
10.7561/SACS.2019.2.113

Publication date
2019

Document Version
Final published version

Published in
Scientific Annals of Computer Science

Scientific Annals of Computer Science vol. 29 (2), 2019, pp. 113–139
doi: 10.7561/SACS.2019.2.113

Program Algebra for Turing-Machine Programs

J.A. Bergstra[1], C.A. Middelburg[1]

Abstract

This paper presents an algebraic theory of instruction sequences with instructions for Turing tapes as basic instructions, the behaviours produced by the instruction sequences concerned under execution, and the interaction between such behaviours and Turing tapes provided by an execution environment. This theory provides a setting for the development of theory in areas such as computability and computational complexity that distinguishes itself by offering the possibility of equational reasoning and being more general than the setting provided by a known version of the Turing-machine model of computation. The theory is essentially an instantiation of a parameterized algebraic theory which is the basis of a line of research in which issues relating to a wide variety of subjects from computer science have been rigorously investigated thinking in terms of instruction sequences.

Keywords: program algebra, thread algebra, model of computation, Turing-machine program, computability, computational complexity.

1 Introduction

This paper introduces an algebraic theory that provides a setting for the development of theory in areas such as computability and computational

[1]Informatics Institute, Faculty of Science, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, the Netherlands, email: {J.A.Bergstra,C.A.Middelburg}@uva.nl.

complexity. The setting in question distinguishes itself by offering the possibility of equational reasoning and being more general than the setting provided by a known version of the Turing-machine model of computation. Many known and unknown versions of the Turing-machine 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 areas such as computability and computational complexity. 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 [7, 8, 9, 10, 11, 13]).

It is often said that a program is an instruction sequence. If this characterization has any value, it must be the case that it is somehow easier to understand the concept of an instruction sequence than to understand the concept of a program. In tune with this, the first objective of the work on instruction sequences that started with [4], and of which an enumeration is available at [21], 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, the work in question has, among other things, yielded an approach to non-uniform computational complexity where instruction sequence length 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, program parallelization for the purpose of explicit multi-threading and virus detection.

The basis of all the work in question (see [21]) 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 instruction sequences under execution, called basic thread algebra, extended to deal with the interaction between such behaviours and components of an execution environment. This combination is parameterized by a set of basic instructions and a set of objects that represent the behaviours exhibited by the components of an execution environment.

The current paper contains a simplified presentation of the instantiation of this combination in which all possible instructions to read out or alter the content of the cell of a Turing tape under the tape's head and to optionally move the head in either direction by one cell are taken as basic instructions and Turing tapes are taken as the components of an execution environment. The rationale for taking certain instructions as basic instructions is that the instructions concerned are sufficient to compute each function on bit strings. However, shorter instruction sequences may be possible if certain additional instructions are taken as basic instructions. That is why we opted for the most general instantiation.

An instantiation in which instructions to read out or alter the content of a Boolean register are taken as basic instructions and Boolean registers are taken as the components of an execution environment turned out to be useful to rigorous investigations of issues relating to non-uniform computational complexity and algorithm efficiency (see e.g. [8, 10]). We expect that the instantiation presented in this paper can be useful to rigorous investigations relating to uniform computational complexity and algorithm efficiency.

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 [5]. 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 execution environment components. In the current paper, a much less involved presentation is obtained by covering only the case where the execution environment components are Turing tapes and the basic instructions are instructions to read out or alter the content of the cell of a Turing tape under the tape's head and to optionally move the head in either direction by one cell.

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, introduce the notion of a single-tape Turing-machine program in the setting, give a result concerning the computational power of such programs, and give a result relating the complexity class **P** to the functions that can be computed by such programs in polynomial time. We also give a simple example of a single-tape Turing-machine program. This example is only given to illustrate the close resemblance of such programs to transition functions of Turing machines. The notation that is used for Turing-machine programs is intended for theoretical purposes and not for actual programming.

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

This paper is organized as follows. First, we introduce program algebra (Section 2) and basic thread algebra (Section 3) and extend their combination to make precise which behaviours are produced by instruction sequences under execution (Section 4). Next, we present the instantiation of the resulting theory in which all possible instructions to read out or alter the content of the cell of a Turing tape under the tape's head and to optionally move the head in either direction by one cell are taken as basic instructions (Section 5), introduce an algebraic theory of Turing-tape families (Section 6), and extend the combination of the theories presented in the two preceding sections to deal with the interaction between the behaviours of instruction sequences under execution and Turing tapes (Section 7). Then, we formalize in the setting of the resulting theory what it means that a given instruction sequence computes a given partial function on bit strings (Section 8) and give as an example an instruction sequence that computes the non-zeroness test function (Section 9). Finally, we make some concluding remarks (Section 10).

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. [16, 23, 27] for notions concerning algebraic specification and [2, 19, 24] for notions concerning computability and computational complexity.

Sections 2–4 are largely shortened versions of Sections 2–4 of [12], which, in turn, draw from the preliminary sections of several earlier papers.

2 Program Algebra

In this section, we introduce PGA (ProGram Algebra). The starting-point of PGA is the perception of a program 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. 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 PGA has been to achieve simple syntax and semantics, while maintaining the expressive power of arbitrary finite control.

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 the instruction sequences considered in PGA. The intuition is that the execution of a basic instruction may modify a state and must produce the

Boolean value 0 or 1 as reply at its completion. The actual reply may be state-dependent. In applications of PGA, the instructions taken as basic instructions vary from instructions relating to Boolean registers via instructions relating to Turing tapes to machine language instructions of actual computers.

The set of instructions of which the instruction sequences considered in PGA 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*.

Primitive instructions are the elements of the instruction sequences considered in PGA. On execution of such an instruction sequence, these primitive instructions 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 1 is produced and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one — if there is no primitive instruction to proceed with, inaction occurs;
- the effect of a negative test instruction $-a$ is the same as the effect of $+a$, but with the role of the value produced reversed;
- the effect of a plain basic instruction a is the same as the effect of $+a$, but execution always proceeds as if 1 is produced;
- the effect of a forward jump instruction $\#l$ is that execution proceeds with the lth next primitive instruction — if l equals 0 or there is no primitive instruction to proceed with, inaction occurs;
- the effect of the termination instruction $!$ is that execution terminates.

Inaction occurs if no more basic instructions are executed, but execution does not terminate.

PGA has one sort: the sort **IS** of *instruction sequences*. We make this sort explicit to anticipate the need for many-sortedness later on. To build terms of sort **IS**, PGA has the following constants and operators:

- for each $u \in \mathcal{I}$, the *instruction* constant $u : \to \mathbf{IS}$;
- the binary *concatenation* operator $\_ ; \_ : \mathbf{IS} \times \mathbf{IS} \to \mathbf{IS}$;
- the unary *repetition* operator $\_^{\omega} : \mathbf{IS} \to \mathbf{IS}$.

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. The instruction sequence represented by a closed term of the form $t \,;\, t'$ is the instruction sequence represented by t concatenated with the instruction sequence represented by t'.[3] The instruction sequence represented by a closed term of the form t^{ω} is the instruction sequence represented by t concatenated infinitely many times with itself. A closed PGA term represents a finite instruction sequence if and only if it is a closed repetition-free PGA term.

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 $\mathbb{N}_1$.[4] 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 $t^{n+1} = t \,;\, t^n$.

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 and making all jumps as short as possible. Moreover, $t = t'$ is derivable from PGA1–PGA4 iff t and t' represent the same instruction sequence. We write $\mathrm{PGA}^{\mathrm{isc}}$ for the algebraic

[3] The concatenation of an infinite sequence with a finite or infinite sequence yields the former sequence.

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

Table 1: 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
$\#k{+}1 ; u_1 ; \ldots ; u_k ; \#0 = \#0 ; u_1 ; \ldots ; u_k ; \#0$	PGA5
$\#k{+}1 ; u_1 ; \ldots ; u_k ; \#l = \#l{+}k{+}1 ; u_1 ; \ldots ; u_k ; \#l$	PGA6
$(\#l{+}k{+}1 ; u_1 ; \ldots ; u_k)^\omega = (\#l ; u_1 ; \ldots ; u_k)^\omega$	PGA7
$\#l{+}k{+}k'{+}2 ; u_1 ; \ldots ; u_k ; (v_1 ; \ldots ; v_{k'+1})^\omega =$ $\#l{+}k{+}1 ; u_1 ; \ldots ; u_k ; (v_1 ; \ldots ; v_{k'+1})^\omega$	PGA8

theory whose sorts, constants and operators are those of PGA, but whose axioms are PGA1–PGA4.

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 the axioms of $\mathrm{PGA}^{\mathrm{isc}}$. This model, which is described in detail in, for example, [6], is an initial model of the axioms of $\mathrm{PGA}^{\mathrm{isc}}$. 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 BTA (Basic Thread Algebra) 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 Boolean 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*. We make this sort explicit to anticipate the need for many-sortedness later on. 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 $\_ \trianglelefteq \alpha \trianglerighteq \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

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 thread represented by a closed term of the form $t \trianglelefteq \alpha \trianglerighteq t'$ models the behaviour that first performs α, and then proceeds as the behaviour modeled by the thread represented by t if the reply from the execution environment is 1 and proceeds as the behaviour modeled by the thread represented by t' if the reply from the execution environment is 0. Performing tau, which is considered performing an internal action, always leads to the reply 1. The thread represented by S models the behaviour that does nothing else but terminate and the thread represented by D models the behaviour that is inactive, i.e. it performs no more basic actions and it does not terminate.

BTA has only one axiom. This axiom is given in Table 2.

Table 2: Axiom of BTA

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

Using the abbreviation introduced above, it can also be written as follows: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

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 a finite upper bound to the number of basic actions that it can perform, can be defined by means of a set of recursion equations (see e.g. [5]).

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. AIP (Approximation Induction Principle) 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.

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
$\pi_{n+1}(x \unlhd \alpha \unrhd y) = \pi_n(x) \unlhd \alpha \unrhd \pi_n(y)$	PR4
$\bigwedge_{n \geq 0} \pi_n(x) = \pi_n(y) \;\Rightarrow\; x = y$	AIP

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

By AIP, we have to deal in BTA$^\infty$ with conditional equational formulas with a countably infinite number of premises. Therefore, infinitary conditional equational logic is used in deriving equations from the axioms of BTA$^\infty$. A complete inference system for infinitary conditional equational logic can be found in, for example, [26].

The depth of a finite thread is the maximum number of basic actions that the thread 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:

$$
\begin{aligned}
&\mathit{depth}(\mathsf{S}) = 0 \;, \\
&\mathit{depth}(\mathsf{D}) = 0 \;, \\
&\mathit{depth}(t \unlhd \alpha \unrhd t') = \max\{\mathit{depth}(t), \mathit{depth}(t')\} + 1 \;.
\end{aligned}
$$

4 Thread Extraction and Behavioural Congruence

In this section, we make precise in the setting of BTA$^\infty$ which behaviours are produced by PGA instruction sequences under execution and introduce the notion of behavioural congruence on PGA instruction sequences.

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 BTA$^\infty$ and extend the combination with the *thread extraction* operator $|\_| : \mathbf{IS} \to \mathbf{T}$ and the axioms given in Table 4.

Table 4: Axioms for the thread extraction operator

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

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 PGA/BTA$^\infty$ for the combination of PGA and BTA$^\infty$ extended with the thread extraction operator and the axioms for the thread extraction operator.

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 the axioms of PGA and TE1–TE13. By contrast, $|t| = \mathsf{D}$ is not derivable from the axioms of PGA$^{\mathrm{isc}}$ 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 the axioms of PGA$^{\mathrm{isc}}$ and TE1–TE13.

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 the axioms of PGA, TE1–TE13, the axioms of BTA, and PR1–PR4. 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: $\bigwedge_{n \geq 0} \pi_n(|t|) = \pi_n(|t'|) \Rightarrow |t| = |t'|$.

The following proposition, proved in [6], puts the expressiveness of PGA in terms of producible behaviours.

Proposition 1 *Let $\mathcal{M}$ be a model of* PGA/BTA$^\infty$*. Then, for each element p from the domain associated with the sort $\mathbf{T}$ in $\mathcal{M}$, there exists a closed* PGA *term t such that p is the interpretation of $|t|$ in $\mathcal{M}$ iff p is a component of the solution of a finite set of recursion equations $\{V = t_V \mid V \in \mathcal{V}\}$, where $\mathcal{V}$ is a set of variables of sort $\mathbf{T}$ and each t_V is a* BTA *term that is not a variable and contains only variables from $\mathcal{V}$.*

More results on the expressiveness of PGA can be found in [6].

PGA instruction sequences are behaviourally equivalent if they produce the same behaviour under execution. Behavioural equivalence is not a congruence. Instruction sequences are behaviourally congruent if they produce the same behaviour irrespective of the way they are entered and the way they are left.

Let t and t' be closed PGA terms. Then:

- t and t' are *behaviourally equivalent*, written $t \equiv_{\mathsf{be}} t'$, if $|t| = |t'|$ is derivable from the axioms of PGA/BTA$^\infty$;
- t and t' are *behaviourally congruent*, written $t \cong_{\mathsf{bc}} t'$, if, for each $l, n \in \mathbb{N}$, $\#l \,;\, t \,;\, !^n \equiv_{\mathsf{be}} \#l \,;\, t' \,;\, !^n$.[5]

Behavioural congruence is the largest congruence contained in behavioural equivalence.

5 The Case of Instructions for Turing Tapes

In this section, we present the instantiation of PGA in which the instructions taken as basic instructions are all possible instructions to read out or alter the

[5]We use the convention that $t \,;\, t'^0$ stands for t.

content of the cell of a Turing tape under the tape's head and to optionally move the head in either direction by one cell.

The instructions concerned are meant for Turing tapes of which each cell contains a symbol from the input alphabet $\{0,1\}$ or the symbol $\sqcup$. Turing proposed computing machines with a tape of which each cell contains a symbol from a finite alphabet, the so-called tape alphabet, that includes the input alphabet $\{0,1\}$ and the symbol $\sqcup$ (see [25]).[6] The tape alphabet may differ from one machine to another. The choice between the tape alphabet $\{0,1,\sqcup\}$ and any tape alphabet that includes $\{0,1,\sqcup\}$ is rather arbitrary because it has no effect on the computability and the order-of-magnitude time complexity of partial functions from $(\{0,1\}^*)^n$ to $\{0,1\}^*$ $(n \geq 0)$. We have chosen for the tape alphabet $\{0,1,\sqcup\}$ because it allows of presenting part of the material to come in a more comprehensible manner.

In the present instantiation of PGA, it is assumed that a fixed but arbitrary set $\mathcal{F}$ of *foci* has been given. Foci serve as names of Turing tapes.

The set of basic instructions used in this instantiation consists of the following:

> for each $p : \{0,1,\sqcup\} \to \{0,1\}$, $q : \{0,1,\sqcup\} \to \{0,1,\sqcup\}$, $d \in \{-1,0,1\}$, and $f \in \mathcal{F}$, a *basic Turing-tape instruction* $f.p/(q,d)$.

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

Each basic Turing-tape instruction consists of two parts separated by a dot. The part on the left-hand side of the dot plays the role of the name of a Turing tape and the part on the right-hand side of the dot plays the role of an operation to be carried out on the named Turing tape when the instruction is executed. The intuition is basically that carrying out the operation concerned produces as a reply 0 or 1 depending on the content of the cell under the head of the named Turing tape, modifies the content of this cell, and optionally moves the head in either direction by one cell. More precisely, the execution of a basic Turing-tape instruction $f.p/(q,d)$ has the following effects:

- if the content of the cell under the head of the Turing tape named f is b when the execution of $f.p/(q,d)$ starts, then the reply produced on termination of the execution of $f.p/(q,d)$ is $p(b)$;

[6] In many publications in which Turing machines are defined, the input alphabet may even be any non-empty finite set of symbols.

- if the content of the cell under the head of the Turing tape named f is b when the execution of $f.p/(q,d)$ starts, then the content of this cell is $q(b)$ when the execution of $f.p/(q,d)$ terminates;
- if the cell under the head of the Turing tape named f is the ith cell of the tape when the execution of $f.p/(q,d)$ starts, then the cell under the head of this Turing tape is the $\max(i+d,1)$th cell when the execution of $f.p/(q,d)$ terminates.

The execution of $f.p/(q,d)$ has no effect on Turing tapes other than the one named f.

We write $[\mathrm{PGA/BTA}^{\infty}](\mathcal{A}_{\mathrm{tt}})$ for $\mathrm{PGA/BTA}^{\infty}$ with $\mathcal{A}$ instantiated by $\mathcal{A}_{\mathrm{tt}}$. Notice that $[\mathrm{PGA/BTA}^{\infty}](\mathcal{A}_{\mathrm{tt}})$ is itself parameterized by a set of foci.

Some functions from $\{0,1,\sqcup\}$ to $\{0,1,\sqcup\}$ are:

- the function $0?$, satisfying $0?(0)=1$ and $0?(1)=0$ and $0?(\sqcup)=0$;
- the function $1?$, satisfying $1?(0)=0$ and $1?(1)=1$ and $1?(\sqcup)=0$;
- the function $\sqcup?$, satisfying $\sqcup?(0)=0$ and $\sqcup?(1)=0$ and $\sqcup?(\sqcup)=1$;
- the function 0, satisfying $0(0)=0$ and $0(1)=0$ and $0(\sqcup)=0$;
- the function 1, satisfying $1(0)=1$ and $1(1)=1$ and $0(\sqcup)=1$;
- the function $\sqcup$, satisfying $1(0)=\sqcup$ and $1(1)=\sqcup$ and $0(\sqcup)=\sqcup$;
- the function i, satisfying $\mathrm{i}(0)=0$ and $\mathrm{i}(1)=1$ and $0(\sqcup)=\sqcup$;
- the function c, satisfying $\mathrm{c}(0)=1$ and $\mathrm{c}(1)=0$ and $0(\sqcup)=\sqcup$.

The first five of these functions are also functions from $\{0,1,\sqcup\}$ to $\{0,1\}$.

For some instances of $p/(q,d)$, we introduce a special notation. We write:

$$
\begin{array}{lll}
\mathsf{test}{:}0 \text{ for } 0?/(\mathrm{i},0)\,, & \mathsf{set}{:}0{:}d \text{ for } 1/(0,d)\,, & \mathsf{skip}{:}d \text{ for } 1/(\mathrm{i},d)\,, \\
\mathsf{test}{:}1 \text{ for } 1?/(\mathrm{i},0)\,, & \mathsf{set}{:}1{:}d \text{ for } 1/(1,d)\,, & \\
\mathsf{test}{:}\sqcup \text{ for } \sqcup?/(\mathrm{i},0)\,, & \mathsf{set}{:}\sqcup{:}d \text{ for } 1/(\sqcup,d)\,, &
\end{array}
$$

where $d \in \{-1,0,1\}$.

6 Turing-Tape Families

PGA instruction sequences under execution may interact with the named services from a family of services provided by their execution environment. In applications of PGA, the services provided by an execution environment vary from Boolean registers via Turing tapes to random access memories of actual computers.[7] In this section, we consider service families in which the services are Turing tapes and introduce an algebraic theory of Turing-tape families called TTFA (Turing-Tape Family Algebra).

A *Turing-tape state* is a pair (τ, i), where $\tau : \mathbb{N}_1 \to \{0, 1, \sqcup\}$ and $i \in \mathbb{N}_1$, satisfying the condition that, for some $j \in \mathbb{N}_1$, for all $k \in \mathbb{N}$, $\tau(j + k) = \sqcup$. We write $\mathcal{S}$ for the set of all Turing-tape states.

Let (τ, i) be a Turing-tape state. Then, for all $j \in \mathbb{N}_1$, $\tau(j)$ is the content of the jth cell of the Turing tape concerned and the ith cell is the cell under its head.

Our Turing tapes are one-way infinite tapes. Turing proposed computing machine with two-way infinite tapes (see [25]). In many publications in which Turing machine are defined, Turing machines are a variant of Turing's computing machines with one or more one-way infinite tapes (cf. the textbooks [1, 2, 18, 19, 20, 24]). The choice between one-way infinite tapes and two-way infinite tapes is rather arbitrary because it has no effect on the computability and the order-of-magnitude time complexity of partial functions from $(\{0, 1\}^*)^n$ to $\{0, 1\}^*$ $(n \geq 0)$. We have chosen for one-way infinite tapes because it allows of presenting part of the material to come in a more comprehensible manner.

In Section 7, we will use the notation $(\tau{:}i{\mapsto}b)$. For each $\tau{:}\mathbb{N}_1 \to \{0, 1, \sqcup\}$, $i \in \mathbb{N}_1$, and $b \in \{0, 1, \sqcup\}$, $(\tau : i \mapsto b)$ is defined as follows: $(\tau : i \mapsto b)(i) = b$ and, for all $j \in \mathbb{N}_1$ with $j \neq i$, $(\tau : i \mapsto b)(j) = \tau(j)$.

In TTFA, as in $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})$, it is assumed that a fixed but arbitrary set $\mathcal{F}$ of foci has been given.

TTFA has one sort: the sort **TTF** of *Turing-tape families*. To build terms of sort **TTF**, TTFA has the following constants and operators:

- the *empty Turing-tape family* constant $\emptyset : \to \mathbf{TTF}$;
- for each $f \in \mathcal{F}$ and $s \in \mathcal{S} \cup \{*\}$, the *singleton Turing-tape family* constant $f.\mathtt{tt}(s) : \to \mathbf{TTF}$;

[7] A Boolean register consists of a single cell that contains a symbol from the alphabet $\{0, 1\}$. Carrying out an operation on a Boolean register produces as a reply 0 or 1, depending on the content of the cell, and/or modifies the content of the cell.

- the binary *Turing-tape family composition* operator $\_ \oplus \_ : \mathbf{TTF} \times \mathbf{TTF} \to \mathbf{TTF}$;
- for each $F \subseteq \mathcal{F}$, the unary *encapsulation* operator $\partial_F : \mathbf{TTF} \to \mathbf{TTF}$.

We assume that there are infinitely many variables of sort **TTF**, including u, v, w. We use infix notation for the Turing-tape family composition operator.

The Turing-tape family denoted by $\emptyset$ is the empty Turing-tape family. The Turing-tape family denoted by a closed term of the form $f.\mathsf{tt}(s)$, where $s \in \mathcal{S}$, consists of one named Turing tape only, the Turing tape concerned is an operative Turing tape named f whose state is s. The Turing-tape family denoted by a closed term of the form $f.\mathsf{tt}(*)$ consists of one named Turing tape only, the Turing tape concerned is an inoperative Turing tape named f. The Turing-tape family denoted by a closed term of the form $t \oplus t'$ consists of all named Turing tapes that belong to either the Turing-tape family denoted by t or the Turing-tape family denoted by t'. In the case where a named Turing tape from the Turing-tape family denoted by t and a named Turing tape from the Turing-tape family denoted by t' have the same name, they collapse to an inoperative Turing tape with the name concerned. The Turing-tape family denoted by a closed term of the form $\partial_F(t)$ consists of all named Turing tapes with a name not in F that belong to the Turing-tape family denoted by t.

An inoperative Turing tape can be viewed as a Turing tape whose state is unavailable. Carrying out an operation on an inoperative Turing tape is impossible.

The axioms of TTFA are given in Table 5.

Table 5: Axioms of TTFA

$u \oplus \emptyset = u$	TTFC1	$\partial_F(\emptyset) = \emptyset$		TTFE1
$u \oplus v = v \oplus u$	TTFC2	$\partial_F(f.\mathsf{tt}(s)) = \emptyset$	if $f \in F$	TTFE2
$(u \oplus v) \oplus w = u \oplus (v \oplus w)$	TTFC3	$\partial_F(f.\mathsf{tt}(s)) = f.\mathsf{tt}(s)$	if $f \notin F$	TTFE3
$f.\mathsf{tt}(s) \oplus f.\mathsf{tt}(s') = f.\mathsf{tt}(*)$	TTFC4	$\partial_F(u \oplus v) = \partial_F(u) \oplus \partial_F(v)$		TTFE4

In this table, f stands for an arbitrary focus from $\mathcal{F}$, F stands for an arbitrary subset of $\mathcal{F}$, and s and s' stand for arbitrary members of $\mathcal{S} \cup \{*\}$. These axioms simply formalize the informal explanation given above.

The following proposition, proved in [6], is a representation result for closed TTFA terms.

Proposition 2 *For all closed* TTFA *terms t, for all $f \in \mathcal{F}$, either $t = \partial_{\{f\}}(t)$ is derivable from the axioms of* TTFA *or there exists an $s \in \mathcal{S} \cup \{*\}$ such that $t = f.\mathsf{tt}(s) \oplus \partial_{\{f\}}(t)$ is derivable from the axioms of* TTFA.

In Section 8, we will use the notation $\bigoplus_{i=1}^{n} t_i$. For each $i \in \mathbb{N}_1$, let t_i be a terms of sort **TTF**. Then, for each $n \in \mathbb{N}_1$, the term $\bigoplus_{i=1}^{n} t_i$ is defined by induction on n as follows: $\bigoplus_{i=1}^{1} t_i = t_1$ and $\bigoplus_{i=1}^{n+1} t_i = \bigoplus_{i=1}^{n} t_i \oplus t_{n+1}$. We use the convention that $\bigoplus_{i=1}^{n} t_i$ stands for $\emptyset$ if $n = 0$.

7 Interaction of Threads with Turing Tapes

If instructions from $\mathcal{A}_{\mathrm{tt}}$ are taken as basic instructions, a PGA instruction sequence under execution may interact with named Turing tapes from a family of Turing tapes provided by its execution environment. In line with this kind of interaction, a thread may perform a basic action basically for the purpose of modifying the content of a named Turing tape or receiving a reply value that depends on the content of a named Turing tape. In this section, we introduce related operators.

We combine PGA/BTA$^\infty(\mathcal{A}_{\mathrm{tt}})$ with TTFA and extend the combination with the following operators for interaction of threads with Turing tapes:

- the binary *use* operator $\_ \,/\, \_ : \mathbf{T} \times \mathbf{TTF} \to \mathbf{T}$;
- the binary *apply* operator $\_ \bullet \_ : \mathbf{T} \times \mathbf{TTF} \to \mathbf{TTF}$;
- the unary *abstraction* operator $\tau_{\mathsf{tau}} : \mathbf{T} \to \mathbf{T}$;

and the axioms given in Tables 6.[8]

In these tables, f stands for an arbitrary focus from $\mathcal{F}$, p stands for an arbitrary function from $\{0, 1, \sqcup\}$ to $\{0, 1\}$, q stands for an arbitrary function from $\{0, 1, \sqcup\}$ to $\{0, 1, \sqcup\}$, d stands for an arbitrary member of $\{-1, 0, 1\}$, τ stands for an arbitrary function from $\mathbb{N}_1$ to $\{0, 1, \sqcup\}$, i stands for an arbitrary natural number from $\mathbb{N}_1$, n stands for an arbitrary natural number from $\mathbb{N}$, and t and t' stand for arbitrary terms of sort **TTF**. We use infix notation for the use and apply operators. We write [PGA/BTA$^\infty$]$(\mathcal{A}_{\mathrm{tt}})$/TTI for the combination of [PGA/BTA$^\infty$]$(\mathcal{A}_{\mathrm{tt}})$ and TTFA extended with the use operator, the apply operator, the abstraction operator, and the axioms for these operators.

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

Table 6: Axioms for the use, apply and abstraction operator

$\mathsf{S} / u = \mathsf{S}$	U1
$\mathsf{D} / u = \mathsf{D}$	U2
$(\mathsf{tau} \circ x) / u = \mathsf{tau} \circ (x / u)$	U3
$(x \unlhd f.p/(q,d) \unrhd y) / \partial_{\{f\}}(u) = (x / \partial_{\{f\}}(u)) \unlhd f.p/(q,d) \unrhd (y / \partial_{\{f\}}(u))$	U4
$(x \unlhd f.p/(q,d) \unrhd y) / (f.\mathsf{tt}(\tau,i) \oplus \partial_{\{f\}}(u))$	
$\quad = \mathsf{tau} \circ (x / (f.\mathsf{tt}((\tau : i \mapsto q(\tau(i))), \max(i+d,1)) \oplus \partial_{\{f\}}(u)))$ $\;\mathsf{if}\; p(\tau(i)) = 1$	U5
$(x \unlhd f.p/(q,d) \unrhd y) / (f.\mathsf{tt}(\tau,i) \oplus \partial_{\{f\}}(u))$	
$\quad = \mathsf{tau} \circ (y / (f.\mathsf{tt}((\tau : i \mapsto q(\tau(i))), \max(i+d,1)) \oplus \partial_{\{f\}}(u)))$ $\;\mathsf{if}\; p(\tau(i)) = 0$	U6
$(x \unlhd f.p/(q,d) \unrhd y) / (f.\mathsf{tt}(*) \oplus \partial_{\{f\}}(u)) = \mathsf{D}$	U7
$\pi_n(x / u) = \pi_n(x) / u$	U8
$\mathsf{S} \bullet u = u$	A1
$\mathsf{D} \bullet u = \emptyset$	A2
$(\mathsf{tau} \circ x) \bullet u = \mathsf{tau} \circ (x \bullet u)$	A3
$(x \unlhd f.p/(q,d) \unrhd y) \bullet \partial_{\{f\}}(u) = \emptyset$	A4
$(x \unlhd f.p/(q,d) \unrhd y) \bullet (f.\mathsf{tt}(\tau,i) \oplus \partial_{\{f\}}(u))$	
$\quad = x \bullet (f.\mathsf{tt}((\tau : i \mapsto q(\tau(i))), \max(i+d,1)) \oplus \partial_{\{f\}}(u))$ $\;\mathsf{if}\; p(\tau(i)) = 1$	A5
$(x \unlhd f.p/(q,d) \unrhd y) \bullet (f.\mathsf{tt}(\tau,i) \oplus \partial_{\{f\}}(u))$	
$\quad = y \bullet (f.\mathsf{tt}((\tau : i \mapsto q(\tau(i))), \max(i+d,1)) \oplus \partial_{\{f\}}(u))$ $\;\mathsf{if}\; p(\tau(i)) = 0$	A6
$(x \unlhd f.p/(q,d) \unrhd y) \bullet (f.\mathsf{tt}(*) \oplus \partial_{\{f\}}(u)) = \emptyset$	A7
$\bigwedge_{k \geq n} t[\pi_k(x)/z] = t'[\pi_k(y)/z] \;\Rightarrow\; t[x/z] = t'[y/z]$	A8
$\tau_{\mathsf{tau}}(\mathsf{S}) = \mathsf{S}$	C1
$\tau_{\mathsf{tau}}(\mathsf{D}) = \mathsf{D}$	C2
$\tau_{\mathsf{tau}}(\mathsf{tau} \circ x) = \tau_{\mathsf{tau}}(x)$	C3
$\tau_{\mathsf{tau}}(x \unlhd f.p/(q,d) \unrhd y) = \tau_{\mathsf{tau}}(x) \unlhd f.p/(q,d) \unrhd \tau_{\mathsf{tau}}(y)$	C4
$\bigwedge_{k \geq 0} \tau_{\mathsf{tau}}(\pi_k(x)) = \tau_{\mathsf{tau}}(\pi_k(y)) \;\Rightarrow\; \tau_{\mathsf{tau}}(x) = \tau_{\mathsf{tau}}(y)$	C5

Axioms U1–U7 and A1–A7 formalize the informal explanation of the use operator and the apply operator given below and in addition stipulate what is the result of apply if an unavailable focus is involved (A4) and what is the result of use and apply if an inoperative Turing tape is involved (U7 and A7). Axioms U8 and A8 allow of reasoning about infinite threads, and therefore about the behaviour produced by infinite instruction sequences under execution, in the context of use and apply, respectively.

On interaction between a thread and a Turing tape, the thread affects the Turing tape and the Turing tape affects the thread. The use operator concerns the effects of Turing tapes on threads and the apply operator concerns the effects of threads on Turing tapes. The thread denoted by a closed term of the form $t \,/\, t'$ and the Turing-tape family denoted by a closed term of the form $t \bullet t'$ are the thread and Turing-tape family, respectively, that result from carrying out the operation that is part of each basic action performed by the thread denoted by t on the Turing tape in the Turing-tape family denoted by t' with the focus that is part of the basic action as its name. When the operation that is part of a basic action performed by a thread is carried out on a Turing tape, the content of the Turing tape is modified according to the operation concerned and the thread is affected as follows: the basic action turns into the internal action tau and the two ways to proceed reduce to one on the basis of the reply value produced according to the operation concerned.

With the use operator the internal action tau is left as a trace of each basic action that has led to carrying out an operation on a Turing tape. The abstraction operator serves to abstract fully from such internal activity by concealing tau. Axioms C1–C4 formalizes the concealment of tau. Axiom C5 allows of reasoning about infinite threads in the context of abstraction.

The following two theorems are elimination results for closed $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$ terms.

Theorem 1 *For all closed* $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$ *terms* t *of sort* $\mathbf{T}$ *in which all subterms of sort* $\mathbf{IS}$ *are repetition-free, there exists a closed* $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})$ *term* t' *of sort* $\mathbf{T}$ *such that* $t = t'$ *is derivable from the axioms of* $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$.

Proof: It is easy to prove by structural induction that, for all closed repetition-free $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})$ terms s of sort $\mathbf{IS}$, there exists a closed $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})$ term s' of sort $\mathbf{T}$ such that $|s| = s'$ is derivable from the axioms of $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})$. Therefore, it is sufficient to prove the proposition for all closed $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$ terms t of sort $\mathbf{T}$ in which no subterms of sort $\mathbf{IS}$ occur. This is proved similarly to part (1) of Theorem 3.1 from [6]. □

Theorem 2 *For all closed* $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$ *terms* t *of sort* $\mathbf{TTF}$ *in which all subterms of sort* $\mathbf{IS}$ *are repetition-free, there exists a closed*

$[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})$ *term* t' *of sort* **TTF** *such that* $t = t'$ *is derivable from the axioms of* $[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$.

Proof: As in the proof of Theorem 1, it is sufficient to prove the proposition for all closed $[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$ terms t of sort **TTF** in which no subterms of sort **IS** occur. This is proved similarly to part (2) of Theorem 3.1 from [6]. □

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

In this section, we make precise in the setting of the algebraic theory $[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$ 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 write $\mathcal{F}^k_{\mathsf{tt}}$, where $k \in \mathbb{N}_1$, for the set $\{\mathsf{t}{:}i \mid 1 \leq i \leq k\}$ of foci. We write $[[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}](\mathcal{F}^k_{\mathsf{tt}})$ for $[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$ with $\mathcal{F}$ instantiated by $\mathcal{F}^k_{\mathsf{tt}}$.

Below, we use the function $\underline{\mathsf{c}}{:}\{\tau{:}\mathbb{N}_1 \to \{0,1,\sqcup\} \mid (\tau,1) \in \mathcal{S}\} \to \{0,1,\sqcup\}^*$ for extracting the content of a Turing tape. This function is defined as follows:

for all $n \in \mathbb{N}_1$, for all $b_1,\ldots,b_n \in \{0,1,\sqcup\}$,
$\underline{\mathsf{c}}(\tau) = b_1 \ldots b_n$ iff $\tau(i) = b_i$ for all $i \leq n$, $\tau(i) = \sqcup$ for all $i > n$, and $\tau(n) \neq \sqcup$;

$\underline{\mathsf{c}}(\tau) = \epsilon$ iff $\tau(i) = \sqcup$ for all $i \geq 1$.[9]

Let $k \in \mathbb{N}_1$, let t be a closed $[[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}](\mathcal{F}^k_{\mathsf{tt}})$ term of sort **IS**, let $n \in \mathbb{N}$, let $F : (\{0,1\}^*)^n \rightharpoonup \{0,1\}^*$,[10] and let $T : \mathbb{N} \to \mathbb{N}$. Then t *computes* F *with* k *tapes in time* T if:

- for all $w_1,\ldots,w_n \in \{0,1\}^*$ such that $F(w_1,\ldots,w_n)$ is defined, there exist $(\tau'_1,i_1),\ldots,(\tau'_{k-1},i_{k-1}) \in \mathcal{S}$ such that

$$|t| \bullet \bigoplus_{j=1}^{k} \mathsf{t}{:}j.\mathsf{tt}(\tau_j,1) = \bigoplus_{j=1}^{k-1} \mathsf{t}{:}j.\mathsf{tt}(\tau'_j,i_j) \oplus \mathsf{t}{:}k.\mathsf{tt}(\tau'_k,1)\;,$$

$$\mathit{depth}(|t| \mathbin{/} \bigoplus_{j=1}^{k} \mathsf{t}{:}j.\mathsf{tt}(\tau_j,1)) \leq T(\mathit{len}(w_1) + \ldots + \mathit{len}(w_n))\;,$$

where

[9] We write ϵ for the empty string.

[10] We write $f : A \rightharpoonup B$ to indicate that f is a partial function from A to B.

τ_1 is the unique $\tau : \mathbb{N}_1 \to \{0, 1, \sqcup\}$ with $(\tau, 1) \in \mathcal{S}$ and $\underline{\mathsf{c}}(\tau) = w_1 \sqcup \ldots \sqcup w_n$,

for $j \neq 1$, τ_j is the unique $\tau : \mathbb{N}_1 \to \{0, 1, \sqcup\}$ with $(\tau, 1) \in \mathcal{S}$ and $\underline{\mathsf{c}}(\tau) = \epsilon$,

τ'_k is the unique $\tau : \mathbb{N}_1 \to \{0, 1, \sqcup\}$ with $(\tau, 1) \in \mathcal{S}$ and $\underline{\mathsf{c}}(\tau) = F(w_1, \ldots, w_n)$;

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

$$|t| \bullet \bigoplus_{j=1}^{k} \mathsf{t}{:}j.\mathsf{tt}(\tau_j, 1) = \emptyset ,$$

  where

τ_1 is the unique $\tau : \mathbb{N}_1 \to \{0, 1, \sqcup\}$ with $(\tau, 1) \in \mathcal{S}$ and $\underline{\mathsf{c}}(\tau) = w_1 \sqcup \ldots \sqcup w_n$,

for $j \neq 1$, τ_j is the unique $\tau : \mathbb{N}_1 \to \{0, 1, \sqcup\}$ with $(\tau, 1) \in \mathcal{S}$ and $\underline{\mathsf{c}}(\tau) = \epsilon$.

We say that t *computes* F *in time* T if there exists a $k \in \mathbb{N}_1$ such that t computes F with k tapes in time T, and we say that t *computes* F if there exists a $T : \mathbb{N} \to \mathbb{N}$ such that t computes F in time T.

With the above definition, we can establish whether an instruction sequence of the kind considered in $[[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}](\mathcal{F}^k_{\mathsf{tt}})$ $(k \in \mathbb{N}_1)$ computes a given partial function from $(\{0,1\}^*)^n$ to $\{0,1\}^*$ $(n \in \mathbb{N})$ by equational reasoning using the axioms of $[[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}](\mathcal{F}^k_{\mathsf{tt}})$.

A *single-tape Turing-machine program* is a closed $[[\mathrm{PGA/BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}](\mathcal{F}^1_{\mathsf{tt}})$ term of sort **IS** that is of the form $(t_1 \,;\ldots;\, t_n)^\omega$, where each t_i is of the form

$$\begin{gathered}\mathsf{test}{:}0 \,;\, \#3 \,;\, \mathsf{set}{:}b_0{:}d_0 \,;\, u_0 \,; \\ \mathsf{test}{:}1 \,;\, \#3 \,;\, \mathsf{set}{:}b_1{:}d_1 \,;\, u_1 \,; \\ \mathsf{test}{:}\sqcup \,;\, \#3 \,;\, \mathsf{set}{:}b_\sqcup{:}d_\sqcup \,;\, u_\sqcup ,\end{gathered}$$

where $b_0, b_1, b_\sqcup \in \{0, 1, \sqcup\}$, $d_0, d_1, d_\sqcup \in \{-1, 0, 1\}$, and

u_0 is of the form $\#l$ with $l \in \{12 \cdot i + 9 \mid 0 \leq i < n\}$ or $\#0$ or $!$,

u_1 is of the form $\#l$ with $l \in \{12 \cdot i + 5 \mid 0 \leq i < n\}$ or $\#0$ or $!$,

$u_\sqcup$ is of the form $\#l$ with $l \in \{12 \cdot i + 1 \mid 0 \leq i < n\}$ or $\#0$ or $!$.

We refrain from defining a k-tape Turing-machine program (for $k > 1$), which is much more involved than defining a single-tape Turing-machine program. However, we remark that the theorems given below go through for k-tape Turing-machine programs.

The following theorem is a result concerning the computational power of single-tape Turing-machine programs.

Theorem 3 *For each $F : (\{0,1\}^*)^n \nrightarrow \{0,1\}^*$, there exists a single-tape Turing-machine program t such that t computes F iff F is Turing-computable.*

Proof: For each $F : (\{0,1\}^*)^n \nrightarrow \{0,1\}^*$, F is Turing-computable iff there exists a Turing machine with a single semi-infinite tape and stay option that computes F. There is an obvious one-to-one correspondence between the transition functions of such Turing machines and single-tape Turing-machine programs by which the Turing machines concerned can be simulated when they are applied to a single tape. Hence, for each $F : (\{0,1\}^*)^n \nrightarrow \{0,1\}^*$, there exists a single-tape Turing-machine program t such that t computes F iff F is Turing-computable. □

Below, we write TMP_{st} for the set of all single-tape Turing-machine programs, and POLY for $\{T \mid T : \mathbb{N} \to \mathbb{N} \land T \text{ is a polynomial function}\}$.

The following theorem is a result relating the complexity class **P** to the functions that can be computed by a single-tape Turing-machine program in polynomial time.

Theorem 4 **P** *is equal to the class of all functions $F : \{0,1\}^* \to \{0,1\}$ for which there exist an $t \in \mathit{TMP}_{st}$ and a $T \in \mathit{POLY}$ such that t computes F in time T.*

Proof: This follows from the proof of Theorem 3 and the fact that, if a function $F : \{0,1\}^* \to \{0,1\}$ is computed on a Turing machine in time T, then the one-to-one correspondence referred to in the proof of Theorem 3 yields for this Turing machine a single-tape Turing-machine program that computes F in a time of $O(T)$. □

We think that Theorems 3 and 4 above provide evidence of the claim that $[\mathrm{PGA}/\mathrm{BTA}^\infty](\mathcal{A}_{\mathrm{tt}})/\mathrm{TTI}$ is a suitable setting for the development of theory in areas such as computability and computational complexity. Moreover, in this setting variations on Turing machines that have not attracted attention yet come into the picture and can be studied.

9 A Turing-Machine Program Example

In this section, we give a simple example of a Turing-machine program. We consider the *non-zeroness test* function $NZT : (\{0,1\}^*)^1 \to \{0,1\}^*$ defined by

$$
\begin{aligned}
NZT(b_1 \ldots b_n) &= 0 \;\text{ if }\; b_1 = 0 \text{ and } \ldots \text{ and } b_n = 0\ ,\\
NZT(b_1 \ldots b_n) &= 1 \;\text{ if }\; b_1 = 1 \;\text{ or }\; \ldots \;\text{ or }\; b_n = 1\ .
\end{aligned}
$$

NZT models the function $nzt : \mathbb{N} \to \mathbb{N}$ defined by $nzt(0) = 0$ and $nzt(k+1) = 1$ with respect to the binary representations of the natural numbers.

We define a Turing-machine program $NZTIS$ that computes NZT as follows:

$$
\begin{aligned}
NZTIS \triangleq (&-\mathsf{test{:}0}\,;\#3\,;\mathsf{set{:}0{:}1}\,;\#33\,;\\
&-\mathsf{test{:}1}\,;\#3\,;\mathsf{set{:}1{:}1}\,;\#29\,;\\
&-\mathsf{test{:}\sqcup}\,;\#3\,;\mathsf{set{:}\sqcup{:}{-1}}\,;\#1\,;\\
&-\mathsf{test{:}0}\,;\#3\,;\mathsf{set{:}\sqcup{:}{-1}}\,;\#33\,;\\
&-\mathsf{test{:}1}\,;\#3\,;\mathsf{set{:}\sqcup{:}{-1}}\,;\#5\,;\\
&-\mathsf{test{:}\sqcup}\,;\#3\,;\mathsf{set{:}0{:}0}\,;!\,;\\
&-\mathsf{test{:}0}\,;\#3\,;\mathsf{set{:}\sqcup{:}{-1}}\,;\#33\,;\\
&-\mathsf{test{:}1}\,;\#3\,;\mathsf{set{:}\sqcup{:}{-1}}\,;\#29\,;\\
&-\mathsf{test{:}\sqcup}\,;\#3\,;\mathsf{set{:}1{:}0}\,;!)^\omega\ .
\end{aligned}
$$

First, the head is moved to the right cell-by-cell until the first cell whose content is $\sqcup$ has been reached and after that the head is moved to the left by one cell. Then, the head is moved to the left cell-by-cell until the first cell of the tape has been reached and on top of that the content of each cell that comes under the head is replaced by $\sqcup$. Finally, the content of the first cell is replaced by 1 if at least one cell with content 1 has been encountered during the moves to the left and the content of the first cell is replaced by 0 if no cell with content 1 has been encountered during the moves to the left.

Because Turing-machine programs closely resemble the transition functions of Turing machines, they have built-in inefficiencies. We use $NZTIS$ to illustrate this. We define an instruction sequence $NZTIS'$ that computes NZT according to the same algorithm in less time than $NZTIS$ as follows:

$$
\begin{aligned}
NZTIS' \triangleq (&+\mathsf{test}{:}\sqcup \,;\, \#3 \,;\, \mathsf{skip}{:}1 \,;\, \#18 \,; \\
&\mathsf{skip}{:}{-1} \,; \\
&-\mathsf{test}{:}0 \,;\, \#3 \,;\, \mathsf{set}{:}\sqcup{:}{-1} \,;\, \#18 \,; \\
&-\mathsf{test}{:}1 \,;\, \#3 \,;\, \mathsf{set}{:}\sqcup{:}{-1} \,;\, \#3 \,; \\
&\mathsf{set}{:}0{:}0 \,;\, ! \,; \\
&+\mathsf{test}{:}\sqcup \,;\, \#3 \,;\, \mathsf{set}{:}\sqcup{:}{-1} \,;\, \#18 \,; \\
&\mathsf{set}{:}1{:}0 \,;\, !)^{\omega} \;.
\end{aligned}
$$

In $NZTIS'$, which is clearly not a single-tape Turing-machine program, all instructions of the form $\mathsf{test}{:}b$ that are redundant or can be made redundant by using instructions of the form $\mathsf{skip}{:}d$ are removed.

In [13], we have presented instruction sequences that compute the restriction of NZT to $\{0,1\}^n$, for $n > 0$. The instruction sequences concerned are instruction sequences that, under execution, can act on Boolean registers instead of Turing tapes.

10 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. The parameterized theory concerned is the basis of a line of research in which issues relating to a wide variety of subjects from computer science have been rigorously investigated thinking in terms of instruction sequences (see [21]). In the presented instantiation of this parameterized theory, all possible instructions to read out or alter the content of the cell of a Turing tape under the tape's head and to optionally move the head in either direction by one cell are taken as basic instructions and Turing tapes are taken as the components of an execution environment.

The instantiated theory provides a setting for the development of theory in areas such as computability and computational complexity that distinguishes itself by offering the possibility of equational reasoning and being more general than the setting provided by a known version of the Turing-machine model of computation. Many known and unknown versions of the Turing-machine model of computation can be dealt with by imposing apposite restrictions.

We have defined the notion of a single-tape Turing-machine program in the setting of the instantiated theory and have provided evidence for the claim that the theory provides a suitable setting for the development of theory in areas such as computability and computational complexity. Single-tape Turing-machine programs and multiple-tape Turing-machine programs make up only small parts of the instruction sequences that can be considered in this setting. This largely explains why it is more general than the setting provided by a known version of the Turing-machine model of computation. From our experience in previous work with a comparable algebraic theory of instruction sequences, with instructions that operate on Boolean registers instead of Turing tapes, we expect that the generality is conducive to the investigation of novel issues in areas such as computability and computational complexity.

The given presentation of the instantiated theory is set up in a way where the introduction of services, the generic kind of execution-environment components from the parameterized theory in question, is circumvented. In [12], the presentation of another instantiation of the same parameterized theory, with instructions that operate on Boolean registers, is set up in the same way. The distinguishing feature of this way of presenting an instantiation of this 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.

The closed terms of the instantiated theory that are of sort **IS** can be considered to constitute a programming language of which the syntax and semantics is defined following an algebraic approach. This approach is more operational than the usual algebraic approach which is among other things followed in [14, 15, 17]. A more operational approach is needed to make it possible to investigate issues in the area of computational complexity.

Broadly speaking, the work presented in this paper is concerned with formalization in subject areas, such as computability and computational complexity, that traditionally relies on a version of the Turing-machine model of computation. To the best of our knowledge, very little work has been done in this area. Three notable exceptions are [3, 22, 28]. However, in those papers, formalization means formalization in a theorem prover (Matita, HOL4, Isabelle/HOL). Little or nothing is said in these papers about the syntax and semantics of the notations used — which are probably the ones that have to be used in the theorem provers. This makes it impracticable to compare the work presented in those papers with our work, but it is of course clear that the approach followed in the work presented in those papers is completely different from the algebraic approach followed in our work.

Acknowledgement

We thank two anonymous referees for their helpful suggestions.

References

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] S. Arora, B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, Cambridge, 2009.

[3] A. Asperti, W. Ricciotti. A Formalization of Multi-tape Turing Machines. *Theoretical Computer Science* 603, 23–42, 2015. doi:10.1016/j.tcs.2015.07.013.

[4] J. A. Bergstra, M. 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] J. A. Bergstra, C. A. Middelburg. Instruction Sequence Processing Operators. *Acta Informatica* 49(3), 139–172, 2012. doi:10.1007/s00236-012-0154-2.

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

[7] J. A. Bergstra, C. A. Middelburg. Instruction Sequence Expressions for the Secure Hash Algorithm SHA-256. 2013. arXiv:1308.0219.

[8] J. A. Bergstra, C. 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.

[9] J. A. Bergstra, C. 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.

[10] J. A. Bergstra, C. A. Middelburg. Instruction Sequence Size Complexity of Parity. *Fundamenta Informaticae* 149(3), 297–309, 2016. doi:10.3233/fi-2016-1450.

[11] J. A. Bergstra, C. A. Middelburg. Instruction Sequences Expressing Multiplication Algorithms. *Scientific Annals of Computer Science* 28(1), 39–66, 2018. doi:10.7561/sacs.2018.1.39.

[12] J. A. Bergstra, C. A. Middelburg. A Short Introduction to Program Algebra with Instructions for Boolean Registers. *Computer Science Journal of Moldova* 26(3), 199–232, 2018.

[13] J. A. Bergstra, C. 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.

[14] M. Broy, W. Dosch, B. Möller, M. Wirsing. GOTOs – A Study in the Algebraic Specification of Programming Languages (Extended Abstract). In W. Brauwer (Ed.) *GI — 11. Jahrestagung*, volume 50 of *Informatik-Fachberichte*, 109–121, 1981. doi:10.1007/978-3-662-01089-1_13.

[15] M. Broy, M. Wirsing, P. 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.

[16] H. Ehrig, B. Mahr. Fundamentals of Algebraic Specification I: Equations and Initial Semantics, *EATCS Monographs on Theoretical Computer Science* 6, 62-88, 1985. doi:10.1007/978-3-642-69962-7_4.

[17] J. A. Goguen, G. Malcolm. *Algebraic Semantics of Imperative Programs*. Foundations of Computing. MIT Press, Cambridge, MA, 1996. doi:10.7551/mitpress/1188.001.0001.

[18] O. Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, Cambridge, 2008. doi:10.1017/CBO9780511804106.

[19] S. Homer, A. L. Selman. *Computability and Complexity Theory*. Springer-Verlag, Berlin, second edition, 2011. doi:10.1007/978-1-4614-0682-2.

[20] D. C. Kozen. *Theory of Computation*. Springer-Verlag, Berlin, 2006. doi:10.1007/1-84628-477-5.

[21] C. A. Middelburg. Instruction Sequences as a Theme in Computer Science. 2015. Springer-Verlag, Berlin, 2006. https://instructionsequence.wordpress.com/.

[22] M. Norrish. Mechanised Computability Theory. In M. van Eekelen, H. Geuvers, J. Schmaltz, F. Wiedijk (Eds.) *Interactive Theorem Proving* (ITP 2011), *Lecture Notes in Computer Science* 6898, 297–311, 2011. doi:10.1007/978-3-642-22863-6_22.

[23] D. Sannella, A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. An EATCS Seriess. Springer-Verlag, 2012. doi:10.1007/978-3-642-17336-3.

[24] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, Boston, MA, third edition, 2013.

[25] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungs Problem. *Proceedings of the London Mathematical Society, Series 2,* 42, 230–265, 1937. doi:10.1112/plms/s2-42.1.230. Correction: *ibid* 43, 544–546, 1938. doi:10.1112/plms/s2-43.6.544.

[26] R. J. van Glabbeek, F. W. Vaandrager. Modular Specification of Process Algebras. *Theoretical Computer Science* 113(2), 293–348, 1993. doi:10.1016/0304-3975(93)90006-F.

[27] M. Wirsing. Algebraic Specification. In J. van Leeuwe (Ed.) *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, 675–788. Elsevier, 1990. doi:10.1016/B978-0-444-88074-1.50018-4.

[28] J. Xu, X. Zhang, C. Urban. Mechanising Turing Machines and Computability Theory in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, D. Pichardie (Eds.) *Proceedings 4th International Conference on Interactive Theorem Proving* (ITP 2013), *Lecture Notes in Computer Science* 7998, 147–162, 2013. doi:10.1007/978-3-642-39634-2_13.