Scientific Annals of Computer Science vol. 19, 2009
"Alexandru Ioan Cuza" University of Iaşi, Romania

# An Instruction Sequence Semigroup with Involutive Anti-Automorphisms

J.A. BERGSTRA[1] and A. PONSE[1]

**Abstract**

We introduce an algebra of instruction sequences by presenting a semigroup $C$ in which programs can be represented without directional bias: in terms of the next instruction to be executed, $C$ has both forward and backward instructions and a $C$-expression can be interpreted starting from any instruction. We provide equations for thread extraction, i.e., $C$'s program semantics. Then we consider thread extraction compatible (anti-)homomorphisms and (anti-)automorphisms. Finally we discuss some expressiveness results.

## 1 Introduction

In this paper three types of mathematical objects play a basic role:

1. Pieces of code, i.e., finite *sequences of instructions*, given some set $\mathcal{I}$ of instructions. A (computer) *program* is in our case a piece of code that satisfies the additional property that each state of its execution is prescribed by an instruction (typically, there are no jumps outside the range of instructions).

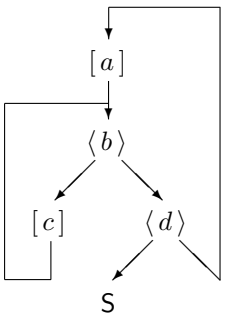
2. Finite and infinite sequences of *primitive* instructions (briefly, SPIs), the mathematical objects denoted by pieces of code (in particular by programs). Primitive instructions are taken from a set $\mathcal{U}$ that

[1]Section Theoretical Computer Science, Informatics Institute, University of Amsterdam. The authors acknowledge support from the NWO project Thread Algebra for Strategic Interleaving. Email: {J.A.Bergstra,A.Ponse}@uva.nl.

(possibly after some renaming) is a strict subset of $\mathcal{I}$. The execution of a SPI is *single-pass*: it starts with executing the first primitive instruction, and each primitive instruction is dropped after it has been executed or jumped over.

3. *Threads*, the mathematical objects representing the execution behavior of programs and used as their program semantics. Threads are defined using polarized actions and a certain form of conditional composition.

While each (computer) program can be considered as representing a sequence of instructions, the converse is not true. Omitting a few lines of code from a (well-formed) program usually results in an ill-formed program, if the remainder can be called a program at all. Before we discuss the instruction sequence semigroup mentioned in the title of this paper we briefly consider "threads", the mathematical objects representing the execution behavior of programs, or, more generally, of instruction sequences. Threads as considered here resemble finite state schemes that represent the execution of imperative programs in terms of their (control) actions. We take an abstract point of view and only consider actions and tests with symbolic names $(a, b, \ldots)$:




In this picture,

$[\,a\,]$ models the execution of action $a$ and its descent leads to the state thereafter (and likewise for $[\,c\,]$);

$\langle\, b\,\rangle$ models a the execution of test action $b$; its left descent models the "true-case" and its right one the "false-case" (and likewise for $\langle\, d\,\rangle$);

$\mathsf{S}$ is the state that models termination.

Finite state threads as the one above can be produced in many ways, and a primary goal of program algebra (PGA) is to study which primitives and program notations serve that purpose well. The first publication on PGA is the paper [7]. A basic expressiveness result states that the class of SPIs that can be directly represented in PGA (the so-called *periodic* SPIs) corresponds with these finite state threads: each PGA-program produces upon execution a finite state thread, and conversely, each finite state thread is produced by some PGA-program.

In this paper we introduce a set of instructions that also suits the above-mentioned purpose well and that at the same time has nice mathematical properties. Together with concatenation—its natural operation—it forms a semigroup with involutions that we call $C$ (for "code"). A simple involutive anti-automorphism[2] transforms each $C$-program into one of which the interpretation from right to left produces the same thread as the original program. Furthermore we define some homomorphisms and automorphisms that preserve the threads produced by $C$-expressions, thereby exemplifying a simple case of systematic program transformation. We generalize this approach by defining bijections on finite state threads and describe the associated automorphisms and anti-automorpisms on $C$, which all are generated from simple involutions. Finally, we study a few basic expressiveness questions about $C$.

The paper is structured as follows: In Section 2 we review threads in the setting of program algebra. Then, in Section 3 we introduce the semigroup $C$ of sequences of instructions that this paper is about. In Section 4 we define thread extraction on $C$, thereby giving semantics to $C$-expressions: each $C$-expression produces a finite state thread. In Section 5 we define '$C$-programs' and show that these are sufficient to produce finite state threads. Furthermore, only certain test instructions in $C$ are necessary to preserve $C$'s expressive power.

Section 6 is about a thread extraction preserving homomorphism on $C$ and a related anti-homomorphism. Then, in Section 7 we define a natural class of bijections on threads and establish a relation with a class of automorphisms on $C$, and in Section 8 we do the same thing with respect to a related class of anti-automorphisms on $C$.

In Section 9 we further consider $C$'s instructions in the perspective of expressiveness and show that restricting to a bound on the counters of jump instructions yields a loss in expressive power. In Section 10 we use Boolean registers to facilitate easy programming of finite state threads, and in Section 11 we relate the length of a $C$-program to the number of states of the thread it produces.

In Section 12 we discuss $C$ as a context in which some fundamental questions about programming can be further investigated and come up with some conclusions.

The paper is ended with an appendix that contains some background information (Sections A, B and C).

[2]We refer to [11] as a general reference for algebraic notions.

## 2 Basic Thread Algebra

In this section we review threads as they emerge from the behavioral abstraction from programs. Most of this text is taken from [14].

Basic Thread Algebra (BTA) is a form of process algebra which is tailored to the description of sequential program behavior. Based on a set $A$ of *actions*, it has the following constants and operators:

- the *termination* constant $\mathsf{S}$,
- the *deadlock* or *inaction* constant $\mathsf{D}$,
- for each $a \in A$, a binary *postconditional composition* operator $\_ \trianglelefteq a \trianglerighteq \_$.

We use *action prefixing* $a \circ P$ as an abbreviation for $P \trianglelefteq a \trianglerighteq P$ and take $\circ$ to bind strongest. Furthermore, for $n \geq 1$ we define $a^n \circ P$ by $a^1 \circ P = a \circ P$ and $a^{n+1} \circ P = a \circ (a^n \circ P)$.

The operational intuition is that each action represents a command which is to be processed by the execution environment of the thread. The processing of a command may involve a change of state of this environment.[3] At completion of the processing of the command, the environment produces a reply value `true` or `false`. The thread $P \trianglelefteq a \trianglerighteq Q$ proceeds as $P$ if the processing of $a$ yields `true`, and it proceeds as $Q$ if the processing of $a$ yields `false`.

Every thread in BTA is finite in the sense that there is a finite upper bound to the number of consecutive actions it can perform. The *approximation operator* $\pi : \mathbb{N} \times \mathrm{BTA} \to \mathrm{BTA}$ gives the behavior up to a specified depth. It is defined by

1. $\pi(0, P) = \mathsf{D}$,
2. $\pi(n+1, \mathsf{S}) = \mathsf{S}$, $\pi(n+1, \mathsf{D}) = \mathsf{D}$,
3. $\pi(n+1, P \trianglelefteq a \trianglerighteq Q) = \pi(n, P) \trianglelefteq a \trianglerighteq \pi(n, Q)$,

for $P, Q \in \mathrm{BTA}$ and $n \in \mathbb{N}$. We further write $\pi_n(P)$ instead of $\pi(n, P)$. We find that for every $P \in \mathrm{BTA}$, there exists an $n \in \mathbb{N}$ such that

$$\pi_n(P) = \pi_{n+1}(P) = \cdots = P.$$

[3] For the definition of threads we completely abstract from the environment. In Appendix C we define services which model (part of) the environment, and thread-service composition.

Following the metric theory of [1] in the form developed as the basis of the introduction of processes in [5], BTA has a completion $\mathrm{BTA}^\infty$ which comprises also the infinite threads. Standard properties of the completion technique yield that we may take $\mathrm{BTA}^\infty$ as the cpo consisting of all so-called *projective* sequences:[4]

$$\mathrm{BTA}^\infty = \{(P_n)_{n\in\mathbb{N}} \mid \forall n \in \mathbb{N}\ (P_n \in \mathrm{BTA}\ \&\ \pi_n(P_{n+1}) = P_n)\}.$$

For a detailed account of this construction see [3] or [15]. On $\mathrm{BTA}^\infty$, equality is defined componentwise: $(P_n)_{n\in\mathbb{N}} = (Q_n)_{n\in\mathbb{N}}$ if for all $n \in \mathbb{N}$, $P_n = Q_n$.

Overloading notation, we now define the constants and operators of BTA on $\mathrm{BTA}^\infty$:

1. $\mathsf{D} = (\mathsf{D}, \mathsf{D}, \ldots)$ and $\mathsf{S} = (\mathsf{D}, \mathsf{S}, \mathsf{S}, \ldots)$;
2. $(P_n)_{n\in\mathbb{N}} \trianglelefteq a \trianglerighteq (Q_n)_{n\in\mathbb{N}} = (R_n)_{n\in\mathbb{N}}$ with $\begin{cases} R_0 = \mathsf{D}, \\ R_{n+1} = P_n \trianglelefteq a \trianglerighteq Q_n. \end{cases}$

The elements of BTA are included in $\mathrm{BTA}^\infty$ by a mapping following this definition. E.g.,

$$a \circ \mathsf{S} \mapsto (P_n)_{n\in\mathbb{N}} \quad \text{with } P_0 = \mathsf{D},\ P_1 = a \circ \mathsf{D} \text{ and for } n \geq 2,\ P_n = a \circ \mathsf{S}.$$

It is not difficult to show that the projective sequence of $P \in \mathrm{BTA}$ thus defined equals $(\pi_n(P))_{n\in\mathbb{N}}$. We further use this inclusion of finite threads in $\mathrm{BTA}^\infty$ implicitly and write $P, Q, \ldots$ to denote elements of $\mathrm{BTA}^\infty$.

We define the set $Res(P)$ of *residual threads* of $P$ inductively as follows:

1. $P \in Res(P)$,
2. $Q \trianglelefteq a \trianglerighteq R \in Res(P)$ implies $Q \in Res(P)$ and $R \in Res(P)$.

A residual thread may be reached (depending on the execution environment) by performing zero or more actions. A thread $P$ is *regular* if $Res(P)$ is finite. Regular threads are also called *finite state threads*.

A *finite linear recursive specification* over $\mathrm{BTA}^\infty$ is a set of equations

$$x_i = t_i$$

for $i \in I$ with $I$ some finite index set, variables $x_i$, and all $t_i$ terms of the form $\mathsf{S}$, $\mathsf{D}$, or $x_j \trianglelefteq a \trianglerighteq x_k$ with $j, k \in I$. Finite linear recursive specifications represent continuous operators having unique fixed points [15].

[4]The cpo is based on the partial ordering $\sqsubseteq$ defined by $\mathsf{D} \sqsubseteq P$, and $P \sqsubseteq P'$, $Q \sqsubseteq Q'$ implies $P \trianglelefteq a \trianglerighteq Q \sqsubseteq P' \trianglelefteq a \trianglerighteq Q'$.

**Theorem 1.** *For all $P \in \mathrm{BTA}^\infty$, $P$ is regular iff $P$ is the solution of a finite linear recursive specification.*

*Proof.* Suppose $P$ is regular. Then $Res(P)$ is finite, so $P$ has residual threads $P_1, \ldots, P_n$ with $P = P_1$. We construct a finite linear recursive specification with variables $x_1, \ldots, x_n$ as follows:

$$x_i = \begin{cases} \mathsf{D} & \text{if } P_i = \mathsf{D}, \\ \mathsf{S} & \text{if } P_i = \mathsf{S}, \\ x_j \trianglelefteq a \trianglerighteq x_k & \text{if } P_i = P_j \trianglelefteq a \trianglerighteq P_k. \end{cases}$$

For the converse, assume that $P$ is the solution of some finite linear recursive specification $E$ with variables $x_1, \ldots, x_n$. Because the variables in $E$ have unique fixed points, we know that there are threads $P_1, \ldots, P_n \in \mathrm{BTA}^\infty$ with $P = P_1$, and for every $i \in \{1, \ldots, n\}$, either $P_i = \mathsf{D}$, $P_i = \mathsf{S}$, or $P_i = P_j \trianglelefteq a \trianglerighteq P_k$ for some $j, k \in \{1, \ldots, n\}$. We find that $Q \in Res(P)$ iff $Q = P_i$ for some $i \in \{1, \ldots, n\}$. So $Res(P)$ is finite, and $P$ is regular. □

**Example 1.** *The regular threads $a^2 \circ \mathsf{D}$ and $a^\infty = a \circ a \circ \cdots$ are the respective fixed points for $x_1$ in the finite linear recursive specifications*

1. $\{x_1 = a \circ x_2,\ x_2 = a \circ x_3,\ x_3 = \mathsf{D}\}$,
2. $\{x_1 = a \circ x_1\}$.

In reasoning with finite linear recursive specifications, we shall often identify variables and their fixed points. For example, we say that $P$ *is* the thread defined by $P = a \circ P$ instead of stating that $P$ equals the fixed point for $x$ in the finite linear recursive specification $x = a \circ x$. In this paper we write

$$\mathbb{T}_{reg}$$

for the set of regular threads in $\mathrm{BTA}^\infty$.

An elegant result based on [2] is that equality of recursively specified regular threads can be easily decided. Because one can always take the disjoint union of two finite linear recursive specifications it suffices to consider a single finite linear recursive specification $\{P_i = t_i \mid 1 \leq i \leq n\}$. Then $P_i = P_j$ follows from $\pi_{n-1}(P_i) = \pi_{n-1}(P_j)$. Thus, it is sufficient to decide whether two certain finite threads are equal. In Appendix B we provide a proof sketch.

## 3 *C*, a Semigroup for Code

In this section we introduce the sequences of instructions that form the main subject of this paper. We call these sequences "pieces of code" and use the letter $C$ to represent the resulting semigroup. The set $A$ of actions represents a parameter for $C$ (as it does for BTA).

For $a \in A$ and $k$ ranging over $\mathbb{N}^+$ (i.e., $\mathbb{N} \setminus \{0\}$), $C$-expressions are of the following form:

$$P \quad ::= \quad /a \;\Big|\; +/a \;\Big|\; -/a \;\Big|\; /\#k \;\Big|\; \backslash a \;\Big|\; +\backslash a \;\Big|\; -\backslash a \;\Big|\; \backslash\#k \;\Big|\; ! \;\Big|\; \# \;\Big|\; P; P$$

In $C$ the operation ";" is called *concatenation* and all other syntactical categories are called *$C$-instructions*:

$/a$ is a *forward basic instruction*. It prescribes to perform action $a$ and then (irrespective of the Boolean reply) to execute the instruction concatenated to its right-hand side; if there is no such instruction, deadlock follows.

$+/a$ and $-/a$ are *forward test instructions*. The *positive* forward test instruction $+/a$ prescribes to perform action $a$ and upon reply `true` to execute the instruction concatenated to its right-hand side, and upon reply `false` to execute the second instruction concatenated to its righthand side; if there is no such instruction to be executed, deadlock follows. For the *negative* forward test instruction $-/a$, execution of the next instruction is prescribed by the complementary replies.

$/\#k$ is a *forward jump instruction*. It prescribes to execute the instruction that is $k$ positions to the right and deadlock if there is no such instruction.

$\backslash a$, $+\backslash a$, $-\backslash a$ and $\backslash\#k$ are the *backward* versions of the instructions mentioned above. For these instructions, orientation is from right to left. For example, $\backslash a$ prescribes to perform action $a$ and then to execute the instruction concatenated to its left-hand side; if there is no such instruction, deadlock follows.

$!$ is the *termination instruction* and prescribes successful termination.

$\#$ is the *abort instruction* and prescribes deadlock.

For $C$ there is one axiom:

$$(X;Y);Z = X;(Y;Z). \tag{1}$$

By this axiom, $C$ is a semigroup and we shall not use brackets in repeated concatenations. As an example,

$$+/a;\ !\ ;\backslash\#2$$

is considered an appropriate $C$-expression. The instructions for termination and deadlock are the only instructions that do not specify further control of execution.

Perhaps the most striking aspect of $C$ is that its sequences of instructions have no directional bias. Although most program notations have a left to right (and top to bottom) natural order, symmetry arguments clarify that an orientation in the other direction might be present as well.

It is an empirical fact that imperative program notations in the vast majority of cases make use of a default direction, inherited from the natural language in which a program notation is naturally embedded. This embedding is caused by the language designers, or by the language that according to the language designers will be the dominant mother tongue of envisaged programmers. None of these matters can be considered core issues in computer science.

The fact, however, that imperative programs invariably show a default directional bias itself might admit an explanation in terms of complexity of design, expression or execution, and $C$ provides a context in which this advantage may be investigated.

Thus, in spite of an overwhelming evidence of the presence of directional bias in 'practice' we propose that the primary notation for sequences of instructions to be used for theoretical work is $C$ which refutes this bias. Obviously, from $C$ one may derive a dialect $C'$ by writing $a$ for $/a$, $+a$ for $+/a$, $-a$ for $-/a$ and $\#k$ for $/\#k$. Now there is a directional bias and in terms of bytes, the instructions are shorter. As explained in Section 5, the instructions $\backslash a$, $+\backslash a$ and $-\backslash a$ can be eliminated, thus obtaining a smaller instruction set which is more easily parsed. One may also do away with $a$ and $-a$ in favor of $+a$, again reducing the number of instructions. Reduction of the number of instructions leads to longer sequences, however, and where the optimum of this trade off is found is a matter which lies outside the theory of instruction sequences per se. We further discuss the nature of $C$ in Section 12.

# 4 Thread Extraction and $C$-Expressions

In this section we define thread extraction on $C$. For a $C$-expression $X$, $|X|^{\rightarrow}$ denotes the thread produced by $X$ when execution started at the leftmost or "first" instruction, thus $|..|^{\rightarrow}$ is an operator that assigns a thread to a $C$-expression. We prove that this is always a regular thread. We also consider right-to-left thread extraction where thread extraction starts at the righmost of a $C$-expression.

We will use auxiliary functions $|X|_j$ with $j$ ranging over the integers $\mathbb{Z}$ and we define

$$|X|^{\rightarrow} = |X|_1,$$

meaning that thread extraction starts at the first (or leftmost) instruction of $X$. For $j \in \mathbb{Z}$, $|X|_j$ is defined in Table 1.

---

Let $X = i_1; \ldots; i_n$ and $j \in \mathbb{Z}$.

For $j \in \{1, \ldots, n\}$,

$$|X|_j = \begin{cases} a \circ |X|_{j+1} & \text{if } i_j = /a, \\ |X|_{j+1} \trianglelefteq a \trianglerighteq |X|_{j+2} & \text{if } i_j = +/a, \\ |X|_{j+2} \trianglelefteq a \trianglerighteq |X|_{j+1} & \text{if } i_j = -/a, \\ |X|_{j+k} & \text{if } i_j = /\#k, \\ \\ a \circ |X|_{j-1} & \text{if } i_j = \backslash a, \\ |X|_{j-1} \trianglelefteq a \trianglerighteq |X|_{j-2} & \text{if } i_j = +\backslash a, \\ |X|_{j-2} \trianglelefteq a \trianglerighteq |X|_{j-1} & \text{if } i_j = -\backslash a, \\ |X|_{j-k} & \text{if } i_j = \backslash\#k, \\ \\ \mathsf{S} & \text{if } i_j = !\,, \\ \mathsf{D} & \text{if } i_j = \#\,, \end{cases}$$

$$\text{for } j \notin \{1, \ldots, n\}, \quad |X|_j = \mathsf{D}. \tag{2}$$

---

Table 1: Equations for thread extraction

A special case arises if these equations applied from left to right define a loop without any actions, as in

$$\begin{aligned}|/\#2;/a;\backslash\#2|_1 &= |/\#2;/a;\backslash\#2|_3\\ &= |/\#2;/a;\backslash\#2|_1.\end{aligned}$$

For this case we have the following rule:

> If the equations in Table 1 applied from left to right yield a loop without any actions the extracted thread is $\mathsf{D}$. (3)

Rule (3) applies if and only if a loop in a thread extraction is the result of consecutive jumps to jump instructions.

In the following we show that thread extraction on $C$-expressions produces regular threads. For a $C$-expression $X$ we define $\ell(X) \in \mathbb{N}^+$ to be the length of $X$, i.e., its number of instructions.

**Theorem 2.** *If $X$ is a $C$-expression and $i \in \mathbb{Z}$, then $|X|_i$ defines a regular thread.*

*Proof.* Assume $X$ is a $C$-expression with $\ell(X) = n$. If $i \notin \{1, \ldots, n\}$, then $|X|_i = \mathsf{D}$ by rule (2). In the other case, a single application of the matching equation in Table 1 determines for each $i \in \{1, \ldots, n\}$ an equation of the form

$$|X|_i = |X|_j \trianglelefteq a \trianglerighteq |X|_k, \quad \text{or } |X|_i = |X|_j, \quad \text{or } |X|_i = \mathsf{D}, \quad \text{or } |X|_i = \mathsf{S} \tag{4}$$

where by rule (2) we may assume that all expressions $|X|_j$ and $|X|_k$ occurring in the right-hand sides satisfy $j, k \in \{1, \ldots, n\}$ (otherwise they are replaced by $\mathsf{D}$). We construct $n$ linear equations $x_i = t_i$ with the property that $|X|_i$ as given by the rules for thread extraction is a fixed point for $x_i$:

1. Define $x_i = t_i$ from (4) by replacing each $|X|_j$ by $x_j$.
2. Determine with Rule (3) all equations $|X|_i = |X|_j$ that define a loop without actions, and replace all associated equations $x_i = x_j$ by
   $$x_i = \mathsf{D}.$$
3. Replace any remaining equation of the form $x_i = x_j$ by
   $$x_i = t_j$$
   where $t_j$ is the right-hand side of the equation for $x_j$. Repeating this procedure exhaustively yields a finite linear specification with variables $x_1, \ldots, x_n$.

For each $i \in \{1, \ldots, n\}$ the thread defined by thread extraction on $|X|_i$ is a fixed point for $x_i$. Hence $|X|^{\rightarrow}$ is a regular thread, and so is $|X|^{\leftarrow}$. $\square$

Given some $C$-expression $X$, we shall often use $|X|_i$ as the identifier of the thread defined by $|X|_i$ as meant in Theorem 2, and similar for $|X|^{\rightarrow}$. As an example of thread extraction, consider the $C$-expression

$$X = /a; +/b; \backslash c; +/d; \, !\,; \backslash\#5 \tag{5}$$

It is not hard to check that $X$ produces the regular thread $P_1$ (i.e., $|X|^{\rightarrow} = P_1$) defined by[5]

$$\begin{aligned} P_1 &= a \circ P_2 \\ P_2 &= P_3 \trianglelefteq b \trianglerighteq P_4 \\ P_3 &= c \circ P_2 \\ P_4 &= P_5 \trianglelefteq d \trianglerighteq P_1 \\ P_5 &= \mathsf{S} \end{aligned}$$

Thread extraction defines an equivalence on $C$-expressions, say $X \equiv_{\rightarrow} Y$ if $|X|^{\rightarrow} = |Y|^{\rightarrow}$, that is not a congruence, e.g.,

$$\# \equiv_{\rightarrow} /\#1 \quad \text{but} \quad \#\,; /a \not\equiv_{\rightarrow} /\#1; /a.$$

We define right-to-left thread extraction, notation

$$|X|^{\leftarrow},$$

as the thread extraction that starts from the rightmost position of a piece of code:

$$|X|^{\leftarrow} = |X|_{\ell(X)}$$

where $\ell(X) \in \mathbb{N}^+$ is the length of $X$, i.e., its number of instructions. Taking $X$ as defined in Example (5), we find $|X|^{\leftarrow} = |X|^{\rightarrow}$ because for that particular $X$, $|X|_6 = |X|_1$. Right-to-left thread extraction also defines an equivalence on $C$-expressions, say $X \equiv_{\leftarrow} Y$ if $|X|^{\leftarrow} = |Y|^{\leftarrow}$, that is not a congruence, e.g.,

$$\# \equiv_{\leftarrow} \backslash\#1 \quad \text{but} \quad /a; \# \not\equiv_{\leftarrow} /a; \backslash\#1.$$

[5]This regular thread $P_1$ can be visualized as was done in Section 1.

## 5 Expressiveness of $C$-Programs

In this section we introduce the notion of a '$C$-program'. Furthermore we discuss a basic expressiveness result: we show that each regular thread is the thread extraction of some $C$-program. Finally we establish that we do not need all of $C$'s instructions to preserve expressiveness.

**Definition 1.** *A **C-program** is a piece of code $X = i_1; \ldots; i_n$ with $n > 0$ such that the computation of $|X|_j$ for each $j = 1, \ldots, n$ does not use equation (2). In other words, there are no jumps outside the range of $X$ and execution can only end by executing either the termination instruction ! or the abort instruction # .*

In the setting of program algebra we explicitly distinguished in [9] a "program" from an instruction sequence (or a piece of code) in the sense that a program has a natural and preferred semantics, while this is not the case for the latter one. Observe that if $X$ and $Y$ are $C$-programs, then so is $X; Y$. A piece of code that is not a program can be called a *program fragment* because it can be extended to a program that yields the same thread extraction. This follows from the next proposition, which states that position numbers can be relativized.

**Proposition 1.** *For $k \in \mathbb{N}$ and $X$ a $C$-expression,*

1. $|X|_k = |\# \,; X|_{k+1}$,
2. $|X|_k = |X; \# \,|_k$.

*Moreover, in the case that $X$ is a $C$-program and $1 \leq k \leq \ell(X)$,*

3. $|X|_k = |/\#k; X|^{\rightarrow}$,
4. $|X|_k = |X; \backslash\#\ell(X) + 1 - k|^{\leftarrow}$.

With properties 1 and 2 we find for example

$$\begin{aligned} |+/a; \backslash\#2|^{\rightarrow} &= |+/a; \backslash\#2|_1 \\ &= |\# \,; +/a; \backslash\#2; \# \,|_2, \end{aligned}$$

and since the latter piece of code is a $C$-program, we find with property 3 another one that produces the same thread with left-to-right thread extraction:

$$|\# \,; +/a; \backslash\#2; \# \,|_2 = |/\#2; \# \,; +/a; \backslash\#2; \# \,|^{\rightarrow}.$$

Of course, for property 3 to be valid it is crucial that $X$ is a $C$-program: for example

$$\begin{aligned} |+/a;\backslash\#2|^{\rightarrow} &= |+/a;\backslash\#2|_1 \\ &\neq |/\#1;+/a;\backslash\#2|^{\rightarrow}. \end{aligned}$$

A similar example contradicting property 4 for $X$ not a $C$-program is easily found.

**Theorem 3.** *Each regular thread in $\mathbb{T}_{reg}$ is produced by a $C$-program.*

*Proof.* Assume that a regular thread $P_1$ is specified by linear equations $P_1 = t_1, \ldots, P_n = t_n$. We transform each equation into a piece of $C$-code:

$$\begin{aligned} P_i = \mathsf{S} &\mapsto !\,;\#\,;\#\,, \\ P_i = \mathsf{D} &\mapsto \#\,;\#\,;\#\,, \\ P_i = P_j \trianglelefteq a \trianglerighteq P_k &\mapsto \begin{cases} +/a;/\#p;/\#q & \text{if } p,q>0, \\ +/a;/\#p;\backslash\#(-q) & \text{if } p>0,\ q<0, \\ +/a;\backslash\#(-p);/\#q & \text{if } p<0,\ q>0, \\ +/a;\backslash\#(-p);\backslash\#(-q) & \text{if } p,q<0, \end{cases} \end{aligned}$$

where $p = 3(j-i)-1$ and $q = 3(k-i)-2$ (so $p, q \in \mathbb{Z}\setminus\{0\}$). Concatenating these pieces of code in the order given by $P_1, \ldots, P_n$ yields a $C$-expression $X$ with $|X|^{\rightarrow} = P_1$. By construction $X$ contains no jumps outside the range of instructions and therefore $X$ is a $C$-program. Finally, note that the instructions of $X$ are in the set $\{+/a, /\#k, \backslash\#k, !, \# \mid a \in A,\ k \in \mathbb{N}^+\}$. $\square$

From the proof of Theorem 3 we infer that only positive forward test instructions, jumps and termination are needed to preserve $C$'s expressiveness:

**Corollary 1.** *Let $C^-$ be defined by allowing only instructions from the set*

$$\{+/a, /\#k, \backslash\#k, !\ \mid a \in A,\ k \in \mathbb{N}^+\}.$$

*Then each regular thread in $\mathbb{T}_{reg}$ can be produced by a program in $C^-$.*

*Proof.* With $\#$ added to the instruction set mentioned, the result follows immediately from the proof of Theorem 3. The use of $\#$ in that proof can easily be avoided, for example by setting

$$\begin{aligned} P_i = \mathsf{S} &\mapsto !\,;/\#1;\backslash\#1 && \text{(instead of } !\,;\#\,;\#\,), \\ P_i = \mathsf{D} &\mapsto /\#1;/\#1;\backslash\#1 && \text{(instead of } \#\,;\#\,;\#\,). \end{aligned}$$

The resulting expression clearly contains no jumps outside its range and is hence a $C$-program. □

# 6 Thread Extraction Preserving Homomorphisms

In this section we consider functions on $C$ that preserve thread extraction. We start with a homomorphism that turns all basic and test instructions into their forward counterparts, and another one that only yields positive forward test instructions. Then we consider an anti-homomorphism that relates extraction with right-to-left thread extraction. So, these functions are very basic examples of program transformation.

Let the function $h : C \to C$ be defined on $C$-instructions as follows:

$$\begin{aligned}
/a &\mapsto /a; /\#2; \# ,\\
+/a &\mapsto +/a; /\#2; /\#4,\\
-/a &\mapsto -/a; /\#2; /\#4,\\
/\#k &\mapsto /\#3k; \# ; \# ,
\end{aligned}$$

$$\begin{aligned}
\backslash a &\mapsto /a; \backslash\#4; \# ,\\
+\backslash a &\mapsto +/a; \backslash\#4; \backslash\#8,\\
-\backslash a &\mapsto -/a; \backslash\#4; \backslash\#8,\\
\backslash\#k &\mapsto \backslash\#3k; \# ; \# ,
\end{aligned}$$

$$\begin{aligned}
! &\mapsto ! ; \# ; \# ,\\
\# &\mapsto \# ; \# ; \# .
\end{aligned}$$

So, $h$ replaces all basic and test instructions by fragments containing only their forward counterparts. Defining

$$h(X;Y) = h(X); h(Y)$$

makes $h$ an injective homomorphism (a 'monomorphism') that preserves the equivalence obtained by (left-to-right) thread extraction, i.e.,

$$|X|^{\rightarrow} = |h(X)|^{\rightarrow}.$$

This follows from the more general property

$$|X|_{j+1} = |h(X)|_{3j+1}$$

for all $j < \ell(X)$, which is easy to prove by case distinction. So, $|X|^{\rightarrow} = |h^k(X)|^{\rightarrow}$, and, moreover, if $X$ is a $C$-program, then so is $h^k(X)$.

Of course many variants of the homomorphism $h$ satisfy the latter two properties. A particular one is the homomorphism obtained from $h$ by replacement with the following defining clauses:

$$\begin{aligned}
/a &\mapsto +/a; /\#2; /\#1,\\
-/a &\mapsto +/a; /\#5; /\#1,\\
\backslash a &\mapsto +/a; \backslash\#4; \backslash\#5,\\
-\backslash a &\mapsto +/a; \backslash\#4; \backslash\#8,
\end{aligned}$$

because now only forward positive test instructions occur in the homomorphic image. In other words: with respect to thread extraction, $C$'s expressive power is preserved if its set of instructions is reduced to

$$\{+/a, /\#k, \backslash\#k, \,!\,, \# \mid a \in A,\ k \in \mathbb{N}^+\}.$$

This is the syntactic counterpart of Corollary 1 in Section 5.

Let $g : C \to C$ be defined on $C$-instructions as follows:

$$\begin{aligned}
/a &\mapsto \# \,; \backslash\#2; \backslash a,\\
+/a &\mapsto \backslash\#4; \backslash\#2; +\backslash a,\\
-/a &\mapsto \backslash\#4; \backslash\#2; -\backslash a,\\
/\#k &\mapsto \# \,; \# \,; \backslash\#3k,
\end{aligned}$$

$$\begin{aligned}
\backslash a &\mapsto \# \,; /\#4; \backslash a,\\
+\backslash a &\mapsto /\#8; /\#4; +\backslash a,\\
-\backslash a &\mapsto /\#8; /\#4; -\backslash a,\\
\backslash\#k &\mapsto \# \,; \# \,; /\#3k,
\end{aligned}$$

$$\begin{aligned}
! &\mapsto \# \,; \# \,; \,!\,,\\
\# &\mapsto \# \,; \# \,; \# \,.
\end{aligned}$$

So, $g$ replaces all basic and test instructions by $C$-fragments containing only their backward counterparts. Defining $g(X;Y) = g(Y); g(X)$ makes $g$ an *anti-homomorphism* that satisfies

$$|X|^{\rightarrow} = |g(X)|^{\leftarrow}.$$

This follows from a more general property discussed in Section 8.

# 7 Structural Bijections and TEC-Automorphisms

In this section we define *structural bijections* on the finite state threads over $A$ as a natural type of (bijective) thread transformations. We then describe and analyze the associated class of automorphisms on $C$, which appear to be generated from simple involutions.

Given a bijection $\phi$ on $A$ (thus a permutation of $A$) and a partitioning of $A$ in $A_{\mathtt{true}}$ and $A_{\mathtt{false}}$, we extend $\phi$ to a *structural bijection* on BTA by defining for all $a \in A$ and $P, Q \in \mathrm{BTA}$,

$$\begin{aligned} \phi(\mathsf{D}) &= \mathsf{D}, \\ \phi(\mathsf{S}) &= \mathsf{S}, \\ \phi(P \trianglelefteq a \trianglerighteq Q) &= \begin{cases} \phi(P) \trianglelefteq \phi(a) \trianglerighteq \phi(Q) & \text{if } \phi(a) \in A_{\mathtt{true}}, \\ \phi(Q) \trianglelefteq \phi(a) \trianglerighteq \phi(P) & \text{if } \phi(a) \in A_{\mathtt{false}}. \end{cases} \end{aligned}$$

Structural bijections naturally extend to $\mathbb{T}_{\mathrm{reg}}$: if $P_i$ is a fixed point for $x_i$ in the finite linear specification $\{x_i = t_i(\overline{x}) \mid i = 1, \ldots, n\}$, then $\phi(P_i)$ is a fixed point for $y_i$ in

$$\{y_i = \phi(t_i(\overline{x})) \mid i = 1, \ldots, n,\ \phi(x_i) = y_i\}. \tag{6}$$

As an example, assume that $\phi(a) = b \in A_{\mathtt{false}}$ and thread $P$ is given by

$$P = P \trianglelefteq a \trianglerighteq Q,\ Q = \mathsf{D}$$

then $P' = \phi(P)$ is defined by

$$P' = Q' \trianglelefteq b \trianglerighteq P',\ Q' = \mathsf{D}.$$

**Theorem 4.** *There are* $2^{|A|} \cdot |A|!$ *structural bijections on* BTA*, and thus on* $\mathbb{T}_{reg}$*.*

*Proof.* Trivial: if $|A| = n$, there are $2^n$ different partitionings in $A_{\mathtt{true}}$ and $A_{\mathtt{false}}$, and $n!$ different bijections on $A$. □

Each structural bijection can be written as the composition of a (possibly empty) series of transpositions or 'swaps' (its permutation part) and a (possibly empty) series of postconditional 'flips' that model the `false`-part of its partitioning. So, for a fixed $\phi$ there exist $k$ and $m$ such that

$$\phi = \overline{flip}_{c_1} \circ \ldots \circ \overline{flip}_{c_m} \circ \overline{swap}_{a_1,b_1} \circ \ldots \circ \overline{swap}_{a_k,b_k}$$

where $\overline{swap}_{a,b}$ models the exchange of actions $a$ and $b$, and $\overline{flip}_c$ the postconditional flips for $A_{\mathtt{false}} = \{c_1, \ldots, c_m\}$, and $\phi$ is the identity if $k = m = 0$. More precisely,

$$\overline{swap}_{a,b}(P \trianglelefteq c \trianglerighteq Q) =$$

$$\overline{swap}_{a,b}(P) \trianglelefteq \overline{c} \trianglerighteq \overline{swap}_{a,b}(Q) \quad \text{with} \quad \begin{cases} \overline{c} = b & \text{if } c = a, \\ \overline{c} = a & \text{if } c = b, \\ \overline{c} = c & \text{otherwise,} \end{cases}$$

and

$$\overline{flip}_c(P \trianglelefteq a \trianglerighteq Q) = \begin{cases} \overline{flip}_c(Q) \trianglelefteq a \trianglerighteq \overline{flip}_c(P) & \text{if } a = c, \\ \overline{flip}_c(P) \trianglelefteq a \trianglerighteq \overline{flip}_c(Q) & \text{otherwise.} \end{cases}$$

For $A = \{a_1, \ldots, a_n\}$ we can do with $n - 1$ swaps $\overline{swap}_{a_1,a_j}$ $(1 < j \leq n)$ as these define any other swap by $\overline{swap}_{a_i,a_j} = \overline{swap}_{a_1,a_j} \circ \overline{swap}_{a_1,a_i} \circ \overline{swap}_{a_1,a_j}$, and $n$ flips $\overline{flip}_{a_i}$ $(1 \leq i \leq n)$.

We show that structural bijections naturally correspond with a certain class of automorphisms on $C$.

**Definition 2.** *An automorphism $\alpha$ on $C$ is **thread extraction compatible** (TEC) if there exists a structural bijection $\beta$ such that the following diagram commutes:*

$$\begin{array}{ccc} C & \xrightarrow{|-|^{\rightarrow}} & \mathbb{T}_{reg} \\ \downarrow \alpha & & \downarrow \beta \\ C & \xrightarrow{|-|^{\rightarrow}} & \mathbb{T}_{reg} \end{array}$$

**Theorem 5.** *The TEC-automorphisms on $C$ are generated by*

$$\begin{array}{rl} swap_{a,b}: & \text{exchanges } a \text{ and } b \text{ in all instructions containing } a \text{ or } b, \\ flip_a: & \text{exchanges } + \text{ and } - \text{ in all test instructions containing } a, \end{array}$$

*where $a$ and $b$ range over $A$.*

*Proof.* First we have to show that if $\alpha$ is generated from $swap_{a,b}$ and $flip_a$ $(a, b \in A)$, then $\alpha$ is a TEC-automorphism. This follows from the fact that the diagram in Definition 2 commutes for $swap_{a,b}$ if we take $\beta = \overline{swap}_{a,b}$ and for $flip_a$ if we take $\beta = \overline{flip}_a$. We show this below.

Then we have to show that if $\alpha$ is a TEC-automorphism, then $\alpha$ is generated from swaps and flips. Above we argued that each structural bijection can be characterized by zero or more $\overline{swap}_{a,b}$ and $\overline{flip}_a$ applications. So, again it suffices to argue that for $\beta = \overline{swap}_{a,b}$, the diagram commutes if $\alpha = swap_{a,b}$ and for $\beta = \overline{flip}_c$ if $\alpha = flip_c$. The general case follows from repeated applications.

Let $X \in C$. First assume $\beta = \overline{flip}_c$. Following the construction in the proof of Theorem 2 we find a finite linear specification $\{x_i = t_i \mid i = 1, \ldots, n\}$ with $n = \ell(X)$ such that $|X|_i$ is a fixed point for $x_i$. Transforming this specification according to (6) with $\phi = \overline{flip}_c$ yields $\{y_i = \overline{flip}_c(t_i(\overline{x})) \mid i = 1, \ldots, n,\ \overline{flip}_c(x_i) = y_i\}$. Now $|flip_c(X)|_i$ is a fixed point for $y_i$: this also follows from the construction in the proof of Theorem 2 and the fact that $flip_c$ only changes the sign of $\pm/c$ and $\pm\backslash c$ in $X$.

We now show that $\overline{flip}_c(|X|_i)$ is a fixed point for $y_i$ by a case distinction on the form of $t_i$ in the equations $x_i = t_i$ $(i = 1, \ldots, n)$:

- If $x_i = x_j \trianglelefteq c \trianglerighteq x_k$ then $|X|_i = |X|_j \trianglelefteq c \trianglerighteq |X|_k$, so
$$\begin{aligned}\overline{flip}_c(|X|_i) &= \overline{flip}_c(|X|_j \trianglelefteq c \trianglerighteq |X|_k)\\ &= \overline{flip}_c(|X|_k) \trianglelefteq c \trianglerighteq \overline{flip}_c(|X|_j).\end{aligned}$$
Note that in this case $y_i = y_k \trianglelefteq c \trianglerighteq y_j$.

- If $x_i = x_j \trianglelefteq a \trianglerighteq x_k$ with $a \neq c$, then $|X|_i = |X|_j \trianglelefteq a \trianglerighteq |X|_k$, so
$$\begin{aligned}\overline{flip}_c(|X|_i) &= \overline{flip}_c(|X|_j \trianglelefteq a \trianglerighteq |X|_k)\\ &= \overline{flip}_c(|X|_j) \trianglelefteq a \trianglerighteq \overline{flip}_c(|X|_k).\end{aligned}$$
Note that in this case $y_i = y_j \trianglelefteq a \trianglerighteq y_k$.

- If $x_i = \mathsf{S}$, then $|X|_i = \mathsf{S}$ and $y_i = \mathsf{S}$. Also $\overline{flip}_c(|X|_i) = \mathsf{S}$.

- If $t_i = \mathsf{D}$, then $|X|_i = \mathsf{D}$ and $y_i = \mathsf{D}$. Also $\overline{flip}_c(|X|_i) = \mathsf{D}$.

So in all cases $\overline{flip}_c(|X|_i)$ is a fixed point for $y_i$. Hence, $|flip_c(X)|_i = \overline{flip}_c(|X|_i)$ and thus $|flip_c(X)|^{\rightarrow} = \overline{flip}_c(|X|^{\rightarrow})$.

In a similar way it follows that $|swap_{a,b}(X)|_i = \overline{swap}_{a,b}(|X|_i)$. □

Note that $swap_{a,a}$ is the identity and so is $flip_a \circ flip_a$. Furthermore, for $a \neq b$ we have $swap_{a,b} = swap_{b,a}$ and

$$swap_{a,b} \circ flip_c = \begin{cases} flip_c \circ swap_{a,b} & \text{if } c \notin \{a, b\},\\ flip_d \circ swap_{a,b} & \text{if } \{a, b\} = \{c, d\}.\end{cases}$$

This implies that each TEC-automorphism can be represented as

$$\mathit{flip}_{c_1} \circ \ldots \circ \mathit{flip}_{c_m} \circ \mathit{swap}_{a_1,b_1} \circ \ldots \circ \mathit{swap}_{a_k,b_k}.$$

Similarly as remarked above, for $A = \{a_1, \ldots, a_n\}$ we can do with $n-1$ swaps $\mathit{swap}_{a_1,a_j}$ $(1 < j \leq n)$ as these define any other swap.

We further write *TEC-AUT* for the set of TEC-automorphisms, and we say that $\mathit{swap}_{a,b}$ and the structural bijection $\overline{\mathit{swap}}_{a,b}$ are *associated*, and similar for $\mathit{flip}_a$ and $\overline{\mathit{flip}}_a$. So, the above result states that for the associated pair $\alpha \in \mathit{TEC\text{-}AUT}$ and structural bijection $\overline{\alpha}$ the following diagram commutes:

$$\begin{array}{ccc} C & \stackrel{|-|^{\rightarrow}}{\longrightarrow} & \mathbb{T}_{\mathrm{reg}} \\ \downarrow \alpha & & \downarrow \overline{\alpha} \\ C & \stackrel{|-|^{\rightarrow}}{\longrightarrow} & \mathbb{T}_{\mathrm{reg}} \end{array}$$

The following corollary of Theorem 5 follows immediately.

**Corollary 2.** *If $\alpha \in \mathit{TEC\text{-}AUT}$, then $\alpha$ preserves the orientation of all instructions and $\alpha(i) = i$ for $i \in \{/\#k, \backslash\#k, !, \# \mid k \in \mathbb{N}^+\}$. Furthermore, for each $a \in A$, $\alpha$ is determined by its value on one of the possible four test instructions. If for example $\alpha(+/a) = -/b$, then $\alpha(/a) = /b$, $\alpha(-/a) = +/b$, and the remaining identities are given by replacing all forward slashes by backward slashes.*

Each element $\alpha \in \mathit{TEC\text{-}AUT}$ that satisfies $\alpha^2(u) = u$ for all $C$-instructions $u$ is an *involution*, i.e.

$$\alpha^2(X) = X.$$

Obvious examples of involutions are $\mathit{swap}_{a,b}$ and $\mathit{flip}_c$, and a counter-example is

$$\alpha = \mathit{flip}_b \circ \mathit{swap}_{a,b}$$

because

$$\alpha^2 = \mathit{flip}_b \circ \mathit{swap}_{a,b} \circ \mathit{flip}_b \circ \mathit{swap}_{a,b} = \mathit{flip}_b \circ \mathit{flip}_a \circ \mathit{swap}_{a,b} \circ \mathit{swap}_{a,b} = \mathit{flip}_b \circ \mathit{flip}_a.$$

However, $\alpha^2$ is an involution (because compositions of flip commute).

# 8 TEC-Anti-Automorphisms

In this section we consider the relation between structural bijections on threads and an associated class of anti-automorphisms on $C$. Recall that a function $\phi$ is an anti-homomorphism if it satisfies $\phi(X;Y) = \phi(Y);\phi(X)$. Furthermore, we show how the monomorphism $h$ defined in Section 6 is systematically related to the anti-homomorphism $g$ defined in that section.

Define the anti-automorphism $rev : C \to C$ (reverse) on $C$-instructions by the exchange of all forward and backward orientations:

$$\begin{aligned} /a &\mapsto \backslash a, \\ +/a &\mapsto +\backslash a, \\ -/a &\mapsto -\backslash a, \\ /\#k &\mapsto \backslash \#k, \end{aligned}$$

$$\begin{aligned} \backslash a &\mapsto /a, \\ +\backslash a &\mapsto +/a, \\ -\backslash a &\mapsto -/a, \\ \backslash \#k &\mapsto /\#k, \end{aligned}$$

$$\begin{aligned} ! &\mapsto !\,, \\ \# &\mapsto \#\,. \end{aligned}$$

Then $rev^2(X) = X$, so $rev$ is an involution. Furthermore, it is immediately clear that for all $X \in C$,

$$|X|^{\rightarrow} = |rev(X)|^{\leftarrow}.$$

**Definition 3.** *An anti-automorphism $\alpha$ on $C$ is **thread extraction compatible** (TEC) if there exists a structural bijection $\beta$ such that the following diagram commutes:*

$$\begin{array}{ccc} C & \stackrel{|-|^{\rightarrow}}{\longrightarrow} & \mathbb{T}_{reg} \\ \downarrow \alpha & & \downarrow \beta \\ C & \stackrel{|-|^{\leftarrow}}{\longrightarrow} & \mathbb{T}_{reg} \end{array}$$

We write *TEC-AntiAUT* for the set of thread extraction compatible anti-automorphisms on $C$. The following result establishes a strong connection between *TEC-AUT* and *TEC-AntiAUT*.

**Theorem 6.** *TEC-AntiAUT* $= \{rev \circ \alpha \mid \alpha \in$ *TEC-AUT*$\}$.

*Proof.* Let $\gamma \in$ *TEC-AntiAUT*, so $\gamma$ is an anti-automorphism and there is a structural bijection $\beta$ such that $|\gamma(X)|^{\leftarrow} = \beta(|X|^{\rightarrow})$ for all $X$. By Theorem 5, $\beta = \overline{\alpha}$ for some $\alpha \in$ *TEC-AUT* and $\beta(|X|^{\rightarrow}) = |\alpha(X)|^{\rightarrow}$ for all $X$, and thus

$$|\gamma(X)|^{\leftarrow} = |rev \circ \alpha(X)|^{\leftarrow} \quad \text{for all } X. \tag{7}$$

This defines $\gamma$ on $\{/\#k, \backslash\#k, \,!\,, \# \mid k \in \mathbb{N}^+\}$. By Corollary 2, $\alpha$ is determined by its definition on all positive forward test instructions. So, if for $a, b \in A$, $\alpha(+/a) = \pm/b$ then we find by (7) with $X = +/a;\,!$ that $\gamma(-\backslash a) = \mp\backslash b$. Since $\alpha$ is determined for all other instructions containing $a$, also $\gamma$ is fully determined for all instructions containing $a$. It follows that $\gamma = rev \circ \alpha$, thus $\gamma \in \{rev \circ \alpha \mid \alpha \in$ *TEC-AUT*$\}$.

Conversely, if $\gamma \in \{rev \circ \alpha \mid \alpha \in$ *TEC-AUT*$\}$, say $\gamma = rev \circ \alpha$ with $\alpha \in$ *TEC-AUT*, then $|\gamma(X)|^{\leftarrow} = |\alpha(X)|^{\rightarrow} = \beta(|X|^{\rightarrow})$ for some structural bijection $\beta$ and all $X$. Furthermore, $\gamma$ is an anti-automorphism, so $\gamma \in$ *TEC-AntiAUT*. □

Observe that for all $\alpha \in$ *TEC-AUT*, $\alpha \circ rev = rev \circ \alpha$ and for all $\alpha, \beta \in$ *TEC-AntiAUT*, $\alpha \circ \beta \in$ *TEC-AUT*. Using the notation for associated pairs we find for $\beta = rev \circ \alpha \in$ *TEC-AntiAUT* that the following diagram commutes:

$$\begin{array}{ccc} C & \stackrel{|-|^{\rightarrow}}{\longrightarrow} & \mathbb{T}_{\mathrm{reg}} \\ \downarrow \beta & & \downarrow \overline{\alpha} \\ C & \stackrel{|-|^{\leftarrow}}{\longrightarrow} & \mathbb{T}_{\mathrm{reg}} \end{array}$$

Note that we use $\overline{\alpha}$, i.e., the associated structural bijection of $\alpha$, in this diagram.

Another application with *rev* is the following: for $h : C \to C$ a monomorphism, the following diagram commutes:

$$\begin{array}{ccc} C & \stackrel{rev \circ h}{\longrightarrow} & C \\ \downarrow h & & \downarrow |-|^{\leftarrow} \\ C & \stackrel{|-|^{\rightarrow}}{\longrightarrow} & \mathbb{T}_{\mathrm{reg}} \end{array}$$

As an example, consider the anti-homomorphism $g$ defined in Section 6: indeed $g = rev \circ h$ for the homomorphism $h$ defined in that section.

# 9 Expressiveness and reduced instruction sets

In this section we further consider $C$'s instructions in the perspective of expressiveness. We show that setting a bound on the size of jump counters in $C$ does have consequences with respect to expressiveness: let

$$C_k$$

be defined by allowing only jump instructions with counter value $k$ or less.

We first introduce some auxiliary notions: following the definition of residual threads in Section 2, we say that thread $Q$ is a 0-*residual* of thread $P$ if $P = Q$, and an $n+1$-*residual* of $P$ if for some $a \in A$, $P = P_1 \trianglelefteq a \trianglerighteq P_2$ and $Q$ is an $n$-residual of $P_1$ or of $P_2$. Note that a finite thread (in BTA) only has $n$-residuals for finitely many $n$, while for the thread $P$ defined by $P = a \circ P$ it holds that $P$ is an $n$-residual of itself for each $n \in \mathbb{N}$.

Let $a \in A$ be fixed and $n \in \mathbb{N}^+$. Thread $P$ has the *$a$-$n$-property* if $\pi_n(P) = a^n \circ \mathsf{D}$ and $P$ has $2^n - 1$ (different) $n$-residuals which all have a first approximation not equal to $a \circ \mathsf{D}$. So, if a thread $P$ has the $a$-$n$-property, then $n$ consecutive $a$-actions can be executed and each sequence of $n$ replies leads to a unique $n$-residual. Moreover, none of these residual threads starts with an $a$-action (by the requirement on their first approximation). We note that for each $n \in \mathbb{N}^+$ we can find a finite thread with the $a$-$n$-property. In the next section we return to this point.

A piece of code $X$ has the $a$-$n$-property if for some $i$, $|X|_i$ has this property. It is not hard to see that in this case $X$ contains at least $2^n - 1$ different $a$-tests. As an example, consider

$$X = !\,; \backslash b; + \backslash a; + /a; \backslash \#2; + /a; /\#2; /c; \#$$

Clearly, $X$ has the $a$-2-property because $|X|_4$ has this property: its 2-residuals are $b \circ \mathsf{S}$, $\mathsf{S}$, $\mathsf{D}$ and $c \circ \mathsf{D}$, so each thread is not equal to one of the others and does not start with an $a$-action.

Note that if a piece of code $X$ has the $a$-$(n+k)$-property, then it also has the $a$-$n$-property. In the example above, $X$ has the $a$-1-property because $|X|_3$ has this property (and $|X|_6$ too).

**Lemma 1.** *For each $k \in \mathbb{N}$ there exists $n \in \mathbb{N}^+$ such that no $X \in C_k$ has the $a$-$n$-property.*

*Proof.* Suppose the contrary and let $k$ be minimal in this respect. Assume for each $n \in \mathbb{N}^+$, $Y_n \in C_k$ has the $a$-$n$-property.

Let $B = \{\texttt{true}, \texttt{false}\}$. For $\alpha, \beta \in B^*$ we write

$$\alpha \preceq \beta$$

if $\alpha$ is a prefix of $\beta$, and we write $\alpha \prec \beta$ or $\beta \succ \alpha$ if $\alpha \preceq \beta$ and $\alpha \neq \beta$. Furthermore, let

$$B^{\leq n} = \bigcup_{i=0}^{n} B^i,$$

thus $B^{\leq n}$ contains all $B^*$-sequences $\alpha$ with $\ell(\alpha) \leq n$ (there are $2^{n+1} - 1$ such sequences).

Let $g : \mathbb{N} \to \mathbb{N}$ be such that $|Y_n|_{g(n)}$ has the $a$-$n$-property. Define

$$f_n : B^{\leq n} \to \mathbb{N}^+$$

by $f_n(\alpha) = m$ if the instruction reached in $Y_n$ when execution started at position $g(n)$ after the replies to $a$ according to $\alpha$ has position $m$. Clearly, $f_n$ is an injective function.

In the following claim we show that under the supposition made in this proof a certain form of squeezing holds: if $k'$ is sufficiently large, then for all $n > 0$ there exist $\alpha, \beta, \gamma \in B^{k'}$ with $f_{k'+n}(\alpha) < f_{k'+n}(\beta) < f_{k'+n}(\gamma)$ with the property that $f_{k'+n}(\alpha) < f_{k'+n}(\beta') < f_{k'+n}(\gamma)$ for each extension $\beta'$ of $\beta$ within $B^{\leq k'+n}$. This claim is proved by showing that not having this property implies that "too many" such extensions $\beta'$ exist. Using this claim it is not hard to contradict the minimality of $k$.

**Claim 1.** *Let $k'$ satisfy $2^{k'} \geq 2k + 3$. Then for all $n > 0$ there exist $\alpha, \beta, \gamma \in B^{k'}$ with*

$$f_{k'+n}(\alpha) < f_{k'+n}(\beta) < f_{k'+n}(\gamma)$$

*such that for each extension $\beta' \succeq \beta$ in $B^{\leq k'+n}$,*

$$f_{k'+n}(\alpha) < f_{k'+n}(\beta') < f_{k'+n}(\gamma).$$

*Proof of Claim 1.* Let $k'$ satisfy $2^{k'} \geq 2k + 3$. Towards a contradiction, suppose the stated claim is not true for some $n > 0$. The sequences in $B^{k'}$ are totally ordered by $f_{k'+n}$, say

$$f_{k'+n}(\alpha_1) < f_{k'+n}(\alpha_2) < \ldots < f_{k'+n}(\alpha_{2^{k'}}).$$

Consider the following list of sequences:

$$\alpha_1, \underbrace{\alpha_2, \ldots, \alpha_{2k+2}}_{\text{choices for } \beta}, \alpha_{2k+3}$$

By supposition there is for each choice $\beta \in \{\alpha_2, \ldots, \alpha_{2k+2}\}$ an extension $\beta' \succ \beta$ in $B^{\leq k'+n}$ with

$$\text{either} \quad f_{k'+n}(\beta') < f_{k'+n}(\alpha_1), \quad \text{or} \quad f_{k'+n}(\beta') > f_{k'+n}(\alpha_{2k+3}).$$

Because there are $2k+1$ choices for $\beta$, assume that at least $k+1$ elements $\beta \in \{\alpha_2, \ldots, \alpha_{2k+2}\}$ have an extension $\beta'$ with

$$f_{k'+n}(\beta') < f_{k'+n}(\alpha_1)$$

(the assumption $f_{k'+n}(\beta') > f_{k'+n}(\alpha_{2k+3})$ for at least $k+1$ elements $\beta$ with extension $\beta'$ leads to a similar argument). Then we obtain a contradiction with respect to $f_{k'+n}$: for each of the sequences $\beta$ in the subset just selected and its extension $\beta'$,

$$f_{k'+n}(\beta') < f_{k'+n}(\alpha_1) < f_{k'+n}(\beta),$$

and there are at least $k+1$ different such pairs $\beta, \beta'$ (recall $f_{k'+n}$ is injective). But this is not possible with jumps of at most $k$ because the $f_{k'+n}$ values of each of these pairs define a path in $Y_{k'+n}$ that never has a gap that exceeds $k$ and that passes position $f_{k'+n}(\alpha_1)$, while different paths never share a position. This finishes the proof of Claim 1. □

Take according to Claim 1 an appropriate value $k'$, some value $n > 0$ and $\alpha, \beta, \gamma \in B^{k'}$. Consider $Y_{k'+n}$ and mark the positions that are used for the computations according to $\alpha$ and $\gamma$: these computations both start in position $g(k'+n)$ and end in $f_{k'+n}(\alpha)$ and $f_{k'+n}(\gamma)$, respectively. Note that the set of marked positions never has a gap that exceeds $k$.

Now consider a computation that starts from instruction $f_{k'+n}(\beta)$ in $Y_{k'+n}$, a position in between $f_{k'+n}(\alpha)$ and $f_{k'+n}(\gamma)$. By Claim 1, the first $n$ $a$-instructions have positions in between $f_{k'+n}(\alpha)$ and $f_{k'+n}(\gamma)$ and none of these are marked. Leaving out all marked positions and adjusting the associated jumps yields a piece of code, say $Y$, with smaller jumps, thus in $C_{k-1}$, that has the $a$-$n$-property. Because $n$ was chosen arbitrarily, this contradicts the initial supposition that $k$ was minimal. □

**Theorem 7.** *For any $k \in \mathbb{N}^+$, not all threads in* BTA *can be expressed in $C_k$. This is also the case if thread extraction may start at arbitrary positions.*

*Proof.* Fix some value $k$. Then, by Lemma 1 we can find a value $n$ such that no $X \in C_k$ has the $a$-$n$-property. But we can define a finite thread that has this property. □

In the next section we discuss a systematic approach to define finite threads that have the $a$-$n$-property.

## 10 Boolean Registers for Producing Threads

In this section we briefly discuss the use of Boolean registers to ease programming in $C$. This is an example of so-called *thread-service* composition. In appendix C we provide a brief but general introduction to thread-service composition.

Consider *Boolean registers* named $b1, b2, \ldots, bn$ which all are initially set to $F$ (false) and can be set to $T$ (true). We write $bi(b)$ with $b \in \{T, F\}$ to indicate that $bi$'s value is $b$. The action $bi.set{:}b$ sets register $bi$ to $b$ and yields `true` as its reply. The action $bi.get$ reads the value from register $bi$ and provides this value as its reply. The defining rules for threads in BTA that use one of these registers are for $b, b' \in \{T, F\}$, $i \in \{1, \ldots, n\}$:

$$\begin{aligned}
\mathsf{S} \mathbin{/_{bi}} bi(b) &= \mathsf{S},\\
\mathsf{D} \mathbin{/_{bi}} bi(b) &= \mathsf{D},\\
(P \trianglelefteq bi.set{:}b' \trianglerighteq Q) \mathbin{/_{bi}} bi(b) &= P \mathbin{/_{bi}} bi(b'),\\
(P \trianglelefteq bi.get \trianglerighteq Q) \mathbin{/_{bi}} bi(b) &= \begin{cases} P \mathbin{/_{bi}} bi(b) & \text{if } b = T,\\ Q \mathbin{/_{bi}} bi(b) & \text{if } b = F, \end{cases}
\end{aligned}$$

and, if none of these rules apply,

$$(P \trianglelefteq a \trianglerighteq Q) \mathbin{/_{bi}} bi(b) = (P \mathbin{/_{bi}} bi(b)) \trianglelefteq a \trianglerighteq (Q \mathbin{/_{bi}} bi(b)).$$

The operator $/_{bi}$ is called the *use operator* and stems from [8]. Observe that the requests to the service $bi$ do not occur as actions in the behavior of a thread-service composition. So the composition hides the associated actions.

As a simple example consider the $C$-program $X$ that has extra instructions based on the set $\{bi.set{:}b, bi.get \mid b \in \{T,F\},\ i \in \{1,2\}\}$:

$$\begin{aligned} X = \ & +/a; /b1.set{:}T; \\ & +/a; /b2.set{:}T; \\ & +/b1.get; c; d; \\ & +/b2.get; c; d; \ ! \end{aligned}$$

Then one can derive (recall the initial value of $b1$ and $b2$ is $F$):

$$\begin{aligned} (|X|^{\rightarrow} /_{b1}\ b1) /_{b2}\ b2 &= (|X|_3 /_{b1}\ b1(T) \trianglelefteq a \trianglerighteq |X|_3 /_{b1}\ b1(F)) /_{b2}\ b2 \\ &= (R_1 \trianglelefteq a \trianglerighteq R_2) \trianglelefteq a \trianglerighteq (R_3 \trianglelefteq a \trianglerighteq R_4) \end{aligned}$$

where $R_1 = c \circ d \circ c \circ d \circ \mathsf{S}$ (case $T,T$), $R_2 = c \circ d \circ d \circ \mathsf{S}$ (case $T,F$), $R_3 = d \circ c \circ d \circ \mathsf{S}$ (case $F,T$), and $R_4 = d \circ d \circ \mathsf{S}$ (case $F,F$). So, the four possible combinations of the values of $b1$ and $b2$ yield the different 2-residuals $R_1, \ldots, R_4$. Clearly, $X$ has the $a$-2-property. The particular form of the $C$-program $X$ already suggests how to generalize $X$ to a family of $C$-programs $Z_n$ ($n \in \mathbb{N}^+$) such that

$$((|Z_n|^{\rightarrow} /_{b1}\ b1)...) /_{bn}\ bn$$

has the $a$-$n$-property:

$$\begin{aligned} Z_n = & +/a;\ /b1.set{:}T; \\ & +/a;\ /b2.set{:}T; \\ & \ldots \\ & +/a;\ /bn.set{:}T; \\ & +/b1.get;\ c; d; \\ & +/b2.get;\ c; d; \\ & \ldots \\ & +/bn.get;\ c; d; \ ! \end{aligned}$$

Each series of $n$ replies to the positive testinstructions $+/a$ has a unique continuation after which $Z_n$ terminates successfully: the number of `true`-replies matches the number of $c$-actions, and their ordering that of the occurring $d$-actions. Obviously, each thread $((|Z_n|^{\rightarrow} /_{b1}\ b1)...) /_{bn}\ bn$ is a finite thread in BTA and can thus be produced by a $C$-program not using Boolean registers (cf. Theorem 3).

More information about thread-service composition is given in Appendix C.

# 11 On the Length of *C*-Programs for Producing Threads

$C$-programs can be viewed as descriptions of finite state threads. In this section we consider the question which program length is needed to produce a finite state thread. We also consider the case that auxiliary Boolean registers are used for producing threads, which can be a very convenient feature as was shown in the previous section. We find upper and lower bounds for the lengths of $C$-programs.

For $k, n \in \mathbb{N}^+$ let

$\psi(k, n) \in \mathbb{N}^+$

be the minimal value such that each thread over alphabet $a_1, \ldots, a_k$ with at most $n$ states can be expressed as a $C$-program with at most $\psi(k, n)$ instructions. Furthermore, let

$\psi_{br}(k, n) \in \mathbb{N}^+$

be the minimal value such that each thread over alphabet $a_1, \ldots, a_k$ with at most $n$ states can be expressed as a $C$-program with at most $\psi_{br}(k, n)$ instructions including those to use Boolean registers.

It is not hard to see that

$\psi(k, n) \leq 3n \quad \text{and} \quad \psi_{br}(k, n) \leq 3n$

because each state can be described by either the piece of code

$+/a_i; u; v$

with $u$ and $v$ jumps to the pieces of code that model the two successor states, or by ! or # . Presumably, a sharper upper bound for both $\psi(k, n)$ and $\psi_{br}(k, n)$ can be found.

As for a lower bound for $\psi_{br}(k, n)$, we can use auxiliary Boolean registers by forward basic instructions

$/bi.set{:}T$
$/bi.set{:}F$
$/bi.get$

and their backward and test counterparts. So, each Boolean register $bi$ comes with 18 different instructions, and of course at most $\psi_{br}(k,n)$ of these can be used.

Programs containing at most $l = \psi_{br}(k,n)$ instructions, contain per position $i$ at most $l-1$ jump instructions, namely jumps to all other (at most $l-1$) positions in the program.

So, if we restrict to $k=1$, say $/a$ is the only forward basic instruction involved (with backward and test variants yielding 5 more instructions) and include the termination instruction ! and the abort instruction # , the admissible instruction alphabet counts

$$2 + 6 + (l-1) + 18l$$

instructions. Because $l \geq 1$, this is bounded by $26l$ instructions, and therefore we count

$$(26l)^l$$

syntactically different programs.

A lower bound on the number of threads with $n$ states over one action $a$ can be estimated as follows: let $F$ range over all functions

$$\{1,\ldots,n-1\} \mapsto \{0,1,\ldots,n-1\},$$

thus there are $n^{n-1}$ different $F$. Define threads $P_k^F$ for $k = 0,\ldots,n-1$ by

$$\begin{aligned} P_0^F &= \mathsf{S} \\ P_{i+1}^F &= P_{F(i+1)}^F \trianglelefteq a \trianglerighteq P_i^F \end{aligned}$$

We claim that for a fixed $n$ the threads $P_{n-1}^F$ (each one containing $n$ states $P_0^F,\ldots,P_{n-1}^F$), are for each $F$ different, thus yielding $n^{n-1}$ different threads, so we find

$$(26l)^l \geq n^{n-1}. \tag{8}$$

Assume $n \geq 2$, thus $26 \leq 25n$, thus $n \leq 26n-26$, thus $\dfrac{n}{26} \leq n-1$. Suppose $l < \dfrac{n}{26}$, then $26l < n$ and $l < n-1$, which contradicts (8). Thus

$$l \geq \frac{n}{26}.$$

So, for $k = 1$ and in fact for arbitrary $k \geq 1$ we find

$$\frac{n}{26} \leq \psi_{br}(k, n) \leq 3n.$$

In the case that we do not allow the use of auxiliary Boolean registers, it follows in a same manner as above that for arbitrary $k \geq 1$,

$$\frac{n}{8} \leq \psi(k, n) \leq 3n.$$

We see it as a challenging problem to improve the bounds of $\psi_{br}(k, n)$ and $\psi(k, n)$.

## 12 Discussion

In this paper we proposed an algebra of instruction sequences based on a set of instructions without directional bias. The use of the phrase "instruction sequence" asks for some rigorous motivation. This is a subtle matter which defeats many common sense intuitions regarding the science of computer programming.

The Latin source of the word 'instruction' tells us no more than that the instruction is part of a listing. On that basis, instruction sequence is a pleonasm and justification is problematic.[6] We need to add the additional connotation of instruction as a "unit of command". This puts instructions at a core position. Maurer's paper *A theory of computer instructions* [12] provides a theory of instructions which can be taken on board in an attempt to define what is an instruction in this more narrow sense. Now Maurer's instructions certainly qualify as such but his survey is not exhaustive. His theory has an intentional focus on transformation of data while leaving change of control unexplained. We hold that Maurer's theory, including his ongoing work on this theme in [13], provides a candidate definition for so-called basic instructions.

At this stage different arguments can be used to make progress. Suppose a collection $\mathcal{I}$ is claimed to constitute a set of instructions:

[6] [10]: INSTRUCTION, in Latin *instructio*, comes from *in* and *struo* to dispose or regulate, signifying the thing laid down.

The following is taken from `http://www.etymonline.com/`. INSTRUCTION: from O.Fr. instruction, from L. instructionem (nom. instructio) "building, arrangement, teaching," from instructus, pp. of instruere "arrange, inform, teach," from in- "on" + struere "to pile, build" (see structure).

1. If the mnemonics of elements of $\mathcal{I}$ are reminding of known instructions of some low level program notations, and if the semantics provided complies with that view, the use of these terms may be considered justified.

2. If, however, unknown, uncommon or even novel instructions are included in $\mathcal{I}$, the argument of 1 can not be used. Of course some similarity of explanation can be used to carry the jargon beyond conventional use. At some stage, however, a more intrinsic justification may be needed.

3. A different perspective emerges if one asserts that certain instruction sequences constitute programs, thus considering $\mathcal{I}^+$ (i.e., finite, non-empty sequences of instructions from $\mathcal{I}$) one may determine a subset $\mathcal{P} \subseteq \mathcal{I}^+$ of programs. Now a sequence in $\mathcal{I}^+$ qualifies as a program if and only if it is in $\mathcal{P}$. In the context of $C$-expressions we say that

   $$+/a; \backslash\#10; /b; +/c; /\#8; \,!\,; \,!$$

   is not in $\mathcal{P}$ because the jumps outside the range of instructions cannot be given a natural and preferred semantics, as opposed to $+/a; \backslash\#1; \,!$ and $+/a; /b; +/c; \,!\,; \,!\,$. We here state once more that we do *not* consider the empty sequence of instructions as a program, or even as an instruction sequence because we have no canonical meaning or even intuition about such an empty sequence in this context.

4. The next question is how to determine $\mathcal{P}$. At this point we make use of the framework of PGA [7, 14] (for a brief explanation of PGA see Appendix A). A program is a piece of data for which the preferred and natural meaning is a "sequence of primitive instructions", abbreviated to a SPI. Primitive instructions are defined over some collection $A$ of basic instructions. The meaning of a program $X$ is by definition provided by means of a projection function which produces a SPI for $X$. Using PGA as a notation for SPIs, the projection function can be written `p2pga` ("$\mathcal{P}$ to PGA"). The behavior $|X|_{\mathcal{P}}$ for $X \in \mathcal{P}$ is given by

   $$|X|_{\mathcal{P}} = |\mathtt{p2pga}(X)|$$

   where thread extraction in PGA, i.e., $|\ldots|$, is supposed to be known.

5. In the particular case of $\mathcal{I}$ consisting of $C$'s instructions, we take for $\mathcal{P}$ those instruction sequences for which control never reaches outside the sequence. These are the sequences that we called $C$-programs. First we restrict to $C$-programs composed from instructions in $\{/a, +/a, -/a, /\#k, \backslash\#k, !, \# \mid a \in A, k \in \mathbb{N}^+\}$ and we define

   $$F(i_1; \ldots; i_n) = (\psi(i_1); \ldots; \psi(i_n))^\omega$$

   as a "pre-projection function" that uses an auxiliary function $\psi$ on these instructions:

   $$\begin{aligned} \psi(/a) &= a, \\ \psi(+/a) &= +a, \\ \psi(-/a) &= -a, \\ \psi(/\#k) &= \#k, \\ \psi(\backslash\#k) &= \#n - k, \\ \psi(!) &= !, \\ \psi(\#) &= \#0. \end{aligned}$$

   We can rewrite each $C$-program into this restricted form by applying the behavior preserving homomorphism $h$ defined in Section 6. Thus our final definition of a projection can be $\texttt{p2pga} = F \circ h$. Note that many alternatives for $h$ could have been used as well (as was already noted in Section 6).

6. Conversely, each PGA-program can be embedded into $C$ while its behavior is preserved. For repetition free programs this embedding is defined by the addition of forward slashes and replacing #0 by # .[7] In the other case, a PGA-program can be embedded into PGLB, a variant of PGA with backward jumps and no repetition operator [7], and transformation from PGLB to $C$ is trivial.

In the case of $C$, items 4 and 5 above should of course be *proved*, i.e., for a $C$-program $X$,

$$|X|^{\rightarrow} = |X|_C \quad (= |\texttt{p2pga}(X)|),$$

and for item 6 a similar requirement about the definition of $|\ldots|^{\rightarrow}$ should be substantiated. We omit these proofs as they seem rather clear.

[7] The instruction # already occurred in [6], but was in [7] replaced by #0, thus admitting a more systematic treatment of "jumps".

**Acknowledgements**

We thank Stephan Schroevers and anonymous referees for their useful comments and for pointing out some errors.

## References

[1] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1-2):70-120,1982.

[2] A. Barros and T. Hou. A constructive version of AIP revisited. Technical report PRG0802, University of Amsterdam, January 2008. Available via www.science.uva.nl/research/prog/publications.html.

[3] J.A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J.C.M. Baeten, J.K. Lenstra, J. Parrow, G.J. Woeginger, eds., *Proceedings of ICALP 2003*, LNCS 2719, pages 1-21, Springer-Verlag, 2003.

[4] J.A. Bergstra and I. Bethke. Polarized process algebra with reactive composition. *Theoretical Computer Science*, 343(3):285-304, 2005.

[5] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109-137, 1984.

[6] J.A. Bergstra and M.E. Loots. Program algebra for component code. *Formal Aspects of Computing*, 12(1):1-17, 2000.

[7] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125-156, 2002.

[8] J.A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming* 51(2):175-192, 2002.

[9] J.A. Bergstra and A. Ponse. An instruction sequence semigroup with repeaters. arXiv:0810.1151v1 [cs.PL] at http://arxiv.org/, 2008.

[10] George Crabb. *English Synonyms Explained, in Alphabetical Order: With Copious Illustrations and Examples Drawn from the Best Writers*. Published by Baldwin, Cradock, 1818. Original from the New York Public Library, Digitized Sep 25, 2006, 904 pages.

[11] M. Hazewinkel. Encyclopaedia of Mathematics: an updated and annotated translation of the Soviet "Mathematical Encyclopaedia". Springer-Verlag, 2002.

[12] W.D. Maurer. A theory of computer instructions. *Science of Computer Programming*, 60:244-273, 2006. (A shorter version of this paper was published in the *Journal of the ACM*, 13(2): 226–235, 1966.)

[13] W.D. Maurer. Partially defined computer instructions and guards. *Science of Computer Programming*, 72(3):220-239, 2008.

[14] A. Ponse and M.B. van der Zwaag. An introduction to program and thread algebra. In A. Beckmann et al. (editors), *Logical Approaches to Computational Barriers: Proceedings CiE 2006*, LNCS 3988, pages 445-458, Springer-Verlag, 2006.

[15] T.D. Vu. Denotational semantics for thread algebra. *Journal of Logic and Algebraic Programming*, 74(2):94-111, 2008.

# A PGA, a summary

Let a set $A$ of constants with typical elements $a, b, c, \ldots$ be given. PGA-programs are of the following form ($a \in A,\ k \in \mathbb{N}$):

$$P ::= a \mid +a \mid -a \mid \#k \mid ! \mid P;P \mid P^\omega.$$

Each of the first five forms above is called a *primitive instruction*. We write $\mathcal{U}$ for the set of primitive instructions and we define each element of $\mathcal{U}$ to be a SPI (Sequence of Primitive Instructions).

Finite SPIs are defined using *concatenation*: if $P$ and $Q$ are SPIs, then so is

$$P;Q$$

which is the SPI that lists $Q$'s primitive instructions right after those of $P$, and we take concatenation to be an *associative* operator.

Periodic SPIs are defined using the repetition operator: if $P$ is a SPI, then

$$P^\omega$$

is the SPI that repeats $P$ forever, thus $P;P;P;\ldots$. Typical identities that relate repetition and concatenation of SPIs are

$$(P;P)^\omega = P^\omega \quad \text{and} \quad (P;Q)^\omega = P;(Q;P)^\omega.$$

Another typical identity is

$$P^\omega; Q = P^\omega,$$

expressing that nothing "can follow" an infinite repetition.

The execution of a SPI is *single-pass*: it starts with the first (left-most) instruction, and each instruction is dropped after it has been executed or jumped over.

Equations for thread extraction on SPIs, notation $|X|$, are the following, where $a$ ranges over $A$, $u$ over the primitive instructions $\mathcal{U}$, and $k \in \mathbb{N}$:

$$\begin{aligned}
|!| &= \mathsf{S} & |!; X| &= \mathsf{S} \\
|\mathsf{a}| &= \mathsf{a} \circ \mathsf{D} & |\mathsf{a}; X| &= \mathsf{a} \circ |X| \\
|{+}\mathsf{a}| &= \mathsf{a} \circ \mathsf{D} & |{+}\mathsf{a}; X| &= |X| \trianglelefteq \mathsf{a} \trianglerighteq |\#2; X| \\
|{-}\mathsf{a}| &= \mathsf{a} \circ \mathsf{D} & |{-}\mathsf{a}; X| &= |\#2; X| \trianglelefteq \mathsf{a} \trianglerighteq |X| \\
|\#k| &= \mathsf{D} & |\#0; X| &= \mathsf{D} \\
& & |\#1; X| &= |X| \\
& & |\#k{+}2; u| &= \mathsf{D} \\
& & |\#k{+}2; u; X| &= |\#k{+}1; X|
\end{aligned}$$

For more information on PGA we refer to [7, 14].

# B Basic Thread Algebra and Finite Approximations

An elegant result based on [2] is that equality of recursively specified regular threads can be easily decided. Because one can always take the disjoint union of two finite linear recursive specifications, it suffices to consider a single specification $\{P_i = t_i \mid 1 \leq i \leq n\}$. Then $P_i = P_j$ follows from

$$\pi_{n-1}(P_i) = \pi_{n-1}(P_j).$$

Thus, it is sufficient to decide whether two certain finite threads are equal. We provide a proof sketch:

For $k \geq 0$ consider the equivalence relation $\cong_k$ on $\{P_1, \ldots, P_n\}$ defined by $P_i \cong_k P_j$ if $\pi_k(P_i) = \pi_k(P_j)$. Then

$$\cong_0 \supseteq \cong_1 \supseteq \cong_2 \supseteq \ldots \tag{9}$$

If $\cong_k = \cong_{k+1}$ then $\cong_{k+1} = \cong_{k+2}$. This follows from (9) and $\cong_{k+1} \subseteq \cong_{k+2}$. Suppose the latter is not true, then $\pi_{k+1}(P_i) = \pi_{k+1}(P_j)$ while $\pi_{k+2}(P_i) \neq \pi_{k+2}(P_j)$.

The only possible cases are that $P_i = P_m \trianglelefteq a \trianglerighteq P_l$ and $P_j = P_{m'} \trianglelefteq a \trianglerighteq P_{l'}$ and $\pi_{k+1}(P_m) \neq \pi_{k+1}(P_{m'})$ or $\pi_{k+1}(P_l) \neq \pi_{k+1}(P_{l'})$. So by $\cong_k = \cong_{k+1}$, at least one of $\pi_k(P_m) \neq \pi_k(P_{m'})$ and $\pi_k(P_l) \neq \pi_k(P_{l'})$ must be true, but this refutes $\pi_{k+1}(P_i) = \pi_{k+1}(P_j)$. So, once the sequence (9) becomes constant, it remains constant. Since this sequence is decreasing and the maximum number of equivalence classes on $\{P_1, \ldots, P_n\}$ is $n$, at most the first $n$ relations in the sequence can be unequal, hence $\cong_{n-1} = \cong_n$, and thus $\pi_{n-1}(P_i) = \pi_{n-1}(P_j)$ implies $\pi_k(P_i) = \pi_k(P_j)$ for all $k \in \mathbb{N}$.

It is not difficult to show for threads $P$ and $Q$: if $\pi_k(P) = \pi_k(Q)$ for all $k \in \mathbb{N}$ then $P = Q$. First, each (infinite) thread is a projective sequence on which $\pi_k$ is defined componentwise. Secondly, for a projective sequence $(P_n)_{n\in\mathbb{N}}$ it follows that $\pi_k(P_k) = \pi_k(\pi_k(P_{k+1}) = \pi_k(P_{k+1}) = P_k$ for all $k \in \mathbb{N}$. So, for $(Q_n)_{n\in\mathbb{N}}$ a projective sequence, $P_k = \pi_k(P_k) = \pi_k(Q) = Q_k$ for all $k$ implies $(P_n)_{n\in\mathbb{N}} = (Q_n)_{n\in\mathbb{N}}$.

# C Thread-Service Composition

Most of this text is taken from [14]. A *service*, or a *state machine*, is a pair $\langle \Sigma, F \rangle$ consisting of a set $\Sigma$ of so-called *co-actions* and a reply function $F$. The reply function is a mapping that gives for each non-empty finite sequence of co-actions from $\Sigma$ a reply `true` or `false`.

**Example 2.** *A* stack *can be defined as a service with co-actions push:$i$, topeq:$i$, and pop, for $i = 1, \ldots, n$ for some $n$, where push:$i$ pushes $i$ onto the stack and yields* `true`*, the action topeq:$i$ tests whether $i$ is on top of the stack, and pop pops the stack with reply* `true` *if it is non-empty, and it yields* `false` *otherwise.*

Services model (part of) the execution environment of threads. In order to define the interaction between a thread and a service, we let actions be of the form $c.m$ where $c$ is the so-called *channel* or *focus*, and $m$ is the co-action or *method*. For example, we write *s.pop* to denote the action which pops a stack via channel $s$. For service $\mathcal{H} = \langle \Sigma, F \rangle$ and thread $P$, $P \mathbin{/_c} \mathcal{H}$ represents $P$ *using the service* $\mathcal{H}$ via channel $c$. The defining rules for threads in BTA are:

$$\begin{aligned}
\mathsf{S} \mathbin{/_c} \mathcal{H} &= \mathsf{S},\\
\mathsf{D} \mathbin{/_c} \mathcal{H} &= \mathsf{D},\\
(P \trianglelefteq c'.m \trianglerighteq Q) \mathbin{/_c} \mathcal{H} &= (P \mathbin{/_c} \mathcal{H}) \trianglelefteq c'.m \trianglerighteq (Q \mathbin{/_c} \mathcal{H}) \quad \text{if } c' \neq c,\\
(P \trianglelefteq c.m \trianglerighteq Q) \mathbin{/_c} \mathcal{H} &= P \mathbin{/_c} \mathcal{H}' \quad \text{if } m \in \Sigma \text{ and } F(m) = \texttt{true},\\
(P \trianglelefteq c.m \trianglerighteq Q) \mathbin{/_c} \mathcal{H} &= Q \mathbin{/_c} \mathcal{H}' \quad \text{if } m \in \Sigma \text{ and } F(m) = \texttt{false},\\
(P \trianglelefteq c.m \trianglerighteq Q) \mathbin{/_c} \mathcal{H} &= \mathsf{D} \quad \text{if } m \notin \Sigma,
\end{aligned}$$

where $\mathcal{H}' = \langle \Sigma, F' \rangle$ with $F'(\sigma) = F(m\sigma)$ for all co-action sequences $\sigma \in \Sigma^+$.

The operator $/_c$ is called the *use operator* and stems from [8]. An expression $P \mathbin{/_c} \mathcal{H}$ is sometimes referred to as a *thread-service composition*. The use operator

is expanded to infinite threads in $\mathrm{BTA}^\infty$ by defining

$$(P_n)_{n\in\mathbb{N}} \mathbin{/_c} \mathcal{H} = \bigsqcup_{n\in\mathbb{N}} P_n \mathbin{/_c} \mathcal{H}.$$

(Cf. [4].) It follows that the rules for finite threads are valid for infinite threads as well. Observe that the requests to the service do not occur as actions in the behavior of a thread-service composition. So the composition not only reduces the above-mentioned non-determinism of the thread, but also hides the associated actions.

In the next example we show that the use of services may turn regular threads into non-regular ones.

**Example 3.** *We define a thread using a stack as defined in Example 2. We only push the value* 1 *(so the stack behaves as a counter), and write* $S(n)$ *for a stack holding* $n$ *times the value* 1*. By the defining equations for the use operator it follows that for any thread* $P$*,*

$$\begin{aligned}(s.push{:}1 \circ P) \mathbin{/_s} S(n) &= P \mathbin{/_s} S(n{+}1),\\ (P \trianglelefteq s.pop \trianglerighteq \mathsf{S}) \mathbin{/_s} S(0) &= \mathsf{S},\\ (P \trianglelefteq s.pop \trianglerighteq \mathsf{S}) \mathbin{/_s} S(n{+}1) &= P \mathbin{/_s} S(n).\end{aligned}$$

*Now consider the regular thread* $Q$ *defined by*

$$Q = s.push{:}1 \circ Q \trianglelefteq a \trianglerighteq R, \quad R = b \circ R \trianglelefteq s.pop \trianglerighteq \mathsf{S},$$

*where actions* $a$ *and* $b$ *do not use focus* $s$*. Then, for all* $n \in \mathbb{N}$*,*

$$\begin{aligned}Q \mathbin{/_s} S(n) &= (s.push{:}1 \circ Q \trianglelefteq a \trianglerighteq R) \mathbin{/_s} S(n)\\ &= (Q \mathbin{/_s} S(n{+}1)) \trianglelefteq a \trianglerighteq (R \mathbin{/_s} S(n)).\end{aligned}$$

*It is not hard to see that* $Q \mathbin{/_s} S(0)$ *is an infinite thread with the property that for all* $n$*, a trace of* $n+1$ $a$*-actions produced by* $n$ *positive and one negative reply on* $a$ *is followed by* $b^n \circ \mathsf{S}$*. This yields an* non-regular *thread: if* $Q \mathbin{/_s} S(0)$ *were regular, it would be a fixed point of some finite linear recursive specification, say with* $k$ *equations. But specifying a trace* $b^k \circ \mathsf{S}$ *already requires* $k+1$ *linear equations* $x_1 = b \circ x_2, \ldots, x_k = b \circ x_{k+1}, x_{k+1} = \mathsf{S}$*, which contradicts the assumption. So* $Q \mathbin{/_s} S(0)$ *is not regular.*

Finally, we note that the use of finite state services, such as Boolean registers, can *not* turn regular threads into non-regular ones (see [8]). More information on thread-service composition can be found in e.g. [14].