An Instruction Sequence Semigroup with Involutive Anti-Automorphisms

Bergstra, Jan A.; Ponse, Alban

Publication date
2009

Document Version
Final published version

Link to publication

Citation for published version (APA):
Bergstra, J. A., & Ponse, A. (2009). *An Instruction Sequence Semigroup with Involutive Anti-Automorphisms*. (1 ed.) Section Software Engineering, University of Amsterdam. https://arxiv.org/abs/0903.1352v1

An Instruction Sequence Semigroup with Involutive Anti-Automorphisms [*]

Jan A. Bergstra
Alban Ponse

Section Software Engineering, Informatics Institute, University of Amsterdam
URL: `www.science.uva.nl/~{janb/,alban/}`

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, and we distinguish behavioral equivalence and congruence. Then we consider thread extraction compatible (anti-)homomorphisms and (anti-)automorphisms. Finally we axiomatize structural congruence under which we can exclude chained jumps and discuss some expressiveness results.

1 Introduction

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

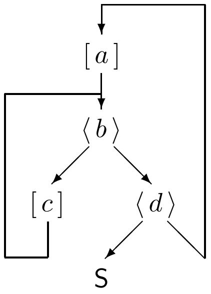
1. Pieces of code, i.e., finite *sequences of instructions*, given some set $\mathcal{I}$ of instructions. A (computer) *program* is in our case a piece of code that satisfies the additional property that each state of its execution is prescribed by an instruction (typically, there are no jumps outside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

[*] The authors acknowledge support from the NWO project Thread Algebra for Strategic Interleaving.

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 $\langle\, d\,\rangle$);

S is the state that models successful 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 [1]. 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[1] transforms each C-program into one of which the interpretation from right to left produces the same thread as the original program. Furthermore, many homomorphisms and automorphisms can be defined in this setting, allowing for a systematic treatment of various behavior preserving congruences.

The paper is structured as follows:

- In Section 2 we briefly review finite state 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-programs.
- Section 5 is about a thread extraction preserving homomorphism on C and a related anti-homomorphism.

[1] We refer to [5] as a general reference for more information about these and related algebraic notions.

- In Section 6 we relate a certain class of bijections on threads to a class of automorphisms on C, and in Section 7 we do the same thing for a related class of anti-automorphisms on C.
- In Section 8 we discuss structural congruence, a congruence that identifies C-programs if they only differ in their jump counters and are behaviorally equivalent. We use structural congruence to prove a first, basic expressiveness result.
- In Section 9 we establish that only certain test instructions in C are necessary to preserve expressiveness, while we cannot restrict to a bound on the counters of jump instructions.
- In Section 10 we relate the length of a C-program to the number of states of the thread it produces.
- In Section 11 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 a small Appendix A on PGA.

2 Threads

In this section we briefly review finite state threads — also called *regular threads* — in the setting of program algebra.

Assume A is a set of *polarized actions* $a, b, c, \ldots$. The adjective 'polarized' means here that upon the execution of an action the (execution) environment provides a Boolean reply `true` or `false`. Polarized actions are used to define *threads*, the mathematical objects that we will use for program semantics. Finite threads are defined inductively:

$$\begin{array}{rcl} \mathsf{S} & - & \textit{stop}\text{, the terminated thread,} \\ \mathsf{D} & - & \textit{inaction}\text{ or }\textit{deadlock}\text{, the inactive thread,} \\ P \trianglelefteq a \trianglerighteq Q & - & \text{for each action } a \in A\text{, the }\textit{postconditional composition}\text{ of } P \text{ and } Q, \\ & & \text{where } P \text{ and } Q \text{ are finite threads.} \end{array}$$

The behavior of a postconditional composition can be characterized as follows:

$$\text{A thread } P \trianglelefteq a \trianglerighteq Q \text{ starts with the } \textit{action } a \text{ and upon reply } \begin{cases} \texttt{true} & \text{continues as } P, \\ \texttt{false} & \text{continues as } Q. \end{cases}$$

Note that finite threads always end in S or D. We use *action prefix*, notation

$$a \circ P,$$

as an abbreviation for $P \trianglelefteq a \trianglerighteq P$ and take $\circ$ to bind strongest. The set of finite threads over A is denoted by

$$\mathbb{T}.$$

A *regular* thread is a finite state thread in which infinite paths can occur. Each regular thread can be defined by a finite number of equations of the form

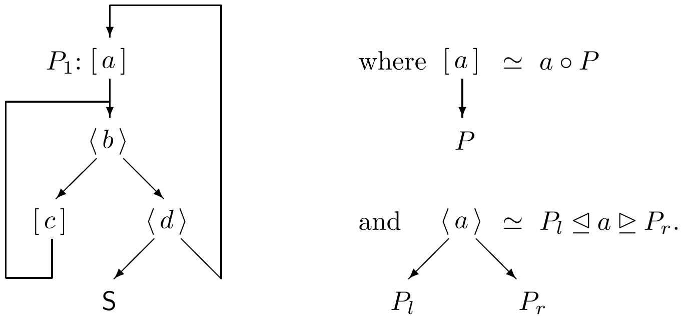
$$P_i = t_i$$

with $t_i ::= \mathsf{S} \mid \mathsf{D} \mid P_j \trianglelefteq a_i \trianglerighteq P_k$ and selecting one the P_i's as its identifier. Of course, there should be an equation for each identifier that occurs in one of the right-hand sides.

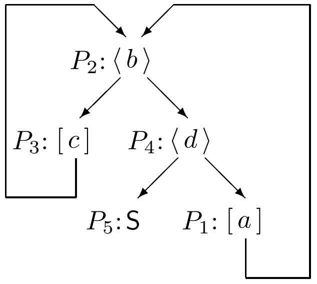
As an example, consider the regular thread P_1 specified by these five equations:

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

The regular thread P_1 can be visualized as follows:


P_1: [a]
⟨ b ⟩
[c]
⟨ d ⟩
S
where [a] $\simeq$ $a \circ P$
P
and ⟨ a ⟩ $\simeq$ $P_l \trianglelefteq a \trianglerighteq P_r$.
P_l
P_r

Also P_2 as defined above is a regular thread (one that starts with a b-action) and it is tempting to draw P_2 as the topmost state in a visualization, and (perhaps) also to provide each state with its "name":



The equations introduced for the specification of regular threads are further called *linear equations.*

Observe that the thread P_2 discussed above can also be specified by a single equation that is not linear:

$$P_2 = c \circ P_2 \trianglelefteq b \trianglerighteq (\mathsf{S} \trianglelefteq d \trianglerighteq a \circ P_2).$$

However, this equation does not provide names for states different from P_2. Therefore we stick to the convention to specify each regular thread by a set of linear equations.

Observe that a set of linear equations need not specify a regular thread that covers each state. For example, replacing P_2's equation above by

$$P_2 = P_2 \trianglelefteq b \trianglerighteq P_4$$

turns P_3 into a state that cannot be reached from one of the other states.

Given a set A of actions, we write

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

for the set of regular threads over A. More information about this type of regular threads can be found in [8].

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 $\mathbb{T}$ and $\mathbb{T}_{\text{reg}}$).

For $a \in A$ and k ranging over $\mathbb{N}$, C-expressions are of the following form:

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

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. The special case $/\#0$ prescribes deadlock.

$\backslash a$, $+\backslash a$, $-\backslash a$ 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*, prescribing successful termination.

For C there are two axioms:

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

$$/\#0 = \backslash\#0. \tag{2}$$

By axiom (1), 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. Axiom (2) implies that deadlock can be defined by both the instructions $/\#0$ and $\backslash\#0$. The instructions for deadlock and termination (!) are the only instructions that do not specify further control of execution.

Note that we could have introduced extended jump instructions $/\#k$ with $k \in \mathbb{Z}$ and axiom $/\#k = \backslash\#{-k}$, but we did not do so because jumps use a jump-*counter* and counters range over $\mathbb{N}$.

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 11.

$$
\text{if } j \in \{1,\dots,n\}, \text{ then } |X|_j = \begin{cases}
a \circ |X|_{j+1} & \text{if } i_j = /a, \\
|X|_{j+1} \unlhd a \unrhd |X|_{j+2} & \text{if } i_j = +/a, \\
|X|_{j+2} \unlhd a \unrhd |X|_{j+1} & \text{if } i_j = -/a, \\
\mathsf{D} & \text{if } i_j = /\#0, \\
|X|_{j+k} & \text{if } i_j = /\#k \text{ and } k > 0, \\
 & \\
a \circ |X|_{j-1} & \text{if } i_j = \backslash a, \\
|X|_{j-1} \unlhd a \unrhd |X|_{j-2} & \text{if } i_j = +\backslash a, \\
|X|_{j-2} \unlhd a \unrhd |X|_{j-1} & \text{if } i_j = -\backslash a, \\
\mathsf{D} & \text{if } i_j = \backslash\#0, \\
|X|_{j-k} & \text{if } i_j = \backslash\#k \text{ and } k > 0, \\
 & \\
\mathsf{S} & \text{if } i_j = !\,,
\end{cases}
$$

$$
\text{if } j \notin \{1,\dots,n\}, \text{ then } |X|_j = \mathsf{D}. \tag{3}
$$

Table 1: Equations for thread extraction $|X|_j$ for $X = i_1; \dots ; i_n$ and $j \in \mathbb{Z}$

4 C-Programs and Thread Extraction

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 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 4.

If these equations define a loop without having generated any behavior, as in

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

the extracted behavior is defined as D. Such loops are necessarily the result of "chained jumps", i.e., jumps to jump instructions. In Section 8 we introduce so-called "canonical C-forms", which exclude this kind of loops in their thread extractions.

Thread extraction defines an equivalence on C-expressions that is not a congruence, e.g.,

$$|/\#0|^{\rightarrow} = |/\#1|^{\rightarrow} \quad \text{but} \quad |/\#0;/a|^{\rightarrow} \neq |/\#1;/a|^{\rightarrow}.$$

We write

$$\mathbb{N}^+ \quad \text{for} \quad \mathbb{N} \setminus \{0\}$$

and 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.

Right-to-left thread extraction also defines an equivalence on C-expressions that is not a congruence, e.g.,

$$|\backslash\#0|^{\leftarrow} = |\backslash\#1|^{\leftarrow} \quad \text{but} \quad |/a;\backslash\#0|^{\leftarrow} \neq |/a;\backslash\#1|^{\leftarrow}.$$

Definition 1. *Two C-expressions X and Y are **behaviorally equivalent**, notation*

$$X \equiv_{be} Y,$$

if $|X|^{\rightarrow} = |Y|^{\rightarrow}$ and $|X|^{\leftarrow} = |Y|^{\leftarrow}$.

As mentioned above, a simple example is $/\#0 \equiv_{be} /\#1$.

Behavioral congruence $=_{be}$ is the largest congruence contained in behavioral equivalence. Some typical axioms for this congruence are

$$+/a;/\#1 =_{be} /a;/\#1 \quad \text{and} \quad \backslash\#1;+\backslash a =_{be} \backslash\#1;\backslash a$$

(and similar for the negative tests), and

$$/a;\backslash\#1 =_{be} /\#1;\backslash a \quad \text{and} \quad +/a;/a;\backslash\#2 =_{be} /\#2;\backslash a;-\backslash a.$$

We refrain from attempting to axiomatize this congruence.

Definition 2. *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* (3)*. In other words, there are no jumps outside the range of X and execution can only end by executing either the termination instruction ! or deadlock $/\#0$ (or $\backslash\#0$).*

In the setting of program algebra we explicitly distinguished in [3] 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 are $X;Y$ and $Y;X$. A piece of code that is not a program can be called a *program fragment* because it can always 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. $|X|_k = |/\#0;X|_{k+1}$,
2. $|X|_k = |X;\backslash\#0|_k$.

Moreover, in the case that X is a C-program,

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

$$\begin{aligned}|{+}/a;\backslash\#2|^{\rightarrow} &= |{+}/a;\backslash\#2|_1\\ &= |/\#0;{+}/a;\backslash\#2;\backslash\#0|_2,\end{aligned}$$

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:

$$|/\#0;{+}/a;\backslash\#2;\backslash\#0|_2 = |/\#2;/\#0;{+}/a;\backslash\#2;\backslash\#0|^{\rightarrow}.$$

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

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

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

5 Thread Extraction Preserving Homomorphisms

In this section we consider some functions on C that preserve the behavior defined.

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

$$\begin{aligned}/a &\mapsto /a;/\#2;/\#0,\\ {+}/a &\mapsto {+}/a;/\#2;/\#4,\\ {-}/a &\mapsto {-}/a;/\#2;/\#4,\\ /\#k &\mapsto /\#3k;/\#0;/\#0,\end{aligned}$$

$$\begin{aligned}\backslash a &\mapsto /a;\backslash\#4;\backslash\#0,\\ {+}\backslash a &\mapsto {+}/a;\backslash\#4;\backslash\#8,\\ {-}\backslash a &\mapsto {-}/a;\backslash\#4;\backslash\#8,\\ \backslash\#k &\mapsto \backslash\#3k;\backslash\#0;\backslash\#0,\end{aligned}$$

$$! \mapsto !\,;/\#0;/\#0.$$

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:

$$\begin{aligned} /a &\mapsto +/a;/\#2;/\#1,\\ -/a &\mapsto +/a;/\#5;/\#1,\\ \backslash a &\mapsto +/a;\backslash\#4;\backslash\#5,\\ -\backslash a &\mapsto +/a;\backslash\#5;\backslash\#7, \end{aligned}$$

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 $\{+/a, /\#k, \backslash\#k, ! \mid a \in A,\ k \in \mathbb{N}\}$. We will come back to this issue in Section 9.

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

$$\begin{aligned} /a &\mapsto \backslash\#0;\backslash\#2;\backslash a,\\ +/a &\mapsto \backslash\#4;\backslash\#2;+\backslash a,\\ -/a &\mapsto \backslash\#4;\backslash\#2;-\backslash a,\\ /\#k &\mapsto \backslash\#0;\backslash\#0;\backslash\#3k,\\[1ex] \backslash a &\mapsto /\#0;/\#4;\backslash a,\\ +\backslash a &\mapsto /\#8;/\#4;+\backslash a,\\ -\backslash a &\mapsto /\#8;/\#4;-\backslash a,\\ \backslash\#k &\mapsto /\#0;/\#0;/\#3k,\\[1ex] ! &\mapsto \backslash\#0;\backslash\#0;\,!\,. \end{aligned}$$

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 7.

6 Structural Bijections and TEC-Automorphisms

In this section we consider the relation between a certain class of bijections on threads and an associated class of automorphisms on C.

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

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

Structural bijections naturally extend to $\mathbb{T}_{\text{reg}}$ by distribution over (linear) equations.

Theorem 1. *There are* $2^{|A|} \cdot |A|!$ *structural bijections on* $\mathbb{T}$*, and thus on* $\mathbb{T}_{reg}$.

Proof. Trivial: if $|A| = n$, there are 2^n different partitionings in $A_{\texttt{true}}$ and $A_{\texttt{false}}$, 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 ϕ there exist k and m such that

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

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

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

and

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

In fact, for $|A| = \{a_1, \ldots, a_n\}$ we can do with $n - 1$ swaps $\overline{\mathit{swap}}_{a_1,a_j}$ for $j = 2, ..., n$ as these define any other swap by $\overline{\mathit{swap}}_{a_i,a_j} = \overline{\mathit{swap}}_{a_1,a_j} \circ \overline{\mathit{swap}}_{a_1,a_i} \circ \overline{\mathit{swap}}_{a_1,a_j}$, and n flips $\overline{\mathit{flip}}_{a_i}$. We show that structural bijections naturally correspond with a certain class of automorphisms on C.

Definition 3. *An automorphism α on C is **thread extraction compatible** (TEC) if there exists a structural bijection β such that the following diagram commutes:*

$$\begin{tikzcd}
C \arrow[r, "|{-}|^{\rightarrow}"] \arrow[d, "\alpha"] & \mathbb{T}_{reg} \arrow[d, "\beta"] \\
C \arrow[r, "|{-}|^{\rightarrow}"] & \mathbb{T}_{reg}
\end{tikzcd}$$

Theorem 2. *The TEC-automorphisms on C are generated by*

$$\begin{aligned}
\mathit{swap}_{a,b}: &\quad \textit{exchanges } a \textit{ and } b \textit{ in all instructions containing } a \textit{ or } b,\\
\mathit{flip}_{a}: &\quad \textit{exchanges } + \textit{ and } - \textit{ in all test instructions containing } a,
\end{aligned}$$

where a and b range over A.

Proof. Each composition $\mathit{swap}_{a,b} \circ \mathit{flip}_c$ equals one of the form $\mathit{flip}_{a'} \circ \mathit{swap}_{b'c'}$, e.g.,

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

This implies that each of the above automorphisms can be represented as $\mathit{flip}_{c_1} \circ \ldots \circ \mathit{flip}_{c_m} \circ \mathit{swap}_{a_1,b_1} \circ \ldots \circ \mathit{swap}_{a_k,b_k}$.

Above we argued that each structural bijection can be characterized by zero or more swap and flip applications. So, it suffices to argue that the diagram commutes for $\alpha = \mathit{swap}_{a,b}$ and $\beta = \overline{\mathit{swap}}_{a,b}$, and for $\alpha = \mathit{flip}_c$ and $\beta = \overline{\mathit{flip}}_c$: the general case follows from repeated applications.

The first case is trivial, so assume $\alpha = \mathit{flip}_c$ and $\beta = \overline{\mathit{flip}}_c$. Then, writing X_i for the i_{th} instruction of a piece of code X, $|\mathit{flip}_c(X)|_i = \overline{\mathit{flip}}_c(|X|_i)$. This follows from a case distinction:

- If $X_i = +/c$, then $(\mathit{flip}_c(X))_i = -/c$, so

  $$|\mathit{flip}_c(X)|_i = |\mathit{flip}_c(X)|_{i+2} \unlhd c \unrhd |\mathit{flip}_c(X)|_{i+1},$$

  and

  $$\begin{aligned}
  \overline{\mathit{flip}}_c(|X|_i) &= \overline{\mathit{flip}}_c(|X|_{i+1} \unlhd c \unrhd |X|_{i+2})\\
  &= \overline{\mathit{flip}}_c(|X|_{i+2}) \unlhd c \unrhd \overline{\mathit{flip}}_c(|X|_{i+1}).
  \end{aligned}$$

  So, both $|\mathit{flip}_c(X)|_i$ and $\overline{\mathit{flip}}_c(|X|_i)$ satisfy the same thread equation.

- If $X_i = -/c$ or $X_i = +\backslash c$ or $X_i = -\backslash c$: similar.

- In the remaining cases, say $X_i = u$, both $|\mathit{flip}_c(X)|_i$ and $\overline{\mathit{flip}}_c(|X|_i)$ also satisfy the same thread equation (the one defined for C-instruction u).

□

We further write $\mathit{TEC\text{-}AUT}$ 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 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:

$$\begin{tikzcd}
C \arrow[r, "|-|^{\rightarrow}"] \arrow[d, "\alpha"] & \mathbb{T}_{\mathrm{reg}} \arrow[d, "\overline{\alpha}"] \\
C \arrow[r, "|-|^{\rightarrow}"] & \mathbb{T}_{\mathrm{reg}}
\end{tikzcd}$$

Example 1. *As an example, take $\alpha = \mathit{flip}_b \circ \mathit{swap}_{a,b}$ and consider $+/a;\,!\,;\backslash\#2;\backslash b$, thus*

$$|+/a;\,!\,;\backslash\#2;\backslash b|^{\rightarrow} = \mathsf{S} \trianglelefteq a \trianglerighteq |+/a;\,!\,;\backslash\#2;\backslash b|^{\rightarrow}.$$

Then

$$\begin{aligned}
\overline{\alpha}(|+/a;\,!\,;\backslash\#2;\backslash b|^{\rightarrow}) &= \overline{\mathit{flip}_b}(\mathsf{S} \trianglelefteq b \trianglerighteq |+/b;\,!\,;\backslash\#2;\backslash a|^{\rightarrow}) \\
&= \overline{\mathit{flip}_b}(|+/b;\,!\,;\backslash\#2;\backslash a|^{\rightarrow}) \trianglelefteq b \trianglerighteq \mathsf{S} \\
&= \overline{\alpha}(|+/a;\,!\,;\backslash\#2;\backslash b|^{\rightarrow}) \trianglelefteq b \trianglerighteq \mathsf{S},
\end{aligned}$$

and

$$\begin{aligned}
|\alpha(+/a;\,!\,;\backslash\#2;\backslash b)|^{\rightarrow} &= |-/b;\,!\,;\backslash\#2;\backslash a|^{\rightarrow} \\
&= |-/b;\,!\,;\backslash\#2;\backslash a|^{\rightarrow} \trianglelefteq b \trianglerighteq \mathsf{S} \\
&= |\alpha(+/a;\,!\,;\backslash\#2;\backslash b)|^{\rightarrow} \trianglelefteq b \trianglerighteq \mathsf{S}.
\end{aligned}$$

Clearly, both equations are of the form $P = P \trianglelefteq b \trianglerighteq \mathsf{S}$ and thus specify the same regular thread.

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 flip_c, and a counter-example is $\alpha = \mathit{flip}_b \circ \mathit{swap}_{a,b}$ as used in Example 1:

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

but α^2 is an involution (because compositions of flip commute).

7 TEC-Anti-Automorphisms and Anti-Homomorphisms

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 ϕ is an anti-homomorphism if it satisfies $\phi(X;Y) = \phi(Y);\phi(X)$. Furthermore, we show how the monomorphism h defined in Section 5 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:

$$\begin{aligned}
/a &\mapsto \backslash a,\\
+/a &\mapsto +\backslash a,\\
-/a &\mapsto -\backslash a,\\
/\#k &\mapsto \backslash\#k,\\[4pt]
\backslash a &\mapsto /a,\\
+\backslash a &\mapsto +/a,\\
-\backslash a &\mapsto -/a,\\
\backslash\#k &\mapsto /\#k,\\[4pt]
! &\mapsto !\,.
\end{aligned}$$

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

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

Definition 4. *An anti-automorphism α on C is **thread extraction compatible** (TEC) if there exists a structural bijection β such that the following diagram commutes:*

$$\begin{tikzcd}
C \arrow[r, "|-|^{\to}"] \arrow[d, "\alpha"] & \mathbb{T}_{reg} \arrow[d, "\beta"] \\
C \arrow[r, "|-|^{\leftarrow}"] & \mathbb{T}_{reg}
\end{tikzcd}$$

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

Theorem 3. $\mathit{TEC\text{-}AntiAUT} = \{rev \circ \alpha \mid \alpha \in \mathit{TEC\text{-}AUT}\}$.

Proof. Let $B = \{rev \circ \alpha \mid \alpha \in \mathit{TEC\text{-}AUT}\}$.

Let $\alpha \in \mathit{TEC\text{-}AntiAUT}$ be given. By Definition 4 there is a structural bijection β such that $|\alpha(X)|^{\leftarrow} = \beta(|X|^{\to})$ for all X. By Theorem 2, $\beta = \overline{\gamma}$ for some $\gamma \in \mathit{TEC\text{-}AUT}$ and $\beta(|X|^{\to}) = |\gamma(X)|^{\to}$ for all X, and thus $|\alpha(X)|^{\leftarrow} = |\gamma(X)|^{\to}$. Because $|\gamma(X)|^{\to} = |rev(\gamma(X))|^{\leftarrow}$ it follows that $\alpha = rev \circ \gamma$, thus $\alpha \in B$.

Conversely, if $\alpha \in B$, then $\alpha = rev \circ \gamma$ with $\gamma \in \mathit{TEC\text{-}AUT}$, so $|\alpha(X)|^{\leftarrow} = |\gamma(X)|^{\to} = \beta(|X|^{\to})$ for some structural bijection β and all X. So, $\alpha \in \mathit{TEC\text{-}AntiAUT}$. □

Observe that for all $\alpha \in \mathit{TEC\text{-}AUT}$, $\alpha \circ rev = rev \circ \alpha$ and for all $\alpha, \beta \in \mathit{TEC\text{-}AntiAUT}$, $\alpha \circ \beta \in \mathit{TEC\text{-}AUT}$.

Using the notation for associated pairs we find for $\beta = \mathit{rev} \circ \alpha \in \mathit{TEC\text{-}AntiAUT}$ that the following diagram commutes:

$$\begin{tikzcd}
C \arrow[r, "|{-}|^{\rightarrow}"] \arrow[d, "\beta"'] & \mathbb{T}_{\text{reg}} \arrow[d, "\overline{\alpha}"] \\
C \arrow[r, "|{-}|^{\leftarrow}"] & \mathbb{T}_{\text{reg}}
\end{tikzcd}$$

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

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

$$\begin{tikzcd}
C \arrow[r, "\mathit{rev} \circ h"] \arrow[d, "h"'] & C \arrow[d, "|{-}|^{\leftarrow}"] \\
C \arrow[r, "|{-}|^{\rightarrow}"] & \mathbb{T}_{\text{reg}}
\end{tikzcd}$$

As an example, consider the anti-homomorphism g defined in Section 5: indeed $g = \mathit{rev} \circ h$ for the homomorphism h defined in that section.

8 Structural Congruence and Expressiveness

In this section we define structural congruence and we discuss a basic expressiveness result: we show that C-programs characterize all regular threads.

Two C-expressions X and Y are *structurally equivalent*, notation

$$X \equiv_{sc} Y,$$

if they are syntactically equal apart from their jump instructions and define the same thread extractions, i.e. $|X|_i = |Y|_i$ for $i = 1, \ldots, \ell(X) \;\; (= \ell(Y))$. A typical example is

$$+/a; /\#3; /b; -\backslash c; \backslash\#3; -\backslash d \equiv_{sc} +/a; /\#0; /b; -\backslash c; \backslash\#0; -\backslash d.$$

The axioms in Table 2 define structural congruence, notation $=_{sc}$, where for $k = 0$,

$$i_1; \ldots; i_k;$$

denotes the empty sequence.

From these axioms and those for C-expressions (axioms (1) and (2)) it follows that

$$/\#k{+}1; i_1; \ldots; i_k; \backslash\#k{+}1 =_{sc} /\#0; i_1; \ldots; i_k; \backslash\#0.$$

Recall that a *chained jump* is a jump to a jump instruction.

Definition 5. *A **canonical** C**-form** is a C-expression not containing chained jumps.*

$$/\#k{+}1; i_1; \ldots; i_k; /\#0 =_{sc} /\#0; i_1; \ldots; i_k; /\#0 \tag{4}$$

$$/\#k{+}1; i_1; \ldots; i_k; /\#m =_{sc} /\#k{+}m{+}1; i_1; \ldots; i_k; /\#m \tag{5}$$

$$/\#k{+}1; i_1; \ldots; i_k; \backslash\#m =_{sc} \begin{cases} /\#k{+}1{-}m; i_1; \ldots; i_k; \backslash\#m & \text{if } m \leq k+1 \\ \backslash\#m{-}(k{+}1); i_1; \ldots; i_k; \backslash\#m & \text{otherwise} \end{cases} \tag{6}$$

$$\backslash\#0; i_1; \ldots; i_k; \backslash\#k{+}1 =_{sc} \backslash\#0; i_1; \ldots; i_k; \backslash\#0 \tag{7}$$

$$\backslash\#m; i_1; \ldots; i_k; \backslash\#k{+}1 =_{sc} \backslash\#k{+}1{+}m; i_1; \ldots; i_k; \backslash\#k{+}1 \tag{8}$$

$$/\#m; i_1; \ldots; i_k; \backslash\#k{+}1 =_{sc} \begin{cases} /\#m; i_1; \ldots; i_k; \backslash\#k{+}1{-}m & \text{if } m \leq k+1 \\ /\#m; i_1; \ldots; i_k; /\#m{-}(k{+}1) & \text{otherwise} \end{cases} \tag{9}$$

Table 2: Axioms for structural congruence, where $k, m \in \mathbb{N}$

Proposition 2. *Structural congruent C-expressions yield the same thread extraction from arbitrary positions, and each C-expression is structurally congruent with a canonical C-form.*

Note that both thread extractions $|X|^{\rightarrow}$ and $|X|^{\leftarrow}$ of a C-program X in canonical C-form only use the equations from Table 4 that either yield an action, or D by $/\#0$ or $\backslash\#0$, or S by $!$, and never generate loops without having generated any behavior.

Theorem 4. *The regular threads in $\mathbb{T}_{reg}$ are characterized by C-programs.*

Proof. First, given a C-expression X determine its structural congruent canonical C-form Y. Applying behavior extraction to Y yields a finite number of equations of the form

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

so $|X|_i = |Y|_i$ is a regular thread.

Secondly, each finite number of linear equations $P_1 = t_1, \ldots, P_n = t_n$ specifying a regular thread P_1 can be turned into a C-program fragment:

$$\begin{aligned} P_i = \mathsf{S} &\mapsto !\,, \\ P_i = \mathsf{D} &\mapsto /\#0, \\ P_i = P_j \trianglelefteq a \trianglerighteq P_k &\mapsto +/a; u_i; v_i, \end{aligned}$$

where u_i and v_i will be jumps to the appropriate position. Concatenating these fragments and instantiating the u_i and v_i correctly yields a program that specifies P_1. Note that this program contains only instructions from the set

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

□

As an example, the regular thread P_1 discussed in Section 2, i.e.,

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

yields the C-template $+/a; u_1; v_1; +/b; u_2; v_2; +/c; u_3; v_3; +/d; u_4; v_4; \,!$. Filling in the appropriate jump counters yields a C-program

$$X = +/a; /\#2; /\#1; +/b; /\#2; /\#4; +/c; \backslash\#4; \backslash\#5; +/d; /\#2; \backslash\#11; ! \quad \text{with} \quad |X|^{\rightarrow} = P_1.$$

9 Expressiveness and C's instructions

In this section we consider C's instructions in the perspective of expressiveness. We establish that we do not need all instructions in C to preserve expressiveness, but that C's expressiveness is reduced if jump instructions are restricted to a finite bound.

From the proof of Theorem 4 we immediately 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 expressed in C^-.

In the remainder of this section 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. As a first example, the thread P defined by

$$P = a \circ b \circ c \circ P$$

can not be expressed in C_2: a C-program expressing P necessarily has a part defining a b-action that upon execution needs to be traversed in both directions. Assume this part contains a basic instruction followed by a jump in one direction, then these two instructions need to be jumped over in the other direction. The latter implies that a jump of at least 3 is needed for expressing P, as is indeed the case in

$$X = /a; /\#2; \backslash\#2; /b; /\#2; \backslash\#3; \backslash c$$

which satisfies $|X|^{\rightarrow} = P$. Of course, using the combination of a test instruction for b followed by two jumps even requires a jump of at least 4 for traversal in the other direction. Instead of making this argument more precise we turn to the general case and prove that for any $k \in \mathbb{N}$, not all regular threads in $\mathbb{T}_{\mathrm{reg}}$ can be expressed in C_k.

We first introduce an auxiliary notion: let $a \in A$ and $n \in \mathbb{N}^+$, then a piece of code X has the *a-n-property* if for some i, $|X|_i$ can perform n consecutive a-tests such that each sequence of replies leads to a unique further behavior. It is not hard to see that in this case X contains at least $2^n - 1$ different a-tests.

As a first example, consider

$$X = !\,;\backslash b;-\backslash a;+/a;\backslash\#2;-/a;/\#2;/c;/\#0$$

Clearly, X has the a-2-property: $|X|_4$ yields

$$\begin{cases} \texttt{true},\texttt{true} & \mapsto \quad |X|_1 = \mathsf{S}, \\ \texttt{true},\texttt{false} & \mapsto \quad |X|_2 = b \circ \mathsf{S}, \\ \texttt{false},\texttt{true} & \mapsto \quad |X|_8 = c \circ \mathsf{D}, \\ \texttt{false},\texttt{false} & \mapsto \quad |X|_9 = \mathsf{D}. \end{cases}$$

Note that X also has the a-1-property. This illustrates a general property: if a piece of code has the a-$(n{+}m)$-property, then it has also the a-n-property.

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 α is a prefix of β, 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 α with $\ell(\alpha) \leq n$ (there are $2^{n+1} - 1$ such sequences).

Let $g : \mathbb{N} \to \mathbb{N}$ be such that $|Y_n|_{g(n)}$ establishes 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 α has position m. Clearly, f_n is an injective function.

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:

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

By supposition there is for each choice $\beta \in \{\alpha_2, ..., \alpha_{2k+2}\}$ 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 β, assume that at least $k+1$ elements $\beta \in \{\alpha_2, ..., \alpha_{2k+2}\}$ have an extension β' 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 β with extension β' leads to a similar argument). Then we obtain a contradiction with respect to $f_{k'+n}$: for each of the sequences β in the subset just selected and its extension β',

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

and there are at least $k+1$ different such pairs β, β' (recall $f_{k'+n}$ is injective). But this is not possible with jumps of at most k. □

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 α and γ: 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 5. *For any $k \in \mathbb{N}$, not all regular threads in $\mathbb{T}_{reg}$ can be expressed in C_k. This is also the case if thread extraction may start at arbitrary positions.*

Proof. Immediately with Lemma 1: we discuss a family of regular threads each of which is only definable by a program with the a-n-property. For each $n \in \mathbb{N}^+$ we first define a program Z_n that uses n *Boolean registers* named $b1, b2, ..., bn$ which all are initially set to F (false). An instruction $/bi.set{:}T$ sets register bi to T (true), and a test instruction $+/bi.get$ reads the value from register bi. Execution of such register instructions does not yield any observable behavior (this type of thread-service composition is explained in detail in e.g. [8, 2]):

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

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. Hence, for each $n \in \mathbb{N}^+$, a defining C_k-program must have the a-n-property. But this is not possible according to Lemma 1. □

10 On the Length of C-Programs for Producing Threads

C-programs can be viewed as descriptions of 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 (this can be a very convenient feature, cf. the proof of Theorem 5). We find upper and lower bounds for the lenghts 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 /#0. 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

$$\begin{aligned}&/bi.set{:}T\\&/bi.set{:}F\\&\quad/bi.get\end{aligned}$$

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 jump instructions, namely a jump to the position itself (deadlock, we identify \#0 and /#0) and 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 ! , the admissible instruction alphabet counts

$$1 + 6 + l + 18l$$

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

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

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

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

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

$$(26l)^l \geq n^{n-1}. \tag{10}$$

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 (10). 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 $\psi(k, n)$.

11 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.[2] 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* [6] 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 [7], 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:

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.

[2] [4]: 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).

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 [1, 8] (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., $|...|$, 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 ψ on these instructions:

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

   We can rewrite each C-program into this restricted form by applying the behavior preserving homomorphism h defined in Section 5. Thus our final definition of a projection can be $\mathtt{p2pga} = F \circ h$. Note that many alternatives for h could have been used as well, as was already noted in Section 5.

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. In the other case, a PGA-program can be brought into so-called second canonical form, i.e., the form $u_1;\ldots;u_k;(u_{k+1};\ldots;u_{k+n})^\omega$ without the occurrence of chained jumps. Then, adding the appropriate slashes to $u_1;\ldots;u_k;\vartheta_1(u_{k+1});\ldots;\vartheta_n(u_{k+n});\backslash\#n;\backslash\#n$ with $\vartheta_j(\#l) = \backslash\#n - l$ and the identity otherwise yields a C-program with the same behavior.

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.

References

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

[2] J.A. Bergstra and C.A. Middelburg. Instruction sequences and non-uniform complexity theory. arXiv:0809.0352v1 [cs.CC] at http://arxiv.org/, 2008.

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

[4] 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.

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

[6] 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.)

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

[8] 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.

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 the basic instructions, u over the primitive instructions, 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 [1, 8].