An instruction sequence semigroup with repeaters

Bergstra, J.A.; Ponse, A.

Publication date
2008
Document Version
Submitted manuscript

An Instruction Sequence Semigroup with Repeaters

Jan A. Bergstra
Alban Ponse

Section Software Engineering, Informatics Institute, University of Amsterdam
Email: {J.A.Bergstra,A.Ponse}@uva.nl

Abstract

In the setting of program algebra (PGA) we consider the repeat instruction. This special instruction was designed to represent infinite sequences of primitive instructions as finite, linear programs. The resulting mathematical structure is a semigroup. We show that a kernel of this syntax can replace PGA as a carrier for program algebra by providing axioms for defining single-pass congruence and structural congruence, and equations for thread extraction. Finally, we discuss the related program notation PGLA that serves as a basis for PGA's tool set.

1 Introduction

This paper is a thoroughly revised version of [5]. A "program" is a piece of data for which the preferred or natural interpretation (or meaning) is a sequence of primitive instructions (SPI), and we say that a program produces such a sequence. 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.

We work in the setting of *Program Algebra* (PGA) as laid out in [3], a setting that provides an algebraic framework and corresponding semantic foundations for sequential programming. A very basic question is how to represent SPIs. PGA was designed to give a very straightforward and simple answer to this question, starting with constants for primitive instructions and two operations for composing SPIs: *concatenation* and *repetition*. Each primitive instruction is a finite SPI, and if P and Q are finite SPIs, then so is their concatenation $P;Q$. An infinite SPI is *periodic* if it can be written as $P;Q^\omega$ with P and Q finite SPIs. Here Q^ω is the repetition of Q, i.e. the SPI that repeats Q infinitely often. PGA has a complete axiomatization for *single-pass congruence* of finite and periodic SPIs, which is the congruence that characterizes extensional equality of SPIs, i.e., the equality defined by having the same primitive instruction at each position. For finite and periodic SPIs, single-pass congruence is decidable.

In this paper (Section 3) we characterize the class of SPIs expressible in PGA without making use of the repetition operator: each periodic SPI can be represented as a finite sequence of instructions with help of the *repeat instruction*. The price to be paid is that such a finite

sequence contains a *non-primitive* repeat instruction and that we lose the property that each sequence of instructions is a (proper) program. The repeat instructions or so-called *repeaters* are defined as follows: for n a natural number larger than 0,

$$\backslash\backslash\#n$$

prescribes to repeat the last preceding n instructions. Repeaters do not comply with the notion of single pass execution. For example, for u a primitive instruction,

$u; \backslash\backslash\#1$ abbreviates the SPI $u; u; u; u; \ldots$.

Thus, $u; \backslash\backslash\#1$ is a sequence of two instructions that *represents* a periodic SPI: the infinite sequence of u's, in PGA's notation u^ω, and the same SPI is represented by $u; u; \backslash\backslash\#1$ and $u; u; \backslash\backslash\#2$ and $u; u; u; \backslash\backslash\#2$, and by many more finite sequences. We write L for the set of all finite sequences built from primitive instructions and repeaters with concatenation.

We also discuss the fact that not each sequence of instructions in L can be called a program: for example, which SPI, if any, is meant by

$$u; \backslash\backslash\#2$$

or by a single repeat instruction? We distinguish a subset K of L containing those sequences that represent the finite and periodic SPIs. All sequences in K can be called "programs". For K, we provide axioms for single-pass congruence and for *structural congruence*, a congruence that admits the chaining of jump counters. The question whether we should deal at all with L-sequences with "too large" repeat counters, thus sequences not in K such as $u; \backslash\backslash\#2$, is briefly discussed in Section 4.

2 PGA basics

In this section some basic information about PGA (based on [3]) is recalled. Furthermore, we briefly discuss Thread Algebra (cf. [7]), earlier described in e.g. [1, 3] under the name Polarized Process Algebra.

2.1 PGA, primitive instructions and SPIs

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

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

Each of the first five forms above is called a *primitive instruction*:

- A *basic instruction* $\mathtt{a}$ is a prescription for a piece of behavior that is considered indivisible and executable in finite time.

- A basic instruction can be turned into a *test instruction* by prefixing it with either a $+$ (positive test instruction) or a $-$ (negative test instruction), thus typically $+\mathtt{a}$, $-\mathtt{b}$ etc. Test instructions control subsequent execution via the result of their execution (which is a Boolean reply).
- A next kind of primitive instruction is the *jump instructions* $\#k$: this instruction prescribes to jump k instructions ahead (if possible; otherwise deadlock occurs) and generate no observable behavior.
- Finally, the *termination instruction* ! prescribes sucessful termination, an event that is taken to be observable.

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 defied 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.

As mentioned before, the execution of a SPI is *single-pass*: it starts with the first (leftmost) instruction, and each instruction is dropped after it has been executed or jumped over. In Section 2.3 the precise *meaning* of primitive instructions in terms of their execution is explained. The representation of a finite or periodic SPI in PGA is henceforth called a "PGA-program".

2.2 Canonical forms and two congruences in PGA

In PGA, different types of equality are discerned, the most simple of which is *single-pass congruence*, identifying PGA-programs that execute identical SPIs.[1] For PGA-programs not containing repetition, single-pass congruence boils down to the associativity of concatenation, and is thus axiomatized by

$$(X; Y); Z = X; (Y; Z).$$

From this point onwards we leave out brackets in repeated concatenations. Define $X^1 = X$ and for $n > 0$, $X^{n+1} = X; X^n$. According to [3], single-pass congruence for PGA-programs is axiomatized by the axioms (schemes) PGA1-PGA4 in Table 1.

[1] Although a bit long, *primitive instruction sequence congruence* would also be an adequate name.

$$(X;Y);Z = X;(Y;Z) \quad \text{(PGA1)}$$
$$(X^n)^\omega = X^\omega \quad \text{(PGA2)}$$
$$X^\omega;Y = X^\omega \quad \text{(PGA3)}$$
$$(X;Y)^\omega = X;(Y;X)^\omega \quad \text{(PGA4)}$$
$$\#n{+}1;u_1;\ldots;u_n;\#0 = \#0;u_1;\ldots;u_n;\#0 \quad \text{(PGA5)}$$
$$\#n{+}1;u_1;\ldots;u_n;\#m = \#n{+}m{+}1;u_1;\ldots;u_n;\#m \quad \text{(PGA6)}$$
$$(\#k{+}n{+}1;u_1;\ldots;u_n)^\omega = (\#k;u_1;\ldots;u_n)^\omega \quad \text{(PGA7)}$$
$$X = u_1;\ldots;u_n;(v_1;\ldots;v_{m+1})^\omega \rightarrow \#n{+}m{+}k{+}2;X = \#n{+}k{+}1;X \quad \text{(PGA8)}$$

Table 1: PGA-axioms for structural congruence, where $k, n, m \in \mathbb{N}$, u_i, v_j range over the primitive instructions, and $u_1;\ldots;u_0$; represents the empty sequence

Proposition 1. *The unfolding law*

$$X^\omega = X;X^\omega$$

follows from the axioms PGA2 and PGA4. [2]

Proof. Straightforward: $X^\omega = (X;X)^\omega = X;(X;X)^\omega = X;X^\omega$. □

Whenever two PGA-programs X and Y are single-pass congruent, this is written

$$X =_{spc} Y.$$

The subscript *spc* will be dropped if no confusion can arise. Using the axioms PGA1–PGA4 (thus preserving single-pass congruence), each PGA-program can be rewritten into one of the following forms:

Y not containing repetition, or

$Y;Z^\omega$ with Y and Z not containing repetition.

Any PGA-program in one of the two above forms is said to be in *first canonical form*. Moreover, in the case of $Y;Z^\omega$, there is a unique first canonical form if the number of instructions in both Y and Z is minimized (using PGA1-PGA4). Single-pass congruence is decidable (as recorded in [3]).

PGA-programs in first canonical form can be converted into *second canonical form*: a first canonical form in which no chained jumps occur, i.e., jumps to jump instructions (apart from #0), and in which each non-chaining jump into the repeating part is minimized. The

[2] Conversely, from unfolding and the conditional proof rule $X = Y;X \Rightarrow X = Y^\omega$, one derives PGA2-4.

associated congruence is called *structural congruence* and is axiomatized in Table 1. Note that axiom PGA8 is an equational axiom, the implication is only used to enhance readability.

We write $X =_{sc} Y$ if X and Y are structurally congruent, and drop the subscript if no confusion can arise. Two examples, of which the right-hand sides are in second canonical form:

$$\#2; \mathtt{a}; (\#5; \mathtt{b}; +\mathtt{c})^\omega =_{sc} \#4; \mathtt{a}; (\#2; \mathtt{b}; +\mathtt{c})^\omega,$$
$$+\mathtt{a}; \#2; (+\mathtt{b}; \#2; -\mathtt{c}; \#2)^\omega =_{sc} +\mathtt{a}; \#0; (+\mathtt{b}; \#0; -\mathtt{c}; \#0)^\omega.$$

For each PGA-program there exists a structurally equivalent second canonical form. Moreover, in the case of $Y; Z^\omega$ this form is unique if the number of instructions in Y and Z is minimized. As a consequence, structural congruence is decidable. In the first example above, $\#4; \mathtt{a}; (\#2; \mathtt{b}; +\mathtt{c})^\omega$ is the unique minimal second canonical form; for the second example it is

$$+\mathtt{a}; (\#0; +\mathtt{b}; \#0; -\mathtt{c})^\omega.$$

For more information on PGA we refer to [3, 7].

2.3 Thread algebra: behavioral semantics for PGA

We briefly discuss thread algebra. Threads model the execution of SPIs. Finite threads are defined inductively:

S	–	*stop*, the termination thread,
D	–	*inaction* or *deadlock*, the inactive thread,
$P \trianglelefteq \mathtt{a} \trianglerighteq Q$	–	the *postconditional composition* of P and Q for action $\mathtt{a}$, where P and Q are finite threads and $\mathtt{a} \in A$.

The behavior of the thread $P \trianglelefteq \mathtt{a} \trianglerighteq Q$ starts with the *action* $\mathtt{a}$ and continues as P upon reply `true` to $\mathtt{a}$, and as Q upon reply `false`. Note that finite threads always end in S or D. We use *action prefix* $\mathtt{a} \circ P$ as an abbreviation for $P \trianglelefteq \mathtt{a} \trianglerighteq P$ and take $\circ$ to bind strongest.

Upon its execution, a basic or test instruction yields the equally named action in a post conditional composition. Thread extraction on PGA, notation

$$|X|$$

with X a PGA-program, is defined by the thirteen equations in Table 2. In particular, note that upon the execution of a positive test instruction $+\mathtt{a}$, the reply `true` to $\mathtt{a}$ prescribes to continue with the next instruction and `false` to skip the next instruction and to continue with the instruction thereafter; if no such instruction is available, deadlock occurs. For the execution of a negative test instruction $-\mathtt{a}$, subsequent execution is prescribed by the complementary replies.

For a PGA-program in second canonical form, these equations either yield a finite thread, or a so-called *regular* thread, i.e., a finite state thread in which infinite paths can occur. Each regular thread can be specified (defined) by a finite number of recursive equations. As a first

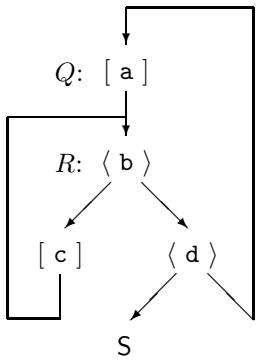
(i)	$\lvert !\rvert = \mathsf{S}$	(ii)	$\lvert !; X\rvert = \mathsf{S}$
(iii)	$\lvert \mathtt{a}\rvert = \mathtt{a} \circ \mathsf{D}$	(iv)	$\lvert \mathtt{a}; X\rvert = \mathtt{a} \circ \lvert X\rvert$
(v)	$\lvert +\mathtt{a}\rvert = \mathtt{a} \circ \mathsf{D}$	(vi)	$\lvert +\mathtt{a}; X\rvert = \lvert X\rvert \trianglelefteq \mathtt{a} \trianglerighteq \lvert \#2; X\rvert$
(vii)	$\lvert -\mathtt{a}\rvert = \mathtt{a} \circ \mathsf{D},$	$(viii)$	$\lvert -\mathtt{a}; X\rvert = \lvert \#2; X\rvert \trianglelefteq \mathtt{a} \trianglerighteq \lvert X\rvert$
(ix)	$\lvert \#k\rvert = \mathsf{D}$	(x)	$\lvert \#0; X\rvert = \mathsf{D}$
		(xi)	$\lvert \#1; X\rvert = \lvert X\rvert$
		(xii)	$\lvert \#k{+}2; u\rvert = \mathsf{D}$
		$(xiii)$	$\lvert \#k{+}2; u; X\rvert = \lvert \#k{+}1; X\rvert$

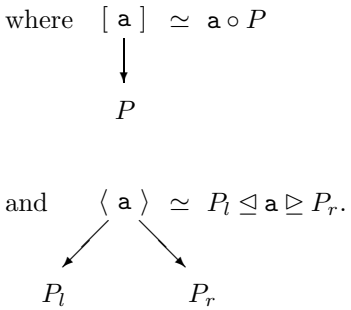
Table 2: Equations for thread extraction, where $\mathtt{a}$ ranges over the basic instructions, and u over the primitive instructions ($k \in \mathbb{N}$)

example, the regular thread Q specified by

$$Q = \mathtt{a} \circ R$$
$$R = \mathtt{c} \circ R \trianglelefteq \mathtt{b} \trianglerighteq (\mathsf{S} \trianglelefteq \mathtt{d} \trianglerighteq Q)$$

and Q can be defined by $\lvert \mathtt{a}; (+\mathtt{b}; \#2; \#3; \mathtt{c}; \#4; +\mathtt{d}; !; \mathtt{a})^\omega\rvert$. A picture of this thread:





Some more examples:

$$\lvert + \mathtt{a}; \#3\rvert = a \circ \mathsf{D},$$
$$\lvert + \mathtt{a}; \#3; (\#0)^\omega\rvert = \lvert + \mathtt{a}; \#0; (\#0)^\omega\rvert = a \circ \mathsf{D},$$
$$\lvert \#4; \mathtt{a}; (\#2; \mathtt{b}; +\mathtt{c})^\omega\rvert = \lvert (+\mathtt{c}; \#2; \mathtt{b};)^\omega\rvert = P \quad \text{with } P = P \trianglelefteq \mathtt{c} \trianglerighteq b \circ P,$$
$$\lvert + \mathtt{a}; \#0; (\mathtt{b}; \#0; -\mathtt{c}; \#0)^\omega\rvert = \mathsf{D} \trianglelefteq \mathtt{a} \trianglerighteq \mathtt{b} \circ \mathsf{D},$$
$$\lvert + \mathtt{a}; \#0; (+\mathtt{b}; \#0; -\mathtt{c}; \#0)^\omega\rvert = \mathsf{D} \trianglelefteq \mathtt{a} \trianglerighteq P \quad \text{with } P = \mathsf{D} \trianglelefteq \mathtt{b} \trianglerighteq (P \trianglelefteq \mathtt{c} \trianglerighteq \mathsf{D}).$$

It can be inferred that

$$|u_1;\ldots;u_n| = |u_1;\ldots;u_n;(\#0)^\omega|$$

for u_i ranging over the primitive instructions. We shall use $\mathtt{a}^\infty$ as an informal notation for the thread defined by $|\mathtt{a}^\omega|$.

For basic information on thread algebra we refer to [2, 7]; more advanced matters, such as an operational semantics for thread algebra, are discussed in [4]. We here only mention the fact that each regular thread can be specified in PGA, and, conversely, that each PGA-program defines a regular thread.

3 An instruction semigroup with repeaters

In this section we introduce an instruction sequence semigroup with repeaters. We distinguish a kernel K of this semigroup that can serve as a carrier for program algebra: single-pass congruence, structural congruence and thread extraction can all be defined in K without reference to PGA.

3.1 A semigroup L with canonical forms

We introduce a semigroup L with concatenation as its (associative) operation, starting from the set $\mathcal{U}$ of primitive instructions and so-called *repeaters* (also called *repeat instructions*)

$$\backslash\backslash\#n$$

for all $n \in \mathbb{N}^+ = \mathbb{N} \setminus \{0\}$.

Brackets are not used in L because we are working in a semigroup. A sequence of primitive instructions ending with $\backslash\backslash\#n$ will repeat its last n instructions, excluding the repeat instruction itself. So for $n > 0$,

$$u_1;\ldots;u_n;\backslash\backslash\#n$$

with all u_i primitive instructions represents the same SPI as $(u_1;\ldots;u_n)^\omega$ does. Instructions to the right of a repeat instruction are irrelevant and can be deleted.

For terms not containing a repeat instruction, single-pass congruence boils down to the associativity of concatenation, and is thus axiomatized by axiom (1) in Table 3. We prove below that single-pass congruence for all SPIs that can be expressed with repeaters is axiomatized by the axioms (1)–(4) in Table 3 (and those of equational logic), and we write

$$L_{spc}$$

for this particular proof system. Note that X, Y, Z in axioms (1) and (3) range over sequences that may contain both primitive and repeat instructions. Although all remaining equations in Table 3 are axiom *schemes*, we shall also call these "axioms".

In L_{spc}, axiom (3) implies that each closed term in L can be equated to one that contains *at most* one repeat instruction. We define first canonical L-forms, a preferred representation for closed terms in L.

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

$$u_1;\ldots;u_n;\backslash\backslash\#n = (u_1;\ldots;u_n)^m;\backslash\backslash\#mn \tag{2}$$

$$\backslash\backslash\#n;X = \backslash\backslash\#n \tag{3}$$

$$u_1;\ldots;u_m;v_1;\ldots;v_n;\backslash\backslash\#m{+}n = u_1;\ldots;u_m;v_1;\ldots;v_n;u_1;\ldots;u_m;\backslash\backslash\#m{+}n \tag{4}$$

$$\#k{+}1;u_1;\ldots;u_k;\#0 = \#0;u_1;\ldots;u_k;\#0 \tag{5}$$

$$\#k{+}1;u_1;\ldots;u_k;\#m = \#k{+}m{+}1;u_1;\ldots;u_k;\#m \tag{6}$$

$$\#k{+}1{+}\ell;u_1;\ldots;u_k;\backslash\backslash\#k{+}1 = \#\ell;u_1;\ldots;u_k;\backslash\backslash\#k{+}1 \tag{7}$$

$$\#k{+}1{+}m{+}\ell;u_1;\ldots;u_k;v_1;\ldots;v_m;\backslash\backslash\#m = \#k{+}1{+}\ell;u_1;\ldots;u_k;v_1;\ldots;v_m;\backslash\backslash\#m \tag{8}$$

Table 3: Axioms for SPIs, where $k, \ell \in \mathbb{N}$, $m, n \in \mathbb{N}^+$ and $u_i, v_j \in \mathcal{U}$

Definition 1. *An L-term is a **first canonical L-form** if it is of the form*

$$u_1;\ldots;u_n \quad \textit{or} \quad u_1;\ldots;u_k;\backslash\backslash\#n$$

with $u_i \in \mathcal{U}$, $k \in \mathbb{N}$ and $n \in \mathbb{N}^+$. Here $u_1;\ldots;u_0;$ represents the empty sequence.

For a closed term in L, we say that its first canonical L-form is obtained by applying axiom (3) to the leftmost occurring repeater if present, and otherwise it is that term itself.

Not all closed terms in L have an intuitive meaning. For example,

$$\mathtt{a};\backslash\backslash\#2 \quad \text{and} \quad \#7;{+}\mathtt{a};\backslash\backslash\#5$$

illustrate this situation. Note that such first canonical L-forms can not be rewritten using any of the axioms (2)–(8) in Table 3.

3.2 A kernel K with canonical forms and two congruences

Let K stand for the subset of closed terms whose first canonical L-form has the property that the repeat instruction $\backslash\backslash\#n$ is preceded by at least n primitive instructions. A first canonical L-form in K will henceforth be called a *first canonical K-form*.

Definition 2. *Let $u_1;\ldots;u_k$ be a SPI. The first canonical K-form $u_1;\ldots;u_k$ is **minimal** by definition, and a first canonical K-form*

$$u_1;\ldots;u_k;\backslash\backslash\#n$$

*(so $0 < n \leq k$) is **minimal** if its repeating part (i.e., $u_{k-n+1};\ldots;u_k$) can not be made smaller with axiom scheme* (2) *and its non-repeating part (i.e., $u_1;\ldots;u_{k-n}$) can not be made smaller with axiom scheme* (4)*.*

Two examples, where the right-hand sides are minimal first canonical K-forms:

$$+\mathtt{a};-\mathtt{b};\#4;-\mathtt{b};\#4;\backslash\backslash\#4 =_{spc} +\mathtt{a};-\mathtt{b};\#4;\backslash\backslash\#2,$$
$$-\mathtt{a};+\mathtt{c};\#4;+\mathtt{c};\backslash\backslash\#2 =_{spc} -\mathtt{a};+\mathtt{c};\#4;\backslash\backslash\#2.$$

From this point onwards, first canonical K-forms are called *K-programs*. We state without proof that in L_{spc} each K-program can be rewritten into a unique minimal K-program in terms of its number of instructions. As a consequence, single-pass congruence is decidable for K-programs. Single-pass congruence for K-programs is captured by the next result.

Theorem 1. *Two K-programs P and Q are single-pass congruent if, and only if,*

$$L_{spc} \vdash P = Q.$$

Proof. Trivial: restricting to K, soundness and completeness of the proof system L_{spc} follow from its direct relation with PGA: the correspondence between $u_1;\ldots;u_n;\backslash\backslash\#n$ and $(u_1;\ldots;u_n)^\omega$ and the first four axioms of L_{spc} and PGA1–PGA4 (and the corresponding fact that minimal first canonical K-forms are unique). □

In order to argue that K is a fully fledged alternative for PGA we define second canonical K-forms, and we write

$$L_{sc}$$

for the extension of L_{spc} with all axiom schemes in Table 3 (thus axioms (1)–(8)).

Definition 3. *A **second canonical K-form** is a first canonical K-form in which no chained jumps occur and in the case of $u_1;\ldots;u_m;\backslash\backslash\#n$, all jumps to $u_{m-n+1},\ldots,u_m$ are minimized (cf. Section 2.2).*

Some examples, the first of which is the instance of axiom (7) for $k = \ell = 0$:

$$\begin{aligned}
\#1;\backslash\backslash\#1 &=_{sc} \#0;\backslash\backslash\#1,\\
\#2;\mathtt{a};\#5;\mathtt{b};+\mathtt{c};\backslash\backslash\#3 &=_{sc} \#4;\mathtt{a};\#2;\mathtt{b};+\mathtt{c}\backslash\backslash\#3,\\
+\mathtt{a};\#2;+\mathtt{b};\#2;-\mathtt{c};\#2;\backslash\backslash\#4 &=_{sc} +\mathtt{a};\#0;+\mathtt{b};\#0;-\mathtt{c};\#0;\backslash\backslash\#4\\
&=_{sc} +\mathtt{a};\#0;+\mathtt{b};\#0;-\mathtt{c};\backslash\backslash\#4.
\end{aligned}$$

Here the right-hand sides are second canonical K-forms. We state without proof that in L_{sc}, second canonical K-forms have a unique minimal representation in terms of their number of instructions (cf. the last example above). As a consequence, structural congruence for K-programs is decidable.

Theorem 2. *Two K-programs P and Q are structurally congruent if, and only if,*

$$L_{sc} \vdash P = Q.$$

Proof. Trivial: restricting to K, soundness and completeness of the proof system L_{sc} are captured by its direct relation with PGA (minimal second canonical K-forms are unique). □

Let $X = u_1; \ldots; u_{n+k}; \backslash\backslash \#n$, then

$$
\begin{aligned}
&[\![X]\!]_K = |1, X|,\\
&|j, X| = |j{-}n, X| \quad \text{if } j > n{+}k,\\
&|j, X| = \mathsf{S} \quad \text{if } u_j = !,\\
&|j, X| = a \circ |j{+}1, X| \quad \text{if } u_j = \mathtt{a},\\
&|j, X| = |j{+}1, X| \trianglelefteq a \trianglerighteq |j{+}2, X| \quad \text{if } u_j = +\mathtt{a},\\
&|j, X| = |j{+}2, X| \trianglelefteq a \trianglerighteq |j{+}1, X| \quad \text{if } u_j = -\mathtt{a},\\
&|j, X| = \mathsf{D} \quad \text{if } u_j = \#0,\\
&|j, X| = |j{+}m, X| \quad \text{if } u_j = \#m.
\end{aligned}
$$

Table 4: Equations for thread extraction on K, where $u_i \in \mathcal{U}$, $k \in \mathbb{N}$ and $j, n, m \in \mathbb{N}^+$

3.3 Thread extraction in K

Thread extraction can be defined in a straightforward way on second canonical K-forms. We write

$$[\![X]\!]_K$$

for the thread extraction of K-program X. Of course, structural congruent K-programs define identical threads. In the case that a K-program contains no repeat instruction, we define

$$[\![u_1; \ldots; u_n]\!]_K \stackrel{\text{def}}{=} [\![u_1; \ldots; u_n; \#0; \backslash\backslash \#1]\!]_K.$$

To define behavior extraction on second canonical K-forms $u_1; \ldots; u_{n+k}; \backslash\backslash \#n$ we use an auxiliary function

$$|j, u_1; \ldots; u_{n+k}; \backslash\backslash \#n|$$

where the number j refers to the position of instructions:

$$[\![u_1; \ldots; u_{n+k}; \backslash\backslash \#n]\!]_K \stackrel{\text{def}}{=} |1, u_1; \ldots; u_{n+k}; \backslash\backslash \#n|$$

and $|j, u_1; \ldots; u_n; \backslash\backslash \#n|$ is defined by the case distinctions in Table 4.

We state without proof the following result, implying that for K-program X, $[\![X]\!]_K$ agrees with thread extraction on PGA-programs (see Section 2.3 and recall that

$$|u_1; \ldots; u_n| = |u_1; \ldots; u_n; (\#0)^\omega|$$

can be inferred from the equations in Table 2).

Theorem 3. *Let $k \geq 0$, $n > 0$ and let u_i range over the primitive instructions. Then*

$$[\![u_1; \ldots; u_k; u_{k+1}; \ldots; u_{k+n}; \backslash\backslash \#n]\!]_K = |u_1; \ldots; u_k; (u_{k+1}; \ldots; u_{k+n})^\omega|.$$

4 Discussion and conclusions

We provided an algebraic theory of an ASCII representation of program algebra PGA by replacing its repetition operator by a family of repeat instructions $\backslash\backslash\#k$, where the counter k ranges over the natural numbers larger than 0. The resulting semigroup L admits representation by first canonical L-forms (axiom (3)). We distinguished a kernel K of L and provided axioms for single-pass congruence, structural congruence and thread extraction. As of yet, we see no other application for K than that it confirms the point of view that a program is "a sequence of instructions", and that it highlights that the converse is not true: not each sequence of instructions can be called a "program".

The contents of this paper adheres to the philosophy of PGA, i.e.,

1. The mathematical object denoted by a program is a SPI, while the program's meaning is a thread to be extracted from that SPI, and
2. A SPI is the sort of object for which single-pass execution is the preferred operational semantics.

Our main motivation to undertake this research is the idea that in the setting of PGA the notion of programming languages or program notations as defined in [3] should be reconsidered. In particular, it is questionable whether the program notation PGLA, which is in fact L as defined in this paper, is an appropriate example of a programming language. The criterion formulated in [3] to use this terminology is the existence of a projection function `pgla2pga` (PGLAtoPGA) that maps any PGLA-program (L-sequence) to a PGA-program. In fact, PGLA was considered a first and basic candidate for a PGA-based programming language and served as the basis for a tool set and programming environment for program algebra [6].

However, a typical property of the projection function `pgla2pga` is that if the repeater of a first canonical form in L is "too large", it adds #0-instructions to obtain a first canonical form in K. This solution does not combine in an elegant way with jumps, as witnessed by the following examples where we abbreviate $|\texttt{pgla2pga}(X)|$ by $|X|_{pgla}$ (as is done in [3]):

$$|\mathsf{a};\#1;\backslash\backslash\#3|_{pgla} = |\mathsf{a};\#1;\#0;\backslash\backslash\#3|_{pgla} = |(\mathsf{a};\#1;\#0)^\omega| = \mathsf{a}\circ\mathsf{D},$$
$$|\mathsf{a};\#2;\backslash\backslash\#3|_{pgla} = |\mathsf{a};\#2;\#0;\backslash\backslash\#3|_{pgla} = |(\mathsf{a};\#2;\#0)^\omega| = \mathsf{a}^\infty,$$

and, more generally,

$$|\mathsf{a};\#k;\backslash\backslash\#3|_{pgla} = |(\mathsf{a};\#k;\#0)^\omega| = \begin{cases} \mathsf{a}^\infty & \text{if } k \bmod 3 = 2, \\ \mathsf{a}\circ\mathsf{D} & \text{otherwise.} \end{cases}$$

So in PGLA's projection of $|\mathsf{a};\#k;\backslash\backslash\#3|_{pgla}$, deadlock D *either* arises from the added #0 instruction *or* from the interplay with $\backslash\backslash\#3$ and the original jump instruction $\#k$. This we now consider rather arbitrary, and we prefer to view K, a proper subset of PGLA, as the programming language that is closest to PGA.

Our conclusion is that we consider PGA the more basic theory for providing semantics for sequential programming (instead of L or K), if only because there is no canonical interpretation

of L outside K. This agrees with the point of departure adopted in [3]: a programming language is a pair (E, ϕ) with E a set of expressions (the programs) and ϕ a projection function to PGA. In particular, the projection function `pgla2pga` maintains its definitional status: it agrees with the original definition of PGLA and at the same time with K as the (proper) subset of its programs. We note that K satisfies a property that is often seen in imperative programming: if

$$P; Q$$

is a program, then P and Q need not be (well-formed) programs. This is not the case in PGA where decomposition of concatenated programs is valid, or in the setting of SPIs.

Finally, a word on related work: PGA can be viewed as a theory of instruction sequences with our kernel K or PGLA as one of its many representations. Unfortunately, we have not been able to identify any pre-existing theory by other authors to which this work can be related in a convincing manner. The phrase *instruction sequence* seems not to play a clear role in the theory of programming. The software engineering literature at large features many uses of this phrase, but only in a casual setting.

References

[1] J.A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J.C.M. Baeten, J.K. Lenstra, J. Parrow, and G.J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4*, Springer-Verlag, LNCS 2719:1-21, 2003.

[2] J.A. Bergstra, I. Bethke, and A. Ponse. Decision problems for pushdown threads. *Acta Informatica*, 44(2):75–90, 2007.

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

[4] J.A. Bergstra and C.A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 19(4):445–474, 2007.

[5] J.A. Bergstra and A. Ponse. Program algebra with repeat instruction. Electronic report PRG0602, Programming Research Group, University of Amsterdam, June 2006.

[6] B. Diertens. *PGA - ProGram Algebra*. Website containing a Toolset for PGA: `www.science.uva.nl/research/prog/projects/pga/`, Last modified: July 3, 2006.

[7] 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.