Periodic single-pass instruction sequences

Bergstra, J.A.; Ponse, A.

Publication date
2013

Document Version
Final published version

Periodic Single-Pass Instruction Sequences

Jan A. Bergstra   Alban Ponse

Informatics Institute
University of Amsterdam

www.science.uva.nl/~janb/   www.science.uva.nl/~alban/

Abstract

A program is a finite piece of data that produces a (possibly infinite) sequence of primitive instructions. From scratch we develop a linear notation for sequential, imperative programs, using a familiar class of primitive instructions and so-called repeat instructions, a particular type of control instructions. The resulting mathematical structure is a semigroup. We relate this set of programs to program algebra (PGA) and show that a particular subsemigroup is a carrier for PGA by providing axioms for single-pass congruence, structural congruence, and thread extraction. This subsemigroup characterizes periodic single-pass instruction sequences and provides a direct basis for PGA's toolset.

Keywords: Program algebra, Repeat instruction, Equational specification.

Contents

1 Introduction **2**

2 K-basics **3**

2.1 Primitive instructions, K and K_r . . . 3

2.2 Single-pass congruence and first canonical forms . . . 4

2.3 K_r-programs and their first canonical forms . . . 6

3 Execution of K_r-programs **7**

3.1 Thread algebra . . . 7

3.2 Behavioral semantics for K_r-programs: threads . . . 9

3.3 K_r-programs, second canonical forms and thread extraction . . . 11

4 Discussion and conclusions **13**

4.1 Program algebra . . . 13

4.2 PGLA . . . 15

References **16**

1 Introduction

Our starting point of view is that a "program" is a finite 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* a SPI. Primitive instructions comprise jump instructions, test instructions and basic instructions that upon execution may alter some state; in the next section we introduce a set $\mathcal{U}$ of primitive instructions that we used in previous research.

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. This point of departure represents a most basic view of what a program constitutes: the syntactic denotation of a SPI to be executed in single-pass mode.[1] A SPI can either be finite or infinite.

A very basic question is how to define programs that produce the classes of SPIs we are interested in, adopting the point of view that a program itself is a finite sequence of instructions. A first, very straightforward and simple approach to this question is to start from constants for primitive instructions and to adopt *concatenation* as an operation for composing programs: each primitive instruction is a program, and if P and Q are programs, then so is their concatenation $P; Q$. Furthermore, it is useful to postulate that concatenation is associative and (thus) to leave out brackets in repeated concatenations. This implies that programs built in this way represent the most simple set of programs that produce finite SPIs, and that in mathematical terms, this set constitutes a *semigroup*. We shall use the notation

$$K$$

for this very basic semigroup, where K abbreviates *Kernel instruction sequence notation*.

However, in order to give an account of sequential, imperative programming one needs programs that can produce certain *infinite* SPIs (cf. programs that define the finite-state control of a Turing machine). An infinite SPI is *periodic* if it can be produced by a program of the form

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

with $u_i \in \mathcal{U}$, the set of primitive instructions, $n > 0$ and $k \geq 0$, and with the *repeat instruction* $\backslash\backslash\#n$, which is defined as follows: for n a natural number larger than 0,

$$\backslash\backslash\#n$$

prescribes to repeat the last preceding n instructions. The repeat instruction, or briefly *repeater* is a *control instruction* to be used for the definition of a program that produces a periodic SPI. As an example with u a primitive instruction, the program $u; \backslash\backslash\#1$ (which consists of two instructions) produces the periodic SPI that consists of an infinite number of u-instructions, and the same SPI is produced by $u; u; \backslash\backslash\#1$ and by many more programs, for example by $u; u; \backslash\backslash\#2$ and $u; u; u; \backslash\backslash\#2$.

In order to provide a setting for defining programs as finite sequences of instructions, we define below two semigroups, where we write S^+ for the set of finite sequences with elements from alphabet S and for which we (also) use ";" as a separator:

[1] In [4], SPIs are referred to as *program objects*.

- The semigroup with domain $\mathcal{U}^+$ and concatenation as its operation, representing the finite SPIs *and* the finite programs over $\mathcal{U}$. As stated above, we use the name K for this semigroup.
- The semigroup with domain $(\mathcal{U} \cup \{\backslash\backslash\#n \mid n \in \mathbb{N}_{>0}\})^+$ and concatenation as its operation; this semigroup will be used to represent periodic SPIs. We use the name K_r for this semigroup.

In Section 2 we introduce the primitive instructions we work with and for programs in K and K_r, we provide an axiomatization of *single-pass congruence*, the congruence that identifies programs that produce identical SPIs. We discuss the fact that not each sequence of instructions in K_r can be called a program. For example, the question whether the K_r-expression

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

produces a SPI — and if so, which one — has no obvious answer. We distinguish a subset of the domain of K_r that rules out this question and contains the K_r*-programs* that produce all finite and periodic SPIs. In Section 3 we discuss the behavior of K_r-programs using *thread algebra* and we define a thread extraction operator that can be applied to K_r-programs. For K-programs and K_r-programs we provide axioms for *structural congruence*, a congruence that admits the unchaining of jump counters and preserves the behavioral semantics of programs (thread extraction applied to structural congruent programs yields equal threads). In Section 4, we relate our approach to PGA (program algebra, [4]), which represents the analysis of SPIs starting from a more mathematically oriented design of a program notation for periodic SPIs (comprising a repetition *operator* instead of repeaters), and to our program notation C [7], a program notation design based on primitive instructions that explicitly prescribe whether the orientation of the execution order is left-to-right or vice versa.

2 K-basics

In this section we formally define our set of primitive instructions (taken from [4]) and the semigroups K and K_r. Then we discuss canonical forms as a preferred form of representation of K_r-expressions and define K_r*-programs*.

2.1 Primitive instructions, K and K_r

Let A be a set of constants and write $\mathbb{N}_{>0}$ for $\mathbb{N} \setminus \{0\}$, where $\mathbb{N}$ represents the natural numbers.

Definition 1. *K**-expressions**, also called K**-programs**, are defined by the following grammar, where $\mathtt{a} \in A$ and $k \in \mathbb{N}$:*

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

*and where the operation ; is called **concatenation**.*

*K_r**-expressions** are defined by the following grammar, where $\mathtt{a} \in A$ and $k \in \mathbb{N},\ n \in \mathbb{N}_{>0}$:*

$$P ::= \mathtt{a} \mid +\mathtt{a} \mid -\mathtt{a} \mid \#k \mid ! \mid \backslash\backslash\#n \mid P; P$$

*(K_r**-programs** are defined in Definition 4).*

Let $\mathtt{a} \in A$ and $k \in \mathbb{N}$. Then each of $\mathtt{a}, +\mathtt{a}, -\mathtt{a}, \#k, !$ is called a *primitive instruction* and primitive instructions occurring in K-expressions can be explained as follows:

- A *basic instruction* $\mathtt{a} \in A$ prescribes an atomic piece of behavior that is considered indivisible and executable in finite time. After completion of its execution, it prescribes to execute the next instruction (if available). One can consider various specific instances of A and we mention here the set of *molecular programming primitives*, see, e.g., [1].

- A basic instruction can be turned into a *test instruction* by prefixing it with either the symbol $+$ (positive test instruction) or with the symbol $-$ (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 that may depend on the execution state:[2] we explain this in detail in Section 3.1.

- A next kind of primitive instruction is the *jump instruction* $\#k$ where $k \in \mathbb{N}$: this instruction prescribes to jump k primitive instructions ahead (if possible; otherwise deadlock occurs) and generates no observable behavior. The special case $\#0$ prescribes deadlock.

- The *termination instruction* ! prescribes successful termination, an event that is taken to be observable.

We write $\mathcal{U}$ for the set of primitive instructions and we shall use $u, u_1, u_2, ..., v, v_1, v_2, ...$ as typical variables for elements in $\mathcal{U}$. We define each element of $\mathcal{U}$ to be a SPI (Sequence of Primitive Instructions).

Finite SPIs are produced by K-expressions (see Definition 1). We take concatenation to be an *associative* operator and leave out brackets in repeated concatenations, so we simply write

$$u_1; u_2; \ldots; u_n$$

for the K-expression built up from the primitive instructions $u_1, ..., u_n$ by $n-1$ repeated concatenations. Thus K is the free semigroup with generators from $\mathcal{U}$.

Finally, the non-primitive *repeat instruction* $\backslash\backslash\#n$, where $n \in \mathbb{N}_{>0}$, prescribes to repeat the last preceding n instructions. Repeat instructions are also called *repeaters*. So-called *periodic* SPIs are produced by K_r-expressions (see Definition 1). Again, we take concatenation to be an *associative* operator and leave out brackets in repeated concatenations, thus K_r is the free semigroup generated by $\mathcal{U} \cup \{\backslash\backslash\#n \mid n \in \mathbb{N}\}$. By definition, K is a subsemigroup of K_r.

2.2 Single-pass congruence and first canonical forms

In this section we define *single-pass congruence*, the congruence that characterizes extensional equality of SPIs, i.e., the equality defined by having the same primitive instruction at each position in the SPI that is produced.[3]

For K-expressions, thus K_r-expressions not containing repeaters, single-pass congruence boils down to the associativity of concatenation.

[2] Upon reply `true`, a positive test instruction prescribes to execute the next instruction (if available) and

$$(u_1;\ldots;u_n)^m;\backslash\backslash\#mn = u_1;\ldots;u_n;\backslash\backslash\#n \tag{1}$$

$$\backslash\backslash\#n;X = \backslash\backslash\#n \tag{2}$$

$$u_1;\ldots;u_m;v_1;\ldots;v_n;u_1;\ldots;u_m;\backslash\backslash\#m{+}n = u_1;\ldots;u_m;v_1;\ldots;v_n;\backslash\backslash\#m{+}n \tag{3}$$

Table 1: The axiom set E_{spc} for single-pass congruence on K_r-expressions, where $m,n \in \mathbb{N}_{>0}$ and $u_i, v_j \in \mathcal{U}$

Define for $n > 0$ and X an K_r-expression, $X^{n+1} = X;X^n$ and $X^1 = X$. Single-pass congruence for K_r-expressions is axiomatized by the axiom schemes (1)–(3) in Table 1 and equational logic, and we write

$$E_{spc}$$

for this proof system. Although equations (1)–(3) are in fact *schemes* in $m,n \in \mathbb{N}_{>0}$, we further refer to these as "axioms". Whenever two K_r-expressions X and Y are single-pass congruent, this is written

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

and the subscript *spc* will be dropped if no confusion can arise.

Proposition 1. *The unfolding property*

$$u_1;\ldots;u_n;\backslash\backslash\#n = (u_1;\ldots;u_n)^2;\backslash\backslash\#n$$

follows from E_{spc}.

Proof.

$$\begin{aligned}
E_{spc} \vdash u_1;\ldots;u_n;\backslash\backslash\#n &= (u_1;\ldots;u_n)^2;\backslash\backslash\#2n && \text{by (1)}\\
&= u_1;\ldots;u_n;(u_1;\ldots;u_n)^2;\backslash\backslash\#2n && \text{by (3)}\\
&= u_1;\ldots;u_n;u_1;\ldots;u_n;\backslash\backslash\#n && \text{by (1)}\\
&= (u_1;\ldots;u_n)^2;\backslash\backslash\#n.
\end{aligned}$$

□

In E_{spc}, axiom (2) implies that each expression in K_r can be equated to one that contains *at most* one repeat instruction. This leads to the following preferred representation of K_r-expressions.

upon reply `false` it prescribes to skip the next instruction and to proceed execution with the instruction thereafter. A negative test instruction has the same effect, but with the role of the replies reversed.

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

Definition 2. *A K_r-expression is a* ***first canonical form*** *if it is of the form*

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

with $u_i \in \mathcal{U}$, $k \in \mathbb{N}$ and $n \in \mathbb{N}_{>0}$, where $u_1; \ldots; u_0;$ represents the empty sequence.

For each K_r-expression, its first canonical form is obtained by applying axiom (2) to the leftmost occurring repeater if present, and otherwise it is that expression itself.

Not all K_r-expressions 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 forms can not be rewritten using any of the axioms (1)–(3) in Table 1. As a consequence, single-pass congruence is not a meaningful notion for such first canonical forms and in the next section we will exclude such K_r-expressions.

2.3 K_r-programs and their first canonical forms

Let K_r^- stand for the subset of K_r-expressions whose first canonical form has the property that the repeat instruction $\backslash\backslash\#n$ (if present) is preceded by at least n primitive instructions. In fact, K_r^- is a subsemigroup of K_r: if $X, Y \in K_r^-$, then $X; Y \in K_r^-$. In the following definition, we refine the notion of a first canonical form.

Definition 3. *Let $u_1; \ldots; u_k$ with $k \geq 1$ be a SPI (thus all u_i are primitive instructions). Then $u_1; \ldots; u_k$ is a* ***first canonical*** ***K_r-form****. This first canonical K_r-form is* ***minimal*** *by definition.*

The K_r-expression $u_1; \ldots; u_k; \backslash\backslash\#n$ is a ***first canonical K_r-form*** *if $0 < n \leq k$. This first canonical K_r-form is* ***minimal*** *if its repeating part $u_{k-n+1}; \ldots; u_k$ can not be made smaller with axiom (1), and its non-repeating part $u_1; \ldots; u_{k-n}$ can not be made smaller with axiom (3).*

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

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

Definition 4. *Elements in K_r^- are referred to as* ***K_r-programs****.*

Recall that two K_r-programs P and Q are single-pass congruent if, and only if,

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

Single-pass congruence for K_r-programs is captured by the next result.

Theorem 1. *Single-pass congruence of K_r-programs is decidable.*

Proof. Assume P and Q are two K_r-programs that denote identical SPIs. If both programs do not contain repeat instructions, they are syntactically identical, apart from the possible use of brackets, which is then captured by the associativity of concatenation, which we adopted throughout this paper. In the other case, application of axiom (2) yields first canonical K_r-forms and these expressions still denote the same SPI. With axiom (3) the non-repeating parts of the two K_r^--expressions (if present) can be made as short as possible, so these should be identical for both expressions. Removal of these non-repeating parts yields two expressions of the form $u_1;\ldots;u_n;\backslash\backslash\#n$ and $v_1;\ldots;v_m;\backslash\backslash\#m$ that denote identical SPIs. With axiom (1) one then derives

$$\begin{aligned}E_{spc} \vdash u_1;\ldots;u_n;\backslash\backslash\#n &= (u_1;\ldots;u_n)^m;\backslash\backslash\#nm\\ &= (v_1;\ldots;v_m)^n;\backslash\backslash\#nm\\ &= v_1;\ldots;v_m;\backslash\backslash\#m.\end{aligned}$$

Of course, the values n and m in these repeating parts can be effectively minimized with axiom (1). Then single-pass congruence coincides with the syntactic equality of both minimal first canonical K_r-forms, which immediately implies the mentioned decidability. □

Without loss of generality, we further only consider K_r-programs that contain at most one repeat instruction.

3 Execution of K_r-programs

We briefly discuss Thread Algebra (cf. [9]), earlier described in e.g. [2, 4]. For basic information on thread algebra we refer to [3, 9]; more advanced matters, such as an operational semantics for thread algebra, are discussed in [5].

3.1 Thread algebra

Threads model the execution of SPIs. In order to define threads, we consider the set A of basic instructions also as a set of so-called *actions* that model the execution of basic and test instructions, where it is assumed that execution of the action $\mathtt{a}$ yields a Boolean reply `true` or `false`. Finite threads are defined inductively in the following way:

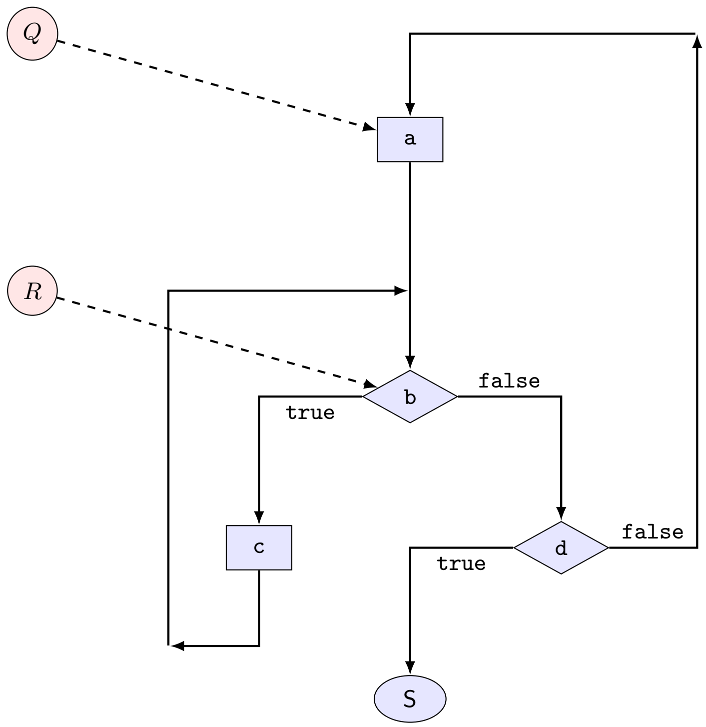
S	*stop*, the termination thread,
D	*inaction* or *deadlock*, the inactive thread,
$P \trianglelefteq \mathtt{a} \trianglerighteq Q$	the *postconditional composition* $\_ \trianglelefteq \mathtt{a} \trianglerighteq \_$ of finite threads P and Q, where $\mathtt{a} \in A$.

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.

A so-called *regular* thread over A is a finite-state thread in which infinite paths can occur (so, finite threads form a special subset of regular threads). Each regular thread can be defined by a finite number of recursive equations. As a first example, consider the regular thread Q defined by

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

This regular thread Q can be depicted in the following way:



Each regular thread can be specified using a so-called linear recursive specification.

Definition 5. *A* ***linear recursive specification*** *is a set of equations*

$$\{P_i = t_i(\overline{P}) \mid i = 0, ..., n\}$$

with $t_i(\overline{P})$ *of the form* S, D*, or* $P_{i_1} \trianglelefteq \mathtt{a}_i \trianglerighteq P_{i_2}$*, where* $\mathtt{a}_i \in A$ *and* $i_1, i_2 \leq n$*.*

For the example above, we find $Q = P_0$ for P_0 defined by the following linear equations:

$$P_0 = P_1 \trianglelefteq \mathtt{a} \trianglerighteq P_1,$$
$$P_1 = P_2 \trianglelefteq \mathtt{b} \trianglerighteq P_3,$$
$$P_2 = P_1 \trianglelefteq \mathtt{c} \trianglerighteq P_1,$$
$$P_3 = P_4 \trianglelefteq \mathtt{d} \trianglerighteq P_0,$$
$$P_4 = \mathsf{S}.$$

In the next section we explain in what way K_r-programs define regular threads.

Let $X = u_1; \dots ; u_{n+k}; \backslash\backslash\#n$, then $[\![X]\!]_{K_r} = |1, X|$, where

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

Table 2: Equations for thread extraction on K_r^-, where $u_i \in \mathcal{U}$, $k \in \mathbb{N}$ and $j, n \in \mathbb{N}_{>0}$

3.2 Behavioral semantics for K_r-programs: threads

As mentioned before, the execution of a SPI is *single-pass*: it starts with the first instruction, and each instruction is dropped after it has been executed or jumped over. In this section we explain the precise *meaning* of primitive instructions and K_r-programs in terms of their execution.

Let X be a K_r-program of the form

$$X = u_1; \dots ; u_{n+k}; \backslash\backslash\#n,$$

thus X is a first canonical K_r-form. In Table 2 we define the *thread extraction* of X, notation

$$[\![X]\!]_{K_r},$$

where the auxiliary function $|j, X|$ models the thread extraction of program X when started at its j'th instruction. In the general case of a K_r-program X, its thread extraction is defined by

$$[\![X]\!]_{K_r} = [\![X; \#0; \backslash\backslash\#1]\!]_{K_r},$$

thus the SPI produced by X that — if it is finite — is extended with an infinite number of #0-instructions. Because each K_r-program of the form $X; \#0; \backslash\backslash\#1$ can be converted to a first canonical form $u_1; \dots ; u_{n+k}; \backslash\backslash\#n$, the equations in Table 2 match all possible cases:

- Repeaters and the termination instruction ! are dealt with in the first two equations for $|j, X|$. Observe that termination must always be explicitly defined using !.
- A basic or test instruction yields the equally named action in a post conditional composition. In the case of a positive test instruction $+\mathtt{a}$, the reply `true` to the associated action `a` prescribes to continue with the next instruction and the reply `false` prescribes to skip the next instruction and to continue with the instruction at the position thereafter; for

the execution of a negative test instruction $-\mathsf{a}$, subsequent execution is prescribed by the complementary replies. If there is no next instruction to be executed, deadlock follows.

- A #0-instruction yields deadlock upon execution, and a jump instruction $\#k{+}1$ shifts $|j, X|$ to $|j{+}k{+}1, X|$.

A first, very simple example is the regular thread obtained by thread extraction of the K_r-program $+\mathsf{a}; \backslash\backslash\#1$. We find that this program prescribes the execution of an infinite sequence of a-actions:

$$\begin{aligned}
[\![+\mathsf{a}; \backslash\backslash\#1]\!]_{K_r} = |1, +\mathsf{a}; \backslash\backslash\#1| &= |2, +\mathsf{a}; \backslash\backslash\#1| \trianglelefteq \mathsf{a} \trianglerighteq |3, +\mathsf{a}; \backslash\backslash\#1| \\
&= |1, +\mathsf{a}; \backslash\backslash\#1| \trianglelefteq \mathsf{a} \trianglerighteq |2, +\mathsf{a}; \backslash\backslash\#1| \\
&= \mathsf{a} \circ |1, a; \backslash\backslash\#1|,
\end{aligned}$$

and thus the regular thread captured by the single recursive equation

$$P = \mathsf{a} \circ P \tag{e_1}$$

and we may write $[\![+\mathsf{a}; \backslash\backslash\#1]\!]_{K_r} = P$ for P defined as in equation (e_1).

The equations in Table 2 need not immediately yield a regular thread for each K_r-program: it can be the case that these equations can be consecutively applied without yielding any action, as for example for the program $X = \#4; \mathsf{a}; \backslash\backslash\#2$, for which we derive

$$\begin{aligned}
[\![X]\!]_{K_r} = |1, X| &= |5, X| \\
&= |3, X| \\
&= |1, X|.
\end{aligned}$$

In such cases we define the extracted behavior to be D, and with this *default-rule* for thread extraction, each K_r-program defines a regular thread.

Example 1. *Let* $X = +\mathsf{a}; \#0; +\mathsf{b}; \#4; -\mathsf{c}; \#0; \backslash\backslash\#4$*. We show that* $[\![X]\!]_{K_r} = \mathsf{D} \trianglelefteq \mathsf{a} \trianglerighteq P$ *with* P *defined by the recursive equation* $P = \mathsf{D} \trianglelefteq \mathsf{b} \trianglerighteq (P \trianglelefteq \mathsf{c} \trianglerighteq \mathsf{D})$*.*

Let $Y = +\mathsf{a}; \#0; +\mathsf{b}; \#4; -\mathsf{c}; \backslash\backslash\#4$*, then* $X =_{spc} Y$ *by axiom* (3) *and hence* $[\![X]\!]_{K_r} = [\![Y]\!]_{K_r}$*. We first derive an intermediate result:*

$$\begin{aligned}
|4, Y| &= |8, Y| \\
&= |4, Y|,
\end{aligned}$$

so by the default-rule, $|4, Y| = \mathsf{D}$*. Finally, we derive*

$$\begin{aligned}
[\![Y]\!]_{K_r} &= |1, Y| \\
&= |2, Y| \trianglelefteq a \trianglerighteq |3, Y| \\
&= \mathsf{D} \trianglelefteq a \trianglerighteq P,
\end{aligned}$$

$$\begin{aligned}
\textit{where } P &= |3, Y| \\
&= |4, Y| \trianglelefteq b \trianglerighteq |5, Y| \\
&= \mathsf{D} \trianglelefteq b \trianglerighteq (|7, Y| \trianglelefteq c \trianglerighteq |6, Y|) \\
&= \mathsf{D} \trianglelefteq b \trianglerighteq (|3, Y| \trianglelefteq c \trianglerighteq |2, Y|) \\
&= \mathsf{D} \trianglelefteq b \trianglerighteq (P \trianglelefteq c \trianglerighteq \mathsf{D}).
\end{aligned}$$

Conversely, each regular thread over A can be specified (programmed) by a K_r-program, as we will discuss in Section 4.1. For example, the regular thread Q that was discussed above and that was specified by the equations

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

satisfies $Q = [\![\mathtt{a}; +\mathtt{b}; \#2; \#3; \mathtt{c}; \#4; +\mathtt{d}; !; \backslash\backslash\#8]\!]_{\mathtt{K_r}}$.

To conclude this section, we mention the fact that in terms of execution behavior, certain different regular threads should be considered equal, e.g.,

$$[\![\mathtt{a}; \backslash\backslash\#1]\!]_{K_r} \quad \text{and} \quad [\![+\mathtt{a}; \mathtt{a}; \backslash\backslash\#2]\!]_{K_r}$$

because both perform repeatedly the action $\mathtt{a}$ and are thus behaviorally equivalent. A formal way to prove this behavioral equivalence is discussed in [4] (and summarized in [9]) and is considered outside the scope of this paper. Finally, observe that behavioral equivalence of K_r-programs, say $\equiv_{be}$, is not a congruence: although $\mathtt{a} \equiv_{be} +\mathtt{a}$ because both define the thread $\mathtt{a} \circ \mathsf{D}$, we find $\mathtt{a}; ! \not\equiv_{be} +\mathtt{a}; !$ because $\mathtt{a} \circ \mathsf{S} \neq \mathsf{S} \trianglelefteq \mathtt{a} \trianglerighteq \mathsf{D}$.

3.3 K_r-programs, second canonical forms and thread extraction

One can change the jump counters in K_r-programs while preserving execution behavior, for example

$$+\mathtt{a}; \#2; +\mathtt{b}; \#2; c; d; e \quad \text{and} \quad +\,\mathtt{a}; \#5; +\mathtt{b}; \#2; c; d; e$$

execute apart from their jump counters the same instructions and their thread extraction yields identical threads. The crucial difference between these programs is that the rightmost program contains no *chained jumps*. In Table 3 we introduce the axiom schemes (4)–(7) for the unchaining of jump instructions and we write

$$E_{sc}$$

for the extension of E_{spc} with these axiom schemes. Although (1)–(7) are axiom schemes in $m, n \in \mathbb{N}_{>0}$ and $k, \ell \in \mathbb{N}$, we shall refer to all of these as "axioms". The congruence defined by E_{sc} is called *structural congruence*, and whenever two K_r-programs X and Y are structurally congruent, this is written

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

Note that first canonical forms not in K_r^- (thus, with a repeat counter that is too large) can not be rewritten using any of the axioms (1)–(7) in Table 3 that contain repeaters. As a consequence, structural congruence is not a meaningful notion for such first canonical forms.

Definition 6. *A* ***second canonical*** K_r***-form*** *is a first canonical K_r-form in which no chained jumps occur, and in the case of $u_1; \ldots; u_m; \backslash\backslash\#n$, in which all jumps to $u_{m-n+1}, \ldots, u_m$ are minimized using axioms* (6) *and* (7).

$$(u_1;\ldots;u_n)^m;\backslash\backslash\#mn = u_1;\ldots;u_n;\backslash\backslash\#n \tag{1}$$
$$\backslash\backslash\#n;X = \backslash\backslash\#n \tag{2}$$
$$u_1;\ldots;u_m;v_1;\ldots;v_n;u_1;\ldots;u_m;\backslash\backslash\#m{+}n = u_1;\ldots;u_m;v_1;\ldots;v_n;\backslash\backslash\#m{+}n \tag{3}$$

$$\#k{+}1;u_1;\ldots;u_k;\#0 = \#0;u_1;\ldots;u_k;\#0 \tag{4}$$
$$\#k{+}1{+}n;u_1;\ldots;u_k;\#n = \#k{+}1;u_1;\ldots;u_k;\#n \tag{5}$$
$$\#k{+}1{+}\ell;u_1;\ldots;u_k;\backslash\backslash\#k{+}1 = \#\ell;u_1;\ldots;u_k;\backslash\backslash\#k{+}1 \tag{6}$$
$$\#k{+}1{+}\ell{+}n;u_1;\ldots;u_k;v_1;\ldots;v_n;\backslash\backslash\#n = \#k{+}1{+}\ell;u_1;\ldots;u_k;v_1;\ldots;v_n;\backslash\backslash\#n \tag{7}$$

Table 3: The axiom set E_{sc} for structural congruence on SPIs, where $k,\ell\in\mathbb{N}$, $m,n\in\mathbb{N}_{>0}$, $u_i,v_j\in\mathcal{U}$, and $u_1;\ldots;u_0;$ represents the empty sequence

Two typical examples, where the K_r-programs in the right-hand column are second canonical K_r-forms (and those in the left-hand column are not):

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

The first example is an instance of axiom (6) ($k=\ell=0$). The last example also provides second canonical K_r-forms for the K_r-program considered in Example 1.

It is easily seen that in E_{sc}, second canonical K_r-forms have a unique minimal representation in terms of their number of instructions and we have the following result.

Theorem 2. *Structural congruence of K_r-programs is decidable, and two K_r-programs P and Q are structurally congruent if, and only if,*

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

Proof. By the proof of Theorem 1 it suffices to consider minimal first canonical K_r-forms, and it is trivial to convert such a form to a second canonical K_r-form. Minimization of the length of the repeating part with (1) then yields a unique K_r-program (cf. the last example above). □

Thread extraction is more straightforward when applied to second canonical K_r-forms: structural congruent K_r-programs define identical threads and because the infinite chaining of jumps is excluded, the rules in Table 2 are then complete and there is no more need for the *default-rule* (that stated that whenever the equations do not yield any action, the resulting behavior is D).

4 Discussion and conclusions

Our main motivation to undertake the current research is that in the setting of program algebra (PGA), the notion of a 'program notation' as defined in [4] should be strengthened, and we return to this question in Section 4.2. Program algebra was introduced as a general approach to model and analyze the notion of a sequential, imperative program in the form of a rather 'non-formal' and theoretical style. An algebra of these programs named PGA is used as the carrier for a further development of this matter, and the syntax of PGA serves as a very simple and basic program notation, underlying many other program notations. In Section 4.1 we relate our approach to PGA.

In Section 4.2 we conclude the paper with a consideration about PGLA, an earlier account of the semigroup K_r that was proposed as a machine-readable version of PGA and that underlies the current toolset for PGA [8]: K_r-expressions are precisely the programs that can be processed by this toolset.

PGA can be viewed as a theory of instruction sequences with our subsemigroup K_r^- 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, and the software engineering literature at large features many uses of this phrase, but only in a casual setting.

4.1 Program algebra

PGA was set up in a very similar way as K_r, with the only difference that instead of repeat instructions, a unary operator called *repetition* is used. The notation for this operator is

$$(\_)^\omega$$

and its relation with K_r is captured by the equation scheme

$$u_1;\ldots;u_k;\backslash\backslash\#k = (u_1;\ldots;u_k)^\omega \quad \text{for } k \in \mathbb{N}_{>0}, \tag{e_2}$$

where $u_1, ..., u_k \in \mathcal{U}$, the set of primitive instructions that PGA and K_r are based on.

The associativity of concatenation and all further axiomatizations discussed previously, that is, the axiomatizations for single-pass congruence and structural congruence for K_r are the direct counterparts of those provided for PGA when equation scheme (e_2) is applied, and the same holds for the equations that define thread extraction. From a mathematical point of view, all such axioms and equations formulated in the setting of PGA are more elegant. The axiomatization of single-pass congruence in PGA is indeed so simple that it can be easily remembered by heart:

$$(X;Y);Z = X;(Y;Z) \tag{PGA1}$$
$$(X^n)^\omega = X^\omega \tag{PGA2}$$
$$X^\omega;Y = X^\omega \tag{PGA3}$$
$$(X;Y)^\omega = X;(Y;X)^\omega \tag{PGA4}$$

However, in terms of a *program notation* for finite or periodic SPIs, there is something to be said *against* PGA: its notation is not conforming to ASCII and exploits a scope-dependent unary operator $(\_)^\omega$.

Because of the immediate correspondence between K_r-programs and PGA-programs as characterized by equation scheme (e_2), many PGA-results also hold for K_r-programs. For example, each regular thread over A can be specified (programmed) by a K_r-program: assume a regular thread P_0 is given by the linear recursive specification

$$\{P_i = t_i(\overline{P}) \mid i = 0, ..., n\}$$

with $t_i(\overline{P})$ of the form S, D, or $P_{i_1} \trianglelefteq \mathtt{a}_i \trianglerighteq P_{i_2}$, where $\mathtt{a}_i \in A$ and $i_1, i_2 \leq n$. Then

$$P_0 = [\![\sigma_0(P_0); \ldots; \sigma_n(P_n); \backslash\backslash\#3n + 3]\!]_{K_r},$$

where

$$\sigma_i(P_i) = \begin{cases} !; \#0^2 & \text{if } t_i = \mathsf{S}, \\ \#0^3 & \text{if } t_i = \mathsf{D}, \\ +\mathtt{a}; \#f(n, i_1, i_2); \#g(n, i_1, i_2) & \text{if } t_i = P_{i_1} \trianglelefteq \mathtt{a}_i \trianglerighteq P_{i_2}, \end{cases}$$

for appropriate target functions f and g. For example, the regular thread Q discussed in Section 3.1 and specified by the equations

$$\begin{aligned} &Q = \mathtt{a} \circ R, \\ &R = \mathtt{c} \circ R \trianglelefteq \mathtt{b} \trianglerighteq (\mathsf{S} \trianglelefteq \mathtt{d} \trianglerighteq Q), \end{aligned}$$

and thus by P_0 in the linear recursive specification (see Definition 5) that consists of the linear equations

$$\begin{aligned} &P_0 = P_1 \trianglelefteq \mathtt{a} \trianglerighteq P_1, \\ &P_1 = P_2 \trianglelefteq \mathtt{b} \trianglerighteq P_3, \\ &P_2 = P_1 \trianglelefteq \mathtt{c} \trianglerighteq P_1, \\ &P_3 = P_4 \trianglelefteq \mathtt{d} \trianglerighteq P_0, \\ &P_4 = \mathsf{S}, \end{aligned}$$

satisfies

$$\begin{aligned} Q = P_0 = [\![&+\mathtt{a}; \#\mathtt{2}; \#\mathtt{1}; \\ &+\mathtt{b}; \#\mathtt{2}; \#\mathtt{4}; \\ &+\mathtt{c}; \#\mathtt{11}; \#\mathtt{10}; \\ &+\mathtt{d}; \#\mathtt{2}; \#\mathtt{4}; \\ &!; \#\mathtt{0}; \#\mathtt{0}; \backslash\backslash\#\mathtt{15} \\ ]\!]&_{K_r}. \end{aligned}$$

Observe that this result implies that negative test instructions and basic instructions do not increase expressiveness; indeed their sole purpose is to provide ease of specification. On the other hand, jump instructions with counters of unbounded size are crucial for the above-mentioned expressiveness result (cf. [4, 9]).

In [7] we introduced an alternative for K_r: the set $\mathcal{U}$ of primitive instructions that underlies PGA and K_r is replaced by a set of programming instructions that specifically prescribe whether the next instruction to be executed is concatenated to the right or to the left. The resulting semigroup C also produces all periodic SPIs over $\mathcal{U}$. More results on C are discussed in [10]. We also mention here [6], in which SPISA is extensively introduced (Single Pass Instruction Sequence Algebra), a variant of PGA that comprises next to the termination instruction also a positive termination instruction !t and a negative termination instruction !f.

4.2 PGLA

The program notation PGLA, which is in fact K_r as defined in this paper, was introduced in [4] as a first example of a 'programming language'. The criterion formulated in [4] to use this terminology is the existence of a projection function

`pgla2pga`

(PGLA to PGA) that maps any PGLA-program (K_r-expression) to a PGA-program. In fact, PGLA inspired a toolset and programming environment for program algebra [8]. However, we now conclude that we did not deal in a proper way with the non-standard case of programs with repeaters with a counter that is too large: the projection function `pgla2pga` then adds #0-instructions to obtain a first canonical K_r-form. This solution does not combine in an elegant way with jumps as witnessed by the following examples, where we write $|X|$ for the thread extraction of PGA-program X and use the following abbreviations:

$|Y|_{pgla}$ for $|\texttt{pgla2pga}(Y)|$ (as is done in [4]), and
a^∞ for the thread defined by $P = \mathsf{a} \circ P$.

Some examples:

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

and, more generally, for $k \in \mathbb{N}$ we find

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

So in PGLA's projection of $|\mathsf{a};\#k;\backslash\backslash\#3|_{pgla}$, deadlock D *either* arises from the added jump instruction #0, *or* from the interplay with \\#3 and the original jump instruction $\#k$, *or* does not arise. This we now consider rather arbitrary and we prefer to view K_r^-, a proper subset of PGLA, as the program notation that is closest to PGA.

Thus, our final conclusion is to avoid the question of "too large repeat counters" and to state that $u_1;\ldots;u_k;\backslash\backslash\#n$ is not a program whenever $k > n$. This agrees with the point of view to consider PGA the more basic theory for providing semantics for sequential programming from a *mathematical* point of view (instead of K_r or K_r^-) and with the point of departure adopted in [4]: a programming language is a pair (E, ϕ) with E a set of expressions (the programs) and ϕ a projection function to PGA. Finally, we note that K_r^- shares a property that is often seen in imperative programming: if

$$P; Q$$

is a K_r-program, then P and Q need not be K_r-programs (while K_r-expressions satisfy by definition the property that their decomposition yields K_r-expressions, and the same can be said for SPIs).

References

[1] J.A. Bergstra and I. Bethke. Molecular dynamics. *Journal of Logic and Algebraic Programming*, 51(2):193–214, 2002.

[2] 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 (Eds.), *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4*, Springer-Verlag, LNCS 2719:1-21, 2003.

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

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

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

[6] J.A. Bergstra and C.A. Middelburg. *Instruction Sequences for Computer Science.* Atlantis Studies in Computing, Vol. 2, Atlantis Press, 2012. ISBN: 978-94-91216-64-0.

[7] J.A. Bergstra and A. Ponse. An instruction sequence semigroup with involutive anti-automorphisms. *Scientific Annals of Computer Science*, 19:57-92, 2009. Also available at arXiv:0903.1352, November 7, 2009.

[8] B. Diertens. *PGA - ProGram Algebra*. Website containing a Toolset for PGA: `www.science.uva.nl/research/prog/projects/pga/` (Last modified: February 2, 2009).

[9] A. Ponse and M.B. van der Zwaag. An introduction to program and thread algebra. In A. Beckmann, U. Berger, B. Löwe, and J.V. Tucker (Eds.), *Logical Approaches to Computational Barriers: Proceedings CiE 2006*, LNCS 3988, pages 445-458, Springer-Verlag, 2006.

[10] S.H.P. Schroevers. Expressiveness and extensions of an instruction sequence semigroup. MSc Thesis Master of Logic, University of Amsterdam, 2010. Also available at arXiv:1003.1572., February 15, 2010.