University of Amsterdam
Programming Research Group

Program Algebra with Repeat Instruction

J.A. Bergstra
A. Ponse

Report PRG0602 June 2006

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

A. Ponse

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7592
e-mail: alban@science.uva.nl

Programming Research Group Electronic Report Series

Program Algebra with Repeat Instruction

Jan A. Bergstra [a,b] and Alban Ponse [a]

[a] *University of Amsterdam, Programming Research Group, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

[b] *Utrecht University, Department of Philosophy, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands*

Abstract

In the setting of program algebra (PGA) we consider the *repeat instruction*. This special instruction is designed to represent infinite sequences of instructions as finite, linear programs. The resulting program notation is called PGLA and can be considered a string syntax for PGA. We redefine the repeat instruction by allowing its counter to be also zero. Then we show that PGLAcore, a kernel of PGLA, can replace PGA as a carrier for program algebra by providing axioms for instruction sequence congruence, structural congruence and thread extraction. Finally, we provide three alternative projection semantics for PGLA that all coincide on PGLAcore.

Key words: Program algebra, Repetition operator, Repeat instruction, Equational specification.

1 Introduction

Program algebra (PGA) is a logical theory which provides an algebraic framework and semantical foundations for sequential programming. In this paper we consider PGLA, a variant of PGA in which programs are represented in a purely linear fashion, i.e., as a sequence of instructions. In particular we show that a kernel of PGLA can serve as a carrier for program algebra. This may be of importance for tool building or (semi-) automated theorem proving.

Given a set of *basic* instructions, a *primitive instruction* is either a basic one, a test-instruction converted from a basic instruction by prefixing it with $+$ or $-$, or a jump or termination instruction. In Section 2 we explain the syntax and meaning of primitive instructions in detail. A *program object* is a non-empty sequence of primitive instructions: each primitive instruction is a program, and if X and Y are programs, then so is $X;Y$, the *concatenation* of X and Y.

In particular, program objects may be infinite, e.g., if $\mathtt{a}$ is a basic instruction, an infinite sequence of $\mathtt{a}$'s constitutes a program object. Program objects are considered *single-pass instruction sequences*: during the run of the instruction sequence, each instruction is visited at most once and dropped after having been executed. Program objects are our main subject of interest.

In PGA all primitive instructions are considered constants and there are two operations on terms: concatenation and repetition. As sketched above, concatenation is a binary, infix operator and is taken to be associative (brackets are not used in repeated applications). In order to represent infinite program objects, PGA has repetition as a unary operator, written $(\_)^\omega$. For example, for $\mathtt{a}$ a basic instruction, $\mathtt{a}^\omega$ represents the infinite program object $\mathtt{a};\mathtt{a};\mathtt{a};\ldots$. In Section 2 we recall some basic information about PGA.

The program notation PGLA ([3]) is defined to represent PGA-programs (closed terms over PGA) in linear form: repetition is replaced by a family of single instructions and sequences are formed with concatenation only. This makes sense if the transition from theory (PGA) to practice (such as a Toolset for PGA [4]) is undertaken. Apart from the fact that ω is not an ASCII character, we intend in a setting with tools to suppress context-dependencies such as the opening bracket in a repetition. The instruction used in PGLA to replace the repetition operator is the *repeat instruction*:

$$\backslash\backslash\#n$$

prescribes to repeat the last n instructions if $n > 0$, and deadlock if $n = 0$. So for $n > 0$, $u_1;\ldots;u_n;\backslash\backslash\#n$ with all u_i primitive instructions represents the same program object as $(u_1;\ldots;u_n)^\omega$. A peculiarity of the repeat instruction is that it is *not* a primitive instruction and does not comply with the notion of a single pass instruction sequence. For example, the instruction $\backslash\backslash\#1$ in $\mathtt{a};\backslash\backslash\#1$ is never "visited and dropped", rather, upon its execution (thus after $\mathtt{a}$) it repeats $\mathtt{a}$ and preserves itself. Thus, $\mathtt{a};\backslash\backslash\#1$ is a sequence of two PGLA-instructions, but *represents* an infinite program object (namely an infinite sequence of $\mathtt{a}$'s, just as $\mathtt{a}^\omega$ does). In Section 3 we distinguish PGLAcore as a kernel of PGLA that can serve as a fully fledged basis for program algebra: a PGLA-program $u_1;\ldots;u_k;\backslash\backslash\#n$ is in PGLAcore if $k \geq n \geq 0$. For PGLAcore we provide axioms for *instruction sequence congruence*, axiomatizing the equivalence of program objects, and axioms for *structural congruence*, admitting the optimization of jump counters. The price to be paid for this linear representation of repetition is that the question how to deal with "too large" repeat counters has to be answered: what program object is represented by sequences such as $\mathtt{a};\backslash\backslash\#2$ or by a single repeat instruction? In Section 4 we discuss these matters in detail and provide some answers. Finally, in Section 5 we end with some conclusions.

2 PGA, two congruences and thread extraction

In this section some basic information about PGA (based on [3]) is recalled. Furthermore, we shortly discuss Thread Algebra (cf. [2]), earlier described in e.g. [1,3] under the name Polarized Process Algebra.

2.1 PGA, primitive instructions and program objects

The program notation PGA is based on a parameter set A of so-called *basic instructions*. These are regarded as indivisible units and execute in finite time. Furthermore, a basic instruction is viewed as a request to the environment, and it is assumed that upon its execution a boolean value (`true` or `false`) is returned that may be used for subsequent program control. The language PGA has two composition constructs:

Concatenation. If X and Y are PGA-programs, i.e., closed terms, then $X;Y$ is one as well.
Repetition. If X is a PGA-program, then so is X^ω.

Given A, the primitive instructions of PGA are the following:

Basic instruction. All elements of A, typically $\mathtt{a}, \mathtt{b}, \ldots$. When executed, a basic instruction generates a boolean value and the associated behavior may modify a state. After execution, a program has to enact its subsequent instruction. If that instruction fails to exist, inaction occurs. Subsequent execution is not influenced by the returned boolean value.
Termination instruction. The termination instruction ! yields termination of the program. It does not modify a state, and it does not return a boolean value.
Positive test instruction. For each element `a` of A there is a positive test instruction $+\mathtt{a}$. When executed, the state is affected according to `a`, and in case `true` is returned, the remaining sequence of instructions is executed. If there are no remaining instructions, inaction occurs. In the case that `false` is returned, the next instruction is skipped and execution proceeds with the instruction following the skipped one. If no such instruction exists, inaction occurs.
Negative test instruction. For each element `a` of A there is a negative test instruction $-\mathtt{a}$. When executed, the state is affected according to `a`, and in case `false` is returned, the remaining sequence of instructions is executed. If there are no remaining instructions, inaction occurs. In the case that `true` is returned, the next instruction is skipped and execution proceeds with the instruction following the skipped one. If no such instruction exists, inaction occurs.

Table 1.
PGA-axioms for instruction sequence congruence, where $n > 0$.

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

Forward jump instruction. For any natural number k, the instruction $\#k$ denotes a jump of length k and k is called the counter of this instruction. If $k = 0$, this jump is to the instruction itself and inaction occurs (one can say that $\#0$ *defines* divergence, which is a particular form of inaction). If $k = 1$, the instruction skips itself, and execution proceeds with the subsequent instruction if available, otherwise inaction occurs. If $k > 1$, the instruction $\#k$ skips itself and the subsequent $k-1$ instructions. If there are not that many instructions left in the remaining part of the program, inaction occurs.

Each primitive instruction is considered a PGA-program. We note that with *unfolding*, captured by the identity $X^\omega = X;X^\omega$ and explained in Section 2.2, PGA-programs refer to an execution mechanism that is left-sequential (from left to right) and single-pass (each instruction is executed at most once). This is closer to the behavioral semantics defined in [3] (and discussed in Section 2.4) than would be possible when more 'advanced' programming features as *goto*'s or *backward jumps* were included from the start, and hence may clarify why we distinguish PGA as most basic.

2.2 Instruction sequence congruence and first canonical forms

In PGA, different types of equality are discerned, the most simple of which is *instruction sequence congruence*, identifying programs that execute identical sequences of primitive instructions (program objects).[1] For PGA-programs not containing repetition, instruction sequence congruence boils down to the associativity of concatenation, and is thus axiomatized by the axiom

$$(X;Y);Z = X;(Y;Z) \qquad \text{(PGA1)}$$

We further leave out brackets in repeated concatenations. Define $X^1 = X$ and for $n > 0$, $X^{n+1} = X;X^n$. Then instruction sequence congruence for infinite program objects is axiomatized by the axioms (schemes) in Table 1. It is straightforward to derive from the axioms PGA2-4 the unfolding identity of repetition: $X^\omega = (X;X)^\omega = X;(X;X)^\omega = X;X^\omega$.[2] Whenever two PGA-

[1] Although a bit long, *primitive instruction sequence congruence* would have been a more adequate name.

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

Table 2.
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.

$$(X;Y);Z = X;(Y;Z) \quad \text{(PGA1)}$$

$$(X^n)^\omega = X^\omega \quad \text{(PGA2)}$$

$$X^\omega;Y = X^\omega \quad \text{(PGA3)}$$

$$(X;Y)^\omega = X;(Y;X)^\omega \quad \text{(PGA4)}$$

$$\#n{+}1;u_1;\ldots;u_n;\#0 = \#0;u_1;\ldots;u_n;\#0 \quad \text{(PGA5)}$$

$$\#n{+}1;u_1;\ldots;u_n;\#m = \#n{+}m{+}1;u_1;\ldots;u_n;\#m \quad \text{(PGA6)}$$

$$(\#k{+}n{+}1;u_1;\ldots;u_n)^\omega = (\#k;u_1;\ldots;u_n)^\omega \quad \text{(PGA7)}$$

$$X = u_1;\ldots;u_n;(v_1;\ldots;v_{m+1})^\omega \;\rightarrow\; \#n{+}m{+}k{+}2;X = \#n{+}k{+}1;X \quad \text{(PGA8)}$$

programs X and Y are instruction sequence congruent, this is written

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

The subscript $_{isc}$ will be dropped if no confusion can arise. Instruction sequence congruence is decidable (see [3]).

Each PGA-program can be rewritten into one of the following forms:

(1) Y not containing repetition, or
(2) $Y;Z^\omega$, with Y and Z not containing repetition.

Any program term in one of the two above forms is said to be in *first canonical form*. According to [3], for each PGA-program there is a PGA-program in first canonical form that is instruction sequence congruent. Moreover, in the case of $Y;Z^\omega$, there is a unique first canonical form if the number of instructions in Y and Z is minimized (using PGA1-4). First canonical forms are useful as input for further transformations (cf. [3]).

2.3 Structural congruence and second canonical forms

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 associated congruence is called *structural congruence* and is axiomatized in Table 2. 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 $+\mathtt{a};(\#0;+\mathtt{b};\#0;-\mathtt{c})^\omega$.

2.4 Thread algebra: behavioral semantics for PGA

In this section we shortly discuss thread algebra. Threads model the execution of PGA-programs. Finite threads are defined inductively as follows:

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 under 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 the reply `true` to $\mathtt{a}$, and as Q upon the 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.

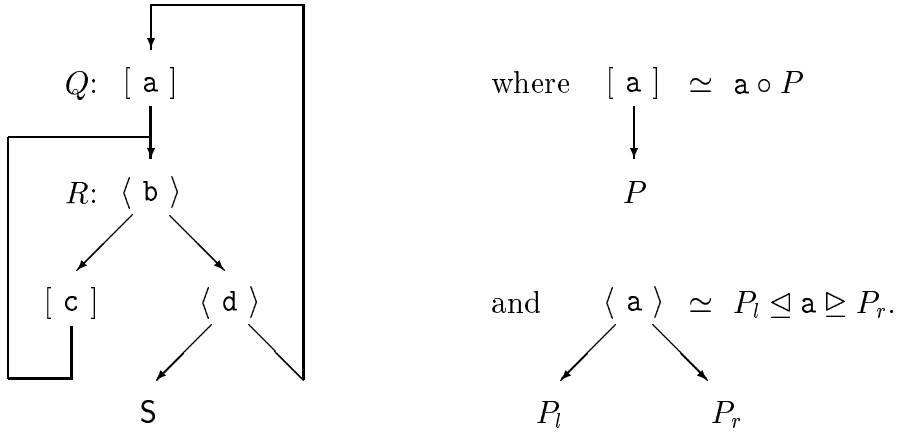
Upon its execution, a basic or test instruction yields the equally named action in a post conditional composition. Thread extraction on PGA, notation $|X|$ with X a PGA-program, is defined by the thirteen equations in Table 3. For a PGA-program in second canonical form, these equations either yield a 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 example, the regular thread Q specified by

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

is the thread defined by $|\mathtt{a};(+\mathtt{b};\#2;\#3;\mathtt{c};\#4;+\mathtt{d};!;\mathtt{a})^\omega|$. A picture of this tread:

Table 3.
Equations for thread extraction, where a ranges over the basic instructions, and u over the primitive instructions ($k \in \mathbb{N}$).

(1)	$\lvert !\rvert = \mathsf{S}$,	(2)	$\lvert !; X\rvert = \mathsf{S}$,
(3)	$\lvert \mathsf{a}\rvert = \mathsf{a} \circ \mathsf{D}$,	(4)	$\lvert \mathsf{a}; X\rvert = \mathsf{a} \circ \lvert X\rvert$,
(5)	$\lvert +\mathsf{a}\rvert = \mathsf{a} \circ \mathsf{D}$,	(6)	$\lvert +\mathsf{a}; X\rvert = \lvert X\rvert \trianglelefteq \mathsf{a} \trianglerighteq \lvert \#2; X\rvert$,
(7)	$\lvert -\mathsf{a}\rvert = \mathsf{a} \circ \mathsf{D}$,	(8)	$\lvert -\mathsf{a}; X\rvert = \lvert \#2; X\rvert \trianglelefteq \mathsf{a} \trianglerighteq \lvert X\rvert$,
(9)	$\lvert \#k\rvert = \mathsf{D}$,	(10)	$\lvert \#0; X\rvert = \mathsf{D}$,
		(11)	$\lvert \#1; X\rvert = \lvert X\rvert$,
		(12)	$\lvert \#k{+}2; u\rvert = \mathsf{D}$,
		(13)	$\lvert \#k{+}2; u; X\rvert = \lvert \#k{+}1; X\rvert$.



Some more examples:

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

It can be inferred that $\lvert u_1; \ldots; u_n\rvert = \lvert u_1; \ldots; u_n; (\#0)^\omega\rvert$ for u_i ranging over the primitive instructions. We shall use a^∞ as an informal notation for the thread defined by $\lvert \mathsf{a}^\omega\rvert$. For more information on thread algebra we refer to [2].

Table 4.
PGAr-axioms, where $k \in \mathbb{N}$ and u_i ranges over the primitive instructions.

$u_1;\ldots;u_{k+1};\backslash\backslash\#k{+}1 = (u_1;\ldots;u_{k+1})^\omega$	(PGAr)
$\backslash\backslash\#k;X = \backslash\backslash\#k$	(PGLA1)
$\backslash\backslash\#0 = \#0;\backslash\backslash\#1$	(PGLA2)

3 PGLA: a linear notation for PGA-programs

In this section we introduce PGLA in detail. PGLA is designed to represent each PGA-program in ASCII-based notation, and at the same time as a finite list of instructions: the repetition operator is not in PGLA and instead there are *repeat instructions* $\backslash\backslash\#k$ for all $k \in \mathbb{N}$. Furthermore, brackets are not used in PGLA. A sequence of primitive instructions ending with $\backslash\backslash\#k$ will repeat its last k instructions, excluding the repeat instruction itself. Instructions to the right of a repeat instruction are irrelevant and can be deleted. A special case is of course $\backslash\backslash\#0$: this instruction represents deadlock as it keeps repeating "nothing" and is further discussed below.

We first consider the combination of PGA and PGLA and present axioms, first canonical PGLA-forms and representation results for this particular program algebra. Then we distinguish PGLAcore as an interesting subset of PGLA: instruction sequence congruence, structural congruence and thread extraction can all be defined on PGLAcore without reference to PGA.

3.1 PGArep, PGA with repeat instructions

Let PGArep stand for the extension of PGA with repeat instructions, so both PGA and PGLA can be seen as subsets of PGArep. As a program algebra, PGArep can be defined by the axioms for PGA and the three axiom schemes in Table 4, where the axiom PGAr connects PGA and PGLA. Note that in a context with at least k preceding instructions, PGLA1 is derivable from PGAr and PGA3 (i.e., $X^\omega;Y = X^\omega$). Furthermore, PGLA2 simply postulates that $\backslash\backslash\#0$ equals $\#0;\backslash\backslash\#1$ and thus by PGAr the program object $(\#0)^\omega$. Therefore we call $\backslash\backslash\#0$ the *abort* or *deadlock* instruction. We write

`PGA4r`, respectively `PGA8r`

for the axioms in Table 4 combined with PGA1-4 (Table 1), respectively PGA1-8 (Table 2). Note that X, Y, Z in the axioms PGA1-4 range in PGArep over sequences that may contain both primitive and repeat instructions.

Table 5.
More axioms for PGLA where $k, l \in \mathbb{N}$, $m, n \in \mathbb{N} \setminus \{0\}$ and u_i, v_j are primitive instructions (note that PGLA5=PGA5 and PGLA6=PGA6, see Table 2).

$$u_1; \ldots; u_n; \backslash\backslash \#n = (u_1; \ldots; u_n)^m; \backslash\backslash \#mn \qquad \text{(PGLA3)}$$

$$u_1; \ldots; u_m; v_1; \ldots; v_n; \backslash\backslash \#m{+}n = u_1; \ldots; u_m; v_1; \ldots; v_n; u_1; \ldots; u_m; \backslash\backslash \#m{+}n \qquad \text{(PGLA4)}$$

$$\#k{+}1; u_1; \ldots; u_k; \#0 = \#0; u_1; \ldots; u_k; \#0 \qquad \text{(PGLA5)}$$

$$\#k{+}1; u_1; \ldots; u_k; \#m = \#k{+}m{+}1; u_1; \ldots; u_k; \#m \qquad \text{(PGLA6)}$$

$$\#k{+}1{+}l; u_1; \ldots; u_k; \backslash\backslash \#k{+}1 = \#l; u_1; \ldots; u_k; \backslash\backslash \#k{+}1 \qquad \text{(PGLA7)}$$

$$\#k{+}1{+}m{+}l; u_1; \ldots; u_k; v_1; \ldots; v_m; \backslash\backslash \#m = \#k{+}1{+}l; u_1; \ldots; u_k; v_1; \ldots; v_m; \backslash\backslash \#m \qquad \text{(PGLA8)}$$

In the combined setting of PGArep, the axiom PGLA1 implies that each PGLA-program (i.e, a sequence of PGLA-instructions) can be equated to one that contains *at most* one repeat instruction and by axiom PGLA2, the counter of this repeat instruction can be made larger than 0. The following proposition illustrates how the PGA4r axioms can be used to reason on PGLA-programs.

Proposition 1 *For all* $k \in \mathbb{N}$,

$$\texttt{PGA4r} \vdash \backslash\backslash \#0 = (\#0)^{k+1}; \backslash\backslash \#k{+}1.$$

Proof. For $k = 0$ this is PGLA2. For the remaining cases, use $(\#0)^\omega = ((\#0)^{k+1})^\omega$ (which follows from PGA2) and PGAr. □

It is an easy exercise to derive from PGA8r the axioms for PGLA displayed in Table 5. These axioms are related to PGA2,4-8, respectively (see Table 2). Note that the axioms PGLA2,3 immediately imply the identity proved in Proposition 1. Better than that, the axioms PGLA1-8 (PGLA1-4) are sufficiently strong to prove all identities between PGLA-programs that follow from PGA8r (PGA4r):

Theorem 2 *For* PGLA*-programs* p *and* q,

(1) $\texttt{PGA8r} \vdash p = q \iff$ PGLA1-8 $\vdash p = q$, *and*
(2) $\texttt{PGA4r} \vdash p = q \iff$ PGLA1-4 $\vdash p = q$.

Proof. It is not hard to see that with the axiom PGAr all "missing" PGA-

axioms are derivable from PGLA1-8. Conversely, with PGAr, all PGLA1-8-axioms can be derived from PGA8r. Similar for result (2). □

In the above we focused on PGLA-programs. In the remainder of this paper we show that PGLA with its axiom system PGLA1-8 can serve as a fully fledged variant of PGA. To this end we define first canonical PGLA-forms for PGArep. Then we show that these can be used to represent any PGArep-program in the string syntax PGLA.

Definition 3 *A PGArep-program is in* first canonical PGLA-form *if it is of the form*

$$u_1; \ldots; u_{k+1} \quad or \quad u_1; \ldots; u_k; \backslash\backslash \#n$$

with u_i primitive instructions and $k, n \in \mathbb{N}$. (Recall that $u_1; \ldots; u_0;$ represents the empty sequence.)

Theorem 4 *For each PGArep-program p there is a first canonical PGLA-form q such that* PGA4r $\vdash p = q$.

Proof. In the case that p is a finite sequence of PGArep-instructions, so without the repetition operator, we are done if there is no repeat instruction, or if all instructions following the leftmost repeat instruction are deleted (axiom PGLA1).

In the other case, derive from p with PGA1-4 a form $X; Y^\omega$ with X and Y containing no repetition operator (cf. first canonical forms in PGA). Then replace Y^ω by $Y; \backslash\backslash \#k$ with k the number of instructions in Y (axiom PGAr). Finally, delete all instructions (if any) following the leftmost repeat instruction in $X; Y; \backslash\backslash \#k$ (axiom PGLA1). □

3.2 PGLAcore, two congruences and thread extraction

Not all PGLA-programs have an intuitive meaning. For example, $\mathsf{a}; \backslash\backslash \#2$ or $\#7; +\mathsf{a}; \backslash\backslash \#5$ are first canonical forms that illustrate this situation. Let

PGLAcore

stand for the subset of first canonical PGLA-forms with the property that each repeat instruction $\backslash\backslash \#n$ is preceded by at least n primitive instructions.

The next result follows immediately from Theorems 2 and 4.

Corollary 5 *Two PGLA-core programs are instruction sequence congruent if and only if they can be equated with the axioms* PGLA1-4.

For example,

$$\begin{aligned} +\mathsf{a}; -\mathsf{b}; \#4; \backslash\backslash \#2 &=_{isc} +\mathsf{a}; -\mathsf{b}; \#4; -\mathsf{b}; \backslash\backslash \#2 \\ &=_{isc} +\mathsf{a}; -\mathsf{b}; \#4; -\mathsf{b}; \#4; \backslash\backslash \#4. \end{aligned}$$

In order to argue that PGLAcore is a fully fledged alternative for PGA we define second canonical PGLA-forms on PGLAcore.

Definition 6 *A* second canonical PGLA-form *is a first canonical PGLA-form in which no chained jumps occur and in the case of* $u_1; \ldots; u_m; \backslash\backslash \#n$, $m \geq n > 0$ *and all jumps to* $u_{m-n+1}, ..., u_m$ *are minimized (cf. axioms PGLA5-8 in Table 5).*

Examples, where the right-hand side is in second canonical PGLA-form:

$$\begin{aligned} \#1; \backslash\backslash \#1 &=_{sc} \#0; \backslash\backslash \#1, \\ +\mathsf{a}; \#7; +\mathsf{b}; \backslash\backslash \#0 &=_{sc} +\mathsf{a}; \#0; +\mathsf{b}; (\#0)^6; \backslash\backslash \#6. \end{aligned}$$

The next result again follows immediately from Theorems 2 and 4.

Corollary 7 *Two PGLAcore programs are structural congruent if and only if they can be equated with the axioms* PGLA1-8.

We state without proof that both first and second canonical PGLA-forms have a unique minimal representation in PGLAcore in terms of their number of primitive instructions; for the last example above this is $+\mathsf{a}; \#0; +\mathsf{b}; \#0; \backslash\backslash \#1$. Furthermore, both congruences are decidable.

Also, thread extraction can be defined on PGLAcore in a straightforward way on second canonical forms. We write

$$[\![X]\!]_{pgla}$$

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

$$[\![u_1; \ldots; u_k]\!]_{pgla} \stackrel{\text{def}}{=} [\![u_1; \ldots; u_k; \backslash\backslash \#0]\!]_{pgla}$$

and use axiom PGLA2 to obtain a second canonical PGLA-form. To define behavior extraction on second canonical forms $u_1; \ldots; u_{n+k}; \backslash\backslash \#n$ we use an

Table 6.
Equations for thread extraction on PGLAcore, where u_i ranges over the primitive instructions, $j, k \in \mathbb{N}$ and $n \in \mathbb{N} \setminus \{0\}$.

Let $X = u_1; \ldots; u_{n+k}; \backslash\backslash \#n$, then $[\![X]\!]_{pgla} = |1, X|$ with

$|j, X| = |j{-}n, X|$ if $j > n{+}k$,

$|j, X| = \mathsf{S}$ if $u_j = !$,

$|j, X| = a \circ |j{+}1, X|$ if $u_j = \mathtt{a}$,

$|j, X| = |j{+}1, X| \trianglelefteq a \trianglerighteq |j{+}2, X|$ if $u_j = +\mathtt{a}$,

$|j, X| = |j{+}2, X| \trianglelefteq a \trianglerighteq |j{+}1, X|$ if $u_j = -\mathtt{a}$,

$|j, X| = \mathsf{D}$ if $u_j = \#0$,

$|j, X| = |j{+}k{+}1, X|$ if $u_j = \#k{+}1$.

auxiliary function $|j, u_1; \ldots; u_{n+k}; \backslash\backslash \#n|$ where j makes reference to the position of instructions:

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

and $|j, u_1; \ldots; u_n; \backslash\backslash \#n|$ is defined by case distinction in Table 6.

We state without proof[3] the following result, implying that on PGLAcore $[\![X]\!]_{pgla}$ agrees with thread extraction on PGA-programs (see Section 2.4).

Theorem 8 *Let $k > 0$, $n > 0$, and let u_i range over the primitive instructions. Then $[\![u_1; \ldots; u_k; u_{k+1}; \ldots; u_{k+n}; \backslash\backslash \#n]\!]_{pgla} = |u_1; \ldots; u_k; (u_{k+1}; \ldots; u_{k+n})^\omega|$.*

Recall that $|u_1; \ldots; u_k; (\#0)^\omega| = |u_1; \ldots; u_k|$ can be inferred from the equations in Table 3 and also that $\mathtt{PGA4r} \vdash u_1; \ldots; u_k; u_{k+1}; \ldots; u_{k+n}; \backslash\backslash \#n = u_1; \ldots; u_k; (u_{k+1}; \ldots; u_{k+n})^\omega$.

4 PGLA as a programming language

In this section we discuss various semantics for full PGLA. According to Theorem 4 it suffices to consider programs in first canonical PGLA-form. Thus, it remains to consider the case in which a sequence of primitive instructions ends with a repeat instruction of which the counter is too large. First we discuss PGLA as it is defined in [3]. Then we provide three alternative definitions of PGLA, differing in how they define the meaning of programs ending with a repeat instruction with a too large counter, such as $\mathtt{a}; \backslash\backslash \#2$. Finally, we discuss an embedding from PGA into PGLA.

[3] But see equation (1) in Section 4.1.

Given a program notation PGLX, a *projection* is a function `pglx2pga` that maps each PGLX-program to PGA while preserving the intended meaning, i.e., the thread defined by that program. Conversely, an *embedding* or *co-projection* is a mapping `pga2pglx` in the reverse direction that preserves the intended meaning. Typically, for each PGA-program X it should hold that

$$|\texttt{pglx2pga}(\texttt{pga2pglx}(X))| = |X|.$$

According to the principles of program algebra, a *programming language* is a pair (PGLX, `pglx2pga`). From this point of view, (PGLA, `pgla2pga`) was introduced in [3] as the programming language that is as close as possible to PGA. The only modification of (PGLA, `pgla2pga`) proposed in this paper is to add the repeat instruction $\backslash\backslash\#0$, as this smoothes the set of PGLA instructions.

However, considering PGLA as the basis for a stand-alone programming language environment — a point of view that a tool-builder might propose or prefer — implies that we should extend our PGLAcore framework to full PGLA and provide all its analytical means independent of PGA. This implies that it remains to define thread extraction on full PGLA.

Both these points of view combine very well, and are in a technical sense interchangeable. Either thread extraction on PGLA conforms to a given projection, or the other way around. We prefer the first point of view (and explain this in Section 5). The projection `pgla2pga` is defined in [3] only on first canonical PGLA-forms (as they are called in this paper) as follows:

Definition 9 *Let u_i range over the primitive instructions, then*

$$\texttt{pgla2pga}(u_1;\ldots;u_{n+1}) = u_1;\ldots;u_{n+1},$$

$$\texttt{pgla2pga}(u_1;\ldots;u_k;\backslash\backslash\#n) = \begin{cases} u_1;\ldots;u_k;(\#0)^\omega & \text{if } k \geq n = 0,\\ u_1;\ldots;u_{k-n};(u_{k-n+1};\ldots;u_k)^\omega & \text{if } k \geq n > 0,\\ (u_1;\ldots;u_k;(\#0)^{n-k})^\omega & \text{if } n > k \geq 0. \end{cases}$$

The special case that $n = 0$ is not considered in [3]. If also $k = 0$ in the first clause, then $\texttt{pgla2pga}(\backslash\backslash\#0) = (\#0)^\omega$. In the last clause, $k = 0$ (thus $n > 0$) yields

$$\texttt{pgla2pga}(\backslash\backslash\#n) = ((\#0)^n)^\omega.$$

The latter PGA-program object is instruction sequence congruent with $(\#0)^\omega$. Observe that the projection `pgla2pga` fully dictates thread extraction on

PGLA: if $k < n$, then

$$[\![u_1; \ldots; u_k; \backslash\backslash \#n]\!]_{pgla} \stackrel{\text{def}}{=} [\![u_1; \ldots; u_k; (\#0)^{n-k}; \backslash\backslash \#n]\!]_{pgla}.$$

The correctness of thread extraction on PGLA as defined here and in Section 3.2 for PGLAcore can now be proved by showing that for all programs X in first canonical PGLA-form,

$$[\![X]\!]_{pgla} = |\texttt{pgla2pga}(X)|. \tag{1}$$

We do not give a proof of this equation, but we note that $[\![X]\!]_{pgla}$ is defined (designed) to make it valid. Furthermore, observe that equation (1) implies Theorem 8.

4.2 Three alternative definitions of PGLA

We consider three alternative projections for PGLA, each of which might be more attractive than `pgla2pga` from a practical point of view (we elaborate on this point in Section 5). All these projections conform to the single-pass character of program objects, and are of course identical on PGLAcore. Below we present these semantics by way of definitions for thread extraction, substantiating our claim that defining thread extraction also fixes the associated projection. Let u_i range over the primitive instructions and let $0 \leq k < n$.

(1) A first alternative is to fill a "repeat gap" with #1-instructions (i.e., with *skip* instructions):

$$[\![u_1; \ldots; u_k; \backslash\backslash \#n]\!]_{pglas} \stackrel{\text{def}}{=} [\![u_1; \ldots; u_k; (\#1)^{n-k}; \backslash\backslash \#n]\!]_{pgla},$$

and for all X in PGLAcore, $[\![X]\!]_{pglas} \stackrel{\text{def}}{=} [\![X]\!]_{pgla}$. We call this interpretation *pglas* (soft PGLA) and the associated projection `pglas2pga` is obtained by replacing the last clause in Definition 9 by

$$\texttt{pglas2pga}(u_1; \ldots; u_k; \backslash\backslash \#n) = (u_1; \ldots; u_k; (\#1)^{n-k})^{\omega}.$$

(2) Another idea is to fill a repeat gap with $\backslash\backslash\#0$ instructions (i.e., with *deadlock* instructions):

$$\begin{aligned}[\![u_1; \ldots; u_k; \backslash\backslash \#n]\!]_{pglah} &= [\![u_1; \ldots; u_k; (\backslash\backslash \#0)^{n-k}; \backslash\backslash \#n]\!]_{pgla} = \\ &= [\![u_1; \ldots; u_k; \backslash\backslash \#0]\!]_{pgla},\end{aligned}$$

and for all X in PGLAcore, $[\![X]\!]_{pglah} \stackrel{\text{def}}{=} [\![X]\!]_{pgla}$. We call this interpretation *pglah* (hard PGLA) and the associated projection `pglas2pga` is obtained by replacing the last clause in Definition 9 by

$$\texttt{pglah2pga}(u_1; \ldots; u_k; \backslash\backslash \#n) = u_1; \ldots; u_k; (\#0)^{\omega}.$$

(3) Yet another idea is to truncate the counter of the repeat instruction:

$$[\![u_1;\ldots;u_k;\backslash\backslash\#n]\!]_{pglat} = [\![u_1;\ldots;u_k;\backslash\backslash\#k]\!]_{pgla},$$

and for all X in PGLAcore, $[\![X]\!]_{pglat} \stackrel{\text{def}}{=} [\![X]\!]_{pgla}$. We call this interpretation *pglat* (truncated PGLA) and the associated projection `pglas2pga` is obtained by replacing the last clause in Definition 9 by

$$\texttt{pglat2pga}(u_1;\ldots;u_k;\backslash\backslash\#n) = (u_1;\ldots;u_k)^\omega.$$

Note that the soft interpretation of PGLA, i.e., `pglas2pga` or $[\![\_]\!]_{pglas}$, can be directly associated with the PGA thread extraction equation $|\#1;X| = |X|$ (equation (11) in Table 3).

4.3 A single embedding into PGLA

The embedding `pga2pgla` from PGA to PGLA is defined in [3] on PGA-programs in first canonical form and yields programs that are in PGLAcore: let $k > 0$, $n > 0$, and let u_i range over the primitive instructions, then

$$\texttt{pga2pgla}(u_1;\ldots;u_k) = u_1;\ldots;u_k,$$
$$\texttt{pga2pgla}(u_1;\ldots;u_k;(u_{k+1};\ldots;u_{k+n})^\omega) = u_1;\ldots;u_k;u_{k+1};\ldots;u_{k+n}\backslash\backslash\#n.$$

Clearly, for each PGA-program X in first canonical form and for all $\phi \in \{\texttt{pgla2pga}, \texttt{pglas2pga}, \texttt{pglah2pga}, \texttt{pglat2pga}\}$,

$$\phi(\texttt{pga2pgla}(X)) = X,$$

and thus $|\phi(\texttt{pga2pgla}(X))| = |X|$.

5 Conclusions

We provided an algebraic theory of the ASCII representation PGLA of program algebra PGA. In PGLA the repetition operator is replaced by a family of repeat instructions $\backslash\backslash\#k$, where the counter k ranges over the naturals $\mathbb{N}$ (including 0). A repeat instruction $\backslash\backslash\#k$ prescribes to repeat the last k instructions. In particular, $\backslash\backslash\#0$ prescribes deadlock.

Then, PGA and PGLA were considered in a combined setting, involving both the repeat instructions of PGLA and the repetition operator of PGA. The

resulting program algebra PGArep is used to establish axioms and representation results for PGLA. As of yet, we see no other application for the program algebra PGArep.

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

(1) A program object is a (possibly infinite) sequence of primitive instructions that is single-pass, and
(2) A programming language is a pair (L, ϕ) where L is a a set of expressions (the programs) and ϕ is a projection to PGA.

Our main motivation to undertake this research is the understanding that in the setting of PGA's hierarchy of programming languages [3], (PGLA, ϕ) is the single one that admits axiomatizations of instruction sequence congruence, structural congruence and thread extraction, and that at the same time can serve as the basis for a programming environment for program algebra [4]. This holds for

$$\phi \in \{\mathtt{pgla2pga}, \mathtt{pglas2pga}, \mathtt{pglah2pga}, \mathtt{pglat2pga}\},$$

and of course for many more definitions of projections to PGA. We proposed the latter three projections because they fit the philosophy sketched above and because a disadvantage of the (original) projection `pgla2pga` is that it does not combine in an elegant way with jumps, as witnessed by the following examples where we abbreviate $|\mathtt{pgla2pga}(X)|$ by $|X|_{pgla}$ (as is done in [3]):

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

In general, $|\mathsf{a}; \#k; \backslash\backslash \#3|_{pgla} = |(\mathsf{a}; \#k; \#0)^\omega| = \mathsf{a}^\infty$ if $k \bmod 3 = 2$, and $\mathsf{a} \circ \mathsf{D}$ otherwise. So in the original approach, D *either* arises from the added #0 instruction *or* from the interplay with $\backslash\backslash \#3$ and the original jump instruction; this we now consider very arbitrary.

According to the proposed "soft projection" `pglas2pga`, $[\![\mathsf{a}; \#k; \backslash\backslash \#3]\!]_{pglas} = \mathsf{a} \circ \mathsf{D}$ if $k \bmod 3 = 0$ and a^∞ otherwise, so inaction only arises if $k = 0$ or from the interplay between $\#k$ and $\backslash\backslash \#3$. We also proposed a "hard interpretation" `pglah2pga` that yields deadlock upon a repeat instruction that prescribes to repeat too many instructions. For instance, $[\![\mathsf{a}; \#k; \backslash\backslash \#3]\!]_{pglah} = \mathsf{a} \circ \mathsf{D}$ for all $k \in \mathbb{N}$. Finally, we proposed `pglat2pga`, a projection that truncates a too large repeat instruction: $[\![\mathsf{a}; \#k; \backslash\backslash \#3]\!]_{pglat} = [\![\mathsf{a}; \#k; \backslash\backslash \#2]\!]_{pglat} = \mathsf{a} \circ \mathsf{D}$ if k is even, and a^∞ otherwise.

Irrespective of which projection is given definitional status, we argued that PGLA and its axioms form a fully fledged alternative for PGA. Our conclusion is that we consider PGA as the most basic theory (instead of PGLA), if

only because there is no canonical interpretation of PGLA outside PGLAcore. Therefore all projection semantics map to PGA. In particular, the projection function `pgla2pga` maintains its definitional status: it embodies the original definition of PGLA as a programming language. However, each of the three alternative projections discussed in this paper has its own methodological advantages, and each of (PGLA, ϕ) for ϕ one of these projections may be an attractive candidate for further establishing PGLA as the basis of a programming environment for program algebra.

A last remark on related work: PGA can be viewed as a theory of instruction streams with PGLA as one of many representations of it. 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 stream* 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. Electronic report PRG0502, Programming Research Group, University of Amsterdam, June 2005. Available at `www.science.uva.nl/research/prog/publications.html`.

[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] B. Diertens. *PGA - ProGram Algebra.* Web site containing a Toolset for PGA: `www.science.uva.nl/research/prog/projects/pga/`, Last modified: September 20, 2005.

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0601] J.A. Bergstra and A. Ponse, *Interface Groups for Analytic Execution Architectures*, Programming Research Group - University of Amsterdam, 2006.

[PRG0505] B. Diertens, *Software (Re-)Engineering with PSF*, Programming Research Group - University of Amsterdam, 2005.

[PRG0504] P.H. Rodenburg, *Piecewise Initial Algebra Semantics*, Programming Research Group - University of Amsterdam, 2005.

[PRG0503] T.D. Vu, *Metric Denotational Semantics for BPPA*, Programming Research Group - University of Amsterdam, 2005.

[PRG0502] J.A. Bergstra, I. Bethke, and A. Ponse, *Decision Problems for Pushdown Threads*, Programming Research Group - University of Amsterdam, 2005.

[PRG0501] J.A. Bergstra and A. Ponse, *A Bypass of Cohen's Impossibility Result*, Programming Research Group - University of Amsterdam, 2005.

[PRG0405] J.A. Bergstra and I. Bethke, *An Upper Bound for the Equational Specification of Finite State Services*, Programming Research Group - University of Amsterdam, 2004.

[PRG0404] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Strategic Interleaving*, Programming Research Group - University of Amsterdam, 2004.

[PRG0403] B. Diertens, *A Compiler-projection from PGLEc.MSPio to Parrot*, Programming Research Group - University of Amsterdam, 2004.

[PRG0402] J.A. Bergstra and I. Bethke, *Linear Projective Program Syntax*, Programming Research Group - University of Amsterdam, 2004.

[PRG0401] B. Diertens, *Molecular Scripting Primitives*, Programming Research Group - University of Amsterdam, 2004.

[PRG0302] B. Diertens, *A Toolset for PGA*, Programming Research Group - University of Amsterdam, 2003.

[PRG0301] J.A. Bergstra and P. Walters, *Projection Semantics for Multi-File Programs*, Programming Research Group - University of Amsterdam, 2003.

[PRG0201] I. Bethke and P. Walters, *Molecule-oriented Java Programs for Cyclic Sequences*, Programming Research Group - University of Amsterdam, 2002.

The above reports and more are available through the website: www.science.uva.nl/research/prog/

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

www.science.uva.nl/research/prog/