An Introduction to Program and Thread Algebra

Alban Ponse and Mark B. van der Zwaag

Programming Research Group, Informatics Institute, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
alban@science.uva.nl, mbz@science.uva.nl

Abstract. We provide an introduction to Program Algebra (PGA, an algebraic approach to the modeling of sequential programming) and to Thread Algebra (TA). PGA is used as a basis for several low- and higher-level programming languages. As an example we consider a simple language with *goto*'s. Threads in TA model the execution of programs. Threads may be composed with services which model (part of) the execution environment, such as a stack. Finally, we discuss briefly the expressiveness of PGA and allude to current work on multithreading and security hazard risk assessment.

Keywords: PGA, Program Algebra, Thread Algebra.

1 Introduction

In this paper we report on a recent line of programming research conducted at the University of Amsterdam. This research comprises *program algebra* and *thread algebra*. A first major publication about this project is [7] (2002).

Program algebra (PGA, for ProGram Algebra) provides a rigid framework for the understanding of imperative sequential programming. Starting point is the perception of a *program object* as a possibly infinite sequence of primitive instructions. PGA programs are composed from primitive instructions and two operators: sequential composition and iteration. Based on this, a family of programming languages is built, containing well-known constructs such as labels and goto's, conditionals and while-loops, etc. These languages are defined with a projection to PGA which defines the program object described by a program expression.

Execution of a program object is single-pass: the instructions are visited in order and are dropped after having been executed. Execution of a basic instruction or test is interpreted as a request to the execution environment: the environment processes the request and replies with a Boolean value. This has lead to the modeling of the behavior of program objects as threads, i.e., as elements of Thread Algebra (TA). The primary operation of TA is postconditional composition:

$$P \trianglelefteq a \trianglerighteq Q$$

stands for the execution of action a which is followed by execution of P if `true` is returned and by execution of Q if `false` is returned. Threads can be composed with services which model (part of) the environment.

A. Beckmann et al. (Eds.): CiE 2006, LNCS 3988, pp. 445–458, 2006.

In Section 2 we present PGA, and in Section 3 we overview thread algebra and the interpretation of programs as threads. Then, in Section 4 we go into the expressiveness of PGA. Finally, in Section 5 we allude briefly to current work on multithreading and security hazard risk assessment. For further discussion on the *why*'s and *why not*'s of PGA, see [2], and of TA, see [3].

2 Program Algebra

Program Algebra (PGA) is based on a parameter set A. The *primitive instructions* of PGA are the following:

Basic instruction. All elements of A, written, typically, as $a, b, \ldots$ are *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 associated behavior may modify a state.

Termination instruction. The termination instruction ! yields termination of the program. It does not modify a state, and it does not return a boolean value.

Test instruction. For each element a of A there is a *positive test* instruction $+a$ and a *negative test* instruction $-a$. When a positive test is executed, the state is affected according to a, and in case `true` is returned, the remaining sequence of actions is performed. 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. Execution of a negative test is the same, except that the roles of `true` and `false` are interchanged.

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.

PGA *program terms* are defined inductively as follows:

1. Primitive instructions are program terms.
2. If X and Y are program terms, then $X; Y$, called the *concatenation* of X and Y, is a program term.
3. If X is a program term, then X^ω (the *repetition* of X) is a program term.

2.1 Instruction Sequence Congruence and Canonical Forms

On PGA, different types of equality can be discerned, the most simple of which is *instruction sequence congruence*, identifying programs that execute identical sequences of instructions. Such a sequence is further called a *program object*. For programs not containing repetition, instruction sequence congruence boils down to the associativity of concatenation, and is axiomatized by

$$(X;Y);Z = X;(Y;Z). \tag{PGA1}$$

As a consequence, brackets are not meaningful in repeated concatenations and will be left out.

Now let $X^1 = X$ and $X^{n+1} = X;X^n$ for $n > 0$. Then instruction sequence congruence for infinite program objects is further axiomatized by the following axioms (schemes):

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

It is straightforward to derive from PGA2–4 the unfolding identity of repetition:

$$X^\omega = (X;X)^\omega = X;(X;X)^\omega = X;X^\omega.$$

Instruction sequence congruence is decidable [7].

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

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

Any program in one of the two above forms is said to be in *first canonical form*. For each PGA program there is a program in first canonical form that is instruction sequence congruent [7]. Canonical forms are useful as input for further transformations.

2.2 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 by PGA1–4 presented above, plus the following axiom schemes, where the u_i and v_i range over primitive instructions:

$$\#n{+}1;u_1;\ldots;u_n;\#0 = \#0;u_1;\ldots;u_n;\#0, \tag{PGA5}$$
$$\#n{+}1;u_1;\ldots;u_n;\#m = \#n{+}m{+}1;u_1;\ldots;u_n;\#m, \tag{PGA6}$$
$$(\#k{+}n{+}1;u_1;\ldots;u_n)^\omega = (\#k;u_1;\ldots;u_n)^\omega, \tag{PGA7}$$

and

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

Two examples, of which the right-hand sides are in second canonical form:

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

Second canonical forms are not unique. However, if in $X; Y^\omega$ the number of instructions in X and Y is minimized, they are. In the first example above, the right-hand side is the unique minimal second canonical form; for the second example it is ${+}a; (\#0; {+}b; \#0; {-}c)^\omega$.

2.3 PGA-Based Languages

On the basis of PGA, a family of programming languages has been developed [7]. The programming constructs in these languages include backward jumps, absolute jumps, labels and goto's, conditionals and while loops, etc. All of these languages are given a projection semantics, that is, they come with a translation to PGA which determines their semantics (together with the semantics of PGA, see Section 3.1). Vice versa, PGA can be embedded in each of these languages, which shows that they share the same expressiveness. As an example we present the language PGLDg and its projection semantics.[1]

PGLDg is a program notation with label and goto instructions as primitives instead of jumps. Repetition is not available, so a PGLDg program is just a finite sequence of instructions. In PGLDg termination takes place when the last instruction has been executed, when a goto to a non-existing label is made, or when a termination instruction ! is executed.

A label in PGLDg is just a natural number. Label and goto-instructions are defined as follows:

Label instruction. The instruction $£k$, for k a natural number, represents a visible label. As an action it is a skip in the sense that it will not have any effect on a state space.

Goto instruction. For each natural number k the instruction $\#\#£k$ represents a jump to the (beginning of) the first (i.e. the left-most) instruction $£k$ in the program. If no such instruction can be found termination of the program execution will occur.

An example of a PGLDg program is: $£0; {-}a; \#\#£1; \#\#£0; £1$. In this program a is repeated until it yields reply value `false`. That is also the functionality of the simpler program $£0; {+}a; \#\#£0$.

[1] The languages presented in [7] are called PGLA, PGLB, PGLC, etc.

A projection from PGLDg to PGA works as follows:

$$\mathtt{pgldg2pga}(u_1;\ldots;u_k) = (\psi_1(u_1);\ldots;\psi_k(u_k);!;!)^\omega,$$

where the u_i range over primitive instructions, the two added termination instructions serve the case that u_k is a test-instruction, and the auxiliary functions ψ_j are defined as follows:

$$\psi_j(\#\#\pounds n) = \begin{cases} ! & \text{if } \mathtt{target}(n) = 0, \\ \#\mathtt{target}(n){-}j & \text{if } \mathtt{target}(n) \geq j, \\ \#k{+}2{-}j{+}\mathtt{target}(n) & \text{otherwise,} \end{cases}$$
$$\psi_j(\pounds n) = \#1,$$
$$\psi_j(u) = u \quad \text{otherwise.}$$

The auxiliary function $\mathtt{target}(k)$ produces for k the smallest number j such that the j-th instruction of the program is of the form $\pounds k$, if such a number exists and 0 otherwise. Projecting the two example programs above yields

$$\mathtt{pgldg2pga}(\pounds 0; -a; \#\#\pounds 1; \#\#\pounds 0; \pounds 1) = (\#1; -a; \#2; \#4; \#1; !; !)^\omega,$$
$$\mathtt{pgldg2pga}(\pounds 0; +a; \#\#\pounds 0) = (\#1; +a; \#3; !; !)^\omega.$$

The projection `pgldg2pga` results from the composition of a number of projections defined in [7] and a tiny bit of smart reasoning.

3 Basic Thread Algebra

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

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

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

The operational intuition is that each action represents a command which is to be processed by the execution environment of the thread. The processing of a command may involve a change of state of this environment.[2] At completion of the processing of the command, the environment produces a reply value `true` or

[2] For the definition of threads we completely abstract from the environment. In Section 3.2 we define services which model (part of) the environment, and thread-service composition.

`false`. The thread $P \trianglelefteq a \trianglerighteq Q$ proceeds as P if the processing of a yields `true`, and it proceeds as Q if the processing of a yields `false`.

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

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

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

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

Following the metric theory of [1] in the form developed as the basis of the introduction of processes in [6], BTA has a completion BTA^∞ which comprises also the infinite threads. Standard properties of the completion technique yield that we may take BTA^∞ as the cpo consisting of all so-called *projective* sequences:[3]

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

For a detailed account of this construction see [4] or [19].

Overloading notation, we now define the constants and operators of BTA on BTA^∞:

1. $\mathsf{D} = (\mathsf{D}, \mathsf{D}, \ldots)$ and $\mathsf{S} = (\mathsf{D}, \mathsf{S}, \mathsf{S}, \ldots)$;
2. $(P_n)_{n\in\mathbb{N}} \trianglelefteq a \trianglerighteq (Q_n)_{n\in\mathbb{N}} = (R_n)_{n\in\mathbb{N}}$ with $R_0 = \mathsf{D}$ and $R_{n+1} = P_n \trianglelefteq a \trianglerighteq Q_n$;
3. $\pi_n((P_m)_{m\in\mathbb{N}}) = (P_0, \ldots, P_{n-1}, P_n, P_n, P_n \ldots)$.

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

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

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

A residual thread may be reached (depending on the execution environment) by performing zero or more actions. A thread P is *regular* if $\mathit{Res}(P)$ is finite.

A finite linear recursive specification over BTA^∞ is a set of equations

$$x_i = t_i$$

for $i \in I$ with I some finite index set, variables x_i, and all t_i terms of the form S, D, or $x_j \trianglelefteq a \trianglerighteq x_k$ with $j, k \in I$. Finite linear recursive specifications

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

represent continuous operators having unique fixed points [19]. In reasoning with finite linear specifications, we shall identify variables and their fixed points. For example, we say that P *is* the thread defined by $P = a \circ P$ instead of stating that P equals the fixed point for x in the finite linear specification $x = a \circ x$.

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

The proof is easy:

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

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

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

Example 1. The regular threads $a^n \circ \mathsf{D}$, $a^n \circ \mathsf{S}$, and $a^\infty = a \circ a \circ \cdots$ are the respective fixed points for x_1 in the specifications

1. $x_1 = a \circ x_2, \ldots, x_n = a \circ x_{n+1}, x_{n+1} = \mathsf{D}$,
2. $x_1 = a \circ x_2, \ldots, x_n = a \circ x_{n+1}, x_{n+1} = \mathsf{S}$,
3. $x_1 = a \circ x_1$.

3.1 Extraction of Threads from Programs

The *thread extraction operator* $|\_|$ assigns a thread to a program. This thread models the behavior of the program. Note that the resulting behavioral equivalence is not a congruence: from $|X|$ equals $|Y|$, one cannot infer that, e.g., $|X; Z|$ equals $|Y; Z|$.

Thread extraction on PGA, notation $|X|$ with X a PGA program, is defined by the following thirteen equations (where a ranges over the basic instructions, and u over the primitive instructions):[4]

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

[4] We generally consider PGA programs modulo instruction sequence congruence, i.e., as program objects, so $|X^\omega| = |X; X^\omega|$.

$$|+a;X| = |X| \trianglelefteq a \trianglerighteq |\#2;X| \qquad |\#1;X| = |X|$$
$$|-a;X| = |\#2;X| \trianglelefteq a \trianglerighteq |X| \qquad |\#k+2;u| = \mathsf{D}$$
$$|\#k+2;u;X| = |\#k+1;X|$$

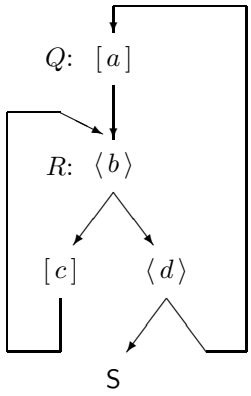
Observe that we interpret basic instructions as actions.

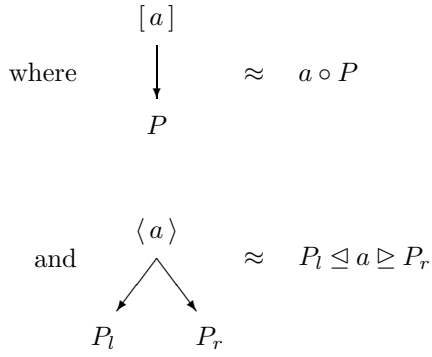
For PGA programs in second canonical form, these equations yield either finite threads, or regular threads (in the case that a non-empty loop occurs, which can be captured by a system of recursive equations).

Example 2. Computation of $Q = |a;(+b;\#2;\#3;c;\#4;+d;!;a)^\omega|$ yields the following regular thread:[5]

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

This thread can be depicted as follows:





Example 3. Observe that thread extraction following the equations does not terminate for the program term

$$+a;\#2;(+b;\#2;-c;\#2)^\omega.$$

However, thread extraction on its second canonical form $+a;(\#0;+b;\#0;-c)^\omega$ yields the thread P defined by

$$P = \mathsf{D} \trianglelefteq a \trianglerighteq Q, \quad Q = \mathsf{D} \trianglelefteq b \trianglerighteq (Q \trianglelefteq c \trianglerighteq \mathsf{D}).$$

Any PGA program defines a regular thread, and conversely, every regular thread can be defined in PGA, see Section 4. Behavioral equivalence is decidable for PGA programs [7].

[5] Note that a *linear* recursive specification of Q requires (at least) five equations.

3.2 Services

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

Example 4. A *stack* can be defined as a service with co-actions $push{:}i$, $topeq{:}i$, and pop, for $i = 1, \ldots, n$ for some n, where $push{:}i$ pushes i onto the stack and yields `true`, the action $topeq{:}i$ tests whether i is on top of the stack, and pop pops the stack with reply `true` if it is non-empty, and it yields `false` otherwise.

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

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

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

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

Example 5. We define a thread using a stack as defined in Example 4. We only push the value 1 (so the stack behaves as a counter), and write $S(n)$ for a stack holding n times the value 1. By the defining equations for the use operator it follows that for any thread P,

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

Now consider the regular thread Q defined by

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

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

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

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

3.3 Classes of Threads

We shall see in Section 4 that *finite* threads (the elements of BTA) correspond exactly to the threads that can be expressed in PGA without iteration, and that *regular* threads (threads definable by finite linear specifications) are exactly those that can be expressed in PGA. Equality is decidable for regular threads [7]. We mention two classes of non-regular threads: *pushdown* threads and *computable* threads. In both cases the non-regularity can be obtained by composing regular threads with certain services.

We call a regular thread that uses a stack as described in Example 4 a *pushdown* thread. In Example 5 we have seen that a pushdown thread may be non-regular. Equality is decidable for pushdown threads, but inclusion (the ordering $\sqsubseteq$ defined in Section 3) is not [5].[6]

Finally, a thread is *computable* if it can be represented by an identifier P_0 and two computable functions f and g as follows ($k \in \mathbb{N}$):

$$P_k = \begin{cases} \mathsf{D} & \text{if } g(k) = 0, \\ \mathsf{S} & \text{if } g(k) = 1, \\ P_{\langle k+f(k),1\rangle} \trianglelefteq a_{g(k)} \trianglerighteq P_{\langle k+f(k),2\rangle} & \text{if } g(k) > 1, \end{cases}$$

where $\langle \_, \_ \rangle$ is a bijective, computable pairing function.

Obviously computable threads can, in general, not be expressed by PGA programs. However, infinite sequences of primitive PGA instructions are universal: for every computable thread P there is such an infinite sequence with P as its behavior [12]. Computable threads can be obtained by composition of regular threads with a Turing machine tape as a service [12].

[6] In [5], the undecidability of inclusion for pushdown threads is proved using a reduction of the halting problem for Minsky machines. In this construction one of the counters is "weakly simulated". This method was found by Jančar and recorded first in 1994 [15], where it was used to prove various undecidability results for Petri nets. In 1999, Jančar et al. [16] used the same idea to prove the undecidability of simulation preorder for processes generated by one-counter machines, and this is most comparable to the approach in [5]. However, in the case of pushdown threads the inclusion relation itself is a little more complex than in process simulation or language theory because $\mathsf{D} \sqsubseteq P$ for any thread P. Moreover, threads have restricted branching, and therefore transforming a regular (control) thread into one that simulates one of the counters of a Minsky machine is more complex than in the related approaches referred to above. See [5] for a further discussion.

4 On the Expressiveness of PGA

We present some expressiveness results for PGA.

Proposition 1. PGA *without repetition characterizes* BTA*, that is, each program without repetition defines a finite thread, and all finite threads can be expressed.*

Proof. It follows immediately from the equations for thread extraction that PGA programs without repetition define finite threads. Vice versa, we give a mapping $[\_]$ from BTA to PGA:

$$\begin{aligned} [\mathsf{D}] &= \#0, \\ [\mathsf{S}] &= !, \\ [P \trianglelefteq a \trianglerighteq Q] &= +a; \#2; \#(n[P] + 1); [P]; [Q], \end{aligned}$$

where $n[P]$ is the number of instructions in $[P]$. □

Proposition 2. PGA *characterizes the regular threads.*

Proof. It follows immediately from the equations for thread extraction that PGA programs define regular threads. Vice versa, any regular thread can be given by a finite linear recursive specification by Theorem 1. Assume a specification with variables $x_1, \ldots, x_n$. We obtain the PGLDg program $[x_1]; [x_2]; \ldots; [x_n]$ for x_1 by the mapping $[\_]$ which is defined as follows:

$$[x_i] = \begin{cases} \pounds i; +a_i; \#\#\pounds j; \#\#\pounds k & \text{if } x_i = x_j \trianglelefteq a \trianglerighteq x_k, \\ \pounds i; ! & \text{if } x_i = \mathsf{S}, \\ \pounds i; \#\#\pounds i & \text{if } x_i = \mathsf{D}. \end{cases}$$

The resulting PGLDg expression for the thread is mapped to a PGA program by `pgldg2pga` (see Section 2.3). □

Corollary 1. *Basic instructions and negative tests instructions do not enhance the expressive power of* PGA*.*

Proof. Take any PGA program. By thread extraction and the method sketched in the proof of Proposition 2 we find an equivalent PGA program without occurrences of basic instructions or negative tests. □

This corollary establishes that PGA's set of primitive instructions is not minimal with respect to its expressiveness. The next proposition shows that that unbounded jump counters are necessary for the expressiveness of PGA.

Proposition 3. *For $n \in \mathbb{N}$, let* PGA_n *denote the set of* PGA *expressions not containing jump counters strictly greater than n. For every $n \geq 2$, there is a* PGA *behavior that cannot be expressed in* PGA_n*.*

Proof. Take $n \geq 2$ and a basic instruction a. Consider the PGA program X defined by

$$\begin{aligned} X &= Y_1; \ldots; Y_{n+1}; !; (Z_1; \ldots; Z_{n+1})^\omega, \\ Y_i &= +a; \#k_i, \\ Z_i &= a^i; +a; !; \#l_i, \end{aligned}$$

where $k_i = 2n + 1 + i(i+1)/2$, and $l_i = (n+4)(n+5)/2 - (i+8)$.

Note that $\#k_i$ jumps from Y_i to the first instruction of subexpression Z_i, and that $\#l_i$ jumps from Z_i also to the first instruction of Z_i. For example, if $n = 2$, then X equals

$$+a; \#6; +a; \#8; +a; \#11; !; (a; +a; !; \#12; a^2; +a; !; \#11; a^3; +a; !; \#10)^\omega.$$

We oberve that X has these properties:

1. After the execution of $Y_1; \ldots; Y_{n+1}; !$, any of the Z_i can be the first part of the iteration that is executed.
2. Execution of the iterative part is completely determined by one of the Z_i and distguished from the execution determined by another Z_j.

We show that $|X|$ cannot be expressed in PGA_n. First, for $i \leq n+1$, define threads

$$\begin{aligned} Q_i &= a^i \circ (\mathsf{S} \trianglelefteq a \trianglerighteq Q_i), \\ P_i &= Q_i \trianglelefteq a \trianglerighteq P_{i+1}, \\ P_{n+2} &= \mathsf{S}. \end{aligned}$$

We find that

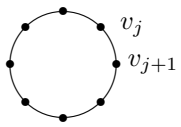
$$\begin{aligned} P_i &= |Y_i; \ldots; Y_{n+1}; !; (Z_1; \ldots; Z_{n+1})^\omega|, \\ Q_i &= |Z_i; \ldots; Z_{n+1}; (Z_1; \ldots; Z_{n+1})^\omega|, \end{aligned}$$

and in particular that $|X| = P_1$.

Now suppose that $|X|$ can be expressed in PGA_n (we shall derive a contradiction). Then there must be a first canonical form

$$u_1; \ldots; u_m; (v_1; \ldots; v_k)^\omega$$

in PGA_n with this behavior. We can picture the iteration of $v_1; \ldots; v_k$ as a circle of k instructions:



Note that each of the v_j serves at most one Q_i.

By the restriction on the values of jump counters, we know that between any two subsequent Q_i-instructions on the circle, there are at most $n-1$ other instructions. Hence, for any i there are at least $\lceil k/n \rceil$ Q_i-instructions on the circle, so in total the circle contains at least $(n+1) \cdot \lceil k/n \rceil$ instructions. Since

$$(n+1) \cdot \lceil k/n \rceil \geq (n+1) \cdot (k/n) > k,$$

this contradicts the fact that the circle contains k instructions. □

5 Current Work

Current work includes research on multithreading. In thread algebra, a multithread consists of a number of basic threads together with an interleaving operator which executes the threads in parallel based on a certain interleaving strategy [8, 9]. This theory is applied in the setting of processor architectures, in particular of so-called *micro-grids* executing micro-threads [17]. For the mathematical modeling of processor architectures, so-called Maurer computers are used [18, 10, 11].

Another branch of research is about the forecasting of certain actions, given the program to be executed. The main purpose of this research is a formal modeling of security hazard risk assessment (or virus detection) [14, 13]. For pushdown threads this type of forecasting is decidable: rename the action(s) to be forecasted and decide whether the thread thus obtained equals the original one. Forecasting becomes much more complicated if a program may contain test instructions that yield a reply according to the result of this type of forecasting. For example, assume that the action to be forecasted is named *risk* and that there is a test action *test* that yields true if its true-branch does not execute *risk*, and false otherwise. Then a current *test* action in the code to be inspected may yield true because a future one will yield false. The reply to these *test* actions can be modeled with a use-application. For regular threads, the associated service has a decidable reply function [13], while for pushdown threads this is still an open question.

References

1. J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1-2):70–120,1982.
2. J.A. Bergstra. `www.science.uva.nl/~janb/ta/`, Februari 2006.
3. J.A. Bergstra. `www.science.uva.nl/~janb/pga/`, Februari 2006.
4. J.A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J.C.M. Baeten, J.K. Lenstra, J. Parrow, G.J. Woeginger, eds., *Proceedings of ICALP 2003*, Springer-Verlag, LNCS 2719:1–21, 2003.
5. J.A. Bergstra, I. Bethke, and A. Ponse. Decision problems for pushdown threads. Report PRG0502, Programming Research Group, University of Amsterdam, June 2005. Available at `www.science.uva.nl/research/prog/publications.html`.

6. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984
7. J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
8. J.A. Bergstra and C.A. Middelburg. A Thread Algebra with Multi-Level Strategic Interleaving. Report 200441, Computing Science Department, Eindhoven University of Technology, 2004.
9. J.A. Bergstra and C.A. Middelburg. Thread Algebra for Strategic Interleaving. Report PRG0404, Programming Research Group, University of Amsterdam, 2004. Available at `www.science.uva.nl/research/prog/publications.html`.
10. J.A. Bergstra and C.A. Middelburg. Simulating Turing Machines on Maurer Machines. Report 200528, Computing Science Department, Eindhoven University of Technology, 2005.
11. J.A. Bergstra and C.A. Middelburg. Maurer Computers with Single-Thread Control. Report 200517, Computing Science Department, Eindhoven University of Technology, 2005.
12. J.A. Bergstra and A. Ponse. Execution architectures for program algebra. To appear in *Journal of Applied Logic*, 2006. An earlier version appeared as Logic Group Preprint Series 230, Department of Philosophy, Utrecht University, 2004.
13. J.A. Bergstra and A. Ponse. A bypass of Cohen's impossibility result. In P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, eds., *Advances in grid computing – EGC 2005*, Springer-Verlag, LNCS 3470:1097–1106, 2005.
14. F. Cohen. Computer viruses — theory and experiments. *Computers & Security*, 6(1):22–35, 1984. Available as `http://vx.netlux.org/lib/afc01.html`.
15. P. Jančar. Decidability questions for bisimilarity of Petri nets and some related problems. *Proceedings of STACS94*, Springer-Verlag, LNCS 775:581-592, 1994.
16. P. Jančar, F. Moller, and Z. Sawa. Simulation problems for one-counter machines. *Proceedings of SOFSEM99: The 26th Seminar on Current Trends in Theory and Practice of Informatics*, Springer-Verlag, LNCS 1725:398-407, 1999.
17. C. Jesshope. Implementing an efficient vector instruction set in a chip multiprocessor using micro-threaded pipelines. *Australian Computer Science Communications*, 23(4):80–88, 2001.
18. W.D. Maurer. A theory of computer instructions. *Journal of the ACM*, 13(2):226–235, 1966.
19. T.D. Vu. Metric denotational semantics for BPPA. Report PRG0503, Programming Research Group, University of Amsterdam, July 2005. Available at `www.science.uva.nl/research/prog/publications.html`.