Scientific Annals of Computer Science vol. 21, 2011, pp. 283–310

Thread Extraction for Polyadic Instruction Sequences

Jan BERGSTRA[1], Cornelis MIDDELBURG[1]

Abstract

In this paper, we study the phenomenon that instruction sequences are split into fragments which somehow produce a joint behaviour. In order to bring this phenomenon better into the picture, we formalize a simple mechanism by which several instruction sequence fragments can produce a joint behaviour. We also show that, even in the case of this simple mechanism, it is a non-trivial matter to explain by means of a translation into a single instruction sequence what takes place on execution of a collection of instruction sequence fragments.

Keywords: Fragmented instruction sequence execution, polyadic instruction sequence, program algebra, basic thread algebra.

1 Introduction

With the work presented in this paper, we carry on a line of research with which a start was made in [4]. This line of research is concerned with sequential programs that take the form of instruction sequences. Its working hypothesis is that instruction sequence is a central notion of computer science, which merits investigation for its own sake.

An instruction sequence is considered to produce on execution a behaviour to be controlled by some execution environment. This behaviour proceeds by performing steps in a sequential fashion. Each step performed actuates the processing of an instruction by the execution environment in question. A reply returned by this execution environment at completion of

[1]Informatics Institute, Faculty of Science, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, the Netherlands.
Email: {J.A.Bergstra,C.A.Middelburg}@uva.nl.

the processing of the instruction determines how the behaviour proceeds further.

The following phenomenon presents itself: instruction sequences are split into fragments which somehow produce a joint behaviour. The objective of this paper is to bring this phenomenon better into the picture. To achieve this, we formalize a simple mechanism by which several instruction sequence fragments can produce a joint behaviour. We show that, even in the case of this simple mechanism, it is a non-trivial matter to explain by means of a translation into a single instruction sequence what takes place on execution of a collection of instruction sequence fragments.

The question is how a joint behaviour of the fragments in a collection of fragments is achieved. The view of this matter is that there can only be a single fragment being executed at any stage, but the fragment in question may make any fragment in the collection the one being executed by means of a special instruction for switching over execution to another fragment. This does not fit in very well with the conception that the collection of fragments constitutes a sequential program. To our knowledge, a theoretical understanding of this matter has not yet been developed. This has motivated us to take up this topic.

The principal reason for splitting instruction sequences into fragments is that the execution environment at hand sets bounds to the size of instruction sequences. In the past, the phenomenon occurred explicitly in many software systems. At present, it often occurs rather implicitly, e.g. on execution of programs written in contemporary object-oriented programming languages, such as Java [1] and C# [11], classes are loaded as they are needed. The mechanisms in question are improvements upon the simple mechanism considered in this paper, but they are also much more complicated. We believe that it is useful to consider the simple mechanism prior to the more complicated ones.

The instruction sequences taken for fragments are called polyadic instruction sequences in this paper. We introduce polyadic instruction sequences in the setting of program algebra [4]. The starting-point of program algebra is the perception of a program as a single-pass instruction sequence, i.e. a finite or infinite sequence of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over. This perception is simple, appealing, and links up with practice.

The behaviours produced by instruction sequences on execution are modelled by threads as considered in basic thread algebra [4].[2] We take the view that the possible joint behaviours produced by polyadic instruction sequences on execution are threads as considered in basic thread algebra as well. In a system that provides an execution environment for polyadic instruction sequences, a polyadic instruction sequence must be loaded in order to become the one being executed. Hence, making a polyadic instruction sequence the one being executed can be looked upon as loading it for execution.

In [4], a hierarchy of program notations rooted in program algebra is presented. Included in this hierarchy are very simple program notations which are close to existing assembly languages up to and including simple program notations that support structured programming by offering a rendering of conditional and loop constructs. All of these program notations are referred to in this paper, but only one of them is actually used. That program notation is introduced under the name PGLD in [4].

This paper is organized as follows. First, we review basic thread algebra and program algebra (Sections 2 and 3). After that, we give an overall picture of the hierarchy of program notations rooted in program algebra and present the program notation PGLD (Sections 4 and 5). Next, we introduce polyadic instruction sequences in the setting of program algebra, explain the possible joint behaviours of a collection of polyadic instruction sequences using basic thread algebra, and give an example of the use of polyadic instruction sequences (Sections 6 and 7). Following this, we extend basic thread algebra to allow for threads to make use of services and give a description of instruction register file services (Sections 8 and 9). After that, we show that, for each possible joint behaviour of a collection of polyadic instruction sequences, a single instruction sequence can be synthesized from the collection of polyadic instruction sequences that produces on execution essentially the behaviour in question by making use of an instruction register file service (Section 10). Finally, we make some concluding remarks (Section 11).

In this paper, we only give brief summaries of program algebra and basic thread algebra. Comprehensive introductions, including examples, can be found in [4, 14].

[2]In [4], basic thread algebra is introduced under the name basic polarized process algebra.

2 Basic Thread Algebra

In this section, we review BTA (Basic Thread Algebra). BTA is a form of process algebra which is concerned with the behaviour that sequential programs produce on execution. Those behaviours are called *threads*.

In BTA, it is assumed that fixed but arbitrary finite sets $\mathcal{A}$ and $\mathcal{I}$ with $\mathcal{A} \cap \mathcal{I} = \emptyset$ and $\mathsf{tau} \in \mathcal{I}$ have been given. The members of $\mathcal{A}$ are called *basic actions* and the members of $\mathcal{I}$ are called *internal actions*. The members of $\mathcal{A} \cup \mathcal{I}$ are referred to as *actions*. In previous work, we take in essence the singleton set $\{\mathsf{tau}\}$ for $\mathcal{I}$. The generalization made here permits internal actions with differences relevant for analysis to be distinguished.

The operational intuition is that a thread has an execution environment which processes each action performed by the thread. A thread performs actions in a sequential fashion. Upon each action performed, a reply from the execution environment of the thread determines how it proceeds. The possible replies are T and F. Performing an internal action, always leads to the reply T.

Although BTA is one-sorted, we make this sort explicit. The reason for this is that we will extend BTA with an additional sort in Section 8.

BTA has one sort: the sort $\mathbf{T}$ of *threads*. To build terms of sort $\mathbf{T}$, BTA has the following constants and operators:

- the *inaction* constant $\mathsf{D} : \mathbf{T}$;
- the *termination* constant $\mathsf{S} : \mathbf{T}$;
- for each $a \in \mathcal{A} \cup \mathcal{I}$, the binary *postconditional composition* operator $\_ \trianglelefteq a \trianglerighteq \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

We assume that there are infinitely many variables of sort $\mathbf{T}$, including x, y, z. Terms of sort $\mathbf{T}$ are built as usual (see e.g. [15, 16]). We use infix notation for the postconditional composition operator. We introduce *basic action prefixing* as an abbreviation: $a \circ p$ abbreviates $p \trianglelefteq a \trianglerighteq p$.

The thread denoted by a closed term of the form $p \trianglelefteq a \trianglerighteq q$ will first perform a, and then proceed as the thread denoted by p if the reply from the execution environment is T and proceed as the thread denoted by q if the reply from the execution environment is F. The threads denoted by D and S will become inactive and terminate successfully, respectively. A thread is inactive if it is neither capable of performing any action nor capable of terminating successfully.

Table 1: Axiom of BTA

$x \trianglelefteq \iota \trianglerighteq y = x \trianglelefteq \iota \trianglerighteq x$	T1

Table 2: Axioms for guarded recursion

$\langle X \vert E\rangle = \langle t_X \vert E\rangle$	if $X = t_X \in E$	RDP
$E \Rightarrow X = \langle X \vert E\rangle$	if $X \in \mathrm{V}(E)$	RSP

BTA has only one axiom. This axiom is given in Table 1. In this table, ι stands for an arbitrary member of $\mathcal{I}$.

Notice that each closed BTA term denotes a thread that will become inactive or terminate after it has performed finitely many actions. Infinite threads can be described by guarded recursion.

A *guarded recursive specification* over BTA is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables of sort $\mathbf{T}$ and each t_X is a BTA term of the form D, S or $t \trianglelefteq a \trianglerighteq t'$ with t and t' that contain only variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur in E. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in [2]. A thread that is the solution of a finite guarded recursive specification over BTA is called a *finite-state* thread.

For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we introduce a constant $\langle X|E\rangle$ of sort $\mathbf{T}$ standing for the unique solution of E for X. The axioms for these constants are given in Table 2. In this table, we write $\langle t_X|E\rangle$ for t_X with, for all $Y \in \mathrm{V}(E)$, all occurrences of Y in t_X replaced by $\langle Y|E\rangle$. X, t_X and E stand for an arbitrary variable of sort $\mathbf{T}$, an arbitrary BTA term of sort $\mathbf{T}$ and an arbitrary guarded recursive specification over BTA, respectively. Side conditions are added to restrict what X, t_X and E stand for. RDP and RSP are abbreviations for Recursive Definition Principle and Recursive Specification Principle, respectively. The equations $\langle X|E\rangle = \langle t_X|E\rangle$ for a fixed E express that the constants $\langle X|E\rangle$ make up a solution of E and the conditional equations $E \Rightarrow X = \langle X|E\rangle$ express that this solution is the only one.

We will use the following abbreviation: a^ω, where $a \in \mathcal{A} \cup \mathcal{I}$, abbreviates $\langle X|\{X = a \circ X\}\rangle$.

We will write BTA+REC for BTA extended with the constants for solutions of guarded recursive specifications and axioms RDP and RSP.

Table 3: Approximation induction principle

$\bigwedge_{n \geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$	AIP
$\pi_0(x) = \mathsf{D}$	P0
$\pi_{n+1}(\mathsf{S}) = \mathsf{S}$	P1
$\pi_{n+1}(\mathsf{D}) = \mathsf{D}$	P2
$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y) = \pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	P3

Closed terms of sort $\mathbf{T}$ from the language of BTA+REC that denote the same infinite thread cannot always be proved equal by means of the axioms of BTA+REC. We introduce AIP (Approximation Induction Principle) to remedy this. AIP is based on the view that two threads are identical if their approximations up to any finite depth are identical. The approximation up to depth n of a thread is obtained by cutting it off after performing a sequence of actions of length n. In AIP, the approximation up to depth n is phrased in terms of the unary *projection* operator $\pi_n : \mathbf{T} \to \mathbf{T}$. AIP and the axioms for the projection operators are given in Table 3. In this table, a stands for an arbitrary member of $\mathcal{A} \cup \mathcal{I}$.

We will write BTA+REC+AIP for BTA+REC extended with the projection operators and the axioms from Table 3.

3 Program Algebra

In this section, we review PGA (ProGram Algebra). The perception of a program as a single-pass instruction sequence is the starting-point of PGA.

In PGA, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of *basic instructions* has been given. The intuition is that the execution of a basic instruction may modify a state and produces a reply at its completion. The possible replies are the Boolean values T and F. The set $\mathfrak{A}$ is the basis for the set of all instructions of which the instruction sequences considered in PGA are made up. These instructions are called primitive instructions.

PGA has the following *primitive instructions*:

- for each $a \in \mathfrak{A}$, a *plain basic instruction* a;
- for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$;

- for each $l \in \mathbb{N}$, a *forward jump instruction* $\#l$;
- a *termination instruction* $!$.

We write $\mathfrak{I}$ for the set of all primitive instructions. Notice that we use the term plain basic instruction to refer to a basic instruction as a primitive instruction of a certain kind.

On execution of an instruction sequence, the effect of a positive test instruction $+a$ is that basic instruction a is executed and execution proceeds with the next primitive instruction if T is produced and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one. If there is no primitive instruction to proceed with, inaction occurs. The effect of a negative test instruction $-a$ is the same as the effect of $+a$, but with the role of the value produced reversed. The effect of a plain basic instruction a is the same as the effect of $+a$, but execution always proceeds as if T is produced. The effect of a forward jump instruction $\#l$ is that execution proceeds with the l-th next primitive instruction of the instruction sequence concerned. If l equals 0 or there is no primitive instructions to proceed with, inaction occurs. The effect of the termination instruction $!$ is that execution terminates.

PGA has the following constants and operators:

- for each $u \in \mathfrak{I}$, an *instruction* constant u;
- the binary *concatenation* operator $\_\,;\,\_$;
- the unary *repetition* operator $\_^{\omega}$.

We assume that there are infinitely many variables, including x, y, z. Terms are built as usual. We use infix notation for the concatenation operator and postfix notation for the repetition operator.

A closed PGA term is considered to denote a non-empty, finite or periodic infinite sequence of primitive instructions.[3] Closed PGA terms are considered equal if they denote the same instruction sequence. The axioms for instruction sequence equivalence are given in Table 4. In this table, n stands for an arbitrary natural number greater than 0. For each PGA term P, the term P^n is defined by induction on n as follows: $P^1 = P$ and $P^{n+1} = P\,;P^n$. The equation $X^{\omega} = X\,;X^{\omega}$ is derivable. Each closed PGA term is derivably equal to one of the form P or $P\,;Q^{\omega}$, where P and Q are closed PGA terms in which the repetition operator does not occur.

[3]An infinite sequence is periodic if the set of all its subsequences is finite.

Table 4: Axioms of PGA

$(x ; y) ; z = x ; (y ; z)$	PGA1
$(x^n)^\omega = x^\omega$	PGA2
$x^\omega ; y = x^\omega$	PGA3
$(x ; y)^\omega = x ; (y ; x)^\omega$	PGA4

Table 5: Defining equations for thread extraction operation of PGA

$\lvert a \rvert = a \circ \mathsf{D}$	$\lvert \#l \rvert = \mathsf{D}$
$\lvert a ; x \rvert = a \circ \lvert x \rvert$	$\lvert \#0 ; x \rvert = \mathsf{D}$
$\lvert +a \rvert = a \circ \mathsf{D}$	$\lvert \#1 ; x \rvert = \lvert x \rvert$
$\lvert +a ; x \rvert = \lvert x \rvert \trianglelefteq a \trianglerighteq \lvert \#2 ; x \rvert$	$\lvert \#l + 2 ; u \rvert = \mathsf{D}$
$\lvert -a \rvert = a \circ \mathsf{D}$	$\lvert \#l + 2 ; u ; x \rvert = \lvert \#l + 1 ; x \rvert$
$\lvert -a ; x \rvert = \lvert \#2 ; x \rvert \trianglelefteq a \trianglerighteq \lvert x \rvert$	$\lvert ! \rvert = \mathsf{S}$
	$\lvert ! ; x \rvert = \mathsf{S}$

Each closed PGA term P is considered to denote an instruction sequence of which the behaviour is a finite-state thread, called the *thread produced by* P. The set $\mathfrak{A}$ of basic instructions is taken for the set $\mathcal{A}$ of basic actions. The *thread extraction* operation $|\_|$ determines, for each closed PGA term P, a finite guarded recursive specification over BTA that defines the thread produced by P. The thread extraction operation is defined by the equations given in Table 5 (for $a \in \mathfrak{A}$, $l \in \mathbb{N}$ and $u \in \mathfrak{I}$) and the rule that $|\#l ; x| = \mathsf{D}$ if $\#l$ is the beginning of an infinite chain of forward jumps. This rule is formalized in e.g. [7].

The behaviour of each closed PGA term, is a thread that is definable by a finite guarded recursive specification over BTA. The other way round, each finite guarded recursive specification over BTA in which no internal actions occur can be translated into a closed PGA term of which the behaviour is the solution of the finite guarded recursive specification concerned.

Closed PGA terms are considered to denote programs and therefore they constitute an elementary program notation. Closed PGA terms are also called PGA programs.

4 A Hierarchy of Program Notations Rooted in Program Algebra

In [4], a hierarchy of program notations rooted in PGA is presented. The program notations that appear in this hierarchy are PGA, PGLA, PGLB, PGLC, PGLD, PGLDg, PGLE, and PGLS. The most interesting ones are PGLC, PGLD and PGLS. PGLC and PGLD are close to existing assembly languages. The main difference between them is that PGLC has relative jump instructions and PGLD has absolute jump instructions. PGLS supports structured programming by offering a rendering of conditional and loop constructs instead of (unstructured) jump instructions.

For each of the program notations that appear in the hierarchy, except PGA, a function is given in [4] by means of which each program from that program notation is translated into a program from the first program notation lower in the hierarchy that produces the same behaviour on execution. These functions are called projections. Moreover, for each of the program notations that appear in the hierarchy, except PGLE and PGLS, a function is given in [4] by means of which each program from that program notation is translated into a program from the first program notation higher in the hierarchy that produces the same behaviour on execution. These functions are called embeddings. There does not exist a function by which each PGLE program is translated into a PGLS program that produces the same behaviour on execution because PGLS is strictly weaker than PGLE. The names of the projections and embeddings referred to above are given in Figure 1.

The program notations, projections, and embeddings referred to above are defined in [4]. We refrain from giving all the definitions in question in this paper as well, because most details of the program notations, projections, and embeddings do not matter here. The important point is that there exist a collection of well-defined program notations rooted in an elementary program notation with projections and embeddings between them. Only PGLD is actually used later on in this paper. Therefore, PGLD is reviewed below in Section 5.

In Section 6, we will take special versions of the above-mentioned program notations for the ones that may be used for fragments. Moreover, we will make use of the projections `pgla2pga` and in addition the projections `pglb2pga`, `pglc2pga`, ... defined in the obvious way by composition of projections given in [4]. In Section 10, we will make use of the embedding

$$\begin{tikzcd}
\mathrm{PGLS} \arrow[d, "\texttt{pgls2pgle}"'] \\
\mathrm{PGLE} \arrow[d, shift right, "\texttt{pgle2pgldg}"'] \\
\mathrm{PGLDg} \arrow[u, shift right, "\texttt{pgldg2pgle}"'] \arrow[d, shift right, "\texttt{pgldg2pgld}"'] \\
\mathrm{PGLD} \arrow[u, shift right, "\texttt{pgld2pgldg}"'] \arrow[d, shift right, "\texttt{pgld2pglc}"'] \\
\mathrm{PGLC} \arrow[u, shift right, "\texttt{pglc2pgld}"'] \arrow[d, shift right, "\texttt{pglc2pglb}"'] \\
\mathrm{PGLB} \arrow[u, shift right, "\texttt{pglb2pglc}"'] \arrow[d, shift right, "\texttt{pglb2pgla}"'] \\
\mathrm{PGLA} \arrow[u, shift right, "\texttt{pgla2pglb}"'] \arrow[d, shift right, "\texttt{pgla2pga}"'] \\
\mathrm{PGA} \arrow[u, shift right, "\texttt{pga2pgla}"']
\end{tikzcd}$$

Figure 1: Hierarchy of program notations rooted in PGA

pglc2pgld and in addition the embedding pga2pglc defined in the obvious way by composition of embeddings given in [4].

5 The Program Notation PGLD

In this section, we review the program notation PGLD. This program notation is reviewed because it will be used later on in Sections 7 and 10.

In PGLD, like in PGA, it is assumed that there is a fixed but arbitrary finite set $\mathfrak{A}$ of *basic instructions*. Again, the intuition is that the execution of a basic instruction a produces either T or F at its completion.

PGLD has the following primitive instructions:

- for each $a \in \mathfrak{A}$, a *plain basic instruction* a;
- for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$;
- for each $l \in \mathbb{N}$, an *absolute jump instruction* $\#\#l$.

PGLD programs have the form $u_1; \ldots ; u_k$, where $u_1, \ldots , u_k$ are primitive instructions of PGLD.

The plain basic instructions, the positive test instructions, and the negative test instructions are as in PGA. The effect of an absolute jump instruction $\#\#l$ is that execution proceeds with the l-th instruction of the program concerned. If $\#\#l$ is itself the l-th instruction, then $\#\#l$ results in inaction. If l equals 0 or l is greater than the length of the program, then termination occurs.

The function `pgld2pga` from the set of all PGLD programs to the set of all PGA programs, which translates each PGLD program into a PGA program that produces the same behaviour on execution, can be defined directly as follows:

$$\texttt{pgld2pga}(u_1 \,;\, \ldots \,;\, u_k) = (\psi_1(u_1) \,;\, \ldots \,;\, \psi_k(u_k) \,;\, ! \,;\, !)^\omega \;,$$

where the auxiliary functions ψ_j from the set of all primitive instructions of PGLD to the set of all primitive instructions of PGA are defined as follows $(1 \leq j \leq k)$:

$$\begin{aligned}
\psi_j(\#\#l) &= \#l - j && \text{if } j \leq l \leq k \;, \\
\psi_j(\#\#l) &= \#k + 2 - (j - l) && \text{if } 0 < l < j \;, \\
\psi_j(\#\#l) &= \,! && \text{if } l = 0 \lor l > k \;, \\
\psi_j(u) \quad &= u && \text{if } u \text{ is not a jump instruction} \;.
\end{aligned}$$

The idea is that each backward jump can be replaced by a forward jump if the entire program is repeated. To enforce termination of the program after execution of its last instruction if the last instruction is a plain basic instruction, a positive test instruction or a negative test instruction, $! \,;\, !$ is appended to $\psi_1(u_1) \,;\, \ldots \,;\, \psi_k(u_k)$.

6 Polyadic Instruction Sequences

In this section, we formalize a simple mechanism by which several instruction sequence fragments can produce a joint behaviour. The instruction sequence fragments are viewed as instruction sequences that contain special instructions for switching over execution from one fragment to another. The instruction sequences in question are called polyadic instruction sequences. It is assumed that a special version of PGLA, PGLB, PGLC, PGLD, PGLDg, PGLE or PGLS, in which the special instructions for switching over execution from one fragment to another are available, is used for each polyadic instruction sequence. Moreover, it is assumed that a collection of polyadic instruction sequences between which execution can be switched takes the form of a sequence, called a polyadic instruction sequence vector, in which each polyadic instruction sequence is coupled with the program notation used for it.

Our general view on the way of achieving a joint behaviour of the polyadic instruction sequences in a polyadic instruction sequence vector is as follows:

- there can only be a single polyadic instruction sequence being executed at any stage;
- the polyadic instruction sequence in question may make any polyadic instruction sequence in the vector the one being executed;
- making another polyadic instruction sequence the one being executed is effected by executing a special instruction for switching over execution;
- any polyadic instruction sequence can be taken for the one being executed initially.

In addition to special instructions for switching over execution, polyadic instruction sequences may contain two other kinds of special instructions:

- special instructions for putting instructions into instruction registers;
- special instructions of which the occurrences in a polyadic instruction sequence are replaced by instructions contained in instruction registers on making the polyadic instruction sequence the one being executed.

The special instructions of the latter kind serve as instruction place-holders. Their presence turns a polyadic instruction sequence into a parameterized instruction sequence of which the parameters are filled in each time it is made the one being executed. This feature accounts for the use of the prefix polyadic. Its merit is primarily that it allows for execution to proceed in effect from different positions each time a polyadic instruction sequence is loaded for execution. An example of this is given in Section 7.

We take the line that different program notations can be used for different polyadic instruction sequences in a polyadic instruction sequence vector. On making a polyadic instruction sequence in the vector the one being executed, it is considered to be translated into a $\mathrm{PGA_p}$ program.

$\mathrm{PGA_p}$ is a variant of PGA in which the above-mentioned special instructions are incorporated. In $\mathrm{PGA_p}$, it is assumed that there is a fixed but arbitrary finite set $\mathfrak{A}_\mathrm{c}$ of *core basic instructions*. In $\mathrm{PGA_p}$, a basic instruction is either a core basic instruction or a supplementary basic instruction.

$\mathrm{PGA_p}$ has the following *core primitive instructions*:

- for each $a \in \mathfrak{A}_\mathrm{c}$, a *plain basic instruction* a;
- for each $a \in \mathfrak{A}_\mathrm{c}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}_\mathrm{c}$, a *negative test instruction* $-a$;
- for each $l \in \mathbb{N}$, a *forward jump instruction* $\#l$;
- a *termination instruction* $!$.

We write $\mathfrak{I}_\mathrm{c}$ for the set of all core primitive instructions. The core primitive instructions of $\mathrm{PGA_p}$ are the counterparts of the primitive instructions of PGA.

$\mathrm{PGA_p}$ has the following *supplementary basic instructions*:

- for each $i \in \mathbb{N}$, a *switch-over instruction* $\#\#\#i$;
- for each $i \in \mathbb{N}$ and $u \in \mathfrak{I}_\mathrm{c}$, a *put instruction* $\$\mathsf{put}{:}i{:}u$;
- for each $i \in \mathbb{N}$, a *get instruction* $\$\mathsf{get}{:}i$.

We write $\mathfrak{A}_\mathrm{s}$ for the set of all supplementary basic instructions. In the presence of a polyadic instruction sequence vector, a switch-over instruction $\#\#\#i$ is the instruction for switching over execution to the i-th polyadic instruction sequence in the vector. A put instruction $\$\mathsf{put}{:}i{:}u$ is the instruction for putting instruction u in the instruction register with number i. A get instruction $\$\mathsf{get}{:}i$ is the instruction of which each occurrence in a polyadic instruction sequence is replaced by the contents of the instruction register with number i on switching over execution to that polyadic instruction sequence. If a get instruction is encountered in the polyadic instruction sequence being executed, inaction occurs.

The supplementary basic instructions of $\mathrm{PGA_p}$ can be viewed as built-in basic instructions. However, as laid down below, supplementary basic instructions do not occur in positive or negative test instructions. Thus, the core primitive instructions and supplementary basic instructions make up the primitive instructions of $\mathrm{PGA_p}$.

$\mathrm{PGA_p}$ has the following constants and operators:

- for each $u \in \mathfrak{I}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$, an *instruction* constant u;
- the binary *concatenation* operator $\_\,;\_$;
- the unary *repetition* operator $\_^\omega$.

The axioms of $\mathrm{PGA_p}$ are the same as the axioms of PGA.

Suppose that in PGA the restriction is dropped that $\mathfrak{A}$ must be a finite set. Then $\mathrm{PGA_p}$ can be viewed as the specialization of PGA obtained by taking the set $\mathfrak{A}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ for $\mathfrak{A}$ and excluding terms in which basic instructions from $\mathfrak{A}_\mathrm{s}$ occur in positive or negative test instructions. Henceforth, we actually drop the restriction that $\mathfrak{A}$ must be a finite set. This simplifies the definitions of the different program notations that can be used for polyadic instruction sequences and also enables the use of the functions `pgla2pga`, `pglb2pga`, etcetera for translating programs in those program notations into $\mathrm{PGA_p}$ programs.

The different program notations that can be used for polyadic instruction sequences are $\mathrm{PGLA_p}$, $\mathrm{PGLB_p}$, $\mathrm{PGLC_p}$, $\mathrm{PGLD_p}$, $\mathrm{PGLDg_p}$, $\mathrm{PGLE_p}$, and $\mathrm{PGLS_p}$. The set of all $\mathrm{PGLA_p}$ programs is the subset of the set of all PGLA programs, taking the set $\mathfrak{A}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ for $\mathfrak{A}$, in which the basic instructions from $\mathfrak{A}_\mathrm{s}$ do not occur in positive or negative test instructions. The other program notations are defined similarly. If the set $\mathfrak{A}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ is taken for $\mathfrak{A}$, the function `pgla2pga` translates each $\mathrm{PGLA_p}$ program into a $\mathrm{PGA_p}$ program that produces the same behaviour on execution. Similar remarks apply to the other program notations.

A *polyadic instruction sequence* is either a $\mathrm{PGLA_p}$ program, a $\mathrm{PGLB_p}$ program, a $\mathrm{PGLC_p}$ program, a $\mathrm{PGLD_p}$ program, a $\mathrm{PGLDg_p}$ program, a $\mathrm{PGLE_p}$ program or a $\mathrm{PGLS_p}$ program.

A *polyadic instruction sequence vector* is a sequence of pairs consisting of a polyadic instruction sequence and a member of the set $\{A, B, C, D, Dg, E, S\}$ of *program notation indices.*

Let α be a polyadic instruction sequence vector, let $P_1, \ldots, P_n$ and $c_1, \ldots, c_n$ be polyadic instruction sequences and program notation indices, respectively, such that $\alpha = \langle (P_1, c_1)\rangle \frown \ldots \frown \langle (P_n, c_n)\rangle$,[4] and let $i \in [1, n]$. Then we write $pg(\alpha, i)$ and $pgn(\alpha, i)$ for P_i and c_i, respectively.

[4]We write D^* for the set of all finite sequences with elements from set D. We use the following notation for finite sequences: $\langle\,\rangle$ for the empty sequence, $\langle d\rangle$ for the sequence having d as sole element, $\sigma \frown \sigma'$ for the concatenation of finite sequences σ and σ', and $\mathrm{len}(\sigma)$ for the length of finite sequence σ.

Let α be a polyadic instruction sequence vector of length n, and let $i \in [1, n]$. Then program notation index $\mathit{pgn}(\alpha, i)$ indicates which program notation is used for polyadic instruction sequence $\mathit{pg}(\alpha, i)$. A stands for $\mathrm{PGLA_p}$, B stands for $\mathrm{PGLB_p}$, etcetera. The program notation used is made explicit because it cannot always be determined uniquely from the polyadic instruction sequence concerned, whereas the behaviour that this polyadic instruction sequence produces on execution may be different for each of the program notations in question. For example, every $\mathrm{PGLC_p}$ program is a $\mathrm{PGLB_p}$ program in which no termination instructions occur. If such a program leads to termination on execution as a $\mathrm{PGLC_p}$ program, it leads to inaction on execution as a $\mathrm{PGLB_p}$ program (for details, see [4]).

Below, we use special notation relating to the program notation indices. Let c be a program notation index. Then we write prj_c for the projection `pgla2pga` if $c = A$, the projection `pglb2pga` if $c = B$, etcetera.

The set of instruction registers that contain an instruction and the contents of each of those registers matter when a polyadic instruction sequence is made the one being executed. That makes us introduce the notion of an instruction register file state and special notation relating to this notion.

An *instruction register file state* is a function $\sigma : I \to \mathfrak{I}_\mathrm{c}$, where I is a finite subset of $\mathbb{N}$.

Let p be a $\mathrm{PGA_p}$ program and σ be an instruction register file state. Then we write $p[\sigma]$ for p with, for all $i \in \mathrm{domain}(\sigma)$, all occurrences of $\mathsf{\$get}{:}i$ in p replaced by $\sigma(i)$.

Let $i, n \in \mathbb{N}$ be such that $1 \leq i \leq n$, let α be a polyadic instruction sequence vector of length n, and let σ be an instruction register file state. Then we write $\mathit{valid}(\alpha, i, \sigma)$ to indicate that instructions of the form $\mathsf{\$get}{:}i$ do not occur in $\mathit{prj}_{\mathit{pgn}(\alpha,i)}(\mathit{pg}(\alpha, i))[\sigma]$.

An obvious choice of the thread extraction operation of $\mathrm{PGA_p}$ is the thread extraction operation of PGA, taking the set $\mathfrak{A}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ for $\mathfrak{A}$, restricted to the set of closed terms of $\mathrm{PGA_p}$. This thread extraction operation is considered not to be the proper one, because it treats the supplementary basic instructions as arbitrary basic instructions and thus disregards the fixed effects that they should produce on execution. For example, a switch-over instruction $\#\#\#i$ would not have the effect that execution is switched over. Moreover, this thread extraction operation would require that in BTA the restriction is dropped that $\mathcal{A}$ must be a finite set.

As regards the proper thread extraction for $\mathrm{PGA_p}$, the idea is that it yields, for each $\mathrm{PGA_p}$ program P, a function that determines, for each polyadic instruction sequence vector α, a finite guarded recursive specification over BTA that defines the thread that is the joint behaviour of P and the polyadic instruction sequences in α in the case where P is the polyadic instruction sequence being executed initially. Because this behaviour depends upon the set of instruction registers that contain an instruction and the contents of each of those registers, we need a thread extraction operation for each instruction register file state.

In the thread extraction for $\mathrm{PGA_p}$, it is assumed that $\mathsf{gl} \in \mathcal{I}$. The internal action gl (generate and load) represents the internal activity involved in switching over execution. The internal actions tau and gl are distinguished because tau is considered to represent a negligible internal activity, whereas gl is considered to represent a substantial internal activity.

For each instruction register file state σ, we introduce the *thread extraction* operation $|\_|_\sigma$. These thread extraction operations are defined by the equations given in Table 6 (for $a \in \mathfrak{A}$, $l, i \in \mathbb{N}$, $u \in \mathfrak{I}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ and $v \in \mathfrak{I}_\mathrm{c}$) and the rule that $|\#l \,;X|_\sigma(\alpha) = \mathsf{D}$ if $\#l$ is the beginning of an infinite chain of forward jumps.

We can couple nominal indices as labels with some of the polyadic instruction sequences in a polyadic instruction sequence vector. This would permit the use of alternative switch-over instructions with nominal indices instead of ordinal indices, like with the goto instructions from PGLDg. In the notational style of [3], the form of those alternative switch-over instructions would be $\#\#\#[i]$.

7 Example

To illustrate the mechanism formalized in Section 6, we consider in this section the splitting of a PGLD program P of 10000 instructions into two fragments.

We write $\nu_1(l)$ for the number of absolute jump instructions $\#\#l'$ with $l' > 5000$ from position 1 up to position l and $\nu_2(l)$ for the number of absolute jump instructions $\#\#l'$ with $l' \leq 5000$ from position 5001 up to position l.

The polyadic instruction sequence P' corresponding to the first half of P is obtained from the first half of P as follows:

- the instruction $\mathsf{\$get}{:}1$ is prefixed to it;
- each absolute jump instruction $\#\#l$ with $l \leq 5000$ is replaced by the absolute jump instructions $\#\#l'$, where $l' = l + \nu_1(l) + 1$;

Table 6: Defining equations for thread extraction operations of $\mathrm{PGA_p}$

$\lvert a\rvert_\sigma(\alpha) = a \circ \mathsf{D}$	
$\lvert a \mathbin{;} x\rvert_\sigma(\alpha) = a \circ \lvert x\rvert_\sigma(\alpha)$	
$\lvert {+a}\rvert_\sigma(\alpha) = a \circ \mathsf{D}$	
$\lvert {+a} \mathbin{;} x\rvert_\sigma(\alpha) = \lvert x\rvert_\sigma(\alpha) \unlhd a \unrhd \lvert \#2 \mathbin{;} x\rvert_\sigma(\alpha)$	
$\lvert {-a}\rvert_\sigma(\alpha) = a \circ \mathsf{D}$	
$\lvert {-a} \mathbin{;} x\rvert_\sigma(\alpha) = \lvert \#2 \mathbin{;} x\rvert_\sigma(\alpha) \unlhd a \unrhd \lvert x\rvert_\sigma(\alpha)$	
$\lvert \#l\rvert_\sigma(\alpha) = \mathsf{D}$	
$\lvert \#0 \mathbin{;} x\rvert_\sigma(\alpha) = \mathsf{D}$	
$\lvert \#1 \mathbin{;} x\rvert_\sigma(\alpha) = \lvert x\rvert_\sigma(\alpha)$	
$\lvert \#l+2 \mathbin{;} u\rvert_\sigma(\alpha) = \mathsf{D}$	
$\lvert \#l+2 \mathbin{;} u \mathbin{;} x\rvert_\sigma(\alpha) = \lvert \#l+1 \mathbin{;} x\rvert_\sigma(\alpha)$	
$\lvert !\rvert_\sigma(\alpha) = \mathsf{S}$	
$\lvert ! \mathbin{;} x\rvert_\sigma(\alpha) = \mathsf{S}$	
$\lvert \#\#\#i\rvert_\sigma(\alpha) = \mathsf{gl} \circ \lvert \mathit{prj}_{\mathit{pgn}(\alpha,i)}(\mathit{pg}(\alpha,i))[\sigma]\rvert_\sigma(\alpha)$	if $1 \leq i \leq n \land \mathit{valid}(\alpha,i,\sigma)$
$\lvert \#\#\#i\rvert_\sigma(\alpha) = \mathsf{D}$	if $1 \leq i \leq n \land \neg\mathit{valid}(\alpha,i,\sigma)$
$\lvert \#\#\#i\rvert_\sigma(\alpha) = \mathsf{S}$	if $i = 0 \lor i > n$
$\lvert \#\#\#i \mathbin{;} x\rvert_\sigma(\alpha) = \mathsf{gl} \circ \lvert \mathit{prj}_{\mathit{pgn}(\alpha,i)}(\mathit{pg}(\alpha,i))[\sigma]\rvert_\sigma(\alpha)$	if $1 \leq i \leq n \land \mathit{valid}(\alpha,i,\sigma)$
$\lvert \#\#\#i \mathbin{;} x\rvert_\sigma(\alpha) = \mathsf{D}$	if $1 \leq i \leq n \land \neg\mathit{valid}(\alpha,i,\sigma)$
$\lvert \#\#\#i \mathbin{;} x\rvert_\sigma(\alpha) = \mathsf{S}$	if $i = 0 \lor i > n$
$\lvert \$\mathsf{put}{:}i{:}v\rvert_\sigma(\alpha) = \mathsf{tau} \circ \mathsf{D}$	
$\lvert \$\mathsf{put}{:}i{:}v \mathbin{;} x\rvert_\sigma(\alpha) = \mathsf{tau} \circ \lvert x\rvert_{\sigma \oplus [i \mapsto v]}(\alpha)$	
$\lvert \$\mathsf{get}{:}i\rvert_\sigma(\alpha) = \mathsf{D}$	
$\lvert \$\mathsf{get}{:}i \mathbin{;} x\rvert_\sigma(\alpha) = \mathsf{D}$	

- each absolute jump instruction $\#\#l$ with $l > 5000$ is replaced by the instruction sequence $\$\mathsf{put}{:}2{:}\#l' \mathbin{;} \#\#\#2$, where $l' = (l - 5000) + \nu_2(l - 5000)$;

and the polyadic instruction sequence P'' corresponding to the second half of P is obtained from the second half of P as follows:

- the instruction $\$\mathsf{get}{:}2$ is prefixed to it;

- each absolute jump instruction $\#\#l$ with $l > 5000$ is replaced by the absolute jump instructions $\#\#l'$, where $l' = (l-5000)+\nu_2(l-5000)+1$;
- each absolute jump instruction $\#\#l$ with $l \leq 5000$ is replaced by the instruction sequence $\mathsf{\$put}{:}1{:}\#l' \,;\, \#\#\#1$, where $l' = l + \nu_1(l)$.

Notice that the positions occurring in jump instructions are adapted to the prefixing of a get instruction to each half of P and the replacement of each jump instructions that gives rise to a jump into the other half of P by two instructions.

For any instruction register file state σ, we have that $|P|$ coincides with $|\mathsf{\$put}{:}1{:}\#1\,;\#\#\#1|_\sigma(\langle(P', D)\rangle \frown \langle(P'', D)\rangle)$ after the presence of the internal actions tau and gl in the latter behaviour has been concealed. In Section 8, we will introduce operators to conceal the presence of internal actions.

In this section, we have illustrated by means of an example that splitting a program into fragments is relatively simple. In Section 10, we will show that synthesizing a program from a collection of fragments is fairly complicated.

8 Threads-Services Interaction and Abstraction

A thread may make use of services. That is, it may perform certain actions for the purpose of having itself affected by a service that takes them as commands to be processed. The processing of an action may involve a change of state of the service and at completion of the processing of the action the service returns a reply value to the thread. The reply value determines how the thread proceeds. In this section, we review the use operators, which are concerned with threads making such use of services.[5] We also introduce abstraction operators. They serve for concealment of the presence of internal actions, which arise among other things from the use operators. Both use operators and abstraction operators will be used later on in Section 10.

It is assumed that a fixed but arbitrary finite set $\mathcal{F}$ of *foci* and a fixed but arbitrary finite set $\mathcal{M}$ of *methods* have been given. Each focus plays the role of a name of a service provided by the execution environment that can be requested to process a command. Each method plays the role of a command proper. For the set $\mathcal{A}$ of basic actions, we take the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. Performing a basic action $f.m$ is taken as making a request to the service named f to process command m.

[5]This version of the use mechanism was first introduced in [6]. In later papers, it is also called thread-service composition.

A *service* H consists of

- a set S of *states*;
- an *effect* function $\mathit{eff} : \mathcal{M} \times S \to S$;
- a *yield* function $\mathit{yld} : \mathcal{M} \times S \to \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$;
- an *initial state* $s_0 \in S$;

satisfying the following condition:

$$\forall m \in \mathcal{M}, s \in S \bullet (\mathit{yld}(m, s) = \mathsf{B} \Rightarrow \forall m' \in \mathcal{M} \bullet \mathit{yld}(m', \mathit{eff}(m, s)) = \mathsf{B}) \;.$$

The set S contains the states in which the service may be, and the functions *eff* and *yld* give, for each method m and state s, the state and reply, respectively, that result from processing m in state s. By the condition imposed on services, once the service has returned B as reply, it keeps returning B as reply.

Let $H = (S, \mathit{eff}, \mathit{yld}, s_0)$ be a service and let $m \in \mathcal{M}$. Then the *derived service* of H after processing m, written $\frac{\partial}{\partial m} H$, is the service $(S, \mathit{eff}, \mathit{yld}, s'_0)$ where $s'_0 = \mathit{eff}(m, s_0)$; and the *reply* of H after processing m, written $H(m)$, is $\mathit{yld}(m, s_0)$.

When a thread makes a request to the service to process m:

- if $H(m) \neq \mathsf{B}$, then the request is accepted, the reply is $H(m)$, and the service proceeds as $\frac{\partial}{\partial m} H$;
- if $H(m) = \mathsf{B}$, then the request is rejected and the service proceeds as a service that rejects any request.

We introduce the sort $\mathbf{S}$ of *services* and, for each $f \in \mathcal{F}$, the binary *use* operator $\_ /_f \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$. The axioms for these operators are given in Table 7. In this table, ι stands for an arbitrary member of $\mathcal{I}$, f and g stand for arbitrary foci from $\mathcal{F}$, and m stands for an arbitrary method from $\mathcal{M}$.

Intuitively, $p /_f H$ is the thread that results from processing all basic actions performed by thread p that are of the form $f.m$ by service H. When a basic action of the form $f.m$ performed by thread p is processed by service H, it is turned into the internal action tau and postconditional composition is removed in favour of action prefixing on the basis of the reply value produced. This intuition is covered by axioms TSU5 and TSU6. Axioms TSU3 and TSU4 express that internal actions and basic actions of the form $g.m$ with

Table 7: Axioms for use operators

$\mathsf{S} /_f H = \mathsf{S}$		TSU1
$\mathsf{D} /_f H = \mathsf{D}$		TSU2
$(\iota \circ x) /_f H = \iota \circ (x /_f H)$		TSU3
$(x \unlhd g.m \unrhd y) /_f H = (x /_f H) \unlhd g.m \unrhd (y /_f H)$	if $f \neq g$	TSU4
$(x \unlhd f.m \unrhd y) /_f H = \mathsf{tau} \circ (x /_f \frac{\partial}{\partial m} H)$	if $H(m) = \mathsf{T}$	TSU5
$(x \unlhd f.m \unrhd y) /_f H = \mathsf{tau} \circ (y /_f \frac{\partial}{\partial m} H)$	if $H(m) = \mathsf{F}$	TSU6
$(x \unlhd f.m \unrhd y) /_f H = \mathsf{D}$	if $H(m) = \mathsf{B}$	TSU7

Table 8: Axioms for abstraction operators

$\tau_\iota(\mathsf{S}) = \mathsf{S}$		TT1
$\tau_\iota(\mathsf{D}) = \mathsf{D}$		TT2
$\tau_\iota(\iota \circ x) = \tau_\iota(x)$		TT3
$\tau_\iota(x \unlhd a \unrhd y) = \tau_\iota(x) \unlhd a \unrhd \tau_\iota(y)$	if $a \neq \iota$	TT4
$\bigwedge_{n \geq 0} \tau_\iota(\pi_n(x)) = \tau_\iota(\pi_n(y)) \Rightarrow \tau_\iota(x) = \tau_\iota(y)$		TT5

$f \neq g$ are not processed. Axiom TSU7 expresses that inaction occurs when a basic action to be processed is not accepted.

Let T stand for either BTA, BTA+REC or BTA+REC+AIP. Then we will write T+TSU for T, taking the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$ for $\mathcal{A}$, extended with the use operators and the axioms from Table 7.

For each $\iota \in \mathcal{I}$, we introduce the unary *abstraction* operator $\tau_\iota : \mathbf{T} \to \mathbf{T}$ to conceal the presence of internal action ι in the case where its presence does not matter. The axioms for the abstraction operators are given in Table 8. In this table, a stands for an arbitrary action from $\mathcal{A} \cup \mathcal{I}$.

A main difference between the version of the use mechanism introduced here and the version of the use mechanism introduced in [10] is that the former version does not incorporate abstraction and the latter version incorporates abstraction.

Let T stand for either BTA, BTA+REC, BTA+REC+AIP, BTA+TSU, BTA+REC+TSU or BTA+REC+AIP+TSU. Then we write T+ABSTR for T extended with the abstraction operators and the axioms from Table 8.

For each $\iota \in \mathcal{I}$, the equation $\tau_\iota(\iota^\omega) = \mathsf{D}$ is derivable from the axioms of BTA+REC+AIP+ABSTR.

9 Instruction Register File Services

In this section, we describe services that make up register files with a finite set of registers that can contain instructions from a finite set of core primitive instructions. These services will be used in Section 10.

It is assumed that a fixed but arbitrary set $I \subseteq \mathbb{N}$ such that $I = [1, n]$ for some $n \in \mathbb{N}$ and a fixed but arbitrary finite set $U \subseteq \mathfrak{I}_\mathsf{c}$ have been given. The set I is considered to contain the positions of the registers in the instruction register file and the set U is considered to contain the instructions that can be put in those registers.

We write *IRFS* for the set $\bigcup_{I' \subseteq I} I' \to U$. The members of *IRFS* are considered to be the possible instruction register file states. It is assumed that a fixed but arbitrary bijection $\theta : \mathit{IRFS} \to [1, \mathrm{card}(\mathit{IRFS})]$ has been given.

The instruction register file services accept the following methods:

- for each $i \in I$ and $u \in U$, a *register put method* $\mathsf{put}{:}i{:}u$;
- for each $j \in \mathrm{range}(\theta)$, a *register file test method* $\mathsf{eq}{:}j$.

We write $\mathcal{M}_\mathsf{irf}$ for the set $\{\mathsf{put}{:}i{:}u \mid i \in I \wedge u \in U\} \cup \{\mathsf{eq}{:}j \mid j \in \mathrm{range}(\theta)\}$.

It is assumed that $\mathcal{M}_\mathsf{irf} \subseteq \mathcal{M}$.

The methods accepted by instruction register file services can be explained as follows:

- $\mathsf{put}{:}i{:}u$: the contents of register i becomes instruction u and the reply is T;
- $\mathsf{eq}{:}j$: if the state of the instruction register file equals $\theta^{-1}(j)$, then nothing changes and the reply is T; otherwise nothing changes and the reply is F.

Let $s \in \mathit{IRFS} \cup \{\uparrow\}$, where $\uparrow \notin \mathit{IRFS}$. Then the *instruction register file service* with initial state s, written $\mathcal{IRF}_s$, is the service $(\mathit{IRFS}, \mathit{eff}, \mathit{yld}, s)$, where the functions *eff* and *yld* are defined as follows ($\sigma \in \mathit{IRFS} \cup \{\uparrow\}$):[6]

$$
\begin{array}{lll}
\mathit{eff}(\mathsf{put}{:}i{:}u, \sigma) = \sigma \oplus [i \mapsto u]\ , & \mathit{yld}(\mathsf{put}{:}i{:}n, \sigma) = \mathsf{T}\ , & \\
\mathit{eff}(\mathsf{eq}{:}j, \sigma) = \sigma\ , & \mathit{yld}(\mathsf{eq}{:}j, \sigma) = \mathsf{T} & \text{if } \theta(\sigma) = j\ , \\
 & \mathit{yld}(\mathsf{eq}{:}j, \sigma) = \mathsf{F} & \text{if } \theta(\sigma) \neq j\ , \\
\mathit{eff}(m, \sigma) = \uparrow \quad \text{if } m \notin \mathcal{M}_{\mathsf{irf}}\ , & \mathit{yld}(m, \sigma) = \mathsf{B} & \text{if } m \notin \mathcal{M}_{\mathsf{irf}}\ , \\
\mathit{eff}(m, \uparrow) = \uparrow\ , & \mathit{yld}(m, \uparrow) = \mathsf{B}\ . &
\end{array}
$$

10 Program Synthesis

In order to establish a connection between collections of instruction sequence fragments and instruction sequences, we show in this section that, for each possible joint behaviour of a collection of instruction sequence fragments, a single instruction sequence can be synthesized from the collection that produces on execution essentially the behaviour in question by making use of an instruction register file service. More precisely, we show that, for each PGA$_\mathrm{p}$ program P and polyadic instruction sequence vector α, a PGA program P' can be synthesized from P and α such that, for all relevant instruction register file states σ, $|P'| /_{\mathsf{irf}} \mathcal{IRF}_\sigma$ coincides with $|P|_\sigma(\alpha)$ after the presence of the internal actions tau and gl has been concealed.

Let P be a PGA$_\mathrm{p}$ program and α be a polyadic instruction sequence vector. The general idea is that:

- each polyadic instruction sequence in α is translated into a PGA$_\mathrm{p}$ program and an appropriate finite collection of instances of this PGA$_\mathrm{p}$ program in which occurrences of get instructions are replaced by core primitive instructions is generated;

- P and all the generated programs are translated into PGLC$_\mathrm{p}$ programs and these PGLC$_\mathrm{p}$ programs are concatenated;

- the resulting PGLC$_\mathrm{p}$ program is translated into a PGLD$_\mathrm{p}$ program and this program is translated into a PGLD program by replacing all occurrences of the supplementary instructions by core primitive instructions as follows:

[6]We use the following notation for functions: $[\,]$ for the empty function; $[d \mapsto r]$ for the function f with domain$(f) = \{d\}$ such that $f(d) = r$; and $f \oplus g$ for the function h with domain$(h) =$ domain$(f) \cup$ domain(g) such that, for all $d \in$ domain(h), $h(d) = f(d)$ if $d \notin$ domain(g) and $h(d) = g(d)$ otherwise.

- a switch-over instruction $\#\#\#i$ is replaced by an absolute jump instruction whose effect is a jump to the beginning of an appended instruction sequence whose execution leads, after the state of the instruction register file has been found by a linear search, to a jump to the beginning of the right instance of the $\mathrm{PGA_p}$ program that corresponds to the ith polyadic instruction sequence in α;
- a put instruction $\$\mathsf{put}{:}i{:}u$ is simply replaced by the plain basic instruction $\mathsf{irf.put}{:}i{:}u$;
- a get instruction $\$\mathsf{get}{:}i$ is simply replaced by the absolute jump instruction whose effect is a jump to the position of the instruction itself.

A collection of instances of the $\mathrm{PGA_p}$ program corresponding to a polyadic instruction sequence in α is considered appropriate if it includes all instances that may become the one being executed. P and all the generated programs are translated into $\mathrm{PGLC_p}$ programs because $\mathrm{PGLC_p}$ programs are relocatable: they can be concatenated without disturbing the meaning of jump instructions. The $\mathrm{PGLC_p}$ program resulting from the concatenation is translated into a $\mathrm{PGLD_p}$ program before the supplementary instructions are replaced because the replacement of a switch-over instruction by an absolute jump instruction is simpler than its replacement by a relative jump instruction.

Following the general idea outlined above, we will define a function `pgap2pgld` that yields, for each $\mathrm{PGA_p}$ program P, a function that gives, for each polyadic instruction sequence vector α, a PGLD program P' such that, for each relevant instruction register file service state σ, $|\mathtt{pgld2pga}(P')| \mathbin{/_{\mathsf{irf}}} \mathcal{IRF}_\sigma$ coincides with $|P|_\sigma(\alpha)$ after the presence of the internal actions tau and gl has been concealed.

The function `pgap2pgld` from the set of all $\mathrm{PGA_p}$ programs to the set all functions from the set of all polyadic instruction sequence vectors to the set of all PGLD programs is defined as follows:

$$
\begin{aligned}
&\mathtt{pgap2pgld}(x)(\alpha) = \\
&\quad \mathit{translate}(\mathtt{pglc2pgld}(\mathit{expand}(x)(\alpha)))\ ; \\
&\quad +\mathsf{irf.eq}{:}1\ ;\ \#\#l_{1,1}\ ;\ \ldots\ ;\ +\mathsf{irf.eq}{:}n'\ ;\ \#\#l_{1,n'}\ ; \\
&\qquad \vdots \\
&\quad +\mathsf{irf.eq}{:}1\ ;\ \#\#l_{n,1}\ ;\ \ldots\ ;\ +\mathsf{irf.eq}{:}n'\ ;\ \#\#l_{n,n'}\ ,
\end{aligned}
$$

where $n = \mathrm{len}(\alpha)$, $n' = \max(\mathrm{range}(\theta))$, the function *expand* from the set of all $\mathrm{PGA_p}$ programs to the set all functions from the set of all polyadic instruction sequence vectors to the set of all PGLCp programs is defined as follows:

$$\begin{aligned}
&\mathit{expand}(x)(\alpha) = \\
&\quad \texttt{pga2pglc}(x)\,; \\
&\quad \texttt{pga2pglc}(\mathit{gen}(\alpha, 1, \theta^{-1}(1)))\,; \ldots ; \texttt{pga2pglc}(\mathit{gen}(\alpha, 1, \theta^{-1}(n')))\,; \\
&\qquad \vdots \\
&\quad \texttt{pga2pglc}(\mathit{gen}(\alpha, n, \theta^{-1}(1)))\,; \ldots ; \texttt{pga2pglc}(\mathit{gen}(\alpha, n, \theta^{-1}(n')))\;,
\end{aligned}$$

where $n = \mathrm{len}(\alpha)$, $n' = \max(\mathrm{range}(\theta))$, and the function *gen* from the set of all polyadic instruction sequence vectors, the set of all natural numbers and the set of all instruction register file states to the set of all $\mathrm{PGA_p}$ programs is defined as follows:

$$\begin{aligned}
&\mathit{gen}(\alpha, i, \sigma) = \mathit{prj}_{\mathit{pgn}(\alpha,i)}(\mathit{pg}(\alpha, i))[\sigma] && \text{if } 1 \leq i \leq \mathrm{len}(\alpha) \land \mathit{valid}(\alpha, i, \sigma)\;, \\
&\mathit{gen}(\alpha, i, \sigma) = \#0 && \text{if } 1 \leq i \leq \mathrm{len}(\alpha) \land \neg\mathit{valid}(\alpha, i, \sigma)\;, \\
&\mathit{gen}(\alpha, i, \sigma) = ! && \text{if } i = 0 \lor i > \mathrm{len}(\alpha)\;,
\end{aligned}$$

the function *translate* from the set of all $\mathrm{PGLD_p}$ programs to the set of all PGLD programs is defined as follows:

$$\mathit{translate}(u_1\,; \ldots ; u_k) = \psi_1(u_1)\,; \ldots ; \psi_1(u_k)\;,$$

where the functions ψ_j from the set of all primitive instructions of $\mathrm{PGLD_p}$ to the set of all primitive instructions of PGLD are defined as follows ($1 \leq j \leq k$):

$$\begin{aligned}
&\psi_j(\#\#\#i) &&= \#\#l_i && \text{if } 1 \leq i \leq \mathrm{len}(\alpha)\;, \\
&\psi_j(\#\#\#i) &&= ! && \text{if } i = 0 \lor i > \mathrm{len}(\alpha)\;, \\
&\psi_j(\$\mathsf{put}{:}i{:}u) &&= \mathsf{irf.put}{:}i{:}u\;, \\
&\psi_j(\$\mathsf{get}{:}i) &&= \#\#j\;, \\
&\psi_j(u) &&= u && \text{if } u \text{ is not a supplementary basic instruction}\;,
\end{aligned}$$

where for each $i \in [1, \mathrm{len}(\alpha)]$:

$$\begin{aligned}
l_i = {} & \mathrm{len}(\texttt{pga2pglc}(x)) \\
& + \sum_{h \in [1,\mathrm{len}(\alpha)], h' \in \mathrm{range}(\theta)} \mathrm{len}(\texttt{pga2pglc}(\mathit{prj}_{\mathit{pgn}(\alpha,h)}(\mathit{pg}(\alpha,h))[\theta^{-1}(h')])) \\
& + 2 \cdot \max(\mathrm{range}(\theta)) \cdot (i-1)\ ,
\end{aligned}$$

and for each $i \in [1, \mathrm{len}(\alpha)]$ and $j \in \mathrm{range}(\theta)$:

$$\begin{aligned}
l_{i,j} = {} & \mathrm{len}(\texttt{pga2pglc}(x)) \\
& + \sum_{h \in [1,i-1], h' \in \mathrm{range}(\theta)} \mathrm{len}(\texttt{pga2pglc}(\mathit{prj}_{\mathit{pgn}(\alpha,h)}(\mathit{pg}(\alpha,h))[\theta^{-1}(h')])) \\
& + \sum_{h' \in [1,j-1]} \mathrm{len}(\texttt{pga2pglc}(\mathit{prj}_{\mathit{pgn}(\alpha,i)}(\mathit{pg}(\alpha,i))[\theta^{-1}(h')]))\ .
\end{aligned}$$

The following theorem states rigorously that, for any $\mathrm{PGA_p}$ program P and polyadic instruction sequence vector α, for all relevant instruction register file states σ, $|\texttt{pgld2pga}(\texttt{pgap2pgld}(P)(\alpha))| /_{\mathsf{irf}}\, \mathcal{IRF}_\sigma$ coincides with $|P|_\sigma(\alpha)$ after the presence of the internal actions tau and gl has been concealed.

Theorem 1 *Let P be a* $\mathrm{PGA_p}$ *program and α be a polyadic instruction sequence vector, and let n be the highest number occurring in instructions of the form* $\$\mathsf{put}{:}i{:}u$ *or* $\$\mathsf{get}{:}i$ *in P or α. Take the interval $[1, n]$ for I and the set of all core primitive instructions occurring in instructions of the form* $\$\mathsf{put}{:}i{:}u$ *in P or α for U, and let $\sigma \in \mathit{IRFS}$. Then* $\tau_{\mathsf{tau}}(|\texttt{pgld2pga}(\texttt{pgap2pgld}(P)(\alpha))| /_{\mathsf{irf}}\, \mathcal{IRF}_\sigma) = \tau_{\mathsf{tau}}(\tau_{\mathsf{gl}}(|P|_\sigma(\alpha)))$.

Proof: The proof of Theorem 1 follows the same line as the proof of Theorem 1 from [8] given in that paper. Here, we give only a brief outline of the proof of the current theorem.

The proof of this theorem proceeds as follows: (i) we give a set T of closed terms of sort $\mathbf{T}$ with $\tau_{\mathsf{tau}}(\tau_{\mathsf{gl}}(|P|_\sigma(\alpha))) \in T$, a set T' of closed terms of sort $\mathbf{T}$ with $\tau_{\mathsf{tau}}(|\texttt{pgld2pga}(\texttt{pgap2pgld}(P)(\alpha))| /_{\mathsf{irf}}\, \mathcal{IRF}_\sigma) \in T'$, and a bijection $\beta : T \to T'$; (ii) we show that there exists a set E consisting of one derivable equation $p = p'$ for each $p \in T$ such that, for all equations $p = p'$ in E:

- $\beta(p) = p''$ is a derivable equation if p'' is p' with, for all $q \in T$, all occurrences of q in p' replaced by $\beta(q)$;

- $p' \in T$ only if p' can be rewritten to a $q' \notin T$ using the equations in E from left to right.

The latter means that $\tau_{\mathsf{tau}}(|\mathtt{pgld2pga}(\mathtt{pgap2pgld}(P)(\alpha))| \mathbin{/_{\mathsf{irf}}} \mathcal{IRF}_\sigma)$ and $\tau_{\mathsf{tau}}(\tau_{\mathsf{gl}}(|P|_\sigma(\alpha)))$ denote solutions of the same guarded recursive specification. Because guarded recursive specifications have unique solutions, it follows immediately that $\tau_{\mathsf{tau}}(|\mathtt{pgld2pga}(\mathtt{pgap2pgld}(P)(\alpha))| \mathbin{/_{\mathsf{irf}}} \mathcal{IRF}_\sigma) = \tau_{\mathsf{tau}}(\tau_{\mathsf{gl}}(|P|_\sigma(\alpha)))$. □

In the proof outlined above, an apposite indexing of the closed terms in the sets T and T' facilitates the definition of the bijection β. Yet, this definition is much more complicated than the definition of the bijection needed in the proof from [8] referred to.

The definition of `pgap2pgld` shows that the synthesis of single instruction sequences from collections of instruction sequence fragments is fairly complicated.

11 Conclusions

We have given a formalization of a simple mechanism by which several instruction sequence fragments can produce a joint behaviour. Thread extraction provides the possible joint behaviours of a collection of instruction sequence fragments. We have shown that, for each possible joint behaviour of a collection of instruction sequence fragments, a single instruction sequence can be synthesized from the collection that produces on execution essentially the behaviour in question by making use of a service. This program synthesis is reminiscent of the service-based variant of projection semantics for program notations used in [5]. The program synthesis shows that it is a non-trivial matter to explain by means of a translation into a single instruction sequence what takes place on execution of a collection of instruction sequence fragments.

In this paper, an instruction sequence fragment in a collection of instruction sequence fragments has two attributes: an ordinal index (position) and a program notation index. We have also mentioned that a nominal index (label) could be an optional attribute. Many other attributes that are relevant in practice can be imagined, e.g. modification date, author, tester, owner, user, and security level. In this paper, we have restricted ourselves to attributes that are indispensable for a theoretical understanding.

The question arises whether all aspects of the mechanism formalized in this paper can be dealt with at the level of threads. This issue has been investigated in [9]. It happens that not all aspects can be dealt with at the level of threads. In particular, the ability to replace special instructions in an instruction sequence fragment by different ordinary instructions every time execution is switched over to that fragment cannot be dealt with at the level of threads. Threads turn out to be too abstract to deal with the mechanism in full.

It is sometimes suggested that program algebra is closely related to Kleene algebra [13] and omega algebra [12] – an extension of Kleene algebra with an infinite iteration operator. In program algebra, programs are considered at a more concrete level than in Kleene algebra and omega algebra. For instance, Kleene algebra and omega algebra are not concerned with instruction sequences, and program algebra is not concerned with non-determinism.

Acknowledgements

The work presented in this paper has been partly carried out in the framework of the Jacquard-project Symbiosis, which is funded by the Netherlands Organisation for Scientific Research (NWO).

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[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, editors, *Proceedings 30th ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.

[3] J. A. Bergstra and I. Bethke. Predictable and reliable program code: Virtual machine based projection semantics. In J. A. Bergstra and M. Burgess, editors, *Handbook of Network and Systems Administration*, pages 653–685. Elsevier, Amsterdam, 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. Instruction sequences with indirect jumps. *Scientific Annals of Computer Science*, 17:19–46, 2007.

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

[7] J. A. Bergstra and C. A. Middelburg. Program algebra with a jump-shift instruction. *Journal of Applied Logic*, 6(4):553–563, 2008.

[8] J. A. Bergstra and C. A. Middelburg. Instruction sequences with dynamically instantiated instructions. *Fundamenta Informaticae*, 96(1–2):27–48, 2009.

[9] J. A. Bergstra and C. A. Middelburg. Thread algebra for poly-threading. *Formal Aspects of Computing*, 23(4):567–538, 2011.

[10] J. A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.

[11] J. Bishop and N. Horspool. *C# Concisely.* Addison-Wesley, Reading, MA, 2004.

[12] E. Cohen. Seperation and reduction. In R. Backhouse and J. N. Oliveira, editors, *MCP 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 45–59. Springer-Verlag, 2000.

[13] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[14] A. Ponse and M. B. van der Zwaag. An introduction to program and thread algebra. In A. Beckmann et al., editors, *CiE 2006*, volume 3988 of *Lecture Notes in Computer Science*, pages 445–458. Springer-Verlag, 2006.

[15] D. Sannella and A. Tarlecki. Algebraic preliminaries. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, pages 13–30. Springer-Verlag, Berlin, 1999.

[16] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, Amsterdam, 1990.