Theory Comput Syst (2012) 50:313–328
DOI 10.1007/s00224-010-9301-8

On the Expressiveness of Single-Pass Instruction Sequences

J.A. Bergstra · C.A. Middelburg

Published online: 11 November 2010

Abstract We perceive programs as single-pass instruction sequences. A single-pass instruction sequence under execution is considered to produce a behaviour to be controlled by some execution environment. Threads as considered in basic thread algebra model such behaviours. We show that all regular threads, i.e. threads that can only be in a finite number of states, can be produced by single-pass instruction sequences without jump instructions if use can be made of Boolean registers. We also show that, in the case where goto instructions are used instead of jump instructions, a bound to the number of labels restricts the expressiveness.

Keywords Single-pass instruction sequence · Regular thread · Expressiveness · Jump-free instruction sequence

1 Introduction

The work presented in this paper is part of a research program which is concerned with different subjects from the theory of computation and the area of computer architectures where we come across the relevancy of the notion of instruction sequence. The working hypothesis of this research program is that this notion is a central notion of computer science. It is clear that instruction sequence is a key concept in practice, but strangely enough it has as yet not come prominently into the picture in theoretical circles.

J.A. Bergstra (✉) · C.A. Middelburg
Informatics Institute, Faculty of Science, University of Amsterdam, Science Park 904,
1098 XH Amsterdam, The Netherlands
e-mail: J.A.Bergstra@uva.nl

C.A. Middelburg
e-mail: C.A.Middelburg@uva.nl

Program algebra [3], which is intended as a setting suited for developing theory from the above-mentioned working hypothesis, is taken for the basis of the development of theory under the research program. 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. A single-pass instruction sequence under execution is considered to produce a behaviour to be controlled by some execution environment. Threads as considered in basic thread algebra [3] model such behaviours: upon each action performed by a thread, a reply from the execution environment determines how the thread proceeds.[1] A thread may make use of services, i.e. components of the execution environment.

Each Turing machine can be simulated by means of a thread that makes use of a service. The thread and service correspond to the finite control and tape of the Turing machine. The threads that correspond to the finite controls of Turing machines are examples of regular threads, i.e. threads that can only be in a finite number of states. The behaviours of all single-pass instruction sequences considered in program algebra are regular threads and each regular thread is produced by some single-pass instruction sequence. In this paper, we show that each regular thread can be produced by some single-pass instruction sequence without jump instructions if use can be made of services that make up Boolean registers.

The primitive instructions of program algebra include jump instructions. An interesting variant of program algebra is obtained by leaving out jump instructions and adding labels and goto instructions. It is easy to see that each regular thread can also be produced by some single-pass instruction sequence with labels and goto instructions. In this paper, we show that a bound to the number of labels restricts the expressiveness of this variant.

As part of the research program of which the work presented in this paper is part, issues concerning the following subjects from the theory of computation have been investigated from the viewpoint that a program is an instruction sequence: semantics of programming languages [4, 12], expressiveness of programming languages [9, 23], computability [10, 13], and computational complexity [7]. Performance related matters of instruction sequences have also been investigated in the spirit of the theory of computation [11, 12]. In the area of computer architectures, basic techniques aimed at increasing processor performance have been studied as part of this research program (see e.g. [5, 8]).

The work referred to above provides evidence for our hypothesis that the notion of instruction sequence is a central notion of computer science. To say the least, it shows that instruction sequences are relevant to diverse subjects. In addition, it is to be expected that the emerging developments with respect to techniques for high-performance program execution on classical or non-classical computers require that programs are considered at the level of instruction sequences. All this has motivated us to continue the above-mentioned research program with the work on expressiveness presented in this paper.

[1] In [3], basic thread algebra is introduced under the name basic polarized process algebra.

This paper is organized as follows. First, we review basic thread algebra and program algebra (Sects. 2 and 3). Next, we present a mechanism for interaction of threads with services and give a description of Boolean register services (Sects. 4 and 5). After that, we show that each regular thread can be produced by some single-pass instruction sequence without jump instructions if use can be made of Boolean register services (Sect. 6). Then, we introduce the variant of program algebra obtained by leaving out jump instructions and adding labels and goto instructions (Sect. 7). Following this, we show that a bound to the number of labels restricts the expressiveness of this variant (Sect. 8). Finally, we make some concluding remarks (Sect. 9).

2 Basic Thread Algebra

In this section, we review BTA (Basic Thread Algebra), which is concerned with the behaviours that sequential programs exhibit on execution. These behaviours are called threads.

In BTA, it is assumed that a fixed but arbitrary set $\mathcal{A}$ of *basic actions* has been given. 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. To simplify matters, there are only two possible replies: T and F.

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

- the *deadlock* constant $\mathsf{D} : \mathbf{T}$;
- the *termination* constant $\mathsf{S} : \mathbf{T}$;
- for each $a \in \mathcal{A}$, 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. We introduce *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, respectively. This implies 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].

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 1. 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

Table 1 Axioms for guarded recursion

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

Table 2 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

arbitrary variable of sort **T**, an arbitrary BTA term of sort **T** and an arbitrary guarded recursive specification over BTA, respectively. Side conditions are added to restrict what X, t_X and E stand for.

Closed terms that denote the same infinite thread cannot always be proved equal by means of the axioms given in Table 1. 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 it has performed n actions. 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 2.

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. 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.

The intuition is that the execution of a basic instruction a produces either T or F at its completion. In the case of a positive test instruction $+a$, a is executed and execution proceeds with the next primitive instruction if T is produced. Otherwise, the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one. If there is no next instruction to be executed, deadlock occurs. In the case of a negative test instruction $-a$, the role of the value produced is reversed. In the case of a plain basic instruction a, 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 instruction. If l equals 0 or the l-th next instruction does

Table 3 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 4 Defining equations for thread extraction operation

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

not exist, deadlock 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.

A closed PGA term is considered to denote a non-empty, finite or eventually periodic infinite sequence of primitive instructions.[2] Closed PGA terms are considered equal if they denote the same instruction sequence. The axioms for instruction sequence equivalence are given in Table 3. 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.

The repetition operator renders backward jump instructions superfluous. In [3], it is shown how programs in a program notation that is close to existing assembly languages with forward and backward jump instructions can be translated into closed PGA terms.

The behaviours of the instruction sequences denoted by closed PGA terms are considered threads, with basic instructions taken for basic actions. The *thread extraction* operation $\lvert\_\rvert$ determines, for each closed PGA term P, a closed term of BTA with guarded recursion that denotes the behaviour of the instruction sequence denoted by P. The thread extraction operation is defined by the equations given in Table 4 (for $a \in \mathfrak{A}$, $l \in \mathbb{N}$ and $u \in \mathfrak{I}$) and the rule that $\lvert \#l\,;\,X\rvert = \mathsf{D}$ if $\#l$ is the beginning of an infinite jump chain. This rule is formalized in e.g. [9].

[2] An eventually periodic infinite sequence is an infinite sequence with only finitely many distinct suffixes.

4 Interaction of Threads with Services

A thread may make use of services. That is, a thread may perform an action for the purpose of interacting with a service that takes the action as a command 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. In this section, we introduce the use operators, which are concerned with this kind of interaction between threads and services.

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

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:

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

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.

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 $(S, \mathit{eff}, \mathit{yld}, \mathit{eff}(m, s_0))$; and the *reply* of H after processing m, written $H(m)$, is $\mathit{yld}(m, s_0)$.

When a thread makes a request to service H 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.

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 5.

Intuitively, $p /_f H$ is the thread that results from processing all actions performed by thread p that are of the form $f.m$ by service H. When an action of the form $f.m$ performed by thread p is processed by service H, the postconditional composition concerned is eliminated on the basis of the reply value produced. No internal action is left as a trace of the processed action, like with the use operators found in papers on thread interleaving (see e.g. [6]).

Combining TSU2 and TSU7, we obtain $\bigwedge_{n \geq 0} \pi_n(x) /_f H = \mathsf{D} \Rightarrow x /_f H = \mathsf{D}$.

Table 5 Axioms for use operators

$\mathsf{S} /_f H = \mathsf{S}$		TSU1
$\mathsf{D} /_f H = \mathsf{D}$		TSU2
$(x \trianglelefteq g.m \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq g.m \trianglerighteq (y /_f H)$	if $f \neq g$	TSU3
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = x /_f \frac{\partial}{\partial m} H$	if $H(m) = \mathsf{T}$	TSU4
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = y /_f \frac{\partial}{\partial m} H$	if $H(m) = \mathsf{F}$	TSU5
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{D}$	if $H(m) = \mathsf{B}$	TSU6
$\bigwedge_{n \geq 0} \pi_n(x) /_f H = \pi_n(y) /_f H \Rightarrow x /_f H = y /_f H$		TSU7

5 Instruction Sequences Acting on Boolean Registers

Our study of jump-free instruction sequences in Sect. 6 is concerned with instruction sequences that act on Boolean registers. In this section, we describe services that make up Boolean registers.

A Boolean register service accepts the following methods:

- a *set to true method* set:T;
- a *set to false method* set:F;
- a *get method* get.

We write $\mathcal{M}_{\mathsf{BR}}$ for the set $\{\mathsf{set{:}T}, \mathsf{set{:}F}, \mathsf{get}\}$. It is assumed that $\mathcal{M}_{\mathsf{BR}} \subseteq \mathcal{M}$.

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

- set:T: the contents of the Boolean register becomes T and the reply is T;
- set:F: the contents of the Boolean register becomes F and the reply is F;
- get: nothing changes and the reply is the contents of the Boolean register.

Let $s \in \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$. Then the *Boolean register service* with initial state s, written BR_s, is the service $(\{\mathsf{T}, \mathsf{F}, \mathsf{B}\}, \mathit{eff}, \mathit{eff}, s)$, where the function eff is defined as follows ($b \in \{\mathsf{T}, \mathsf{F}\}$):

$$\begin{aligned} &\mathit{eff}(\mathsf{set{:}T}, b) = \mathsf{T}, \qquad \mathit{eff}(m, b) = \mathsf{B} \quad \text{if } m \notin \mathcal{M}_{\mathsf{BR}},\\ &\mathit{eff}(\mathsf{set{:}F}, b) = \mathsf{F}, \qquad \mathit{eff}(m, \mathsf{B}) = \mathsf{B}.\\ &\mathit{eff}(\mathsf{get}, b) = b, \end{aligned}$$

Notice that the effect and yield functions of a Boolean register service are the same.

6 Jump-Free Instruction Sequences

In this section, we show that each thread that can only be in a finite number of states can be produced by some single-pass instruction sequence without jump instructions if use can be made of Boolean register services.

First, we make precise what it means that a thread can only be in a finite number of states. We assume that a fixed but arbitrary model $\mathfrak{M}$ of BTA extended with guarded recursion and the use mechanism has been given, we use the term thread only for the

elements from the domain of $\mathfrak{M}$, and we denote the interpretations of constants and operators in $\mathfrak{M}$ by the constants and operators themselves.

Let p be a thread. Then the set of *states* or *residual threads* of p, written $Res(p)$, is inductively defined as follows:

- $p \in Res(p)$;
- if $q \trianglelefteq a \trianglerighteq r \in Res(p)$, then $q \in Res(p)$ and $r \in Res(p)$.

We say that p is a *regular* thread if $Res(p)$ is finite.

We will make use of the fact that being a regular thread coincides with being the solution of a finite guarded recursive specification of a restricted form.

A *linear recursive specification* over BTA is a guarded recursive specification $E = \{X = t_X \mid X \in V\}$, where each t_X is a term of the form D, S or $Y \trianglelefteq a \trianglerighteq Z$ with $Y, Z \in V$.

Proposition 1 *Let p be a thread. Then p is a regular thread iff there exists a finite linear recursive specification E and a variable $X \in \mathrm{V}(E)$ such that p is the solution of E for X.*

Proof This proposition generalizes Theorem 1 from [23] from the projective limit model to an arbitrary model. However, the proof of that theorem is applicable to any model. □

In the proof of the next theorem, we associate a closed PGA term P in which jump instructions do not occur with a finite linear recursive specification

$$E = \left\{X_i = X_{l(i)} \trianglelefteq a_i \trianglerighteq X_{r(i)} \mid i \in [1, n]\right\} \cup \{X_{n+1} = \mathsf{S}, X_{n+2} = \mathsf{D}\}.$$

In P, a number of Boolean register services is used for specific purposes. The purpose of each individual Boolean register is reflected in the focus that serves as its name:

- for each $i \in [1, n+2]$, $\mathsf{s}{:}i$ serves as the name of a Boolean register that is used to indicate whether the current state of $\langle X_1 | E\rangle$ is $\langle X_i | E\rangle$;
- rt serves as the name of a Boolean register that is used to indicate whether the reply upon the action performed by $\langle X_1 | E\rangle$ in its current state is T;
- rf serves as the name of a Boolean register that is used to indicate whether the reply upon the action performed by $\langle X_1 | E\rangle$ in its current state is F;
- e serves as the name of a Boolean register that is used to achieve that instructions not related to the current state of $\langle X_1 | E\rangle$ are passed correctly;
- f serves as the name of a Boolean register that is used to achieve with the instruction $+\mathsf{f.set{:}F}$ that the following instruction is skipped.

Now we turn to the theorem announced above. It states rigorously that the solution of every finite linear recursive specification can be produced by an instruction sequence without jump instructions if use can be made of Boolean register services.

Theorem 1 *Let a finite linear recursive specification*

$$E = \left\{X_i = X_{l(i)} \trianglelefteq a_i \trianglerighteq X_{r(i)} \mid i \in [1, n]\right\} \cup \{X_{n+1} = \mathsf{S}, X_{n+2} = \mathsf{D}\}$$

be given. Then there exists a closed PGA *term* P *in which jump instructions do not occur such that*

$$\langle X_1|E\rangle = ((((\ldots(|P|\mathbin{/_{\mathsf{s}:1}} BR_{\mathsf{F}})\ldots\mathbin{/_{\mathsf{s}:n+2}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{rt}}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{rf}}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{e}}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{f}}} BR_{\mathsf{F}}.$$

Proof We associate a closed PGA term P in which jump instructions do not occur with E as follows:

$$P = \mathsf{s}{:}1.\mathsf{set}{:}\mathsf{T}\,;\,(Q_1\,;\ldots;\,Q_{n+1})^{\omega},$$

where, for each $i \in [1,n]$:

$$\begin{aligned} Q_i = {} & +\mathsf{s}{:}i.\mathsf{get}\,;\,\mathsf{e}.\mathsf{set}{:}\mathsf{T}\,;\\ & +\mathsf{s}{:}i.\mathsf{get}\,;\,\mathsf{s}{:}i.\mathsf{set}{:}\mathsf{F}\,;\\ & +\mathsf{e}.\mathsf{get}\,;\,-a_i\,;\,+\mathsf{f}.\mathsf{set}{:}\mathsf{F}\,;\,\mathsf{rt}.\mathsf{set}{:}\mathsf{T}\,;\\ & +\mathsf{e}.\mathsf{get}\,;\,+\mathsf{rt}.\mathsf{get}\,;\,+\mathsf{f}.\mathsf{set}{:}\mathsf{F}\,;\,\mathsf{rf}.\mathsf{set}{:}\mathsf{T}\,;\\ & +\mathsf{rt}.\mathsf{get}\,;\,\mathsf{s}{:}l(i).\mathsf{set}{:}\mathsf{T}\,;\\ & +\mathsf{rf}.\mathsf{get}\,;\,\mathsf{s}{:}r(i).\mathsf{set}{:}\mathsf{T}\,;\\ & \mathsf{rt}.\mathsf{set}{:}\mathsf{F}\,;\,\mathsf{rf}.\mathsf{set}{:}\mathsf{F}\,;\,\mathsf{e}.\mathsf{set}{:}\mathsf{F}, \end{aligned}$$

and

$$Q_{n+1} = +\mathsf{s}{:}n{+}1.\mathsf{get}\,;\,!.$$

We use the following abbreviations (for $i \in [1,n+1]$ and $j \in [1,n+2]$):

P'_i for $Q_i\,;\ldots;\,Q_{n+1}\,;\,(Q_1\,;\ldots;\,Q_{n+1})^{\omega}$;

$|P'_i|^{\mathrm{br}}_j$ for $((((\ldots(|P'_i|\mathbin{/_{\mathsf{s}:1}} BR_{b_1})\ldots\mathbin{/_{\mathsf{s}:n+2}} BR_{b_{n+2}})\mathbin{/_{\mathsf{rt}}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{rf}}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{e}}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{f}}} BR_{\mathsf{F}}$, where $b_j = \mathsf{T}$ and, for each $j' \in [1,n+2]$ such that $j' \neq j$, $b_{j'} = \mathsf{F}$.

From the definition of thread extraction, the definition of Boolean register services, and axiom TSU4, it follows that

$$((((\ldots(|P|\mathbin{/_{\mathsf{s}:1}} BR_{\mathsf{F}})\ldots\mathbin{/_{\mathsf{s}:n+2}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{rt}}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{rf}}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{e}}} BR_{\mathsf{F}})\mathbin{/_{\mathsf{f}}} BR_{\mathsf{F}} = |P'_1|^{\mathrm{br}}_1.$$

This leaves us to show that $\langle X_1|E\rangle = |P'_1|^{\mathrm{br}}_1$.

Using the definition of thread extraction, the definition of Boolean register services, and axioms P0, P2, TSU1, TSU2, TSU4, TSU5 and TSU7, we easily prove the following:

$$|P'_i|^{\mathrm{br}}_j = |P'_{i+1}|^{\mathrm{br}}_j \qquad \text{if } 1 \leq i \leq n \wedge 1 \leq j \leq n+1 \wedge i \neq j \quad (1)$$

$$|P'_i|^{\mathrm{br}}_j = |P'_1|^{\mathrm{br}}_j \qquad \text{if } i = n+1 \wedge 1 \leq j \leq n+1 \wedge i \neq j \quad (2)$$

$$|P'_i|^{\mathrm{br}}_i = |P'_{i+1}|^{\mathrm{br}}_{l(i)} \trianglelefteq a_i \trianglerighteq |P'_{i+1}|^{\mathrm{br}}_{r(i)} \qquad \text{if } 1 \leq i \leq n \quad (3)$$

$$|P'_i|^{\mathrm{br}}_i = \mathsf{S} \qquad \text{if } i = n+1 \quad (4)$$

$$|P'_i|^{\mathrm{br}}_j = \mathsf{D} \qquad \text{if } 1 \leq i \leq n+1 \wedge j = n+2 \quad (5)$$

From Properties 1 and 2, it follows that

$$|P'_i|^{\mathrm{br}}_j = |P'_j|^{\mathrm{br}}_j \quad \text{if } 1 \leq i \leq n+1 \wedge 1 \leq j \leq n+1 \wedge i \neq j.$$

From this and Property 3, it follows that

$$|P'_i|^{\mathrm{br}}_i = |P'_{l(i)}|^{\mathrm{br}}_{l(i)} \trianglelefteq a_i \trianglerighteq |P'_{r(i)}|^{\mathrm{br}}_{r(i)} \quad \text{if } 1 \leq i \leq n.$$

From this and Properties 4 and 5, it follows that $|P'_1|^{\mathrm{br}}_1$ is a solution of E for X_1. Because linear recursive specifications have unique solutions, it follows that $\langle X_1 | E \rangle = |P'_1|^{\mathrm{br}}_1$. □

Theorem 1 goes through in the case where $E = \{X_1 = \mathsf{D}\}$: a witnessing P is $(\mathsf{f.get})^\omega$. It follows from the proof of Proposition 1 given in [23] that, for each regular thread p, either p is the solution of $\{X_1 = \mathsf{D}\}$ for X_1 or there exists a finite linear recursive specification E of the form considered in Theorem 1 such that p is the solution of E for X_1. Hence, we have the following corollary of Proposition 1 and Theorem 1:

Corollary 1 *For each regular thread p, there exists a closed* PGA *term P in which jump instructions do not occur such that p is the thread denoted by*

$$((((\ldots(|P| \mathbin{/_{\mathsf{s}:1}} BR_{\mathsf{F}}) \ldots \mathbin{/_{\mathsf{s}:n+2}} BR_{\mathsf{F}}) \mathbin{/_{\mathsf{rt}}} BR_{\mathsf{F}}) \mathbin{/_{\mathsf{rf}}} BR_{\mathsf{F}}) \mathbin{/_{\mathsf{e}}} BR_{\mathsf{F}}) \mathbin{/_{\mathsf{f}}} BR_{\mathsf{F}}.$$

In other words, each regular thread can be produced by an instruction sequence without jump instructions if use can be made of Boolean register services.

The construction of such instructions sequences given in the proof of Theorem 1 is weakly reminiscent of the construction of structured programs from flow charts found in [14]. However, our construction is more extreme: it yields programs that contain neither unstructured jumps nor a rendering of the conditional and loop constructs used in structured programming.

7 Program Algebra with Labels and Goto's

In this section, we introduce $\mathrm{PGA}_{\mathrm{g}}$, a variant of PGA obtained by leaving out jump instructions and adding labels and goto instructions.

In $\mathrm{PGA}_{\mathrm{g}}$, like in PGA, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of basic instructions has been given. $\mathrm{PGA}_{\mathrm{g}}$ 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 *label instruction* $[l]$;

– for each $l \in \mathbb{N}$, a *goto instruction* #[l];
– a *termination instruction* !.

We write $\mathfrak{I}_{\mathrm{g}}$ for the set of all primitive instructions of $\mathrm{PGA}_{\mathrm{g}}$.

The plain basic instructions, the positive test instructions, the negative test instructions, and the termination instruction are as in PGA. Upon execution, a label instruction [l] is simply skipped. If there is no next instruction to be executed, deadlock occurs. The effect of a goto instruction #[l] is that execution proceeds with the occurrence of the label instruction [l] next following if it exists. If there is no occurrence of the label instruction [l], deadlock occurs.

$\mathrm{PGA}_{\mathrm{g}}$ has a constant u for each $u \in \mathfrak{I}_{\mathrm{g}}$. The operators of $\mathrm{PGA}_{\mathrm{g}}$ are the same as the operators as PGA. Likewise, the axioms of $\mathrm{PGA}_{\mathrm{g}}$ are the same as the axioms as PGA.

Just like in the case of PGA, the behaviours of the instruction sequences denoted by closed $\mathrm{PGA}_{\mathrm{g}}$ terms are considered threads. The behaviours of the instruction sequences denoted by closed $\mathrm{PGA}_{\mathrm{g}}$ terms are indirectly given by the behaviour preserving function `pgag2pga` from the set of all closed $\mathrm{PGA}_{\mathrm{g}}$ terms to the set of all closed PGA terms defined by

$$\begin{aligned}
&\mathtt{pgag2pga}(u_1 \,;\, \ldots \,;\, u_n) = \mathtt{pgag2pga}(u_1 \,;\, \ldots \,;\, u_n \,;\, (\#[1])^{\omega}),\\
&\mathtt{pgag2pga}(u_1 \,;\, \ldots \,;\, u_n \,;\, (u_{n+1} \,;\, \ldots \,;\, u_m)^{\omega})\\
&\quad = \phi_1(u_1) \,;\, \ldots \,;\, \phi_n(u_n) \,;\, (\phi_{n+1}(u_{n+1}) \,;\, \ldots \,;\, \phi_m(u_m))^{\omega},
\end{aligned}$$

where the auxiliary functions $\phi_j : \mathfrak{I}_{\mathrm{g}} \to \mathfrak{I}$ are defined as follows ($1 \leq j \leq m$):

$$\begin{aligned}
\phi_j([l]) &= \#1,\\
\phi_j(\#[l]) &= \#tgt_j(l),\\
\phi_j(u) &= u \quad \text{if } u \text{ is not a label or goto instruction,}
\end{aligned}$$

where

– $tgt_j(l) = i$ if the leftmost occurrence of [l] in $u_j \,;\, \ldots \,;\, u_m \,;\, u_{n+1} \,;\, \ldots \,;\, u_m$ is the i-th instruction;
– $tgt_j(l) = 0$ if there are no occurrences of [l] in $u_j \,;\, \ldots \,;\, u_m \,;\, u_{n+1} \,;\, \ldots \,;\, u_m$.

Let P be a closed $\mathrm{PGA}_{\mathrm{g}}$ term. Then the behaviour of P is $|\mathtt{pgag2pga}(P)|$. The approach to semantics followed here is introduced under the name *projection semantics* in [3]. The function `pgag2pga` is called a *projection*.

8 A Bounded Number of Labels

In this section, we show that a bound to the number of labels restricts the expressiveness of $\mathrm{PGA}_{\mathrm{g}}$. We will refer to $\mathrm{PGA}_{\mathrm{g}}$ terms that do not contain label instructions [l] with $l > k$ as $\mathrm{PGA}_{\mathrm{g}}^{k}$ terms. Moreover, we will write $\mathfrak{I}_{\mathrm{g}}^{k}$ for the set $\mathfrak{I}_{\mathrm{g}} \setminus \{[l] \mid l > k\}$.

We define an alternative projection for closed $\mathrm{PGA}_{\mathrm{g}}^{k}$ terms, which takes into account that these terms contain only label instructions $[l]$ with $1 \leq l \leq k$. The alternative projection $\mathtt{pgag2pga}^{k}$ from the set of all closed $\mathrm{PGA}_{\mathrm{g}}^{k}$ terms to the set of all closed PGA terms is defined by

$$\begin{aligned}
&\mathtt{pgag2pga}^{k}(u_1 ; \ldots ; u_n) = \mathtt{pgag2pga}^{k}(u_1 ; \ldots ; u_n ; (\#[1])^{\omega}),\\
&\mathtt{pgag2pga}^{k}(u_1 ; \ldots ; u_n ; (u_{n+1} ; \ldots ; u_m)^{\omega})\\
&\quad = \psi(u_1, u_2) ; \ldots ; \psi(u_n, u_{n+1}) ; (\psi(u_{n+1}, u_{n+2}) ; \ldots\\
&\qquad ; \psi(u_{m-1}, u_m) ; \psi(u_m, u_{n+1}))^{\omega},
\end{aligned}$$

where the auxiliary function $\psi : \mathfrak{I}_{\mathrm{g}}^{k} \times \mathfrak{I}_{\mathrm{g}}^{k} \to \mathfrak{I}$ is defined as follows:

$$\psi(u', u'') = \psi'(u') ; \#k{+}2 ; \#k{+}2 ; \psi''(u''),$$

where the auxiliary functions $\psi', \psi'' : \mathfrak{I}_{\mathrm{g}}^{k} \to \mathfrak{I}$ are defined as follows:

$$\begin{aligned}
\psi'([l]) &= \#1,\\
\psi'(\#[l]) &= \#l{+}2 \quad \text{if } l \leq k,\\
\psi'(\#[l]) &= \#0 \quad \text{if } l > k,\\
\psi'(u) &= u \quad \text{if } u \text{ is not a label or goto instruction},\\
\psi''([l]) &= (\#k{+}3)^{l-1} ; \#k{-}l{+}1 ; (\#k{+}3)^{k-l},\\
\psi''(u) &= (\#k{+}3)^{k} \quad \text{if } u \text{ is not a label instruction}.
\end{aligned}$$

In order to clarify the alternative projection, we explain how the intended effect of a goto instruction is obtained. If u_j is $\#[l]$, then $\psi'(u_j)$ is $\#l{+}2$. The effect of $\#l{+}2$ is a jump to the l-th instruction in $\psi''(u_{j+1})$ if $j < m$ and a jump to the l-th instruction in $\psi''(u_{n+1})$ if $j = m$. If this instruction is $\#k{-}l{+}1$, then its effect is a jump to the occurrence of $\#1$ that replaces $[l]$. However, if this instruction is $\#k{+}3$, then its effect is a jump to the l-th instruction in $\psi''(u_{j+2})$ if $j < m - 1$, a jump to the l-th instruction in $\psi''(u_{n+1})$ if $j = m - 1$, and a jump to the l-th instruction in $\psi''(u_{n+2})$ if $j = m$.

In the proof of Theorem 2 below, chains of forward jumps are removed in favour of single jumps. The following proposition justifies these removals.

Proposition 2 *For each* PGA *context* $C[\,]$:

$$|C[\#n + 1 ; u_1 ; \ldots ; u_n ; \#m]| = |C[\#m + n + 1 ; u_1 ; \ldots ; u_n ; \#m]|.$$

Proof Contexts of the forms $C[\,]^{\omega} ; Q$ and $P ; C[\,]^{\omega} ; Q$ do not need to be considered because of axiom PGA3. For eight of the remaining twelve forms, the equation to be proved follows immediately from the equations to be proved for the other forms, to wit $\_ ; Q$, $P ; \_ ; Q$, $P ; \_^{\omega}$ and $P ; (Q ; \_)^{\omega}$, the axioms of PGA, the defining equations for thread extraction, and the easy to prove fact that $|P ; \#0| = |P|$.

In the case of the form $\_ \,;\, Q$, the equation concerned is easily proved by induction on n. In the case of the form $P \,;\, \_ \,;\, Q$, only P in which the repetition operator does not occur need to be considered because of axiom PGA3. For such P, the equation concerned is easily proved by induction on the length of P, using the equation proved for the form $\_ \,;\, Q$. In the case of the form $P \,;\, \_^{\omega}$, only P in which the repetition operator does not occur need to be considered because of axiom PGA3. For such P, the equations for the approximating forms $P \,;\, \_^{k}$ are easily proved by induction on k, using the equation proved for the form $P \,;\, \_ \,;\, Q$. From these equations, the equation for the form $P \,;\, \_^{\omega}$ follows using AIP. In the case of the form $P \,;\, (Q \,;\, \_)^{\omega}$, the equation concerned is proved like in the case of the form $P \,;\, \_^{\omega}$. □

The following theorem states rigorously that the projections $\mathtt{pgag2pga}$ and $\mathtt{pgag2pga}^k$ give rise to instruction sequences with the same behaviour.

Theorem 2 *For each closed* $\mathrm{PGA}_{\mathrm{g}}^{k}$ *term* P, $|\mathtt{pgag2pga}(P)| = |\mathtt{pgag2pga}^k(P)|$.

Proof Because $\mathtt{pgag2pga}(u_1 \,;\, \ldots \,;\, u_n) = \mathtt{pgag2pga}(u_1 \,;\, \ldots \,;\, u_n \,;\, (\#[1])^{\omega})$ and $\mathtt{pgag2pga}^k(u_1 \,;\, \ldots \,;\, u_n) = \mathtt{pgag2pga}^k(u_1 \,;\, \ldots \,;\, u_n \,;\, (\#[1])^{\omega})$, we only consider the case where the repetition operator occurs in P.

We make use of an auxiliary function $|\_, \_|$. This function determines, for each natural number and closed PGA term in which the repetition operator occurs, a closed term of BTA with guarded recursion. The function $|\_, \_|$ is defined as follows:

$$|i, u_1 \,;\, \ldots \,;\, u_n \,;\, (u_{n+1} \,;\, \ldots \,;\, u_m)^{\omega}| = |u_i \,;\, \ldots \,;\, u_m \,;\, (u_{n+1} \,;\, \ldots \,;\, u_m)^{\omega}| \quad \text{if } 1 \leq i \leq m,$$
$$|i, u_1 \,;\, \ldots \,;\, u_n \,;\, (u_{n+1} \,;\, \ldots \,;\, u_m)^{\omega}| = \mathsf{D} \quad \text{if } \neg\, 1 \leq i \leq m.$$

Let $P = u_1 \,;\, \ldots \,;\, u_n \,;\, (u_{n+1} \,;\, \ldots \,;\, u_m)^{\omega}$ be a closed $\mathrm{PGA}_{\mathrm{g}}^{k}$ term, let $P' = \mathtt{pgag2pga}(P)$, and let $P'' = \mathtt{pgag2pga}^k(P)$. Moreover, let $\rho : \mathbb{N} \to \mathbb{N}$ be such that $f(i) = (k+3) \cdot (i-1) + 1$. Then it follows easily from the definitions of $|\_, \_|$, $|\_|$, $\mathtt{pgag2pga}$ and $\mathtt{pgag2pga}^k$, the axioms of PGA and Proposition 2 that for $1 \leq i \leq m$:

$$\begin{aligned}
&|i, P'| = a \circ |i+1, P'| && \text{if } u_i = a,\\
&|i, P'| = |i+1, P'| \trianglelefteq a \trianglerighteq |i+2, P'| && \text{if } u_i = +a,\\
&|i, P'| = |i+2, P'| \trianglelefteq a \trianglerighteq |i+1, P'| && \text{if } u_i = -a,\\
&|i, P'| = |i+1, P'| && \text{if } u_i = [l],\\
&|i, P'| = |i+n, P'| && \text{if } u_i = \#[l] \wedge tgt_i(l) = n,\\
&|i, P'| = \mathsf{S} && \text{if } u_i = !.
\end{aligned}$$

and

$$
\begin{aligned}
&|\rho(i), P''| = a \circ |\rho(i+1), P''| && \text{if } u_i = a,\\
&|\rho(i), P''| = |\rho(i+1), P''| \trianglelefteq a \trianglerighteq |\rho(i+2), P''| && \text{if } u_i = +a,\\
&|\rho(i), P''| = |\rho(i+2), P''| \trianglelefteq a \trianglerighteq |\rho(i+1), P''| && \text{if } u_i = -a,\\
&|\rho(i), P''| = |\rho(i+1), P''| && \text{if } u_i = [l],\\
&|\rho(i), P''| = |\rho(i+n), P''| && \text{if } u_i = \#[l] \wedge tgt_i(l) = n,\\
&|\rho(i), P''| = \mathsf{S} && \text{if } u_i = !
\end{aligned}
$$

(where tgt_i is as in the definition of $\mathtt{pgag2pga}$). Because $|\mathtt{pgag2pga}(P)| = |1, P'|$ and $|\mathtt{pgag2pga}^k(P)| = |\rho(1), P''|$, this means that $|\mathtt{pgag2pga}(P)|$ and $|\mathtt{pgag2pga}^k(P)|$ are solutions of the same guarded recursive specification. Because guarded recursive specifications have unique solutions, it follows that $|\mathtt{pgag2pga}(P)| = |\mathtt{pgag2pga}^k(P)|$. □

The projection $\mathtt{pgag2pga}^k(P)$ yields only closed PGA terms that do not contain jump instructions $\#l$ with $l > k + 3$. Hence, we have the following corollary of Theorem 2:

Corollary 2 *For each closed* $\mathrm{PGA}_{\mathrm{g}}^k$ *term* P, *there exists a closed* PGA *term* P' *not containing jump instructions* $\#l$ *with* $l > k + 3$ *such that* $|\mathtt{pgag2pga}(P)| = |P'|$.

It follows from Corollary 2 that, if a regular thread cannot be denoted by a closed PGA term that does not contain jump instructions $\#l$ with $l > k + 3$, it cannot be denoted by a closed $\mathrm{PGA}_{\mathrm{g}}^k$ term. Moreover, it is known that, for each $k \in \mathbb{N}$, there exists a closed PGA term for which there does not exist a closed PGA term not containing jump instructions $\#l$ with $l > k + 3$ that denotes the same thread (see e.g. [23], Proposition 3). Hence, we also have the following corollary:

Corollary 3 *For each* $k \in \mathbb{N}$, *there exists a closed* PGA *term* P *for which there does not exist a closed* $\mathrm{PGA}_{\mathrm{g}}^k$ *term* P' *such that* $|P| = |\mathtt{pgag2pga}(P')|$.

9 Conclusions

Program algebra is a setting suited for investigating single-pass instruction sequences. In this setting, we have shown that each behaviour that can be produced by a single-pass instruction sequence under execution can be produced by a single-pass instruction sequence without jump instructions if use can be made of Boolean register services. We consider this an interesting expressiveness result. An important variant of program algebra is obtained by leaving out jump instructions and adding labels and goto instructions. We have also shown that a bound to the number of labels restricts the expressiveness of this variant. Earlier expressiveness results on single-pass instruction sequences as considered in program algebra are collected in [23].

Program algebra does not provide a notation for programs that is intended for actual programming. However, to demonstrate that single-pass instruction sequences as considered in program algebra are suited for explaining programs in the form of assembly programs as well as programs in the form of structured programs, a hierarchy of program notations rooted in program algebra is introduced in [3]. One program notation belonging to this hierarchy, called $\mathrm{PGLD}_{\mathrm{g}}$, is a simple program notation, close to existing assembly languages, with labels and goto instructions. We remark that a projection from the set of all $\mathrm{PGLD}_{\mathrm{g}}$ programs to the set of all closed $\mathrm{PGA}_{\mathrm{g}}$ terms can easily be devised.

The idea that programs are in essence single-pass instruction sequences underlies the choice for the name program algebra. The name seems to imply that program algebra is suited for investigating programs in general. We do not intend to claim this generality, which in any case does not matter when investigating single-pass instruction sequences. The name program algebra might as well be used as a collective name for algebras that are based on any viewpoint concerning programs. To our knowledge, it is not common to use the name as such.

Most closely related to our work on instruction sequences is work on Kleene algebras (see e.g. [15–18, 24]), but programs are considered at a higher level in that work. For instance, programming features like jump instructions have never been studied. In most work on computer architecture (see e.g. [1, 19–22]), instruction sequences are under discussion. However, the notion of instruction sequence is not subjected to systematic and precise analysis in the work concerned.

Acknowledgements This research was partly carried out in the framework of the Jacquard-project Symbiosis, which is funded by the Netherlands Organisation for Scientific Research (NWO). We thank Alban Ponse, colleague at the University of Amsterdam, and Stephan Schroevers, graduate student at the University of Amsterdam, for carefully reading a preliminary version of this paper and pointing out some flaws in it. Moreover, we thank an anonymous referee for suggesting improvements of the presentation of the paper.

References

1. Baker, H.G.: Precise instruction scheduling without a precise machine model. SIGARCH Comput. Archit. News **19**(6), 4–8 (1991)
2. Bergstra, J.A., Bethke, I.: Polarized process algebra and program equivalence. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) Proceedings 30th ICALP. Lecture Notes in Computer Science, vol. 2719, pp. 1–21. Springer, Berlin (2003)
3. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. J. Log. Algebr. Program. **51**(2), 125–156 (2002)
4. Bergstra, J.A., Middelburg, C.A.: Instruction sequences with indirect jumps. Sci. Ann. Comput. Sci. **17**, 19–46 (2007)
5. Bergstra, J.A., Middelburg, C.A.: Synchronous cooperation for explicit multi-threading. Acta Inf. **44**(7–8), 525–569 (2007)
6. Bergstra, J.A., Middelburg, C.A.: Distributed strategic interleaving with load balancing. Future Gener. Comput. Syst. **24**(6), 530–548 (2008)
7. Bergstra, J.A., Middelburg, C.A.: Instruction sequences and non-uniform complexity theory. arXiv:0809.0352v3 [cs.CC] (2008)

8. Bergstra, J.A., Middelburg, C.A.: Maurer computers for pipelined instruction processing. Math. Struct. Comput. Sci. **18**(2), 373–409 (2008)
9. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. J. Appl. Log. **6**(4), 553–563 (2008)
10. Bergstra, J.A., Middelburg, C.A.: Autosolvability of halting problem instances for instruction sequences. arXiv:0911.5018v2 [cs.LO] (2009)
11. Bergstra, J.A., Middelburg, C.A.: Indirect jumps improve instruction sequence performance. arXiv:0909.2089v1 [cs.PL] (2009)
12. Bergstra, J.A., Middelburg, C.A.: Instruction sequences with dynamically instantiated instructions. Fundam. Inform. **96**(1–2), 27–48 (2009)
13. Bergstra, J.A., Ponse, A.: Execution architectures for program algebra. J. Appl. Log. **5**(1), 170–192 (2007)
14. Cooper, D.C.: Böhm and Jacopini's reduction of flow charts. Commun. ACM **10**(8), 463, 473 (1967)
15. Fernandes, T., Desharnais, J.: Describing data flow analysis techniques with Kleene algebra. Sci. Comput. Program. **65**, 173–194 (2007)
16. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inf. Comput. **110**(2), 366–390 (1994)
17. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (1997)
18. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Trans. Comput. Log. **1**(1), 60–76 (2000)
19. Lunde, A.: Empirical evaluation of some features of instruction set processor architectures. Commun. ACM **20**(3), 143–153 (1977)
20. Nair, R., Hopkins, M.E.: Exploiting instruction level parallelism in processors by caching scheduled groups. SIGARCH Comput. Archit. News **25**(2), 13–25 (1997)
21. Ofelt, D., Hennessy, J.L.: Efficient performance prediction for modern microprocessors. In: SIGMETRICS '00, pp. 229–239 (2000)
22. Patterson, D.A., Ditzel, D.R.: The case for the reduced instruction set computer. SIGARCH Comput. Archit. News **8**(6), 25–33 (1980)
23. Ponse, A., van der Zwaag, M.B.: An introduction to program and thread algebra. In: Beckmann, A., et al. (eds.) CiE 2006. Lecture Notes in Computer Science, vol. 3988, pp. 445–458. Springer, Berlin (2006)
24. Salomaa, A.: Two complete axiom systems for the algebra of regular events. J. ACM **13**(1), 158–169 (1966)