University of Amsterdam
Programming Research Group

Instruction Sequences with Indirect Jumps

J.A. Bergstra
C.A. Middelburg

Report PRG0709 November 2007

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

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

C.A. Middelburg

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

e-mail: kmiddelb@science.uva.nl

Programming Research Group Electronic Report Series

Instruction Sequences with Indirect Jumps*

J.A. Bergstra[1,2] and C.A. Middelburg[1]

[1] Programming Research Group, University of Amsterdam,
P.O. Box 41882, 1009 DB Amsterdam, the Netherlands
[2] Department of Philosophy, Utrecht University,
P.O. Box 80126, 3508 TC Utrecht, the Netherlands
J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl

Abstract. We study sequential programs that are instruction sequences with direct and indirect jump instructions. The intuition is that indirect jump instructions are jump instructions where the position of the instruction to jump to is the content of some memory cell. We consider several kinds of indirect jump instructions. For each kind, we define the meaning of programs with indirect jump instructions of that kind by means of a translation into programs without indirect jump instructions. For each kind, the intended behaviour of a program with indirect jump instructions of that kind under execution is the behaviour of the translated program under execution on interaction with some memory device.

Keywords: instruction sequence, indirect jump instruction, projection semantics, program algebra, thread algebra.

1998 ACM Computing Classification: D.3.1, D.3.3, F.1.1, F.3.2, F.3.3.

1 Introduction

We take the view that sequential programs are in essence sequences of instructions. Although finite state programs with direct and indirect jump instructions are as expressive as finite state programs with direct jump instructions only, indirect jump instructions are widely used. For example, return instructions, in common use to implement recursive method calls in programming language such as Java [11] and C# [12], are indirect jump instructions. Therefore, we consider a theoretical understanding of both direct jump instructions and indirect jump instructions highly relevant to programming. In [3], sequential programs that are instruction sequences with direct jump instructions are studied. In this paper, we study sequential programs that are instruction sequences with both direct jump instructions and indirect jump instructions.

We believe that interaction with components of an execution environment, in particular memory devices, is inherent in the behaviour of programs under execution. Intuitively, indirect jump instructions are jump instructions where the

* 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).

position of the instruction to jump to is the content of some memory cell. In this paper, we consider several kinds of indirect jump instructions, including return instructions. For each kind, we define the meaning of programs with indirect jump instructions of that kind by means of a translation into programs without indirect jump instructions. For each kind, the intended behaviour of a program with indirect jump instructions of that kind under execution is the behaviour of the translated program under execution on interaction with some memory device. We also describe the memory devices concerned, to wit register files and stacks.

The approach to define the meaning of programs mentioned above is introduced under the name projection semantics in [3]. Projection semantics explains the meaning of programs in terms of known programs instead of more or less sophisticated mathematical objects that represent behaviours. The main advantage of projection semantics is that it does not require a lot of mathematical background. In the present case, another advantage of projection semantics is that it follows immediately that indirect jump instructions of the kinds considered can be eliminated from programs in the presence of an appropriate memory device. We will study sequential programs that are instruction sequences with direct and indirect jump instructions in the setting in which projection semantics has been developed so far: the setting of program algebra and basic thread algebra.[3]

Program algebra is an algebra of deterministic sequential programs based on the idea that such programs are in essence sequences of instructions. Basic thread algebra is a form of process algebra which is tailored to the description of the behaviour of deterministic sequential programs under execution. A hierarchy of program notations rooted in program algebra is introduced in [3]. In this paper, we embroider on two program notations that belong to this hierarchy. The program notations in question, called 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.

A thread proceeds by doing steps in a sequential fashion. A thread may do certain steps only for the sake of having itself affected by some service. In [9], the use mechanism is introduced to allow for such interaction between threads and services. The interaction between behaviours of programs under execution and some memory device referred to above is an interaction of this kind. In this paper, we will use a slightly adapted form of the use mechanism, called thread-service composition, to have behaviours of programs under execution affected by services.

This paper is organized as follows. First, we review basic thread algebra, program algebra, and the program notations PGLC and PGLD (Sections 2, 3, and 4). Next, we extend basic thread algebra with thread-service composition and introduce a state-based approach to describe services (Sections 5 and 6).

[3] In [3], basic thread algebra is introduced under the name basic polarized process algebra. Prompted by the development of thread algebra [7], which is a design on top of it, basic polarized process algebra has been renamed to basic thread algebra.

Following this, we give a state-based description of register file services and introduce variants of the program notations PGLC and PGLD with indirect jump instructions (Sections 7, 8, and 9). We also introduce a variant of one of those program notations with double indirect jump instructions (Section 10). After that, we give a state-based description of stack services and introduce a variant of the program notation PGLD with returning jump instructions and return instructions (Sections 11 and 12). Finally, we make some concluding remarks (Section 13).

2 Basic Thread Algebra

In this section, we review BTA (Basic Thread Algebra), a form of process algebra which is tailored to the description of the behaviour of deterministic sequential programs under execution. The behaviours concerned are called *threads*.

In BTA, it is assumed that there is a fixed but arbitrary finite set of *basic actions* $\mathcal{A}$. The intuition is that each basic action performed by a thread is taken as a command to be processed by a service provided by the execution environment of the thread. The processing of a command may involve a change of state of the service concerned. At completion of the processing of the command, the service produces a reply value. This reply is either T or F and is returned to the thread concerned.

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 5.

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

Terms of sort $\mathbf{T}$ are built as usual (see e.g. [16, 17]). Throughout the paper, we assume that there are infinitely many variables of sort $\mathbf{T}$, including x, y, z.

We use infix notation for postconditional composition. We introduce *action prefixing* as an abbreviation: $a \circ p$, where p is a term of sort $\mathbf{T}$, abbreviates $p \trianglelefteq a \trianglerighteq p$.

Let p and q be closed terms of sort $\mathbf{T}$ and $a \in \mathcal{A}$. Then $p \trianglelefteq a \trianglerighteq q$ will perform action a, and after that proceed as p if the processing of a leads to the reply T (called a positive reply), and proceed as q if the processing of a leads to the reply F (called a negative reply).

Each closed BTA term of sort $\mathbf{T}$ denotes a finite thread, i.e. a thread of which the length of the sequences of actions that it can perform is bounded. Guarded recursive specifications give rise to infinite threads.

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 term of the form D, S or $t \trianglelefteq a \trianglerighteq t'$ with t and t' BTA terms of sort $\mathbf{T}$ that

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

contain only variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur on the left-hand side of an equation 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 [1]. A thread that is the solution of a finite guarded recursive specification over BTA is called a *finite-state* thread.

We extend BTA with guarded recursion by adding constants for solutions of guarded recursive specifications and axioms concerning these additional constants. For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we add a constant of sort $\mathbf{T}$ standing for the unique solution of E for X to the constants of BTA. The constant standing for the unique solution of E for X is denoted by $\langle X|E\rangle$. Moreover, we add the axioms for guarded recursion given in Table 1 to BTA, where 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 $\langle Y|E\rangle$. In this table, 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 the variables, terms and guarded recursive specifications for which X, t_X and E stand. 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. The conditional equations $E \Rightarrow X = \langle X|E\rangle$ express that this solution is the only one.

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

In [5], we show that the threads considered in BTA+REC can be viewed as processes that are definable over ACP [10].

3 Program Algebra

In this section, we review PGA (ProGram Algebra), an algebra of sequential programs based on the idea that sequential programs are in essence sequences of instructions. PGA provides a program notation for finite-state threads.

In PGA, it is assumed that there is a fixed but arbitrary finite set $\mathfrak{A}$ of *basic 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.

Table 2. 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

The intuition is that the execution of a basic instruction a may modify a state and produces T or F at its completion. In the case of a positive test instruction $+a$, 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. In the case where T is produced and there is not at least one subsequent primitive instruction and in the case where F is produced and there are not at least two subsequent primitive instructions, 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, the value produced is disregarded: 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 of the program concerned. If l equals 0 or the l-th next instruction does not exist, then $\#l$ results in deadlock. 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 $\_^\omega$.

Terms are built as usual. Throughout the paper, we assume that there are infinitely many variables, including x, y, z.

We use infix notation for concatenation and postfix notation for repetition.

Closed PGA terms are considered to denote programs. The intuition is that a program is in essence a non-empty, finite or infinite sequence of primitive instructions. These sequences are called *single pass instruction sequences* because PGA has been designed to enable single pass execution of instruction sequences: each instruction can be dropped after it has been executed. Programs are considered to be equal if they represent the same single pass instruction sequence. The axioms for instruction sequence equivalence are given in Table 2. In this table, n stands for an arbitrary natural number greater than 0. For each $n > 0$, the term x^n is defined by induction on n as follows: $x^1 = x$ and $x^{n+1} = x\,;x^n$. The *unfolding* equation $x^\omega = x\,;x^\omega$ is derivable. Each closed PGA term is derivably equal to a term in *canonical form*, i.e. a term of the form P or $P\,;Q^\omega$, where P and Q are closed PGA terms that do not contain the repetition operator.

Each closed PGA term is considered to denote a program of which the behaviour is a finite-state thread, taking the set $\mathfrak{A}$ of basic instructions for the set $\mathcal{A}$

Table 3. Defining equations for thread extraction operator

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

Table 4. Rule for cyclic jump chains

$x \cong \#0\,;y \;\Rightarrow\; \lvert x\rvert = \mathsf{D}$

Table 5. Defining formulas for structural congruence predicate

$\#n+1\,;u_1\,;\ldots\,;u_n\,;\#0 \cong \#0\,;u_1\,;\ldots\,;u_n\,;\#0$
$\#n+1\,;u_1\,;\ldots\,;u_n\,;\#m \cong \#m+n+1\,;u_1\,;\ldots\,;u_n\,;\#m$
$(\#n+l+1\,;u_1\,;\ldots\,;u_n)^\omega \cong (\#l\,;u_1\,;\ldots\,;u_n)^\omega$
$\#m+n+l+2\,;u_1\,;\ldots\,;u_n\,;(v_1\,;\ldots\,;v_{m+1})^\omega \cong$
$\#n+l+1\,;u_1\,;\ldots\,;u_n\,;(v_1\,;\ldots\,;v_{m+1})^\omega$
$x \cong x$
$x_1 \cong y_1 \wedge x_2 \cong y_2 \;\Rightarrow\; x_1\,;x_2 \cong y_1\,;y_2 \wedge {x_1}^\omega \cong {y_1}^\omega$

of actions. The *thread extraction* operator $\lvert\_\rvert$ assigns a thread to each program. The thread extraction operator is defined by the equations given in Table 3 (for $a \in \mathfrak{A}$, $l \in \mathbb{N}$ and $u \in \mathfrak{I}$) and the rule given in Table 4. This rule is expressed in terms of the *structural congruence* predicate $\_ \cong \_$, which is defined by the formulas given in Table 5 (for $n, m, l \in \mathbb{N}$ and $u_1, \ldots, u_n, v_1, \ldots, v_{m+1} \in \mathfrak{I}$).

The equations given in Table 3 do not cover the case where there is a cyclic chain of forward jumps. Programs are structural congruent if they are the same after removing all chains of forward jumps in favour of single jumps. Because a cyclic chain of forward jumps corresponds to #0, the rule from Table 4 can be read as follows: if x starts with a cyclic chain of forward jumps, then $\lvert x\rvert$ equals D. It is easy to see that the thread extraction operator assigns the same thread to structurally congruent programs. Therefore, the rule from Table 4 can be replaced by the following generalization: $x \cong y \Rightarrow \lvert x\rvert = \lvert y\rvert$.

Let E be a finite guarded recursive specification over BTA, and let P_X be a closed PGA term for each $X \in \mathrm{V}(E)$. Let E' be the set of equations that results from replacing in E all occurrences of X by $\lvert P_X\rvert$ for each $X \in \mathrm{V}(E)$. If E' can be obtained by applications of axioms PGA1–PGA4, the defining equations for the thread extraction operator and the rule for cyclic jump chains, then $\lvert P_X\rvert$ is

the solution of E for X. Such a finite guarded recursive specification can always be found. Thus, the behaviour of each closed PGA term, is a thread that is definable by a finite guarded recursive specification over BTA. Moreover, each finite guarded recursive specification over BTA can be translated to a closed PGA term of which the behaviour is the solution of the finite guarded recursive specification concerned.

Closed PGA terms are loosely called PGA *programs*. PGA programs in which the repetition operator do not occur are called *finite* PGA programs.

4 The Program Notations PGLC and PGLD

In this section, we review two program notations which are rooted in PGA. These program notations, called PGLC and PGLD, belong to a hierarchy of program notations introduced in [3].

Both 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. PGLC and PGLD have no explicit termination instruction.

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

PGLC 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 *direct forward jump instruction* $\#l$;
- for each $l \in \mathbb{N}$, a *direct backward jump instruction* $\backslash\#l$.

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

The plain basic instructions, the positive test instructions, and the negative test instructions are as in PGA, except that termination instead of deadlock occurs in the case where there are insufficient subsequent primitive instructions. The effect of a direct forward jump instruction $\#l$ is that execution proceeds with the l-th next instruction of the program concerned. If l equals 0, then deadlock occurs. If the l-th next instruction does not exist, then termination occurs. The effect of a direct backward jump instruction $\backslash\#l$ is that execution proceeds with the l-th previous instruction of the program concerned. If l equals 0, then deadlock occurs. If the l-th previous instruction does not exist, then termination occurs.

We define the meaning of PGLC programs by means of a function `pglc2pga` from the set of all PGLC programs to the set of all PGA programs. This function is defined by

$$\texttt{pglc2pga}(u_1 \,;\, \ldots \,;\, u_k) = (\psi_1(u_1) \,;\, \ldots \,;\, \psi_k(u_k) \,;\, ! \,;\, !)^\omega \,,$$

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

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

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 $\psi_1(u_1)\,;\ldots;\psi_k(u_k)$.

Let P be a PGLC program. Then $\mathtt{pglc2pga}(P)$ represents the meaning of P as a PGA program. The intended behaviour of P is the behaviour of $\mathtt{pglc2pga}(P)$. That is, the *behaviour* of P, written $|P|_{\mathrm{PGLC}}$, is $|\mathtt{pglc2pga}(P)|$.

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 $l \in \mathbb{N}$, a *direct 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 PGLC. The effect of a direct 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 deadlock occurs. If l equals 0 or l is greater than the length of the program, then termination occurs.

We define the meaning of PGLD programs by means of a function $\mathtt{pgld2pglc}$ from the set of all PGLD programs to the set of all PGLC programs. This function is defined by

$$\mathtt{pgld2pglc}(u_1\,;\ldots;u_k) = \psi_1(u_1)\,;\ldots;\psi_k(u_k)\;,$$

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

$$\begin{aligned}
\psi_j(\#\#l) &= \#l - j && \text{if } l \geq j \;, \\
\psi_j(\#\#l) &= \backslash\#j - l && \text{if } l < j \;, \\
\psi_j(u) &= u && \text{if } u \text{ is not a jump instruction .}
\end{aligned}$$

Let P be a PGLD program. Then $\mathtt{pgld2pglc}(P)$ represents the meaning of P as a PGLC program. The intended behaviour of P is the behaviour of $\mathtt{pgld2pglc}(P)$. That is, the *behaviour* of P, written $|P|_{\mathrm{PGLD}}$, is $|\mathtt{pgld2pglc}(P)|_{\mathrm{PGLC}}$.

We use the phrase *projection semantics* to refer to the approach to semantics followed in this section. The meaning functions `pglc2pga` and `pgld2pglc` are called *projections*.

PGLC and PGLD are very simple program notations. The hierarchy of program notations introduced in [3] also includes a program notation, called PGLS, that supports structured programming by offering conditional and loop constructs instead of (unstructured) jumps. Each PGLS program can be translated into a semantically equivalent PGLD program by means of a number of projections.

5 Interaction of Threads with Services

A thread may perform certain actions only for the sake of getting reply values returned by a service and that way having itself affected by that service. In this section, we introduce thread-service composition, which allows for threads to be affected by services in this way. We will only use thread-service composition to have program behaviours affected by a service. Thread-service composition is a slightly adapted form of the use mechanism introduced in [9].

We consider only deterministic services. This will do in the case that we address: services that keep private data for a program. The services concerned are para-target services by the classification given in [6].

It is assumed that there is a fixed but arbitrary finite set of *foci* $\mathcal{F}$ and a fixed but arbitrary finite set of *methods* $\mathcal{M}$. 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 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.

We introduce yet another sort: the sort $\mathbf{S}$ of *services*. However, we will not introduce constants and operators to build terms of this sort. $\mathbf{S}$ is a parameter of theories with thread-service composition. $\mathbf{S}$ is considered to stand for the set of all services. It is assumed that each service can be represented by a function $H : \mathcal{M}^+ \to \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$ with the property that $H(\alpha) = \mathsf{B} \Rightarrow H(\alpha \frown \langle m \rangle) = \mathsf{B}$ for all $\alpha \in \mathcal{M}^+$ and $m \in \mathcal{M}$. This function is called the *reply* function of the service. Given a reply function H and a method $m \in \mathcal{M}$, the *derived* reply function of H after processing m, written $\frac{\partial}{\partial m} H$, is defined by $\frac{\partial}{\partial m} H(\alpha) = H(\langle m \rangle \frown \alpha)$.

The connection between a reply function H and the service represented by it can be understood as follows:

- if $H(\langle m \rangle) = \mathsf{T}$, the request to process command m is accepted by the service, the reply is positive and the service proceeds as $\frac{\partial}{\partial m} H$;
- if $H(\langle m \rangle) = \mathsf{F}$, the request to process command m is accepted by the service, the reply is negative and the service proceeds as $\frac{\partial}{\partial m} H$;

Table 6. Axioms for thread-service composition

$\mathsf{S} /_f H = \mathsf{S}$		TSC1
$\mathsf{D} /_f H = \mathsf{D}$		TSC2
$(x \trianglelefteq g.m \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq g.m \trianglerighteq (y /_f H)$	if $f \neq g$	TSC3
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = x /_f \frac{\partial}{\partial m} H$	if $H(\langle m \rangle) = \mathsf{T}$	TSC4
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = y /_f \frac{\partial}{\partial m} H$	if $H(\langle m \rangle) = \mathsf{F}$	TSC5
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{D}$	if $H(\langle m \rangle) = \mathsf{B}$	TSC6

– if $H(\langle m \rangle) = \mathsf{B}$, the request to process command m is not accepted by the service.

Henceforth, we will identify a reply function with the service represented by it.

For each $f \in \mathcal{F}$, we introduce the binary *thread-service composition* operator $\_ /_f \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$. 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. Service H affects thread p by means of the reply values produced at completion of the processing of the actions performed by p. The actions processed by H are no longer observable.

The axioms for the thread-service composition operator are given in Table 6. In this table, f stands for an arbitrary focus from $\mathcal{F}$, m stands for an arbitrary method from $\mathcal{M}$. Axiom TSC3 expresses that actions of the form $g.m$, where $f \neq g$, are not processed. Axioms TSC4 and TSC5 express that a thread is affected by a service as described above when an action of the form $f.m$ performed by the thread is processed by the service. Axiom TSC6 expresses that deadlock takes place when an action to be processed is not accepted.

Let T stand for either BTA or BTA+REC. Then we will write $T + \mathrm{TSC}$ for T, taking the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$ for $\mathcal{A}$, extended with the thread-service composition operators and the axioms from Table 6.

In [5], we show that the services considered here can be viewed as processes that are definable over an extension of ACP with conditionals introduced in [4].

6 State-Based Description of Services

In this section, we introduce the state-based approach to describe families of services that will be used later on. This approach is similar to the approach to describe state machines introduced in [9].

In this approach, a family of services is described by

– a set of states S;
– an effect function $\mathit{eff} : \mathcal{M} \times S \to S$;
– a yield function $\mathit{yld} : \mathcal{M} \times S \to \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$;

satisfying the following condition:

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

The set S contains the states in which the services 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.

We define, for each $s \in S$, a cumulative effect function $ceff_s : \mathcal{M}^* \to S$ in terms of s and *eff* as follows:

$$ceff_s(\langle\,\rangle) = s \ ,$$
$$ceff_s(\alpha \frown \langle m \rangle) = eff(m, ceff_s(\alpha)) \ .$$

We define, for each $s \in S$, a service $H_s : \mathcal{M}^+ \to \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$ in terms of $ceff_s$ and *yld* as follows:

$$H_s(\alpha \frown \langle m \rangle) = yld(m, ceff_s(\alpha)) \ .$$

H_s is called the service with *initial state* s described by S, *eff* and *yld*. We say that $\{H_s \mid s \in S\}$ is the *family of services* described by S, *eff* and *yld*.

For each $s \in S$, H_s is a service indeed: the condition imposed on S, *eff* and *yld* implies that $H_s(\alpha) = \mathsf{B} \Rightarrow H_s(\alpha \frown \langle m \rangle) = \mathsf{B}$ for all $\alpha \in \mathcal{M}^+$ and $m \in \mathcal{M}$. It is worth mentioning that $H_s(\langle m \rangle) = yld(m,s)$ and $\frac{\partial}{\partial m} H_s = H_{eff(m,s)}$.

7 Register File Services

In this section, we give a state-based description of the very simple family of services that constitute a register file of which the registers can contain natural numbers up to some bound. This register file will be used in Sections 8–10 to describe the behaviour of programs in variants of PGLC and PGLD with indirect jump instructions.

It is assumed that a fixed but arbitrary number I has been given, which is considered the number of registers available. It is also assumed that a fixed but arbitrary number N has been given, which is considered the greatest natural number that can be contained in a register.

The register file services accept the following methods:

- for each $i \in [0,I]$ and $n \in [0,N]$, a *register set method* $\mathsf{set}{:}i{:}n$;
- for each $i \in [0,I]$ and $n \in [0,N]$, a *register test method* $\mathsf{eq}{:}i{:}n$.

We write $\mathcal{M}_{\mathsf{regs}}$ for the set $\{\mathsf{set}{:}i{:}n, \mathsf{eq}{:}i{:}n \mid i \in [0,I] \wedge n \in [0,N]\}$. It is assumed that $\mathcal{M}_{\mathsf{regs}} \subseteq \mathcal{M}$.

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

- $\mathsf{set}{:}i{:}n$: the contents of register i becomes n and the reply is T;
- $\mathsf{eq}{:}i{:}n$: if the contents of register i equals n, then nothing changes and the reply is T; otherwise nothing changes and the reply is F.

Let $s : [1, I] \to [0, N]$. Then we write Regs_s for the service with initial state s described by $S = ([1, I] \to [0, N]) \cup \{\uparrow\}$, where $\uparrow \notin [1, I] \to [0, N]$, and the functions *eff* and *yld* defined as follows ($n \in [0, N]$, $\rho : [1, I] \to [0, N]$):[4]

$$
\begin{aligned}
&\mathit{eff}(\mathsf{set}{:}i{:}n, \rho) = \rho \oplus [i \mapsto n] \;, \\
&\mathit{eff}(\mathsf{eq}{:}i{:}n, \rho) = \rho \;, \\
&\mathit{eff}(m, \rho) = \uparrow && \text{if } m \notin \mathcal{M}_{\mathsf{regs}} \;, \\
&\mathit{eff}(m, \uparrow) = \uparrow \;, \\
\\
&\mathit{yld}(\mathsf{set}{:}i{:}n, \rho) = \mathsf{T} \;, \\
&\mathit{yld}(\mathsf{eq}{:}i{:}n, \rho) = \mathsf{T} && \text{if } \rho(i) = n \;, \\
&\mathit{yld}(\mathsf{eq}{:}i{:}n, \rho) = \mathsf{F} && \text{if } \rho(i) \neq n \;, \\
&\mathit{yld}(m, \rho) = \mathsf{B} && \text{if } m \notin \mathcal{M}_{\mathsf{regs}} \;, \\
&\mathit{yld}(m, \uparrow) = \mathsf{B} \;.
\end{aligned}
$$

We write $\mathit{Regs}_{\mathrm{init}}$ for $\mathit{Regs}_{[1 \mapsto 0] \oplus \ldots \oplus [I \mapsto 0]}$.

8 PGLD with Indirect Jumps

In this section, we introduce a variant of PGLD with indirect jump instructions. This variant is called $\mathrm{PGLD}_{\mathrm{ij}}$.

In $\mathrm{PGLD}_{\mathrm{ij}}$, it is assumed that there is a fixed but arbitrary finite set of *foci* $\mathcal{F}$ with $\mathsf{regs} \in \mathcal{F}$ and a fixed but arbitrary finite set of *methods* $\mathcal{M}$. Moreover, we adopt the assumptions made about register file services in Section 7. The set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$ is taken as the set $\mathfrak{A}$ of basic instructions.

$\mathrm{PGLD}_{\mathrm{ij}}$ 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 *direct absolute jump instruction* $\#\#l$;
- for each $i \in [1, I]$, an *indirect absolute jump instruction* $\mathsf{i}\#\#i$.

$\mathrm{PGLD}_{\mathrm{ij}}$ programs have the form $u_1 \,;\, \ldots \,;\, u_k$, where $u_1, \ldots, u_k$ are primitive instructions of $\mathrm{PGLD}_{\mathrm{ij}}$.

The plain basic instructions, the positive test instructions, the negative test instructions, and the direct absolute jump instructions are as in PGLD. The effect of an indirect absolute jump instruction $\mathsf{i}\#\#i$ is that execution proceeds with the l-th instruction of the program concerned, where l is the content of register i. If $\mathsf{i}\#\#i$ is itself the l-th instruction, then deadlock occurs. If l equals 0 or l is greater than the length of the program, termination occurs.

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

Recall that the content of register i can be set to l by means of the basic instruction $\mathsf{regs.set}{:}i{:}l$. Initially, its content is 0.

Like before, we define the meaning of $\mathrm{PGLD}_{\mathrm{ij}}$ programs by means of a function $\mathtt{pgldij2pgld}$ from the set of all $\mathrm{PGLD}_{\mathrm{ij}}$ programs to the set of all PGLD programs. This function is defined by

$$\begin{aligned}
&\mathtt{pgldij2pgld}(u_1\,;\ldots;u_k) = \\
&\quad \psi(u_1)\,;\ldots;\psi(u_k)\,;\#\#0\,;\#\#0\,; \\
&\quad +\mathsf{regs.eq}{:}1{:}1\,;\#\#1\,;\ldots;+\mathsf{regs.eq}{:}1{:}n\,;\#\#n\,;\#\#0\,; \\
&\quad\ \vdots \\
&\quad +\mathsf{regs.eq}{:}I{:}1\,;\#\#1\,;\ldots;+\mathsf{regs.eq}{:}I{:}n\,;\#\#n\,;\#\#0\,,
\end{aligned}$$

where $n = \min(k, N)$ and the auxiliary function ψ from the set of all primitive instructions of $\mathrm{PGLD}_{\mathrm{ij}}$ to the set of all primitive instructions of PGLD is defined as follows:

$$\begin{aligned}
\psi(\#\#l) &= \#\#l \quad \text{if } l \leq k\,, \\
\psi(\#\#l) &= \#\#0 \quad \text{if } l > k\,, \\
\psi(\mathsf{i}\#\#i) &= \#\#l_i\,, \\
\psi(u) &= u \qquad \text{if } u \text{ is not a jump instruction}\,,
\end{aligned}$$

and for each $i \in [1, I]$:

$$l_i = k + 3 + (2 \cdot \min(k, N) + 1) \cdot (i - 1)\,.$$

The idea is that each indirect absolute jump can be replaced by a direct absolute jump to the beginning of the instruction sequence

$$+\mathsf{regs.eq}{:}i{:}1\,;\#\#1\,;\ldots;+\mathsf{regs.eq}{:}i{:}n\,;\#\#n\,;\#\#0\,,$$

where i is the register concerned and $n = \min(k, N)$. The execution of this instruction sequence leads to the intended jump after the content of the register concerned has been found by a linear search. 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, $\#\#0\,;$ $\#\#0$ is appended to $\psi(u_1)\,;\ldots;\psi(u_k)$. Because the length of the translated program is greater than k, care is taken that there are no direct absolute jumps to instructions with a position greater than k. Obviously, the linear search for the content of a register can be replaced by a binary search.

Let P be a $\mathrm{PGLD}_{\mathrm{ij}}$ program. Then $\mathtt{pgldij2pgld}(P)$ represents the meaning of P as a PGLD program. The intended behaviour of P is the behaviour of $\mathtt{pgldij2pgld}(P)$ on interaction with a register file. That is, the *behaviour* of P, written $|P|_{\mathrm{PGLD_{ij}}}$, is $|\mathtt{pgldij2pgld}(P)|_{\mathrm{PGLD}} \,/_{\mathsf{regs}}\, \mathit{Regs}_{\mathrm{init}}$.

More than one instruction is needed in PGLD to obtain the effect of a single indirect absolute jump instruction. The projection $\mathtt{pgldij2pgld}$ deals with that in such a way that there is no need for the unit instruction operator introduced in [15] or the distinction between first-level instructions and second-level instructions introduced in [2].

9 PGLC with Indirect Jumps

In this section, we introduce a variant of PGLC with indirect jump instructions. This variant is called $\mathrm{PGLC}_{\mathrm{ij}}$.

In $\mathrm{PGLC}_{\mathrm{ij}}$, the same assumptions are made as in $\mathrm{PGLD}_{\mathrm{ij}}$. Like in $\mathrm{PGLD}_{\mathrm{ij}}$, the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$ is taken as the set $\mathfrak{A}$ of basic instructions.

$\mathrm{PGLD}_{\mathrm{ij}}$ 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 *direct forward jump instruction* $\#l$;
- for each $l \in \mathbb{N}$, a *direct backward jump instruction* $\backslash\#l$;
- for each $i \in [1, I]$, an *indirect forward jump instruction* $\mathsf{i}\#i$;
- for each $i \in [1, I]$, an *indirect backward jump instruction* $\mathsf{i}\backslash\#i$.

$\mathrm{PGLC}_{\mathrm{ij}}$ programs have the form $u_1 \,;\, \ldots \,;\, u_k$, where $u_1, \ldots, u_k$ are primitive instructions of $\mathrm{PGLC}_{\mathrm{ij}}$.

The plain basic instructions, the positive test instructions, the negative test instructions, the direct forward jump instructions, and the direct backward jump instructions are as in PGLC. The effect of an indirect forward jump instruction $\mathsf{i}\#i$ is that execution proceeds with the l-th next instruction of the program concerned, where l is the content of register i. If l equals 0, then deadlock occurs. If the l-th next instruction does not exist, then termination occurs. The effect of an indirect backward jump instruction $\mathsf{i}\backslash\#i$ is that execution proceeds with the l-th previous instruction of the program concerned, where l is the content of register i. If l equals 0, then deadlock occurs. If the l-th previous instruction does not exist, then termination occurs.

We define the meaning of $\mathrm{PGLC}_{\mathrm{ij}}$ programs by means of a function `pglcij2pglc` from the set of all $\mathrm{PGLC}_{\mathrm{ij}}$ programs to the set of all PGLC programs. This function is defined by

$$
\begin{aligned}
&\texttt{pglcij2pglc}(u_1 \,;\, \ldots \,;\, u_k) = \\
&\quad \psi_1(u_1) \,;\, \ldots \,;\, \psi_k(u_k) \,;\, \backslash\#k+1 \,;\, \backslash\#k+2 \,; \\
&\quad +\mathsf{regs.eq}{:}1{:}0 \,;\, \backslash\#l'_{1,1,0} \,;\, \ldots \,;\, +\mathsf{regs.eq}{:}1{:}N \,;\, \backslash\#l'_{1,1,N} \,; \\
&\qquad \vdots \\
&\quad +\mathsf{regs.eq}{:}1{:}0 \,;\, \backslash\#l'_{1,k,0} \,;\, \ldots \,;\, +\mathsf{regs.eq}{:}1{:}N \,;\, \backslash\#l'_{1,k,N} \,; \\
&\qquad \vdots \\
&\quad +\mathsf{regs.eq}{:}I{:}0 \,;\, \backslash\#l'_{I,1,0} \,;\, \ldots \,;\, +\mathsf{regs.eq}{:}I{:}N \,;\, \backslash\#l'_{I,1,N} \,; \\
&\qquad \vdots \\
&\quad +\mathsf{regs.eq}{:}I{:}0 \,;\, \backslash\#l'_{I,k,0} \,;\, \ldots \,;\, +\mathsf{regs.eq}{:}I{:}N \,;\, \backslash\#l'_{I,k,N} \,;
\end{aligned}
$$

$$
\begin{array}{l}
+\mathsf{regs.eq}{:}1{:}0\,;\,\backslash\#\underline{l}'_{1,1,0}\,;\,\ldots\,;\,+\mathsf{regs.eq}{:}1{:}N\,;\,\backslash\#\underline{l}'_{1,1,N}\,;\\
\quad\vdots\\
+\mathsf{regs.eq}{:}1{:}0\,;\,\backslash\#\underline{l}'_{1,k,0}\,;\,\ldots\,;\,+\mathsf{regs.eq}{:}1{:}N\,;\,\backslash\#\underline{l}'_{1,k,N}\,;\\
\qquad\vdots\\
+\mathsf{regs.eq}{:}I{:}0\,;\,\backslash\#\underline{l}'_{I,1,0}\,;\,\ldots\,;\,+\mathsf{regs.eq}{:}I{:}N\,;\,\backslash\#\underline{l}'_{I,1,N}\,;\\
\quad\vdots\\
+\mathsf{regs.eq}{:}I{:}0\,;\,\backslash\#\underline{l}'_{I,k,0}\,;\,\ldots\,;\,+\mathsf{regs.eq}{:}I{:}N\,;\,\backslash\#\underline{l}'_{I,k,N}\,,
\end{array}
$$

where the auxiliary functions ψ_j from the set of all primitive instructions of $\mathrm{PGLC}_{\mathrm{ij}}$ to the set of all primitive instructions of PGLC is defined as follows ($1 \leq j \leq k$):

$$
\begin{array}{lll}
\psi_j(\#l) & = \#l & \mathsf{if}\ j + l \leq k\,,\\
\psi_j(\#l) & = \backslash\#j & \mathsf{if}\ j + l > k\,,\\
\psi_j(\backslash\#l) & = \backslash\#l\,, & \\
\psi_j(\mathsf{i}\#i) & = \#l_{i,j}\,, & \\
\psi_j(\mathsf{i}\backslash\#i) & = \#\underline{l}_{i,j}\,, & \\
\psi_j(u) & = u & \mathsf{if}\ u\ \text{is not a jump instruction}\,,
\end{array}
$$

and for each $i \in [1, I]$, $j \in [1, k]$, and $h \in [0, N]$:

$$
\begin{array}{lll}
l_{i,j} & = k + 3 + 2 \cdot (N+1) \cdot (k \cdot (i-1) + (j-1))\,, & \\
\underline{l}_{i,j} & = k + 3 + 2 \cdot (N+1) \cdot (k \cdot (I+i-1) + (j-1))\,, & \\
l'_{i,j,h} & = l_{i,j} + 2 \cdot h + 1 - (j+h) & \mathsf{if}\ j + h \leq k\,,\\
l'_{i,j,h} & = k + 3 + 2 \cdot (N+1) \cdot k \cdot I & \mathsf{if}\ j + h > k\,,\\
\underline{l}'_{i,j,h} & = \underline{l}_{i,j} + 2 \cdot h + 1 - (j-h) & \mathsf{if}\ j - h \geq 0\,,\\
\underline{l}'_{i,j,h} & = k + 3 + 4 \cdot (N+1) \cdot k \cdot I & \mathsf{if}\ j - h < 0\,.
\end{array}
$$

Like in the case of indirect absolute jumps, the idea is that each indirect forward jump and each indirect backward jump can be replaced by a direct forward jump to the beginning of an instruction sequence whose execution leads to the intended jump after the content of the register concerned has been found by a linear search. However, the direct backward jump instructions occurring in that instruction sequence now depend upon the position of the indirect jump concerned in $u_1\,;\,\ldots\,;\,u_k$. 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, $\backslash\#k+1\,;\,\backslash\#k+2$ is appended to $\psi_1(u_1)\,;\,\ldots\,;\,\psi_k(u_k)$. Because the length of the translated program is greater

than k, care is taken that there are no direct forward jumps to instructions with a position greater than k.

Let P be a $\mathrm{PGLC}_{\mathrm{ij}}$ program. Then $\mathtt{pglcij2pglc}(P)$ represents the meaning of P as a PGLC program. The intended behaviour of P is the behaviour of $\mathtt{pglcij2pglc}(P)$ on interaction with a register file. That is, the *behaviour* of P, written $|P|_{\mathrm{PGLC_{ij}}}$, is $|\mathtt{pglcij2pglc}(P)|_{\mathrm{PGLC}} \,/_{\mathsf{regs}}\, \mathit{Regs}_{\mathrm{init}}$.

The projection $\mathtt{pglcij2pglc}$ yields needlessly long PGLC programs because it does not take into account the fact that there is at most one indirect jump instruction at each position in a $\mathrm{PGLC_{ij}}$ program being projected. Taking this fact into account would lead to a projection with a much more complicated definition.

10 PGLD with Double Indirect Jumps

In this section, we introduce a variant of $\mathrm{PGLD_{ij}}$ with double indirect jump instructions. This variant is called $\mathrm{PGLD_{dij}}$.

In $\mathrm{PGLD_{dij}}$, the same assumptions are made as in $\mathrm{PGLD_{ij}}$. Like in $\mathrm{PGLD_{ij}}$, the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$ is taken as the set $\mathfrak{A}$ of basic instructions.

$\mathrm{PGLD_{dij}}$ 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 *direct absolute jump instruction* $\#\#l$;
- for each $i \in [1, I]$, an *indirect absolute jump instruction* $\mathsf{i}\#\#i$;
- for each $i \in [1, I]$, a *double indirect absolute jump instruction* $\mathsf{ii}\#\#i$.

$\mathrm{PGLD_{dij}}$ programs have the form $u_1 \,;\, \ldots \,;\, u_k$, where $u_1, \ldots, u_k$ are primitive instructions of $\mathrm{PGLD_{dij}}$.

The plain basic instructions, the positive test instructions, the negative test instructions, the direct absolute jump instructions, and the indirect absolute jump instruction are as in $\mathrm{PGLD_{ij}}$. The effect of a double indirect absolute jump instruction $\mathsf{ii}\#\#i$ is that execution proceeds with the l-th instruction of the program concerned, where l is the content of register i', where i' is the content of register i. If $\mathsf{ii}\#\#i$ is itself the l-th instruction, then deadlock occurs. If l equals 0 or l is greater than the length of the program, termination occurs.

Like before, we define the meaning of $\mathrm{PGLD_{dij}}$ programs by means of a function $\mathtt{pglddij2pgldij}$ from the set of all $\mathrm{PGLD_{dij}}$ programs to the set of all $\mathrm{PGLD_{ij}}$ programs. This function is defined by

$$
\begin{aligned}
&\mathtt{pglddij2pgldij}(u_1 \,;\, \ldots \,;\, u_k) = \\
&\quad \psi(u_1) \,;\, \ldots \,;\, \psi(u_k) \,;\, \#\#0 \,;\, \#\#0 \,;\, \overbrace{\#\#0 \,;\, \ldots \,;\, \#\#0}^{\max(k+2,N)-(k+2)} \,; \\
&\quad +\mathsf{regs.eq}{:}1{:}1 \,;\, \mathsf{i}\#\#1 \,;\, \ldots \,;\, +\mathsf{regs.eq}{:}1{:}n \,;\, \mathsf{i}\#\#n \,;\, \#\#0 \,; \\
&\qquad \vdots \\
&\quad +\mathsf{regs.eq}{:}I{:}1 \,;\, \mathsf{i}\#\#1 \,;\, \ldots \,;\, +\mathsf{regs.eq}{:}I{:}n \,;\, \mathsf{i}\#\#n \,;\, \#\#0 \,,
\end{aligned}
$$

where $n = \min(I, N)$ and the auxiliary function ψ from the set of all primitive instructions of $\mathrm{PGLD}_{\mathrm{dij}}$ to the set of all primitive instructions of $\mathrm{PGLD}_{\mathrm{ij}}$ is defined as follows:

$$\begin{aligned}
\psi(\#\#l) &= \#\#l \quad \mathsf{if}\ l \leq k\ ,\\
\psi(\#\#l) &= \#\#0 \quad \mathsf{if}\ l > k\ ,\\
\psi(\mathsf{i}\#\#i) &= \mathsf{i}\#\#i\ ,\\
\psi(\mathsf{ii}\#\#i) &= \#\#l_i\ ,\\
\psi(u) &= u \qquad \mathsf{if}\ u \text{ is not a jump instruction}\ ,
\end{aligned}$$

and for each $i \in [1, I]$:

$$l_i = N + 1 + (2 \cdot \min(I, N) + 1) \cdot (i - 1)\ .$$

The idea is that each double indirect absolute jump can be replaced by an indirect absolute jump to the beginning of the instruction sequence

$$+\mathsf{regs.eq}{:}i{:}1\ ;\ \mathsf{i}\#\#1\ ;\ \ldots\ ;\ +\mathsf{regs.eq}{:}i{:}n\ ;\ \mathsf{i}\#\#n\ ;\ \#\#0\ ,$$

where i is the register concerned and $n = \min(I, N)$. The execution of this instruction sequence leads to the intended jump after the content of the register concerned has been found by a linear search. 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, $\#\#0$; $\#\#0$ is appended to $\psi(u_1)\ ;\ \ldots\ ;\ \psi(u_k)$. Because the length of the translated program is greater than k, care is taken that there are no direct absolute jumps to instructions with a position greater than k. To deal properly with indirect absolute jumps to instructions with a position greater than k, the instruction $\#\#0$ is appended to $\psi(u_1)\,;\ldots;\,\psi(u_k)\,;\#\#0\,;\#\#0$ a sufficient number of times.

Let P be a $\mathrm{PGLD}_{\mathrm{dij}}$ program. Then $\mathtt{pglddij2pgldij}(P)$ represents the meaning of P as a $\mathrm{PGLD}_{\mathrm{ij}}$ program. The intended behaviour of program P is the behaviour of $\mathtt{pglddij2pgldij}(P)$. That is, the *behaviour* of P, written $|P|_{\mathrm{PGLD}_{\mathrm{dij}}}$, is $|\mathtt{pglddij2pgldij}(P)|_{\mathrm{PGLD}_{\mathrm{ij}}}$.

The projection `pglddij2pgldij` uses indirect absolute jumps to obtain the effect of a double indirect absolute jump in the same way as the projection `pgldij2pgld` uses direct absolute jumps to obtain the effect of an indirect absolute jump. Likewise, indirect relative jumps can be used in that way to obtain the effect of a double indirect relative jump. Moreover, double indirect jumps can be used in that way to obtain the effect of a triple indirect jump, and so on.

11 Stack Services

In this section, we give a state-based description of the very simple family of services that constitute a bounded stack of which the elements are natural numbers up to some bound. This stack will be used in Section 12 to describe the

behaviour of programs in a variant of PGLD with returning jump instructions and return instructions.

It is assumed that a fixed but arbitrary number J has been given, which is considered the greatest length of the stack. It is also assumed that a fixed but arbitrary number N has been given, which is considered the greatest natural number that can be an element of the stack.

The stack services accept the following methods:

- for each $n \in [0, N]$, a *stack push method* $\mathsf{push}{:}n$;
- for each $n \in [0, N]$, a *stack top test method* $\mathsf{topeq}{:}n$;
- a *stack pop method* pop.

We write $\mathcal{M}_{\mathsf{stack}}$ for the set $\{\mathsf{push}{:}n, \mathsf{topeq}{:}n \mid n \in [0, N]\} \cup \{\mathsf{pop}\}$. It is assumed that $\mathcal{M}_{\mathsf{stack}} \subseteq \mathcal{M}$.

The methods of stack services can be explained as follows:

- $\mathsf{push}{:}n$: if the length of the stack is less than J, then the number n is put on top of the stack and the reply is T; otherwise nothing changes and the reply is F;
- $\mathsf{topeq}{:}n$: if the stack is not empty and the number on top of the stack is n, then nothing changes and the reply is T; otherwise nothing changes and the reply is F;
- pop : if the stack is not empty, then the number on top of the stack is removed from the stack and the reply is T; otherwise nothing changes and the reply is F.

Let $s \in [0, N]^*$ be such that $\mathrm{len}(s) \leq J$. Then we write Stack_s for the service with initial state s described by $S = \{\sigma \in [0, N]^* \mid \mathrm{len}(\sigma) \leq J\} \cup \{\uparrow\}$, where $\uparrow \notin \{\sigma \in [0, N]^* \mid \mathrm{len}(\sigma) \leq J\}$, and the functions *eff* and *yld* defined as follows ($n, n' \in [0, N]$, $\sigma \in [0, N]^*$):[5]

$$
\begin{aligned}
&\mathit{eff}(\mathsf{push}{:}n, \sigma) = \langle n \rangle \frown \sigma \text{ if } \mathrm{len}(\sigma) < J \ ,\\
&\mathit{eff}(\mathsf{push}{:}n, \sigma) = \sigma \qquad \text{if } \mathrm{len}(\sigma) \geq J \ ,\\
&\mathit{eff}(\mathsf{topeq}{:}n, \sigma) = \sigma \ ,\\
&\mathit{eff}(\mathsf{pop}, \langle n \rangle \frown \sigma) = \sigma \ ,\\
&\mathit{eff}(\mathsf{pop}, \langle\,\rangle) = \langle\,\rangle \ ,\\
&\mathit{eff}(m, \sigma) = \uparrow \qquad \text{if } m \notin \mathcal{M}_{\mathsf{stack}} \ ,\\
&\mathit{eff}(m, \uparrow) = \uparrow \ ,
\end{aligned}
$$

[5] 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.

$$
\begin{aligned}
&\mathit{yld}(\mathsf{push}{:}n, \sigma) = \mathsf{T} && \text{if } \mathrm{len}(\sigma) < J \;,\\
&\mathit{yld}(\mathsf{push}{:}n, \sigma) = \mathsf{F} && \text{if } \mathrm{len}(\sigma) \geq J \;,\\
&\mathit{yld}(\mathsf{topeq}{:}n, \langle n' \rangle \frown \sigma) = \mathsf{T} && \text{if } n = n' \;,\\
&\mathit{yld}(\mathsf{topeq}{:}n, \langle n' \rangle \frown \sigma) = \mathsf{F} && \text{if } n \neq n' \;,\\
&\mathit{yld}(\mathsf{topeq}{:}n, \langle\,\rangle) = \mathsf{F} \;,\\
&\mathit{yld}(\mathsf{pop}, \langle n \rangle \frown \sigma) = \mathsf{T} \;,\\
&\mathit{yld}(\mathsf{pop}, \langle\,\rangle) = \mathsf{F} \;,\\
&\mathit{yld}(m, \sigma) = \mathsf{B} && \text{if } m \notin \mathcal{M}_{\mathsf{stack}} \;,\\
&\mathit{yld}(m, \uparrow) = \mathsf{B} \;.
\end{aligned}
$$

We write $\mathit{Stack}_{\mathrm{init}}$ for $\mathit{Stack}_{\langle\,\rangle}$.

12 PGLD with Returning Jumps and Returns

In this section, we introduce a variant of PGLD with returning jump instructions and return instructions. This variant is called $\mathrm{PGLD}_{\mathrm{rj}}$.

In $\mathrm{PGLD}_{\mathrm{rj}}$, like in $\mathrm{PGLD}_{\mathrm{ij}}$, it is assumed that there is a fixed but arbitrary finite set of *foci* $\mathcal{F}$ with $\mathsf{stack} \in \mathcal{F}$ and a fixed but arbitrary finite set of *methods* $\mathcal{M}$. Moreover, we adopt the assumptions made about stack services in Section 11. Like in $\mathrm{PGLD}_{\mathrm{ij}}$, the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$ is taken as the set $\mathfrak{A}$ of basic instructions.

$\mathrm{PGLD}_{\mathrm{rj}}$ 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* $\#\#l$;
- for each $l \in \mathbb{N}$, a *returning absolute jump instruction* $\mathsf{r}\#\#l$;
- an *absolute return instruction* $\#\#\mathsf{r}$.

$\mathrm{PGLD}_{\mathrm{rj}}$ programs have the form $u_1 \,;\, \ldots \,;\, u_k$, where $u_1, \ldots, u_k$ are primitive instructions of $\mathrm{PGLD}_{\mathrm{rj}}$.

The plain basic instructions, the positive test instructions, the negative test instructions, and the absolute jump instructions are as in PGLD. The effect of a returning absolute jump instruction $\mathsf{r}\#\#l$ is that execution proceeds with the l-th instruction of the program concerned, but execution returns to the next primitive instruction on encountering a return instruction. If $\mathsf{r}\#\#l$ is itself the l-th instruction, then deadlock occurs. If l equals 0 or l is greater than the length of the program, termination occurs. The effect of a return instruction $\#\#\mathsf{r}$ is that execution proceeds with the instruction immediately following the last executed returning absolute jump instruction to which a return has not yet taken place.

Like before, we define the meaning of $\mathrm{PGLD}_{\mathrm{rj}}$ programs by means of a function `pgldrj2pgld` from the set of all $\mathrm{PGLD}_{\mathrm{rj}}$ programs to the set of all PGLD programs. This function is defined by

$$
\begin{array}{l}
\texttt{pgldrj2pgld}(u_1 \,;\, \ldots \,;\, u_k) = \\
\quad \psi_1(u_1) \,;\, \ldots \,;\, \psi_k(u_k) \,;\, \#\#0 \,;\, \#\#0 \,; \\
\quad +\mathsf{stack.push}{:}1 \,;\, \#\#1 \,;\, \#\#l'' \,;\, \ldots \,;\, +\mathsf{stack.push}{:}1 \,;\, \#\#k \,;\, \#\#l'' \,; \\
\quad\quad \vdots \\
\quad +\mathsf{stack.push}{:}n \,;\, \#\#1 \,;\, \#\#l'' \,;\, \ldots \,;\, +\mathsf{stack.push}{:}n \,;\, \#\#k \,;\, \#\#l'' \,; \\
\quad -\mathsf{stack.topeq}{:}1 \,;\, \#\#l''_1 \,;\, \mathsf{stack.pop} \,;\, \#\#1 \,; \\
\quad\quad \vdots \\
\quad -\mathsf{stack.topeq}{:}n \,;\, \#\#l''_n \,;\, \mathsf{stack.pop} \,;\, \#\#n \,; \\
\quad \#\#l'' \,,
\end{array}
$$

where $n = \min(k, N)$ and the auxiliary functions ψ_j from the set of all primitive instructions of $\mathrm{PGLD}_{\mathrm{rj}}$ to the set of all primitive instructions of PGLD is defined as follows ($1 \leq j \leq k$):

$$
\begin{array}{lll}
\psi_j(\#\#l) & = \#\#l & \text{if } l \leq k \,, \\
\psi_j(\#\#l) & = \#\#0 & \text{if } l > k \,, \\
\psi_j(\mathsf{r}\#\#l) & = \#\#l_{j,l} \,, & \\
\psi_j(\#\#\mathsf{r}) & = \#\#l' \,, & \\
\psi_j(u) & = u & \text{if } u \text{ is not a jump instruction} \,,
\end{array}
$$

and for each $j \in [1, k]$, $l \in \mathbb{N}$, and $h \in [1, \min(k, N)]$:

$$
\begin{array}{lll}
l_{j,l} = k + 3 + 3 \cdot k \cdot ((j-1) + (l-1)) & & \text{if } l \leq k \wedge j \leq N \,, \\
l_{j,l} = j & & \text{if } l \leq k \wedge j > N \,, \\
l_{j,l} = 0 & & \text{if } l > k \,,
\end{array}
$$

$$
l' = k + 3 + 3 \cdot k \cdot \min(k, N) \,,
$$

$$
l'' = l' + 4 \cdot \min(k, N) \,,
$$

$$
l''_h = l' + 4 \cdot h \,.
$$

The first idea is that each returning absolute jump can be replaced by an absolute jump to the beginning of the instruction sequence

$$
+\mathsf{stack.push}{:}j \,;\, \#\#l \,;\, \#\#l'' \,,
$$

where j is the position of the returning absolute jump instruction concerned and l is the position of the instruction to jump to. The execution of this instruction

sequence leads to the intended jump after the return position has been put on the stack. In the case of stack overflow, deadlock occurs. The second idea is that each return can be replaced by an absolute jump to the beginning of the instruction sequence

$$
\begin{aligned}
&-\mathsf{stack.topeq}{:}1\ ;\ \#\#l_1''\ ;\ \mathsf{stack.pop}\ ;\ \#\#1\ ;\\
&\quad\vdots\\
&-\mathsf{stack.topeq}{:}n\ ;\ \#\#l_n''\ ;\ \mathsf{stack.pop}\ ;\ \#\#n\ ;\\
&\#\#l''\ ,
\end{aligned}
$$

where $n = \min(k, N)$. The execution of this instruction sequence leads to the intended jump after the position on the top of the stack has been found by a linear search and has been removed from the stack. In the case of an empty stack, deadlock occurs. 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, $\#\#0;\#\#0$ is appended to $\psi_1(u_1);\ldots;\psi_k(u_k)$. Because the length of the translated program is greater than k, care is taken that there are no non-returning or returning absolute jumps to instructions with a position greater than k.

Let P be a $\mathrm{PGLD}_{\mathrm{rj}}$ program. Then $\mathtt{pgldrj2pgld}(P)$ represents the meaning of P as a PGLD program. The intended behaviour of P is the behaviour of $\mathtt{pgldrj2pgld}(P)$ on interaction with a stack. That is, the *behaviour* of P, written $|P|_{\mathrm{PGLD}_{\mathrm{rj}}}$, is $|\mathtt{pgldrj2pgld}(P)|_{\mathrm{PGLD}} \,/_{\mathsf{stack}}\, \mathit{Stack}_{\mathrm{init}}$.

According to the definition of the behaviour of $\mathrm{PGLD}_{\mathrm{rj}}$ programs given above, the execution of a returning jump instruction leads to deadlock in the case where its position cannot be pushed on the stack and the execution of a return instruction leads to deadlock in the case where there is no position to be popped from the stack. In the latter case, the return instruction is wrongly used. In the former case, however, the returning jump instruction is not wrongly used, but the finiteness of the stack comes into play. This shows that the definition of the behaviour of $\mathrm{PGLD}_{\mathrm{rj}}$ programs given here takes into account the finiteness of the execution environment of programs.

13 Conclusions

We have studied sequential programs that are instruction sequences with direct and indirect jump instructions. We have considered several kinds of indirect jumps, including return instructions. For each kind, we have defined the meaning of programs with indirect jump instructions of that kind by means of a translation into programs without indirect jump instructions. Each translation determines, together with some memory device (a register file or a stack), the behaviour of the programs concerned under execution.

The increase in the length of a program as a result of translation can be reduced by taking into account which indirect jump instructions actually occur

in the program. The increase in the number of steps needed by a program as a result of translation can be reduced by replacing linear searching by binary searching or another more efficient kind of searching. One option for future work is to look for bounds on the increase in length and the increase in number of steps.

In [8], we have modelled and analysed micro-architectures with pipelined instruction processing in the setting of program algebra, basic thread algebra, and Maurer computers [13, 14]. In that work, which we consider a preparatory step in the development of a formal approach to design new micro-architectures, indirect jump instructions were not taken into account. Another option for future work is to look at the effect of indirect jump instructions on pipelined instruction processing.

References

1. J. A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings 30th ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.
2. 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*. Elsevier, Amsterdam, 2007.
3. J. A. Bergstra and M. E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
4. J. A. Bergstra and C. A. Middelburg. Splitting bisimulations and retrospective conditions. *Information and Computation*, 204(7):1083–1138, 2006.
5. J. A. Bergstra and C. A. Middelburg. Thread algebra with multi-level strategies. *Fundamenta Informaticae*, 71(2/3):153–182, 2006.
6. J. A. Bergstra and C. A. Middelburg. Distributed strategic interleaving with load balancing. *Future Generation Computer Systems*, 2007. In press, doi: 10.1016/j.future.2007.08.001.
7. J. A. Bergstra and C. A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 19(4):445–474, 2007.
8. J. A. Bergstra and C. A. Middelburg. Maurer computers for pipelined instruction processing. *Mathematical Structures in Computer Science*, 2008. In press, doi: 10.1017/S0960129507006548.
9. J. A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
10. W. J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin, 2000.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Reading, MA, second edition, 2000.
12. A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley, Reading, MA, 2003.
13. W. D. Maurer. A theory of computer instructions. *Journal of the ACM*, 13(2):226–235, 1966.
14. W. D. Maurer. A theory of computer instructions. *Science of Computer Programming*, 60:244–273, 2006.

15. A. Ponse. Program algebra with unit instruction operators. *Journal of Logic and Algebraic Programming*, 51(2):157–174, 2002.
16. 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.
17. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, Amsterdam, 1990.

Within this series the following reports appeared.

[PRG0708] B. Diertens, *Software (Re-)Engineering with PSF III: an IDE for PSF*, Programming Research Group - University of Amsterdam, 2007.

[PRG0707] J.A. Bergstra and C.A. Middelburg, *An Interface Group for Process Components*, Programming Research Group - University of Amsterdam, 2007.

[PRG0706] J.A. Bergstra, Y. Hirschfeld, and J.V. Tucker, *Skew Meadows*, Programming Research Group - University of Amsterdam, 2007.

[PRG0705] J.A. Bergstra, Y. Hirschfeld, and J.V. Tucker, *Meadows*, Programming Research Group - University of Amsterdam, 2007.

[PRG0704] J.A. Bergstra and C.A. Middelburg, *Machine Structure Oriented Control Code Logic (Extended Version)*, Programming Research Group - University of Amsterdam, 2007.

[PRG0703] J.A. Bergstra and C.A. Middelburg, *On the Operating Unit Size of Load/Store Architectures*, Programming Research Group - University of Amsterdam, 2007.

[PRG0702] J.A. Bergstra and A. Ponse, *Interface Groups and Financial Transfer Architectures*, Programming Research Group - University of Amsterdam, 2007.

[PRG0701] J.A. Bergstra, I. Bethke, and M. Burgess, *A Process Algebra Based Framework for Promise Theory*, Programming Research Group - University of Amsterdam, 2007.

[PRG0610] J.A. Bergstra and C.A. Middelburg, *Parallel Processes with Implicit Computational Capital*, Programming Research Group - University of Amsterdam, 2006.

[PRG0609] B. Diertens, *Software (Re-)Engineering with PSF II: from architecture to implementation*, Programming Research Group - University of Amsterdam, 2006.

[PRG0608] A. Ponse and M.B. van der Zwaag, *Risk Assessment for One-Counter Threads*, Programming Research Group - University of Amsterdam, 2006.

[PRG0607] J.A. Bergstra and C.A. Middelburg, *Synchronous Cooperation for Explicit Multi-Threading*, Programming Research Group - University of Amsterdam, 2006.

[PRG0606] J.A. Bergstra and M. Burgess, *Local and Global Trust Based on the Concept of Promises*, Programming Research Group - University of Amsterdam, 2006.

[PRG0605] J.A. Bergstra and J.V. Tucker, *Division Safe Calculation in Totalised Fields*, Programming Research Group - University of Amsterdam, 2006.

[PRG0604] J.A. Bergstra and A. Ponse, *Projection Semantics for Rigid Loops*, Programming Research Group - University of Amsterdam, 2006.

[PRG0603] J.A. Bergstra and I. Bethke, *Predictable and Reliable Program Code: Virtual Machine-based Projection Semantics (submitted for inclusion in the Handbook of Network and Systems Administration)*, Programming Research Group - University of Amsterdam, 2006.

[PRG0602] J.A. Bergstra and A. Ponse, *Program Algebra with Repeat Instruction*, Programming Research Group - University of Amsterdam, 2006.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

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