University of Amsterdam
Programming Research Group

Instruction Sequences with Dynamically Instantiated Instructions

J.A. Bergstra
C.A. Middelburg

Report PRG0710 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 Dynamically Instantiated Instructions*

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 dynamically instantiated instructions. We define the meaning of such programs in two different ways. In either case, we give a translation by which each program with dynamically instantiated instructions is turned into a program without them that exhibits on execution the same behaviour by interaction with some service. The complexity of the translations differ considerably, whereas the services concerned are equally simple. However, the service concerned in the case of the simpler translation is far more powerful than the service concerned in the other case.

Keywords: instruction sequence, dynamically instantiated instruction, program algebra, projection semantics, thread algebra, action transforming thread-service composition.

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

1 Introduction

In this paper, we study sequential programs that are instruction sequences with dynamically instantiated instructions. With that we carry on the line of research with which a start was made in [3]. The object pursued with this line of research is the development of a theoretical understanding of possible forms of sequential programs, starting from the simplest form. The view is taken that sequential programs in the simplest form are sequences of instructions. Program algebra, an algebra of programs in which programs are looked upon as sequences of instructions, is taken for the basis of the development aimed at.

The approach to define the meaning of programs followed in this line of research is called projection semantics. It explains the meaning of programs in terms of known programs instead of more or less sophisticated mathematical objects that represent behaviours of programs under execution. The main advantage of projection semantics is that it does not require a lot of mathematical

* 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).

background. Over and above that, the view is taken that the behaviours of sequential programs under execution are threads as considered in basic thread algebra [3].[3] Therefore, the meaning of the programs considered in program algebra is explained in terms of threads. The experience gained so far leads us to believe that sequential programs are nothing but linear representations of threads.

Sequential programs in the form of assembly programs up to and including sequential programs in the form of structured programs are covered in [3]. However, although they are found in existing assembly programming practice, indirect jump instructions are not considered. In [5], several kinds of indirect jump instructions are considered, including a kind by which recursive method calls can easily be explained.

Dynamic instruction instantiation is a programming feature that is not suggested by existing programming practice. However, from the viewpoint that sequential programs are nothing but linear representations of threads, it is a genuine programming feature. It is a useful programming feature as well, as will be illustrated by means of an example in the paper. Therefore, we consider a theoretical understanding of instruction sequences with dynamically instantiated instructions relevant to programming.

We believe that interaction with services provided by an execution environment is inherent in the behaviour of programs under execution. Intuitively, some service provides for dynamic instruction instantiation. In this paper, we define the meaning of programs with dynamically instantiated instructions in two different ways. In either case, we give a translation by which each program with dynamically instantiated instructions is turned into a program without them that exhibits on execution the same behaviour by interaction with some service. In one case, the service concerned provides in effect for the dynamic instruction instantiation and, in the other case, it is largely achieved by the translated programs. We also describe the services concerned.

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. The interaction between behaviours of programs under execution and some service referred to above is an interaction with that purpose. In [6], the use mechanism is introduced to allow for such a kind of interaction between threads and services. In this paper, we will use a generalization of the use mechanism, called action transforming thread-service composition, to have behaviours of programs under execution affected by services. This generalization is reminiscent of the state operator introduced in [1].

A hierarchy of program notations rooted in program algebra is introduced in [3]. In this paper, we embroider on one program notation that belongs to this hierarchy. The program notation in question, called PGLD, is a very simple program notation which is close to existing assembly languages. The hierarchy

[3] In [3], basic thread algebra is introduced under the name basic polarized process algebra. Prompted by the development of thread algebra [6], which is a design on top of it, basic polarized process algebra has been renamed to basic thread algebra.

also includes a program notation, called PGLS, that supports structured programming by offering conditional and loop constructs instead of (unstructured) jump instructions. Each PGLS program can be translated into a semantically equivalent PGLD program by means of the projection semantics of PGLS and some intermediate program notations.

This paper is organized as follows. First, we review basic thread algebra, program algebra, and the program notation PGLD (Sections 2, 3, and 4). Next, we extend basic thread algebra with action transforming thread-service composition and introduce a state-based approach to describe services (Sections 5 and 6). Following this, we give a state-based description of a service that can provide for dynamic instruction instantiation and use that service to define the meaning of the programs from a variant of the program notation PGLD with dynamically instantiated instructions (Sections 7 and 8). Then, we give a state-based description of a register service and use that service to define the meaning of the programs from the variant of the program notation PGLD with dynamically instantiated instructions in another way (Sections 9 and 10). After that, we discuss the semantic approaches followed in the preceding sections and introduce a concrete notation for basic instructions that covers dynamically instantiated 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 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* $\mathcal{A}$ with $\mathsf{tau} \notin \mathcal{A}$. We write $\mathcal{A}_{\mathsf{tau}}$ for $\mathcal{A} \cup \{\mathsf{tau}\}$. The members of $\mathcal{A}_{\mathsf{tau}}$ are referred to as *actions*.

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 $a \in \mathcal{A}_{\mathsf{tau}}$, the binary *postconditional composition* operator $_ \trianglelefteq a \trianglerighteq _ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

Table 1. Axiom of BTA

$x \trianglelefteq \mathsf{tau} \trianglerighteq y = x \trianglelefteq \mathsf{tau} \trianglerighteq x$	T1

Terms of sort $\mathbf{T}$ are built as usual (see e.g. [13, 14]). 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 $a \in \mathcal{A}_{\mathsf{tau}}$. 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). The action tau plays a special role. It is a concrete internal action: performing tau will never lead to a state change and always lead to a positive reply, but notwithstanding all that its presence matters.

BTA has only one axiom. This axiom is given in Table 1. Using the abbreviation introduced above, axiom T1 can be written as follows: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

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 *recursive specification* over BTA is a set of recursion equations $\{X = t_X \mid X \in V\}$ where V is a set of variables of sort $\mathbf{T}$ and each t_X is a BTA term of sort $\mathbf{T}$ that contains only variables from V. Let E be a recursive specification over BTA. Then we write $\mathrm{V}(E)$ for the set of all variables that occur on the left-hand side of an equation in E. A *solution* of a recursive specification E is a set of threads (in some model of BTA) $\{T_X \mid X \in \mathrm{V}(E)\}$ such that the equations of E hold if, for all $X \in \mathrm{V}(E)$, X stands for T_X.

Let t be a BTA term of sort $\mathbf{T}$ containing a variable X of sort $\mathbf{T}$. Then an occurrence of X in t is *guarded* if t has a subterm of the form $t' \trianglelefteq a \trianglerighteq t''$ containing this occurrence of X. Let E be a recursive specification over BTA. Then E is a *guarded recursive specification* if, in each equation $X = t_X \in E$, all occurrences of variables in t_X are guarded or t_X can be rewritten to such a term using the equations in E from left to right. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in [2]. A thread that is the solution of a finite guarded recursive specification over BTA is called a *finite-state* thread.

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 2 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

Table 2. 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

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 use the following abbreviation: a^ω, where $a \in \mathcal{A}_{\mathsf{tau}}$, abbreviates $\langle X|\{X = a \circ X\}\rangle$.

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

In [4], we show that the threads considered in BTA+REC can be viewed as processes that are definable over ACP [9].

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.

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

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

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 3. 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 $\mathcal{A}$ of actions. The *thread extraction* operator $|_|$ assigns a thread to each program. The thread extraction operator 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 given in Table 5. This rule is expressed in terms of the *structural congruence* predicate $_ \cong _$, which is defined by the formulas given in Table 6 (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 4 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 5 can be read as follows: if x starts with a cyclic chain of forward jumps, then $|x|$ 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 5 can be replaced by the following generalization: $x \cong y \Rightarrow |x| = |y|$.

Table 4. 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 5. Rule for cyclic jump chains

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

Table 6. Defining formulas for structural congruence predicate

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

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 (see Proposition 2 of [12]).

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 Notation PGLD

In this section, we review a program notation which is rooted in PGA. This program notation, called PGLD, belongs to a hierarchy of program notations

introduced in [3]. PGLD is close to existing assembly languages. It has absolute jump instructions and no explicit termination instruction.

In 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.

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. We write $\mathcal{P}_{\text{PGLD}}$ for the set of all PGLD programs.

The plain basic instructions, the positive test instructions, and the negative test instructions are as in PGA. 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 `pgld2pga` from the set of all PGLD programs to the set of all PGA programs. This function is defined by

$$\texttt{pgld2pga}(u_1 \,;\ldots;\, u_k) = (\phi_1(u_1) \,;\ldots;\, \phi_k(u_k) \,;\, ! \,;\, !)^\omega \;,$$

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

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

Let P be a PGLD program. Then $\texttt{pgld2pga}(P)$ represents the meaning of P as a PGA program. The intended behaviour of P under execution is the behaviour of $\texttt{pgld2pga}(P)$ under execution. That is, the *behaviour* of P under execution, written $|P|_{\text{PGLD}}$, is $|\texttt{pgld2pga}(P)|$.

We use the phrase *projection semantics* to refer to the approach to semantics followed in this section. The meaning function `pgld2pga` is called a *projection*.

In the hierarchy of program notations introduced in [3], program notations PGLA, PGLB and PGLC appear between PGA and PGLD. In [3], PGLD programs are translated into PGLC programs by means of a projection `pgld2pglc`, etc. Above, `pgld2pga` is defined such that $\texttt{pgld2pga}(P) = \texttt{pgld2pglc}(\texttt{pglc2pglb}(\texttt{pglb2pgla}(\texttt{pgla2pga}(P))))$ for all PGLD program P.

5 Action Transforming Thread-Service Composition

A thread may perform certain basic actions only for the sake of having itself affected by a service. When processing a basic action performed by a thread, a service affect that thread in one of the following ways: (i) by returning a reply value to the thread at completion of the processing of the basic action performed by the thread; (ii) by turning the processed basic action into another basic action. In this section, we introduce action transforming thread-service composition, which allows for such interaction between threads and services. We will only use action transforming thread-service composition to have program behaviours affected by a service. Action transforming thread-service composition is a generalization of the use mechanism introduced in [6].[4]

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 basic actions, we take the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. Performing a basic action $f.m$ is taken as making a request to the service named f to process command m.

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 considered to stand for the set of all services. We identify services with pairs (H_1, H_2), where $H_1 : \mathcal{M}^+ \to \{\mathsf{T}, \mathsf{F}, \mathsf{M}, \mathsf{B}\}$ and $H_2 : \mathcal{M}^+ \to \mathcal{A}_{\mathsf{tau}}$, satisfying the following conditions:

$$\forall m \in \mathcal{M} \bullet$$
$$(\exists \alpha \in \mathcal{M}^* \bullet H_1(\alpha \frown \langle m \rangle) = \mathsf{M} \Rightarrow \forall \alpha' \in \mathcal{M}^* \bullet H_1(\alpha' \frown \langle m \rangle) \notin \{\mathsf{T}, \mathsf{F}\}) \ ,$$

$$\forall \alpha \in \mathcal{M}^+, m \in \mathcal{M} \bullet (H_1(\alpha) = \mathsf{B} \Rightarrow H_1(\alpha \frown \langle m \rangle) = \mathsf{B}) \ ,$$

$$\forall \alpha \in \mathcal{M}^+ \bullet (H_1(\alpha) \neq \mathsf{M} \Leftrightarrow H_2(\alpha) = \mathsf{tau}) \ .$$

Let H be a service, and let H_1 and H_2 be the unique functions such that $H = (H_1, H_2)$. Then we write $rf(H)$ and $af(H)$ for H_1 and H_2, respectively. Given a service H and a method $m \in \mathcal{M}$, the *derived service* of H after processing m, written $\frac{\partial}{\partial m} H$, is defined by $rf(\frac{\partial}{\partial m} H)(\alpha) = rf(H)(\langle m \rangle \frown \alpha)$ and $af(\frac{\partial}{\partial m} H)(\alpha) = af(H)(\langle m \rangle \frown \alpha)$.

A service H can be understood as follows:

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

[4] In later papers, this use mechanism is called thread-service composition.

Table 7. Axioms for action transforming thread-service composition

$\mathsf{S} /_f H = \mathsf{S}$		ATTSC1
$\mathsf{D} /_f H = \mathsf{D}$		ATTSC2
$\mathsf{tau} \circ x /_f H = \mathsf{tau} \circ (x /_f H)$		ATTSC3
$(x \trianglelefteq g.m \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq g.m \trianglerighteq (y /_f H)$	if $f \neq g$	ATTSC4
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{tau} \circ (x /_f \frac{\partial}{\partial m} H)$	if $rf(H)(\langle m \rangle) = \mathsf{T}$	ATTSC5
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{tau} \circ (y /_f \frac{\partial}{\partial m} H)$	if $rf(H)(\langle m \rangle) = \mathsf{F}$	ATTSC6
$(x \trianglelefteq f.m \trianglerighteq y) /_f H =$ $(x /_f \frac{\partial}{\partial m} H) \trianglelefteq af(H)(\langle m \rangle) \trianglerighteq (y /_f \frac{\partial}{\partial m} H)$	if $rf(H)(\langle m \rangle) = \mathsf{M}$	ATTSC7
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{D}$	if $rf(H)(\langle m \rangle) = \mathsf{B}$	ATTSC8

- if $rf(H)(\langle m \rangle) = \mathsf{M}$, then the request to process m is accepted by the service, no reply is produced, m is turned into $af(H)(\langle m \rangle)$, and the service proceeds as $\frac{\partial}{\partial m} H$;
- if $rf(H)(\langle m \rangle) = \mathsf{B}$, then the request to process m is rejected by the service.

The three conditions imposed on services can be paraphrased as follows:

- if it is possible that no reply is produced at completion of the processing of a command, then it is impossible that a positive or negative reply is produced at completion of the processing of that command;
- after a request to process a command has been rejected, any request to process a command will be rejected;
- a reply is produced at completion of the processing of a command if and only if the command is turned into tau.

For each $f \in \mathcal{F}$, we introduce the binary *action transforming 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 basic actions performed by thread p that are of the form $f.m$ by the service H. When a basic action of the form $f.m$ performed by thread p is processed by the service H, it is turned into another action and, if this action is tau, postconditional composition is removed in favour of action prefixing on the basis of the reply value produced.

The axioms for the action transforming thread-service composition operator are given in Table 7. In this table, f and g stand for an arbitrary foci from $\mathcal{F}$ and m stands for an arbitrary method from $\mathcal{M}$. Axioms ATTSC3 and ATTSC4 express that the action tau and basic actions of the form $g.m$, where $f \neq g$, are not processed. Axioms ATTSC5–ATTSC7 express that a thread is affected by a service as described above when a basic action of the form $f.m$ performed by the thread is processed by the service. Axiom ATTSC7 expresses that deadlock takes place when a basic action to be processed is not accepted.

Let T stand for either BTA or BTA+REC. Then we will write T+ATTSC for T, taking the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$ for $\mathcal{A}$, extended with the action transforming thread-service composition operators and the axioms from Table 7.

Table 8. Axioms for abstraction

$\tau_{\mathsf{tau}}(\mathsf{S}) = \mathsf{S}$	TT1
$\tau_{\mathsf{tau}}(\mathsf{D}) = \mathsf{D}$	TT2
$\tau_{\mathsf{tau}}(x \trianglelefteq \mathsf{tau} \trianglerighteq y) = \tau_{\mathsf{tau}}(x)$	TT3
$\tau_{\mathsf{tau}}(x \trianglelefteq a \trianglerighteq y) = \tau_{\mathsf{tau}}(x) \trianglelefteq a \trianglerighteq \tau_{\mathsf{tau}}(y)$	TT4
$\tau_{\mathsf{tau}}(\mathsf{tau}^{\omega}) = \mathsf{D}$	TT5

The use mechanism introduced in [6] deals in essence with services H where $af(H)(\alpha) = \mathsf{tau}$ for all $\alpha \in \mathcal{M}^{+}$. For these services, action transforming thread-service composition coincides with the use mechanism.

The action tau is an internal action whose presence matters. To conceal its presence in the case where it does not matter after all, we also introduce the unary *abstraction* operator $\tau_{\mathsf{tau}} : \mathbf{T} \to \mathbf{T}$.

The axioms for the abstraction operator are given in Table 8. In this table, a stands for an arbitrary basic action from $\mathcal{A}$.

Abstraction is for instance presumably appropriate in the case where tau arises only from turning basic actions of an auxiliary nature into tau on action transforming thread-service composition. Examples of this case will occur later on. Unlike the use mechanism introduced in [6], the use mechanism introduced in [8] incorporates abstraction.

Let T stand for either BTA+REC or BTA+REC+ATTSC. Then we will write T+ABSTR for T extended with the abstraction operator and the axioms from Table 8.

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 [8].

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 $\mathit{yld} : \mathcal{M} \times S \to \{\mathsf{T}, \mathsf{F}, \mathsf{M}, \mathsf{B}\}$;
- an action function $\mathit{act} : M \times S \to \mathcal{A}_{\mathsf{tau}}$;

satisfying the following conditions:

$$\forall m \in M \bullet (\exists s \in S \bullet \mathit{yld}(m, s) = \mathsf{M} \Rightarrow \forall s' \in S \bullet \mathit{yld}(m, s') \notin \{\mathsf{T}, \mathsf{F}\}) \ ,$$

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

$$\forall m \in \mathcal{M}, s \in S \bullet (\mathit{yld}(m, s) \neq \mathsf{M} \Leftrightarrow \mathit{act}(m, s) = \mathsf{tau}) \ .$$

The set S contains the states in which the services may be, and the functions eff, yld and act give, for each method m and state s, the state, reply and action, respectively, that result from processing m in state s.

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

$$\mathit{ceff}_s(\langle\,\rangle) = s \;,$$
$$\mathit{ceff}_s(\alpha \frown \langle m \rangle) = \mathit{eff}(m, \mathit{ceff}_s(\alpha)) \;.$$

We define, for each $s \in S$, a service H_s in terms of ceff_s, yld and act as follows:

$$\mathit{rf}(H_s)(\alpha \frown \langle m \rangle) = \mathit{yld}(m, \mathit{ceff}_s(\alpha)) \;,$$
$$\mathit{af}(H_s)(\alpha \frown \langle m \rangle) = \mathit{act}(m, \mathit{ceff}_s(\alpha)) \;.$$

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

The conditions that are imposed on S, eff, yld and act imply that, for each $s \in S$, H_s is a service indeed. It is worth mentioning that $\frac{\partial}{\partial m} H_s = H_{\mathit{eff}(m,s)}$, $\mathit{rf}(H_s)(\langle m \rangle) = \mathit{yld}(m, s)$, and $\mathit{af}(H_s)(\langle m \rangle) = \mathit{act}(m, s)$.

7 Method to Action Translator Services

In this section, we give a state-based description of the very simple family of services that constitute a register-dependent method to action translator of which the register can contain natural numbers up to some bound. This method to action translator will be used in Section 8 to describe the behaviour of programs in a variant of PGLD with dynamically instantiated instructions.

It is assumed that a fixed but arbitrary number $N \in \mathbb{N}$, a fixed but arbitrary set $\mathcal{A}_{\mathsf{proto}} \subseteq \mathcal{M}$, and a fixed but arbitrary function $\theta : \mathcal{A}_{\mathsf{proto}} \times [0, N] \to \mathcal{A}$ have been given. N is considered the greatest natural number that can be contained in the register involved, $\mathcal{A}_{\mathsf{proto}}$ is considered the set of methods that are transformable to basic actions, and θ is regarded to give, for each method m in $\mathcal{A}_{\mathsf{proto}}$ and natural number n in $[0, N]$, the basic action into which m is turned in the case where the content of the register is n. The methods that belong to $\mathcal{A}_{\mathsf{proto}}$ are called *proto-actions* because they are the methods that are turned into basic actions by the register-dependent method to action translator.

The register-dependent method to action translator services accept the following methods:

- for each $n \in [0, N]$, a *register set method* $\mathsf{set}{:}n$;
- each $m \in \mathcal{A}_{\mathsf{proto}}$.

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

The methods accepted by the method to action translator services can be explained as follows:

- $\mathsf{set}{:}n$: the content of the register becomes n, the reply is T, and $\mathsf{set}{:}n$ is turned into tau;

– m, where $m \in \mathcal{A}_{\mathsf{proto}}$: the content of the register does not change, there is no reply, and m is turned into $\theta(m,n)$ where n is the content of the register.

Let $s \in [0,N]$. Then we write RDT_s for the service with initial state s described by $S = [-1,N]$ and the functions *eff*, *yld* and *act* defined as follows ($n,k \in [0,N]$):

$$
\begin{aligned}
&\mathit{eff}(\mathsf{set}{:}n,k) = n \ , \\
&\mathit{eff}(m,k) = k && \text{if } m \in \mathcal{A}_{\mathsf{proto}} \ , \\
&\mathit{eff}(m,k) = -1 && \text{if } m \notin \mathcal{M}_{\mathsf{set}} \cup \mathcal{A}_{\mathsf{proto}} \ , \\
&\mathit{eff}(m,-1) = -1 \ ,
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{yld}(\mathsf{set}{:}n,k) = \mathsf{T} \ , \\
&\mathit{yld}(m,k) = \mathsf{M} && \text{if } m \in \mathcal{A}_{\mathsf{proto}} \ , \\
&\mathit{yld}(m,k) = \mathsf{B} && \text{if } m \notin \mathcal{M}_{\mathsf{set}} \cup \mathcal{A}_{\mathsf{proto}} \ , \\
&\mathit{yld}(m,-1) = \mathsf{B} \ ,
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{act}(m,k) = \theta(m,k) && \text{if } m \in \mathcal{A}_{\mathsf{proto}} \ , \\
&\mathit{act}(m,k) = \mathsf{tau} && \text{if } m \notin \mathcal{A}_{\mathsf{proto}} \ , \\
&\mathit{act}(m,-1) = \mathsf{tau} \ .
\end{aligned}
$$

We write RDT_{init} for RDT_0.

The following proposition states rigorously that the methods that belong to $\mathcal{A}_{\mathsf{proto}}$ are exactly the methods that are turned into basic actions.

Proposition 1. *For all $s \in [0,N]$:*

$$\mathcal{A}_{\mathsf{proto}} = \{m \in \mathcal{M} \mid \exists \alpha \in \mathcal{M}^* \bullet \mathit{af}(RDT_s)(\alpha \frown \langle m \rangle) \in \mathcal{A}\} \ .$$

Proof. This follows immediately from the definition of the register-dependent method to action translator services. □

8 PGLD with Dynamically Instantiated Instructions

In this section, we introduce a variant of PGLD with dynamically instantiated instructions. This variant is called $\mathrm{PGLD}_{\mathrm{dii}}$. In Section 12, the usefulness of dynamic instruction instantiation will be illustrated by means of an example.

In $\mathrm{PGLD}_{\mathrm{dii}}$, it is assumed that there is a fixed but arbitrary finite set of *foci* $\mathcal{F}$ with $\mathsf{rdt} \in \mathcal{F}$ and a fixed but arbitrary finite set of *methods* $\mathcal{M}$. Moreover, we adopt the assumptions made about register-dependent method to action translator services in Section 7. The set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M} \setminus \mathcal{A}_{\mathsf{proto}}\}$ is taken as the set $\mathfrak{A}$ of basic instructions. In the setting of $\mathrm{PGLD}_{\mathrm{dii}}$, we use the term *proto-instruction* instead of proto-action and write $\mathfrak{A}_{\mathsf{proto}}$ instead of $\mathcal{A}_{\mathsf{proto}}$. A proto-instruction is what becomes a basic instruction by dynamic instantiation.

$\mathrm{PGLD}_{\mathrm{dii}}$ 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 $e \in \mathfrak{A}_{\mathsf{proto}}$, a *plain basic proto-instruction* e;
– for each $e \in \mathfrak{A}_{\mathsf{proto}}$, a *positive test proto-instruction* $+e$;
– for each $e \in \mathfrak{A}_{\mathsf{proto}}$, a *negative test proto-instruction* $-e$;
– for each $l \in \mathbb{N}$, a *direct absolute jump instruction* $\#\#l$.

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

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 a plain basic proto-instruction e is the same as the effect of the plain basic instruction $\theta(e, n)$, where n is the content of the register involved in the instantiation of proto-instructions. The effect of a positive or negative test proto-instruction is similar.

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

We define the meaning of $\mathrm{PGLD}_{\mathrm{dii}}$ programs by means of a function $\mathtt{pglddii2pgld}$ from the set of all $\mathrm{PGLD}_{\mathrm{dii}}$ programs to the set of all PGLD programs. This function is defined by

$$\mathtt{pglddii2pgld}(u_1 \,;\, \ldots \,;\, u_k) = \psi(u_1) \,;\, \ldots \,;\, \psi(u_k) \;,$$

where the auxiliary function ψ from the set of all primitive instructions of $\mathrm{PGLD}_{\mathrm{dii}}$ to the set of all primitive instructions of PGLD is defined as follows:

$$\begin{aligned}
\psi(e) &= \mathsf{rdt}.e && \text{if } e \in \mathfrak{A}_{\mathsf{proto}} \;, \\
\psi(+e) &= +\mathsf{rdt}.e && \text{if } e \in \mathfrak{A}_{\mathsf{proto}} \;, \\
\psi(-e) &= -\mathsf{rdt}.e && \text{if } e \in \mathfrak{A}_{\mathsf{proto}} \;, \\
\psi(u) &= u && \text{if } u \text{ is not a proto-instruction} \;.
\end{aligned}$$

The idea is that each proto-instruction can be replaced by an instruction in which the proto-instruction is taken for the method.

Let P be a $\mathrm{PGLD}_{\mathrm{dii}}$ program. Then $\mathtt{pglddii2pgld}(P)$ represents the meaning of P as a PGLD program. The intended behaviour of P under execution is the behaviour of $\mathtt{pglddii2pgld}(P)$ under execution on interaction with a register-dependent method to action translator when abstracted from tau. That is, the *behaviour* of P under execution, written $|P|_{\mathrm{PGLD}_{\mathrm{dii}}}$, is $\tau_{\mathsf{tau}}(|\mathtt{pglddii2pgld}(P)|_{\mathrm{PGLD}} \,/_{\mathsf{rdt}}\, RDT_{\mathrm{init}})$.

9 Register Services

In this section, we give a state-based description of the very simple family of services that constitute a register that can contain natural numbers up to some bound. This register will be used in Section 10 to describe the behaviour of programs in $\mathrm{PGLD}_{\mathrm{dii}}$.

It is 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 services accept the following methods:

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

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

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

- $\mathsf{set}{:}n$: the content of the register becomes n, the reply is T, and $\mathsf{set}{:}n$ is turned into tau;
- $\mathsf{eq}{:}n$: the content of the register does not change, the reply is T if the content of the register equals n and F otherwise, and $\mathsf{eq}{:}n$ is turned into tau.

Let $s \in [0, N]$. Then we write Reg_s for the service with initial state s described by $S = [-1, N]$ and the functions eff, yld and act defined as follows ($n, k \in [0, N]$):

$$
\begin{aligned}
&\mathit{eff}(\mathsf{set}{:}n, k) = n \ , \\
&\mathit{eff}(\mathsf{eq}{:}n, k) = k \ , \\
&\mathit{eff}(m, k) = -1 \qquad \text{if } m \notin \mathcal{M}_{\mathsf{reg}} \ , \\
&\mathit{eff}(m, -1) = -1 \ ,
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{yld}(\mathsf{set}{:}n, k) = \mathsf{T} \ , \\
&\mathit{yld}(\mathsf{eq}{:}n, k) = \mathsf{T} \qquad \text{if } k = n \ , \\
&\mathit{yld}(\mathsf{eq}{:}n, k) = \mathsf{F} \qquad \text{if } k \neq n \ , \\
&\mathit{yld}(m, k) = \mathsf{B} \qquad \text{if } m \notin \mathcal{M}_{\mathsf{reg}} \ , \\
&\mathit{yld}(m, -1) = \mathsf{B} \ ,
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{act}(m, k) = \mathsf{tau} \ , \\
&\mathit{act}(m, -1) = \mathsf{tau} \ .
\end{aligned}
$$

We write $\mathit{Reg}_{\mathrm{init}}$ for Reg_0.

10 An Alternative Semantics for $\mathrm{PGLD}_{\mathrm{dii}}$

In this section, we give an alternative semantics for $\mathrm{PGLD}_{\mathrm{dii}}$.

We define an alternative meaning of $\mathrm{PGLD}_{\mathrm{dii}}$ programs by means of a function $\mathtt{pglddii2pgld}'$ from the set of all $\mathrm{PGLD}_{\mathrm{dii}}$ programs to the set of all PGLD programs. This function is defined by

$$\mathtt{pglddii2pgld}'(u_1 \,;\, \ldots \,;\, u_k) = \psi'_1(u_1) \,;\, \ldots \,;\, \psi'_k(u_k) \ ,$$

where the auxiliary functions ψ'_j from the set of all primitive instructions of $\mathrm{PGLD}_{\mathrm{dii}}$ to the set of all primitive instructions of PGLD are defined as follows ($1 \leq j \leq k$):

$$
\begin{aligned}
\psi'_j(e) &= -\mathsf{reg.eq}{:}0\ ;\ \#\#l''_{j,0}\ ;\ \theta(e,0)\ ;\ \#\#l'_{j+1}\ ;\ \#\#l'_{j+2};\\
&\quad\ \vdots\\
&\quad -\mathsf{reg.eq}{:}N\ ;\ \#\#l''_{j,N}\ ;\ \theta(e,N)\ ;\ \#\#l'_{j+1}\ ;\ \#\#l'_{j+2}\ ,\\
\psi'_j(+e) &= -\mathsf{reg.eq}{:}0\ ;\ \#\#l''_{j,0}\ ;\ +\theta(e,0)\ ;\ \#\#l'_{j+1}\ ;\ \#\#l'_{j+2};\\
&\quad\ \vdots\\
&\quad -\mathsf{reg.eq}{:}N\ ;\ \#\#l''_{j,N}\ ;\ +\theta(e,N)\ ;\ \#\#l'_{j+1}\ ;\ \#\#l'_{j+2}\ ,\\
\psi'_j(-e) &= -\mathsf{reg.eq}{:}0\ ;\ \#\#l''_{j,0}\ ;\ -\theta(e,0)\ ;\ \#\#l'_{j+1}\ ;\ \#\#l'_{j+2};\\
&\quad\ \vdots\\
&\quad -\mathsf{reg.eq}{:}N\ ;\ \#\#l''_{j,N}\ ;\ -\theta(e,N)\ ;\ \#\#l'_{j+1}\ ;\ \#\#l'_{j+2}\ ,\\
\psi'_j(\#\#l) &= \#\#l'_l\ ,\\
\psi'_j(u) &= u \qquad \text{if } u \text{ is not a proto-instruction or jump instruction}\ ,
\end{aligned}
$$

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

$$
\begin{aligned}
l'_j &= n_j + 1 + 5 \cdot (N+1) \cdot (j - 1 + n_j)\ ,\\
l''_{j,h} &= l'_j + 5 \cdot (h+1)\ ,
\end{aligned}
$$

and n_j is the number of jump instructions preceding position j.

The idea is that each proto-instruction can be replaced by an instruction sequence of which the execution leads to the execution of the intended instruction after the content of the register has been found by a linear search. Because the length of the replacing instruction sequence is greater than 1, the direct absolute jump instructions are adjusted so as to compensate for the introduction of additional instructions. Obviously, the linear search for the content of the register can be replaced by a binary search.

Let P be a $\mathrm{PGLD}_{\mathrm{dii}}$ program. Then $\mathtt{pglddii2pgld}'(P)$ represents an alternative meaning of P as a PGLD program. The alternative behaviour of P under execution is the behaviour of $\mathtt{pglddii2pgld}'(P)$ under execution on interaction with a register when abstracted from $\mathtt{tau}$. That is, the *alternative behaviour* of P under execution, written $|P|'_{\mathrm{PGLD}_{\mathrm{dii}}}$, is $\tau_{\mathsf{tau}}(|\mathtt{pglddii2pgld}'(P)|_{\mathrm{PGLD}} /_{\mathsf{reg}} Reg_{\mathrm{init}})$.

The following theorem states rigorously that the behaviour and alternative behaviour of any $\mathrm{PGLD}_{\mathrm{dii}}$ program under execution coincide.

Theorem 1. *For all* $\mathrm{PGLD}_{\mathrm{dii}}$ *programs* P, $|P|_{\mathrm{PGLD}_{\mathrm{dii}}} = |P|'_{\mathrm{PGLD}_{\mathrm{dii}}}$.

Proof. Strictly speaking, we prove this theorem in the algebraic theory obtained by: (i) combining PGA with BTA+REC+ATTSC+ABSTR, resulting in a theory with three sorts: a sort **P** of programs, a sort **T** of threads, and a sort **S** of services; (ii) extending the result by taking $|_|$ for an additional operator from sort **P** to sort **T** and taking the semantic equations and rule defining thread

extraction for additional axioms. We write $\mathcal{P}$ for the set of all closed terms of sort $\mathbf{P}$ from the language of the resulting theory and $\mathcal{T}$ for the set of all closed terms of sort $\mathbf{T}$ from the language of the resulting theory.

In the proof, we make use of an auxiliary function $|_,_| : \mathbb{N} \times \mathcal{P}_{\mathrm{PGLD}} \to \mathcal{T}$ which gives, for each natural number i and PGLD program $u_1 ; \ldots ; u_k$, a closed term of sort $\mathbf{T}$ that denotes the behaviour of $u_1 ; \ldots ; u_k$ when executed from position i if $1 \leq i \leq k$ and S otherwise. This function is defined as follows:

$$\begin{aligned}
|i, u_1 ; \ldots ; u_k| &= |\phi_i(u_i) ; \ldots ; \phi_k(u_k) ; ! ; ! ; (\phi_1(u_1) ; \ldots ; \phi_k(u_k) ; ! ; !)^\omega| \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{if}\ 1 \leq i \leq k\ , \\
|i, u_1 ; \ldots ; u_k| &= \mathsf{S} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{if}\ \neg\ 1 \leq i \leq k
\end{aligned}$$

(where ϕ_j is as in the definition of `pgld2pga`). It follows easily from the definition of $|_,_|$ and the axioms of PGA that if $1 \leq i \leq k$:

$$\begin{aligned}
|i, u_1 ; \ldots ; u_k| &= a \circ |i+1, u_1 ; \ldots ; u_k| && \mathsf{if}\ u_i = a\ , \\
|i, u_1 ; \ldots ; u_k| &= |i+1, u_1 ; \ldots ; u_k| \trianglelefteq a \trianglerighteq |i+2, u_1 ; \ldots ; u_k| && \mathsf{if}\ u_i = +a\ , \\
|i, u_1 ; \ldots ; u_k| &= |i+2, u_1 ; \ldots ; u_k| \trianglelefteq a \trianglerighteq |i+1, u_1 ; \ldots ; u_k| && \mathsf{if}\ u_i = -a\ , \\
|i, u_1 ; \ldots ; u_k| &= |l, u_1 ; \ldots ; u_k| && \mathsf{if}\ u_i = \#\#l\ .
\end{aligned}$$

Let $v_1, \ldots, v_k$ be primitive instructions of $\mathrm{PGLD}_{\mathrm{dii}}$, let

$$\begin{aligned}
T &= \{\tau_{\mathsf{tau}}(|i, \psi(v_1) ; \ldots ; \psi(v_k)| \,/_{\mathsf{rdt}}\, RDT_s) \mid i \in \mathbb{N} \wedge s \in [0, N]\}\ , \\
T' &= \{\tau_{\mathsf{tau}}(|l'_i, \psi'_1(v_1) ; \ldots ; \psi'_k(v_k)| \,/_{\mathsf{reg}}\, Reg_s) \mid i \in \mathbb{N} \wedge s \in [0, N]\}
\end{aligned}$$

(where ψ, ψ'_j, l'_i are as in the definitions of `pglddii2pgld` and `pglddii2pgld'`), and let $\beta : T \to T'$ be the bijection defined by

$$\begin{aligned}
&\beta(\tau_{\mathsf{tau}}(|i, \psi(v_1) ; \ldots ; \psi(v_k)| \,/_{\mathsf{rdt}}\, RDT_s)) \\
&\quad = \tau_{\mathsf{tau}}(|l'_i, \psi'_1(v_1) ; \ldots ; \psi'_k(v_k)| \,/_{\mathsf{reg}}\, Reg_s)\ .
\end{aligned}$$

For each $p' \in \mathcal{T}$, write $\beta^*(p')$ for p' with, for all $p \in T$, all occurrences of p in p' replaced by $\beta(p)$. Then, using the equations concerning the auxiliary function $|_,_|$ given above, it is straightforward to prove that there exists a function $\gamma : \mathcal{T} \to \mathcal{T}$ such that, for all $p \in T$:

- $p = \gamma(p)$ and $\beta(p) = \beta^*(\gamma(p))$ are derivable;
- there exists an $n \in \mathbb{N}$ such that $\gamma^n(p) \notin T$.[5]

Because $|\psi(v_1) ; \ldots ; \psi(v_k)| = |1, \psi(v_1) ; \ldots ; \psi(v_k)|$ and $|\psi'_1(v_1) ; \ldots ; \psi'_k(v_k)| = |l'_1, \psi'_1(v_1) ; \ldots ; \psi'_k(v_k)|$, this means that $|v_1 ; \ldots ; v_k|_{\mathrm{PGLD}_{\mathrm{dii}}}$ and $|v_1 ; \ldots ; v_k|'_{\mathrm{PGLD}_{\mathrm{dii}}}$ are solutions of the same guarded recursive specification. Because guarded recursive specifications have unique solutions, it follows immediately that $|v_1 ; \ldots ; v_k|_{\mathrm{PGLD}_{\mathrm{dii}}} = |v_1 ; \ldots ; v_k|'_{\mathrm{PGLD}_{\mathrm{dii}}}$. □

[5] The function $\gamma^n : \mathcal{T} \to \mathcal{T}$ is defined by induction on n as usual: $\gamma^0(p) = p$ and $\gamma^{n+1}(p) = \gamma(\gamma^n(p))$.

11 Discussion of Semantic Approaches

In Section 8 and Section 10, the meaning of $\mathrm{PGLD}_{\mathrm{dii}}$ programs is explained by means of different translations into PGLD programs. In both sections, the intended behaviour of a $\mathrm{PGLD}_{\mathrm{dii}}$ program under execution is described as the behaviour of the translated program under execution on interaction with some service. The translation given in Section 8 is extremely simple, but the translation given in Section 10 is fairly complicated. The service used in Section 8 to describe the behaviour of a $\mathrm{PGLD}_{\mathrm{dii}}$ program and the one used in Section 10 are equally simple. However, the former service is far more powerful: it turns a processed method into a basic action if the method corresponds to a basic proto-instruction. By its power, the translation can be kept simple if that service is used. Because of the simpler translation of $\mathrm{PGLD}_{\mathrm{dii}}$ programs into PGLD programs and the equally simple service used, the approach to semantics from Section 8 is preferable in the case of $\mathrm{PGLD}_{\mathrm{dii}}$.

For simplicity, we have considered the case where the content of one register is involved in the instantiation of proto-instructions. In the case where the contents of a number of registers is involved in the instantiation of proto-instructions, the approach to semantics from Section 8 is even more strongly preferable. In this case, the translation of $\mathrm{PGLD}_{\mathrm{dii}}$ programs into PGLD programs remains the same if the approach from Section 8 is followed, whereas the translation becomes far more complicated if the approach from Section 10 is followed.

If a new programming feature is added to a known program notation such as PGLD and the starting-point of the approach to define the meaning of the programs from the extended program notation is translation of those programs into programs from the known program notation, then we can conceive of several approaches:

- give a translation by which each program from the extended program notation is translated into a program from the known program notation that exhibits on execution the same behaviour;
- give a translation by which each program from the extended program notation is translated into a program from the known program notation that exhibits on execution the same behaviour by interaction with a given service that does not turn any processed method into a basic action;
- give a translation by which each program from the extended program notation is translated into a program from the known program notation that exhibits on execution the same behaviour by interaction with a given service that turns certain processed methods into basic actions.

We consider an approach earlier in this list preferable provided that the translation concerned does not become too complicated. In the case where the translation becomes too complicated with all three approaches, it is desirable to look for another starting-point. This may end up in direct thread extraction.

12 Concrete Proto-Instructions

At a fairly concrete level, basic instructions and proto-instructions are strings of characters. In [3], a concrete notation for basic instructions is introduced for the case where each basic instruction consists of a focus and a method. Here, we extend that concrete notation to cover proto-instructions.

First of all, we distinguish neutral strings and active strings. A *neutral string* is an empty string or a string of one or more characters of which the first character is a letter or a colon and each of the remaining characters is a letter, a digit or a colon. An *active string* is a string of two or more characters of which the first character is an asterisk and each of the remaining characters is a digit.

A *concrete proto-instruction* is a string of the form $f'.m'$, where f' and m' are non-empty strings of characters in which neutral strings and active strings alternate starting with a neutral string of which the first character is a letter, in which at least one active string occurs.

A *concrete focus* is a neutral string of which the first character is a letter. A *concrete method* is either a neutral string of which the first character is a letter or a proto-instruction. A *concrete instruction* is a string of the form $f.m$, where f is a concrete focus and m is a concrete method.

The intention is that instantiation of a concrete proto-instruction amounts to simultaneously replacing all active strings occurring in it by neutral strings according to some assignment of neutral strings to active strings. In this way, instantiation turns concrete proto-instructions into concrete instructions.

The concrete notation for basic instructions introduced above covers the case where a number of registers is involved in the instantiation of proto-instructions. It is sufficiently expressive for all applications that we have in mind.

Example 1. Consider a program that on execution at a certain stage receives digit by digit the binary representation of a password and then performs an action to have the password checked by some service. The binary representation of a password is a character sequence of a fixed length, say n, of which all characters are among the digits 0 and 1. Suppose that the service used for checking passwords only accepts methods of the form $\mathsf{chk}{:}pw$, where pw is the binary representation of a password.

In the case where no proto-instructions are available, the program has to distinguish between 2^n cases. In the case where proto-instructions are available, the program has to distinguish between $2 \cdot n$ cases only. In the latter case, using the concrete notation introduced above, the proto-instruction concerned will look like $\mathsf{passw.chk}{:}*\underline{1}\ldots*\underline{n}$, where $\underline{1}, \ldots, \underline{n}$ stand for the decimal representations of the numbers $1, \ldots, n$, respectively.

Take $n = 6$ and suppose that the contents of the registers numbered 1, 2, 3, 4, 5 and 6 are 0, 1, 0, 1, 1 and 0, respectively. Then the proto-instruction $\mathsf{passw.chk}{:}*1\ldots*6$ will be turned into the instruction $\mathsf{passw.chk}{:}010110$.

13 Conclusions

We have studied sequential programs that are instruction sequences with dynamically instantiated instructions. We have defined the meaning of the programs concerned in two different ways which both involve a translation into programs that are instruction sequences without dynamically instantiated instructions. In one of the two ways, the translation is very simple and does not lead to increase in the length of a program or the number of steps needed by a program. That way is considered the preferred one. The preferred way made it necessary for the use mechanism that was introduced in [6] to be generalized. In ongoing work, we find that the generalization is not only useful in the area of semantics.

In [7], we have modelled and analysed micro-architectures with pipelined instruction processing in the setting of program algebra, basic thread algebra, and Maurer computers [10, 11]. In that work, which we consider a preparatory step in the development of a formal approach to design new micro-architectures, dynamically instantiated instructions were not taken into account. Another option for future work is to look at the effect of dynamically instantiated instructions on pipelined instruction processing.

References

1. J. C. M. Baeten and J. A. Bergstra. Global renaming operators in concrete process algebra. *Information and Control*, 78(3):205–245, 1988.
2. J. A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings 30th ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.
3. J. A. Bergstra and 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. Thread algebra with multi-level strategies. *Fundamenta Informaticae*, 71(2/3):153–182, 2006.
5. J. A. Bergstra and C. A. Middelburg. Instruction sequences with indirect jumps. Electronic Report PRG0709, Programming Research Group, University of Amsterdam, November 2007.
6. J. A. Bergstra and C. A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 19(4):445–474, 2007.
7. J. A. Bergstra and C. A. Middelburg. Maurer computers for pipelined instruction processing. *Mathematical Structures in Computer Science*, 2008. In press, doi: 10.1017/S0960129507006548.
8. J. A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
9. W. J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin, 2000.
10. W. D. Maurer. A theory of computer instructions. *Journal of the ACM*, 13(2):226–235, 1966.
11. W. D. Maurer. A theory of computer instructions. *Science of Computer Programming*, 60:244–273, 2006.

12. A. Ponse and M. B. van der Zwaag. An introduction to program and thread algebra. In A. Beckmann et al., editors, *CiE 2006*, volume 3988 of *Lecture Notes in Computer Science*, pages 445–458. Springer-Verlag, 2006.
13. 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.
14. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, Amsterdam, 1990.

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0709] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences with Indirect Jumps,* Programming Research Group - University of Amsterdam, 2007.

[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/