University of Amsterdam
Programming Research Group

Functional Units for Natural Numbers

J.A. Bergstra
C.A. Middelburg

Report PRG0913 November 2009

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

Functional Units for Natural Numbers

J.A. Bergstra and C.A. Middelburg

Informatics Institute, Faculty of Science, University of Amsterdam,
Science Park 107, 1098 XG Amsterdam, the Netherlands
J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl

Abstract. Interaction with services provided by an execution environment forms part of the behaviours exhibited by instruction sequences under execution. Mechanisms related to the kind of interaction in question have been proposed in the setting of thread algebra. Like thread, service is an abstract behavioural concept. The concept of a functional unit is similar to the concept of a service, but more concrete. A state space is inherent in the concept of a functional unit, whereas it is not inherent in the concept of a service. In this paper, we establish the existence of a universal computable functional unit for natural numbers and related results.

Keywords: functional unit, instruction sequence.

1998 ACM Computing Classification: F.1.1, F.4.1.

1 Introduction

We take the view that sequential programs are in essence sequences of instructions, and that interaction with services provided by an execution environment forms part of the behaviours exhibited by instruction sequences under execution (see e.g. [1, 7]). The interaction in question is concerned with the processing of instructions. In earlier work, mechanisms that have a direct bearing on this kind of interaction have been proposed in the setting of basic thread algebra (see e.g. [3, 4]). Both thread and service are abstract behavioural concepts.

We experienced recently limitations of the concept of a service because a state space is not inherent in this concept. This forms the greater part of our motivation for introducing and studying the concept of a functional unit in this paper. This concept is similar to the concept of a service, but it is at a lower level of abstraction. In the concept of a functional unit, a state space is inherent. Rather than first considering functional units in general for an arbitrary state space, we first consider the special case where the state space is the set of natural numbers. This case is arguably the simplest significant case. We establish general results concerning functional units for natural numbers. The main result is the existence of a universal computable functional unit for natural numbers. Results like this one are outside the scope of the concept of a service.

The work presented in this paper belongs to a line of research whose working hypothesis is that instruction sequence is a central notion of computer science.

In this line of research, program algebra [1] is the setting used for investigating issues in which instruction sequences are involved. Instruction sequences are also involved in the issues concerning functional units investigated in this paper. The starting-point of program algebra is the perception of a program as a single-pass instruction sequence, i.e. a finite or infinite sequence of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over. This perception is simple, appealing, and links up with practice. Moreover, basic thread algebra [1] is the setting used for modelling the behaviours exhibited by instruction sequences under execution.[1] In this paper, we use a program notation rooted in program algebra, instead of program algebra itself.

This paper is organized as follows. First, we give a survey of the program notation used in this paper (Section 2) and define its semantics using basic thread algebra (Section 3). Next, we extend basic thread algebra with operators that are related to the processing of instructions by services (Section 4). Then, we introduce the concept of a functional unit and related concepts (Section 5). After that, we investigate functional units for natural numbers (Section 6). We also make some remarks about functional units for finite state spaces (Section 7). Finally, we make some concluding remarks (Section 8).

2 PGLB with Boolean Termination

In this section, we give a survey of the program notation $\mathrm{PGLB}_{\mathrm{bt}}$. This program notation is a variant of the program notation PGLB, which belongs to a hierarchy of program notations rooted in program algebra presented in [1]. $\mathrm{PGLB}_{\mathrm{bt}}$ is PGLB with the Boolean termination instructions $!\mathsf{t}$ and $!\mathsf{f}$ from [3] instead of the termination instruction ! from [1]. PGLB and $\mathrm{PGLB}_{\mathrm{bt}}$ are close to existing assembly languages and have relative jump instructions.

In $\mathrm{PGLB}_{\mathrm{bt}}$, it is assumed that a fixed but arbitrary non-empty finite set $\mathfrak{A}$ of *basic instructions* has been given. The intuition is that the execution of a basic instruction may modify a state and produces t or f at its completion.

$\mathrm{PGLB}_{\mathrm{bt}}$ 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* $\#l$;
- for each $l \in \mathbb{N}$, a *backward jump instruction* $\backslash\#l$;
- a *positive termination instruction* $!\mathsf{t}$;
- a *negative termination instruction* $!\mathsf{f}$.

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

On execution of a $\mathrm{PGLB}_{\mathrm{bt}}$ program, these primitive instructions have the following effects:

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

– the effect of a positive test instruction $+a$ is that basic instruction a is executed and execution proceeds with the next primitive instruction if t is produced and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one – if there is no primitive instruction to proceed with, deadlock occurs;
– the effect of a negative test instruction $-a$ is the same as the effect of $+a$, but with the role of the value produced reversed;
– the effect of a plain basic instruction a is the same as the effect of $+a$, but execution always proceeds as if t is produced;
– the effect of a forward jump instruction $\#l$ is that execution proceeds with the lth next primitive instruction – if l equals 0 or there is no primitive instruction to proceed with, deadlock occurs;
– the effect of a backward jump instruction $\backslash\#l$ is that execution proceeds with the lth previous primitive instruction – if l equals 0 or there is no primitive instruction to proceed with, deadlock occurs;
– the effect of the positive termination instruction $!\mathsf{t}$ is that execution terminates and in doing so delivers the Boolean value t;
– the effect of the negative termination instruction $!\mathsf{t}$ is that execution terminates and in doing so delivers the Boolean value f.

3 Thread Extraction

In this section, we make precise in the setting of $\mathrm{BTA}_{\mathrm{bt}}$ (Basic Thread Algebra with Boolean termination) which behaviours are exhibited on execution by $\mathrm{PGLB}_{\mathrm{bt}}$ programs. We start by reviewing $\mathrm{BTA}_{\mathrm{bt}}$.

In $\mathrm{BTA}_{\mathrm{bt}}$, it is assumed that a fixed but arbitrary non-empty finite set $\mathcal{A}$ of *basic actions*, with $\mathsf{tau} \notin \mathcal{A}$, has been given. We write $\mathcal{A}_{\mathsf{tau}}$ for $\mathcal{A} \cup \{\mathsf{tau}\}$. The members of $\mathcal{A}_{\mathsf{tau}}$ are referred to as *actions*.

A thread is a behaviour which consists of performing actions in a sequential fashion. Upon each basic action performed, a reply from an execution environment determines how it proceeds. The possible replies are the Boolean values t (standing for true) and f (standing for false). Performing the action tau leads always to the reply t.

$\mathrm{BTA}_{\mathrm{bt}}$ has one sort: the sort $\mathbf{T}$ of *threads*. We make this sort explicit because we will extend $\mathrm{BTA}_{\mathrm{bt}}$ with additional sorts in Section 4. To build terms of sort $\mathbf{T}$, $\mathrm{BTA}_{\mathrm{bt}}$ has the following constants and operators:

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

We assume that there is a countably infinite set of variables of sort $\mathbf{T}$ which includes x, y, z. Terms of sort $\mathbf{T}$ are built as usual. We use infix notation for

Table 1. Axiom of $\mathrm{BTA}_{\mathrm{bt}}$

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

Table 2. Approximation induction principle

$\bigwedge_{n \geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$	AIP
$\pi_0(x) = \mathsf{D}$	P0
$\pi_{n+1}(\mathsf{S}+) = \mathsf{S}+$	P1a
$\pi_{n+1}(\mathsf{S}-) = \mathsf{S}-$	P1b
$\pi_{n+1}(\mathsf{D}) = \mathsf{D}$	P2
$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y) = \pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	P3

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

The thread denoted by a closed term of the form $p \trianglelefteq a \trianglerighteq q$ will first perform a, and then proceed as the thread denoted by p if the reply from the execution environment is t and proceed as the thread denoted by q if the reply from the execution environment is f. The threads denoted by D, $\mathsf{S}+$ and $\mathsf{S}-$ will become inactive, terminate with Boolean value t and terminate with Boolean value f, respectively.

$\mathrm{BTA}_{\mathrm{bt}}$ has only one axiom. This axiom is given in Table 1.

Each closed $\mathrm{BTA}_{\mathrm{bt}}$ term of sort $\mathbf{T}$ denotes a thread that will become inactive or terminate after it has performed finitely many actions. Infinite threads can be described by linear recursion. A *linear recursive specification* over $\mathrm{BTA}_{\mathrm{bt}}$ 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 $\mathrm{BTA}_{\mathrm{bt}}$ term of the form D, $\mathsf{S}+$, $\mathsf{S}-$ or $y \trianglelefteq a \trianglerighteq z$ with $y, z \in V$. We are only interested in models of $\mathrm{BTA}_{\mathrm{bt}}$ in which linear recursive specifications have unique solutions. Regular threads, i.e. threads that can only be in a finite number of states, are solutions of finite linear recursive specifications.

To reason about infinite threads, we assume the infinitary conditional equation AIP (Approximation Induction Principle). AIP is based on the view that two threads are identical if their approximations up to any finite depth are identical. The approximation up to depth n of a thread is obtained by cutting it off after it has performed n actions. In AIP, the approximation up to depth n is phrased in terms of the unary *projection* operator $\pi_n : \mathbf{T} \to \mathbf{T}$. AIP and the axioms for the projection operators are given in Table 2. In this table, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$ and n stands for an arbitrary natural number.

The behaviours exhibited on execution by $\mathrm{PGLB}_{\mathrm{bt}}$ programs are considered to be regular threads, with the basic instructions taken for basic actions. The *thread extraction* operation $|_|$ defines, for each $\mathrm{PGLB}_{\mathrm{bt}}$ program, the behaviour exhibited on execution by that $\mathrm{PGLB}_{\mathrm{bt}}$ program. The thread extraction operation is defined by $|u_1 ; \ldots ; u_k| = |1, u_1 ; \ldots ; u_k|$, where the auxiliary operation $|_, _|$ is defined by the equations given in Table 3 (for $a \in \mathfrak{A}$ and $l, i \in \mathbb{N}$) and

Table 3. Defining equations for thread extraction operation

$\lvert i, u_1 ; \ldots ; u_k \rvert = \mathsf{D}$	if not $1 \leq i \leq k$
$\lvert i, u_1 ; \ldots ; u_k \rvert = a \circ \lvert i+1, u_1 ; \ldots ; u_k \rvert$	if $u_i = a$
$\lvert i, u_1 ; \ldots ; u_k \rvert = \lvert i+1, u_1 ; \ldots ; u_k \rvert \trianglelefteq a \trianglerighteq \lvert i+2, u_1 ; \ldots ; u_k \rvert$	if $u_i = +a$
$\lvert i, u_1 ; \ldots ; u_k \rvert = \lvert i+2, u_1 ; \ldots ; u_k \rvert \trianglelefteq a \trianglerighteq \lvert i+1, u_1 ; \ldots ; u_k \rvert$	if $u_i = -a$
$\lvert i, u_1 ; \ldots ; u_k \rvert = \lvert i+l, u_1 ; \ldots ; u_k \rvert$	if $u_i = \#l$
$\lvert i, u_1 ; \ldots ; u_k \rvert = \lvert i \dot{-} l, u_1 ; \ldots ; u_k \rvert$	if $u_i = \backslash\#l$
$\lvert i, u_1 ; \ldots ; u_k \rvert = \mathsf{S}+$	if $u_i = !\mathsf{t}$
$\lvert i, u_1 ; \ldots ; u_k \rvert = \mathsf{S}-$	if $u_i = !\mathsf{f}$

the rule that $\lvert i, u_1 ; \ldots ; u_k \rvert = \mathsf{D}$ if u_i is the beginning of an infinite jump chain.[2]

4 Interaction between Threads and Services

A thread may perform a basic action for the purpose of requesting a named service to process a method and to return a reply value at completion of the processing of the method. In this section, we extend $\mathrm{BTA}_{\mathrm{bt}}$ such that this kind of interaction between threads and services can be dealt with, resulting in $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$. This involves the introduction of service families: collections of named services.

It is assumed that a fixed but arbitrary non-empty finite set $\mathcal{M}$ of *methods* has been given. Methods play the role of commands. A service is able to process certain methods. The processing of a method by a service may involve a change of state of the service and at completion of the processing of the method the service produces a reply value. The set $\mathcal{R}$ of *reply values* is the set $\{\mathsf{t}, \mathsf{f}, \mathsf{d}\}$.

In SF, the algebraic theory of service families introduced below, the following is assumed with respect to services:

- a set $\mathcal{S}$ of services has been given together with:
  - for each $m \in \mathcal{M}$, a total function $\frac{\partial}{\partial m} : \mathcal{S} \to \mathcal{S}$;
  - for each $m \in \mathcal{M}$, a total function $\varrho_m : \mathcal{S} \to \mathcal{R}$;

  satisfying the condition that there exists a unique $S \in \mathcal{S}$ with $\frac{\partial}{\partial m}(S) = S$ and $\varrho_m(S) = \mathsf{d}$ for all $m \in \mathcal{M}$;
- a signature $\Sigma_{\mathcal{S}}$ has been given that includes the following sort:
  - the sort $\mathbf{S}$ of *services*;

  and the following constant and operators:
  - the *empty service* constant $\delta : \mathbf{S}$;
  - for each $m \in \mathcal{M}$, the *derived service* operator $\frac{\partial}{\partial m} : \mathbf{S} \to \mathbf{S}$;
- $\mathcal{S}$ and $\Sigma_{\mathcal{S}}$ are such that:
  - each service in $\mathcal{S}$ can be denoted by a closed term of sort $\mathbf{S}$;
  - the constant δ denotes the unique $S \in \mathcal{S}$ such that $\frac{\partial}{\partial m}(S) = S$ and $\varrho_m(S) = \mathsf{d}$ for all $m \in \mathcal{M}$;

[2] This rule can be formalized, cf. [2].

Table 4. Axioms of SF

$u \oplus \emptyset = u$	SFC1	$\partial_F(\emptyset) = \emptyset$		SFE1
$u \oplus v = v \oplus u$	SFC2	$\partial_F(f.H) = \emptyset$	if $f \in F$	SFE2
$(u \oplus v) \oplus w = u \oplus (v \oplus w)$	SFC3	$\partial_F(f.H) = f.H$	if $f \notin F$	SFE3
$f.H \oplus f.H' = f.\delta$	SFC4	$\partial_F(u \oplus v) = \partial_F(u) \oplus \partial_F(v)$		SFE4

- if closed term t denotes service S, then $\frac{\partial}{\partial m}(t)$ denotes service $\frac{\partial}{\partial m}(S)$.

It is also assumed that a fixed but arbitrary non-empty finite set $\mathcal{F}$ of *foci* has been given. Foci play the role of names of services in the service family offered by an execution environment. A service family is a set of named services where each name occurs only once.

SF has the sorts, constants and operators in $\Sigma_{\mathcal{S}}$ and in addition the following sort:

– the sort **SF** of *service families*;

and the following constant and operators:

– the *empty service family* constant $\emptyset : \mathbf{SF}$;
– for each $f \in \mathcal{F}$, the unary *singleton service family* operator $f._ : \mathbf{S} \to \mathbf{SF}$;
– the binary *service family composition* operator $_ \oplus _ : \mathbf{SF} \times \mathbf{SF} \to \mathbf{SF}$;
– for each $F \subseteq \mathcal{F}$, the unary *encapsulation* operator $\partial_F : \mathbf{SF} \to \mathbf{SF}$.

We assume that there are countably infinite many variables of sort **SF**, including u, v, w. Terms are built as usual in the many-sorted case (see e.g. [8, 10]). We use prefix notation for the singleton service family operators and infix notation for the service family composition operator.

The service family denoted by $\emptyset$ is the empty service family. The service family denoted by a closed term of the form $f.H$ consists of one named service only, the service concerned is H, and the name of this service is f. The service family denoted by a closed term of the form $C \oplus D$ consists of all named services that belong to either the service family denoted by C or the service family denoted by D. In the case where a named service from the service family denoted by C and a named service from the service family denoted by D have the same name, they collapse to an empty service with the name concerned. The service family denoted by a closed term of the form $\partial_F(C)$ consists of all named services with a name not in F that belong to the service family denoted by C.

The axioms of SF are given in Table 4. In this table, f stands for an arbitrary focus from $\mathcal{F}$ and H and H' stand for arbitrary closed terms of sort **S**. The axioms of SF simply formalize the informal explanation given above.

Below we will introduce two operators related to the interaction between threads and services. They are called the apply operator and the reply operator. The apply operator is concerned with the effects of threads on service families and therefore produces service families. The reply operator is concerned with the

Table 5. Axioms for apply operator

$\mathsf{S}+ \bullet u = u$		A1
$\mathsf{S}- \bullet u = u$		A2
$\mathsf{D} \bullet u = \emptyset$		A3
$(\mathsf{tau} \circ x) \bullet u = x \bullet u$		A4
$(x \trianglelefteq f.m \trianglerighteq y) \bullet \partial_{\{f\}}(u) = \emptyset$		A5
$(x \trianglelefteq f.m \trianglerighteq y) \bullet (f.H \oplus \partial_{\{f\}}(u)) = x \bullet (f.\frac{\partial}{\partial m} H \oplus \partial_{\{f\}}(u))$	if $H(m) = \mathsf{t}$	A6
$(x \trianglelefteq f.m \trianglerighteq y) \bullet (f.H \oplus \partial_{\{f\}}(u)) = y \bullet (f.\frac{\partial}{\partial m} H \oplus \partial_{\{f\}}(u))$	if $H(m) = \mathsf{f}$	A7
$(x \trianglelefteq f.m \trianglerighteq y) \bullet (f.H \oplus \partial_{\{f\}}(u)) = \emptyset$	if $H(m) = \mathsf{d}$	A8
$\bigwedge_{n \geq 0} \pi_n(x) \bullet u = \pi_n(y) \bullet v \Rightarrow x \bullet u = y \bullet v$		A9

effects of service families on the Boolean values that threads deliver at their termination. The reply operator does not only produce Boolean values: it produces a special value in cases where no termination takes place.

For the set $\mathcal{A}$ of basic actions, we take the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. Both operators mentioned above relate to the processing of methods by services from a service family in pursuance of basic actions performed by a thread. The service involved in the processing of a method is the service whose name is the focus of the basic action in question.

$\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ has the sorts, constants and operators of both $\mathrm{BTA}_{\mathrm{bt}}$ and SF, and in addition the following sort:

- the sort $\mathbf{R}$ of *replies*;

and the following constants and operators:

- the *reply* constants $\mathsf{t}, \mathsf{f}, \mathsf{d} : \mathbf{R}$;
- the binary *apply* operator $_ \bullet _ : \mathbf{T} \times \mathbf{SF} \to \mathbf{SF}$;
- the binary *reply* operator $_ \,!\, _ : \mathbf{T} \times \mathbf{SF} \to \mathbf{R}$.

We use infix notation for the apply and reply operators.

The service family denoted by a closed term of the form $p \bullet C$ and the reply denoted by a closed term of the form $p \,!\, C$ are the service family and reply, respectively, that result from processing the method of each basic action with a focus of the service family denoted by C that the thread denoted by p performs, where the processing is done by the service in that service family with the focus of the basic action as its name. When the method of a basic action performed by a thread is processed by a service, the service changes in accordance with the method concerned, and affects the thread as follows: the two ways to proceed reduces to one on the basis of the reply value produced by the service. The reply is the Boolean value that the thread denoted by p delivers at termination if it terminates and the value d (standing for divergent) if it does not terminate.

The axioms of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ are the axioms of $\mathrm{BTA}_{\mathrm{bt}}$, the axioms of SF, and the axioms given in Tables 5 and 6. In these tables, f stands for an arbitrary focus

Table 6. Axioms for reply operator

$\mathsf{S}+ \mathbin{!} u = \mathsf{t}$		R1
$\mathsf{S}- \mathbin{!} u = \mathsf{f}$		R2
$\mathsf{D} \mathbin{!} u = \mathsf{d}$		R3
$(\mathsf{tau} \circ x) \mathbin{!} u = x \mathbin{!} u$		R4
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{!} \partial_{\{f\}}(u) = \mathsf{d}$		R5
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{!} (f.H \oplus \partial_{\{f\}}(u)) = x \mathbin{!} (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$	if $H(m) = \mathsf{t}$	R6
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{!} (f.H \oplus \partial_{\{f\}}(u)) = y \mathbin{!} (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$	if $H(m) = \mathsf{f}$	R7
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{!} (f.H \oplus \partial_{\{f\}}(u)) = \mathsf{d}$	if $H(m) = \mathsf{d}$	R8
$\bigwedge_{n \geq 0} \pi_n(x) \mathbin{!} u = \pi_n(y) \mathbin{!} v \Rightarrow x \mathbin{!} u = y \mathbin{!} v$		R9

from $\mathcal{F}$, m stands for an arbitrary method from $\mathcal{M}$, H stands for an arbitrary term of sort $\mathbf{S}$, and n stands for an arbitrary natural number. The axioms simply formalize the informal explanation given above and in addition stipulate what is the result of apply and reply if inappropriate foci or methods are involved. Axioms A9 and R9 allow for reasoning about infinite threads in the contexts of apply and reply, respectively.

5 Functional Units

In this section, we introduce the concept of a functional unit and related concepts such as a functional unit degree.

It is assumed that a non-empty set S of *states* has been given. As before, it is assumed that a non-empty finite set $\mathcal{M}$ of methods has been given. However, in the setting of functional units, methods serve as names of operations on a state space. For that reason, the members of $\mathcal{M}$ will henceforth be called *method names*.

A *method operation* on S is a total function from S to $\mathbb{B} \times S$. A *partial method operation* on S is a partial function from S to $\mathbb{B} \times S$. We write $\mathcal{MO}(S)$ for the set of all method operations on S. We write M^r and M^e, where $M \in \mathcal{MO}(S)$, for the unique functions $R : S \to \mathbb{B}$ and $E : S \to S$, respectively, such that $M(s) = (R(s), E(s))$ for all $s \in S$.

A *functional unit* for S is a finite subset $\mathcal{H}$ of $\mathcal{M} \times \mathcal{MO}(S)$ such that $(m, M) \in \mathcal{H}$ and $(m, M') \in \mathcal{H}$ implies $M = M'$. We write $\mathcal{FU}(S)$ for the set of all functional units for S. We write $\mathcal{I}(\mathcal{H})$, where $\mathcal{H} \in \mathcal{FU}(S)$, for the set $\{m \in \mathcal{M} \mid \exists M \in \mathcal{MO}(S) \bullet (m, M) \in \mathcal{H}\}$. We write $m_{\mathcal{H}}$, where $\mathcal{H} \in \mathcal{FU}(S)$ and $m \in \mathcal{I}(\mathcal{H})$, for the unique $M \in \mathcal{MO}(S)$ such that $(m, M) \in \mathcal{H}$.

The following is a simple illustration of the use of functional units. A variation of a Turing machine tape can be modelled by a functional unit for $\{0,1\}^* \times \{0,1\}^*$ with one method operation for testing the first bit of the second bit sequence, two method operations for overwriting the first bit of the second bit sequence, two method operations for prefixing a bit to the second bit sequence, one method

operation for moving the last bit of the first bit sequence to the second bit sequence, and one method operation for moving the first bit of the second bit sequence to the first bit sequence.

We look upon the set $\mathcal{I}(\mathcal{H})$, where $\mathcal{H} \in \mathcal{FU}(S)$, as the interface of $\mathcal{H}$. It looks to be convenient to have a notation for the restriction of a functional unit to a subset of its interface. We write $(I, \mathcal{H})$, where $\mathcal{H} \in \mathcal{FU}(S)$ and $I \subseteq \mathcal{I}(\mathcal{H})$, for the functional unit $\{(m, M) \in \mathcal{H} \mid m \in I\}$.

According to the definition of a functional unit, $\emptyset \in \mathcal{FU}(S)$. By that we have a unique functional unit with an empty interface, which is not very interesting in itself. However, when considering services that behave according functional units, $\emptyset$ is exactly the functional unit according to which the empty service δ (the service that is not able to process any method) behaves.

The method names attached to method operations in functional units should not be confused with the names used to denote specific method operations in describing functional units. Therefore, we will comply with the convention to use names beginning with a lower-case letter in the former case and names beginning with an upper-case letter in the latter case.

We will use $\mathrm{PGLB}_{\mathrm{bt}}$ programs to derive partial method operations from the method operations of a functional unit. We write $\mathcal{L}(f.I)$, where $I \subseteq \mathcal{M}$, for $\mathrm{PGLB}_{\mathrm{bt}}$ with the set $\{f.m \mid m \in I\}$ taken as the set $\mathfrak{A}$ of basic instructions.

The derivation of partial method operations from the method operations of a functional unit involves services whose processing of methods amounts to replies and state changes according to corresponding method operations of the functional unit concerned. These services can be viewed as the behaviours of a machine, on which the processing in question takes place, in its different states. We take the set $\mathcal{FU}(S) \times S$ as the set $\mathcal{S}$ of services. We write $\mathcal{H}(s)$, where $\mathcal{H} \in \mathcal{FU}(S)$ and $s \in S$, for the service $(\mathcal{H}, s)$. The functions $\frac{\partial}{\partial m}$ and ϱ_m are defined as follows:

$$\frac{\partial}{\partial m}(\mathcal{H}(s)) = \begin{cases} \mathcal{H}(m^e_{\mathcal{H}}(s)) & \text{if } m \in \mathcal{I}(\mathcal{H}) \\ \emptyset(0) & \text{if } m \notin \mathcal{I}(\mathcal{H}) \end{cases},$$

$$\varrho_m(\mathcal{H}(s)) = \begin{cases} m^r_{\mathcal{H}}(s) & \text{if } m \in \mathcal{I}(\mathcal{H}) \\ \mathsf{d} & \text{if } m \notin \mathcal{I}(\mathcal{H}) \end{cases}.$$

We assume that each $\mathcal{H}(s) \in \mathcal{S}$ can be denoted by a closed term of sort $\mathbf{S}$. In this connection, we use the following notational convention: for each $\mathcal{H}(s) \in \mathcal{S}$, $\underline{\mathcal{H}(s)}$ stands for an arbitrary closed term of sort $\mathbf{S}$ that denotes $\mathcal{H}(s)$.

Let $\mathcal{H} \in \mathcal{FU}(S)$, and let $I \subseteq \mathcal{I}(\mathcal{H})$. Then an instruction sequence $x \in \mathcal{L}(f.I)$ produces a partial method operation $|x|_{\mathcal{H}}$ as follows:

$$\begin{aligned} &|x|_{\mathcal{H}}(s) = (|x|^r_{\mathcal{H}}(s), |x|^e_{\mathcal{H}}(s)) && \text{if } |x|^r_{\mathcal{H}}(s) = \mathsf{t} \vee |x|^r_{\mathcal{H}}(s) = \mathsf{f}\ , \\ &|x|_{\mathcal{H}}(s) \text{ is undefined} && \text{if } |x|^r_{\mathcal{H}}(s) = \mathsf{d}\ , \end{aligned}$$

where

$$|x|^r_{\mathcal{H}}(s) = |x| \ ! \ f.\underline{\mathcal{H}(s)}\,,$$
$$|x|^e_{\mathcal{H}}(s) = \text{the unique } s' \in S \text{ such that } |x| \bullet f.\underline{\mathcal{H}(s)} = f.\underline{\mathcal{H}(s')}\,.$$

If $|x|_{\mathcal{H}}$ is total, then it is called a *derived method operation* of $\mathcal{H}$.

The binary relation $\leq$ on $\mathcal{FU}(S)$ is defined by $\mathcal{H} \leq \mathcal{H}'$ iff for all $(m, M) \in \mathcal{H}$, M is a derived method operation of $\mathcal{H}'$. The binary relation $\equiv$ on $\mathcal{FU}(S)$ is defined by $\mathcal{H} \equiv \mathcal{H}'$ iff $\mathcal{H} \leq \mathcal{H}'$ and $\mathcal{H}' \leq \mathcal{H}$.

Theorem 1.

1. *$\leq$ is transitive;*
2. *$\equiv$ is an equivalence relation.*

Proof. Property 1: We have to prove that $\mathcal{H} \leq \mathcal{H}'$ and $\mathcal{H}' \leq \mathcal{H}''$ implies $\mathcal{H} \leq \mathcal{H}''$. It is sufficient to show that we can obtain instruction sequences in $\mathcal{L}(f.\mathcal{I}(\mathcal{H}''))$ that produce the method operations of $\mathcal{H}$ from the instruction sequences in $\mathcal{L}(f.\mathcal{I}(\mathcal{H}'))$ that produce the method operations of $\mathcal{H}$ and the instruction sequences in $\mathcal{L}(f.\mathcal{I}(\mathcal{H}''))$ that produce the method operations of $\mathcal{H}'$. Without loss of generality, we may assume that all instruction sequences are of the form $u_1;\ldots;u_k;!\mathsf{t};!\mathsf{f}$, where, for each $i \in [1, k]$, u_i is a positive test instruction, a forward jump instruction or a backward jump instruction. Let $m \in \mathcal{I}(\mathcal{H})$, let M be such that $(m, M) \in \mathcal{H}$, let $P_m \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}'))$ be such that $M = |P_m|_{\mathcal{H}'}$. Suppose that $\mathcal{I}(\mathcal{H}') = \{m'_1, \ldots, m'_n\}$. For each $i \in [1, n]$, let M'_i be such that $(m'_i, M'_i) \in \mathcal{H}'$, let $P_{m'_i} = u^i_1\,;\ldots;u^i_{k_i}\,;!\mathsf{t}\,;!\mathsf{f} \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}''))$ be such that $M'_i = |P_{m'_i}|_{\mathcal{H}''}$. Consider the $P'_m \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}''))$ obtained from P_m as follows: for each $i \in [1, n]$, (i) first increase each jump over the leftmost occurrence of $+f.m'_i$ in P_m with $k_i + 1$, and next replace this instruction by $u^i_1\,;\ldots;u^i_{k_i}$; (ii) repeat the previous step as long as their are occurrences of $+f.m'_i$. It is easy to see that $M = |P'_m|_{\mathcal{H}''}$.

Property 2: It follows immediately from the definition of $\equiv$ that $\equiv$ is symmetric and from the definition of $\leq$ that $\leq$ is reflexive. From these properties, Property 1 and the definition of $\equiv$, it follows immediately that $\equiv$ is symmetric, reflexive and transitive. □

The members of the quotient set $\mathcal{FU}(S)/\equiv$ are called *functional unit degrees*. Let $\mathcal{H} \in \mathcal{FU}(S)$ and $\mathcal{D} \in \mathcal{FU}(S)/\equiv$. Then $\mathcal{D}$ is a *functional unit degree below* $\mathcal{H}$ if there exists an $\mathcal{H}' \in \mathcal{D}$ such that $\mathcal{H}' \leq \mathcal{H}$.

6 Functional Units for Natural Numbers

In this section, we investigate functional units for natural numbers. The main consequences of considering the special case where the state space is $\mathbb{N}$ are the following: (i) $\mathbb{N}$ is infinite, (ii) there is a notion of computability known which can be used without further preparations.

An example of a functional unit in $\mathcal{FU}(\mathbb{N})$ is an unbounded counter. The method names involved are setzero, succ, pred, and iszero. The method operations

involved are the functions *Setzero*, *Succ*, *Pred*, *Iszero* : $\mathbb{N} \to \mathbb{B} \times \mathbb{N}$ defined as follows:

$$\begin{aligned}
\mathit{Setzero}(x) &= (\mathsf{t}, 0) \ , \\
\mathit{Succ}(x) &= (\mathsf{t}, x+1) \ , \\
\mathit{Pred}(x) &= \begin{cases} (\mathsf{t}, x-1) & \text{if } x > 0 \ , \\ (\mathsf{f}, 0) & \text{if } x = 0 \ , \end{cases} \\
\mathit{Iszero}(x) &= \begin{cases} (\mathsf{t}, x) & \text{if } x = 0 \ , \\ (\mathsf{f}, x) & \text{if } x > 0 \ . \end{cases}
\end{aligned}$$

The functional unit *Counter* is defined as follows:

$$\mathit{Counter} = \{(\mathsf{setzero}, \mathit{Setzero}), (\mathsf{succ}, \mathit{Succ}), (\mathsf{pred}, \mathit{Pred}), (\mathsf{iszero}, \mathit{Iszero})\} \ .$$

Proposition 1. *There are infinitely many functional unit degrees below* $(\{\mathsf{pred}, \mathsf{iszero}\}, \mathit{Counter})$.

Proof. For each $n \in \mathbb{N}$, we define a functional unit $\mathcal{H}_n \in \mathcal{FU}(\mathbb{N})$ such that $\mathcal{H}_n \leq (\{\mathsf{pred}, \mathsf{iszero}\}, \mathit{Counter})$ as follows:

$$\mathcal{H}_n = \{(\mathsf{pred}{:}n, \mathit{Pred}{:}n), (\mathsf{iszero}, \mathit{Iszero})\} \ ,$$

where

$$\mathit{Pred}{:}n = \begin{cases} (\mathsf{t}, x-n) & \text{if } x \geq n \\ (\mathsf{f}, 0) & \text{if } x < n \ . \end{cases}$$

Let $n, m \in \mathbb{N}$ be such that $n < m$. Then $\mathit{Pred}{:}n(m) = (\mathsf{t}, m-n)$. However, there does not exist a $P \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}_m))$ such that $|P|_{\mathcal{H}_m}(m) = (\mathsf{t}, m-n)$ because $\mathit{Pred}{:}m(m) = (\mathsf{t}, 0)$. Hence, $\mathcal{H}_n \not\leq \mathcal{H}_m$ for all $n, m \in \mathbb{N}$ with $n < m$. □

A method operation $M \in \mathcal{MO}(\mathbb{N})$ is *computable* if there exist computable functions $F, G : \mathbb{N} \to \mathbb{N}$ such that $M(n) = (Z(F(n)), G(n))$ for all $n \in \mathbb{N}$, where $Z : \mathbb{N} \to \mathbb{B}$ is inductively defined by $Z(0) = \mathsf{t}$ and $Z(n+1) = \mathsf{f}$. A functional unit $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ is *computable* if, for each $(m, M) \in \mathcal{H}$, M is computable.

Theorem 2. *Let* $\mathcal{H}, \mathcal{H}' \in \mathcal{FU}(\mathbb{N})$ *be such that* $\mathcal{H} \leq \mathcal{H}'$. *Then* $\mathcal{H}$ *is computable if* $\mathcal{H}'$ *is computable.*

Proof. We will show that all derived method operations of $\mathcal{H}'$ are computable.

Take an arbitrary $P \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}'))$ such that $|P|_{\mathcal{H}'}$ is a derived method operations of $\mathcal{H}'$. It follows immediately from the definition of thread extraction that $|P|$ is the solution a finite linear recursive specification over $\mathrm{BTA}_{\mathrm{bt}}$. Let E be a finite linear recursive specification over $\mathrm{BTA}_{\mathrm{bt}}$ of which the solution for x_1 is $|P|$. Because $|P|_{\mathcal{H}'}$ is total, it may be assumed without loss of generality that D does not occur as the right-hand side of an equation in E. Suppose that

$$E = \left\{x_i = x_{l(i)} \trianglelefteq f.m_i \trianglerighteq x_{r(i)} \mid i \in [1, n]\right\} \cup \{x_{n+1} = \mathsf{S}+, x_{n+2} = \mathsf{S}-\} \ .$$

From this set of equations, using the relevant axioms and definitions, we obtain a set of equations of which the solution for F_1 is $|P|^e_{\mathcal{H}'}$:

$$\begin{aligned}&\big\{F_i(s) = F_{l(i)}(m_{i^e_{\mathcal{H}'}}(s)) \cdot \overline{\mathsf{sg}}(\chi_i(s)) + F_{r(i)}(m_{i^e_{\mathcal{H}'}}(s)) \cdot \mathsf{sg}(\chi_i(s)) \mid i \in [1,n]\big\}\\ &\quad \cup \{F_{n+1}(s) = s, F_{n+2}(s) = s\} \ ,\end{aligned}$$

where, for every $i \in [1,n]$, the function $\chi_i : \mathbb{N} \to \mathbb{N}$ is such that for all $s \in \mathbb{N}$:

$$\chi_i(s) = 0 \ \Leftrightarrow \ m_{i^r_{\mathcal{H}'}}(s) = \mathsf{t} \ ,$$

and the functions $\mathsf{sg}, \overline{\mathsf{sg}} : \mathbb{N} \to \mathbb{N}$ are defined as usual:

$$\begin{aligned}\mathsf{sg}(0) &= 0 \ , & \overline{\mathsf{sg}}(0) &= 1 \ ,\\ \mathsf{sg}(n+1) &= 1 \ , & \overline{\mathsf{sg}}(n+1) &= 0 \ .\end{aligned}$$

It follows from the way in which this set of equations is obtained from E, the fact that $m_{i^e_{\mathcal{H}'}}$ and χ_i are computable for each $i \in [1,n]$, and the fact that sg and $\overline{\mathsf{sg}}$ are computable, that this set of equations is equivalent to a set of equations by which $|P|^e_{\mathcal{H}'}$ is defined recursively in the sense of Kleene (see [5]). This means that $|P|^e_{\mathcal{H}'}$ is general recursive, and hence computable.

In a similar way, it is proved that $|P|^r_{\mathcal{H}'}$ is computable. □

A computable $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ is *universal* if for each computable $\mathcal{L} \in \mathcal{FU}(\mathbb{N})$, we have $\mathcal{L} \leq \mathcal{H}$. There exists a universal computable functional unit for natural numbers.

Theorem 3. *There exists a computable $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ that is universal.*

Proof. We will show that there exists a computable $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ with the property that each computable $M \in \mathcal{MO}(\mathbb{N})$ is a derived method operation of $\mathcal{H}$.

As a corollary of Theorem 10.3 from [9],[3] we have that each computable $M \in \mathcal{MO}(\mathbb{N})$ can be computed by means of a register machine with six registers, say $\mathsf{r0}$, $\mathsf{r1}$, $\mathsf{r2}$, $\mathsf{r3}$, $\mathsf{r4}$, and $\mathsf{r5}$. The registers are used as follows: $\mathsf{r0}$ as input register; $\mathsf{r1}$ as output register for the output in $\mathbb{B}$; $\mathsf{r2}$ as output register for the output in $\mathbb{N}$; $\mathsf{r3}$, $\mathsf{r4}$ and $\mathsf{r5}$ as auxiliary registers. The content of $\mathsf{r1}$ represents the Boolean output as follows: 0 represents t and all other natural numbers represent f. For each $i \in [0,5]$, register $\mathsf{r}i$ can be incremented by one, decremented by one, and tested for zero by means of instructions $\mathsf{r}i.\mathsf{succ}$, $\mathsf{r}i.\mathsf{pred}$ and $\mathsf{r}i.\mathsf{iszero}$, respectively. We write $\mathcal{L}(\mathcal{RM}_6)$ for $\mathrm{PGLB}_{\mathrm{bt}}$ with the set $\{\mathsf{r}i.\mathsf{succ}, \mathsf{r}i.\mathsf{pred}, \mathsf{r}i.\mathsf{iszero} \mid i \in [0,5]\}$ taken as the set of basic instructions. Clearly, $\mathcal{L}(\mathcal{RM}_6)$ is adequate to represent all register machine programs using six registers.

We define a computable functional unit $\mathcal{U} \in \mathcal{FU}(\mathbb{N})$ whose method operations can simulate the effects of the register machine instructions by encoding the register machine states by natural numbers such that the contents of the registers can reconstructed by prime factorization. This functional unit is defined as follows:

[3] That theorem can be looked upon as a corollary of Theorem Ia from [6].

$$\mathcal{U} = \{(\mathsf{exp2}, \mathit{Exp2}), (\mathsf{fact5}, \mathit{Fact5})\}$$
$$\cup \{(\mathsf{r}i{:}\mathsf{succ}, \mathit{Ri{:}succ}), (\mathsf{r}i{:}\mathsf{pred}, \mathit{Ri{:}pred}), (\mathsf{r}i{:}\mathsf{iszero}, \mathit{Ri{:}iszero}) \mid i \in [0,5]\} \ ,$$

where the method operations are defined as follows:

$$\mathit{Exp2}(x) = (\mathsf{t}, 2^x) \ ,$$
$$\mathit{Fact5}(x) = (\mathsf{t}, \max\{y \mid \exists z \bullet x = 5^y \cdot z\})$$

and, for each $i \in [0,5]$:[4]

$$\mathit{Ri{:}succ}(x) = (\mathsf{t}, p_i \cdot x) \ ,$$
$$\mathit{Ri{:}pred}(x) = \begin{cases} (\mathsf{t}, x/p_i) & \text{if } p_i \mid x \\ (\mathsf{f}, x) & \text{if } \neg(p_i \mid x) \ , \end{cases}$$
$$\mathit{Ri{:}iszero}(x) = \begin{cases} (\mathsf{t}, x) & \text{if } \neg(p_i \mid x) \\ (\mathsf{f}, x) & \text{if } p_i \mid x \ , \end{cases}$$

where p_i is the $(i{+}1)$th prime number, i.e. $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, $\ldots$.

We define a function `rml2ful` from $\mathcal{L}(\mathcal{RM}_6)$ to $\mathcal{L}(f.\mathcal{I}(\mathcal{U}))$, which gives, for each instruction sequence P from $\mathcal{L}(\mathcal{RM}_6)$, the instruction sequence from $\mathcal{L}(f.\mathcal{I}(\mathcal{U}))$ by which the effect produced by P on a register machine with six registers can be simulated on $\mathcal{U}$. This function is defined as follows:

$$\texttt{rml2ful}(u_1 \,;\, \ldots \,;\, u_k)$$
$$= f.\mathsf{exp2} \,;\, \phi(u_1) \,;\, \ldots \,;\, \phi(u_k) \,;\, -f.\mathsf{r1{:}iszero} \,;\, \#3 \,;\, f.\mathsf{fact5} \,;\, !\mathsf{t} \,;\, f.\mathsf{fact5} \,;\, !\mathsf{f} \ ,$$

where

$$\phi(a) = \psi(a) \ ,$$
$$\phi(+a) = +\psi(a) \ ,$$
$$\phi(-a) = -\psi(a) \ ,$$
$$\phi(u) = u \qquad \text{if } u \text{ is a jump or termination instruction} \ ,$$

where, for each $i \in [0,5]$:

$$\psi(\mathsf{r}i.\mathsf{succ}) = f.\mathsf{r}i{:}\mathsf{succ} \ ,$$
$$\psi(\mathsf{r}i.\mathsf{pred}) = f.\mathsf{r}i{:}\mathsf{pred} \ ,$$
$$\psi(\mathsf{r}i.\mathsf{iszero}) = f.\mathsf{r}i{:}\mathsf{iszero} \ .$$

Take an arbitrary computable $M \in \mathcal{MO}(\mathbb{N})$. Then there exist an instruction sequence in $\mathcal{L}(\mathcal{RM}_6)$ that computes M. Let P be such an instruction sequence. We have that $|\texttt{rml2ful}(P)|_{\mathcal{U}} = M$. Hence, M is a derived method operation of $\mathcal{U}$. $\square$

[4] As usual, we write $x \mid y$ for y is divisible by x.

The universal computable functional unit $\mathcal{U}$ defined in the proof of Theorem 3 has 20 method operations. However, three method operations suffice.

Theorem 4. *There exists a computable $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ with only three method operations that is universal.*

Proof. We know from the proof of Theorem 3 that there exists a computable $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ with 20 method operations, say $M_0, \ldots, M_{19}$. We will show that there exists a computable $\mathcal{H}' \in \mathcal{FU}(\mathbb{N})$ with only three method operations such that $\mathcal{H} \leq \mathcal{H}'$.

We define a computable functional unit $\mathcal{U}' \in \mathcal{FU}(\mathbb{N})$ with only three method operations such that $\mathcal{U} \leq \mathcal{U}'$ as follows:

$$\mathcal{U}' = \{(\mathsf{g1}, G1), (\mathsf{g2}, G2), (\mathsf{g3}, G3)\} \ ,$$

where the method operations are defined as follows:

$$\begin{aligned}
G1(x) &= (\mathsf{t}, 2^x) \ , \\
G2(x) &= \begin{cases} (\mathsf{t}, 3 \cdot x) & \text{if } \neg(3^{19} \mid x) \wedge \forall y \bullet (y \mid x \Rightarrow (y = 2 \vee y = 3)) \\ (\mathsf{t}, x/3^{19}) & \text{if } 3^{19} \mid x \wedge \neg(3^{20} \mid x) \wedge \forall y \bullet (y \mid x \Rightarrow (y = 2 \vee y = 3)) \\ (\mathsf{f}, 0) & \text{if } 3^{20} \mid x \vee \neg\forall y \bullet (y \mid x \Rightarrow (y = 2 \vee y = 3)) \ , \end{cases} \\
G3(x) &= M_{fact3(x)}(fact2(x)) \ ,
\end{aligned}$$

where

$$\begin{aligned}
fact2(x) &= \max\{y \mid \exists z \bullet x = 2^y \cdot z\} \ , \\
fact3(x) &= \max\{y \mid \exists z \bullet x = 3^y \cdot z\} \ .
\end{aligned}$$

We have that, for each $i \in [0, 19]$, $|f.\mathsf{g1}\,;f.\mathsf{g2}^{\,i}\,;+f.\mathsf{g3}\,;!\mathsf{t}\,;!\mathsf{f}|_{\mathcal{U}'} = M_i$.[5] Hence, $M_0, \ldots, M_{19}$ are derived method operations of $\mathcal{U}'$. □

The universal computable functional unit $\mathcal{U}'$ defined in the proof of Theorem 4 has three method operations. We can show that one method operation does not suffice.

Theorem 5. *There does not exist a computable $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ with only one method operation that is universal.*

Proof. We will show that there does not exist a computable $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ with one method operation such that $Counter \leq \mathcal{H}$. Here, *Counter* is the functional unit introduced at the beginning of this section.

Assume that there exists a computable $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ with one method operation such that $Counter \leq \mathcal{H}$. Let $\mathcal{H}' \in \mathcal{FU}(\mathbb{N})$ be such that $\mathcal{H}'$ has one method operation and $Counter \leq \mathcal{H}'$, and let m be the unique method name such that $\mathcal{I}(\mathcal{H}') = \{m\}$. Let $P_1, P_2, P_3, P_4 \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}'))$ be such that $|P_1|_{\mathcal{H}'} = Setzero$,

[5] For each primitive instruction u, the instruction sequence u^n is defined by induction on n as follows: $u^0 = \#1$, $u^1 = u$ and $u^{n+2} = u\,;u^{n+1}$.

$|P_2|_{\mathcal{H}'} = \mathit{Succ}$, $|P_3|_{\mathcal{H}'} = \mathit{Pred}$, and $|P_4|_{\mathcal{H}'} = \mathit{Iszero}$. Then $|P_2|_{\mathcal{H}'}(0) = (\mathrm{t}, 1)$ and $|P_3|_{\mathcal{H}'}(1) = (\mathrm{t}, 0)$. Instruction $f.m$ is processed at least once if P_2 is applied to $\mathcal{H}'(0)$ or P_3 is applied to $\mathcal{H}'(1)$. Let k_0 be the number of times that instruction $f.m$ is processed on application of P_2 to $\mathcal{H}'(0)$ and let k_1 be the number of times that instruction $f.m$ is processed on application of P_3 to $\mathcal{H}'(1)$ (irrespective of replies). Then, from state 0, state 0 is reached again after $f.m$ is processed $k_0 + k_1$ times. Thus, by repeated application of P_2 to $\mathcal{H}'(0)$ at most $k_0 + k_1$ different states can be reached. This contradicts with $|P_2|_{\mathcal{H}'} = \mathit{Succ}$. Hence, there does not exist a computable $\mathcal{H} \in \mathcal{FU}(\mathbb{N})$ with one method operation such that $\mathit{Counter} \leq \mathcal{H}$. □

It is an open problem whether two method operations suffice.

7 Functional Units for Finite State Spaces

In this short section, we make some remarks about functional units for finite state spaces.

In the special case where the state space is $\mathbb{B}$, the state space consists of only two states. Because there are four possible unary functions on $\mathbb{B}$, there are precisely 16 method operations in $\mathcal{MO}(\mathbb{B})$. There are in principle 2^{16} different functional units in $\mathcal{FU}(\mathbb{B})$, for it is useless to include the same method operation more than once under different names in a functional unit. This means that 2^{16} is an upper bound of the number of functional unit degrees in $\mathcal{FU}(\mathbb{B})/{\equiv}$. However, it is straightforward to show that $\mathcal{FU}(\mathbb{B})/{\equiv}$ has only 12 different functional unit degrees.

In the more general case of a finite state space consisting of k states, say S_k, there are in principle $2^{2^k \cdot k^k}$ different functional units in $\mathcal{FU}(S_k)$. Already with $k = 3$, it becomes unclear whether the number of functional unit degrees in $\mathcal{FU}(S_k)$ can be determined manually. Actually, we do not know at the moment whether it can be determined with computer support either.

8 Concluding Remarks

We have defined the concept of a functional unit for a state space and have established general results concerning functional units for natural numbers. The main result is the existence of a universal computable functional unit for natural numbers. The case where the state space is the set of natural numbers is arguably the simplest significant case. We have not yet investigated other significant cases.

An interesting case is the one where the state space is the set of all pairs of sequences over some alphabet: the tape of a Turing machine can be modelled by a functional unit for this state space. Each Turing machine can be simulated by means of a functional unit that corresponds to the tape of the Turing machine and a $\mathrm{PGLB}_{\mathrm{bt}}$ program that corresponds to the finite control of the Turing machine. Variations of the Turing machine theme can be dealt with in this way

as well. Thus, functional units allows for many computability issues to be viewed as issues about programs rather than machines.

In [3], we introduce an extension of program algebra with Boolean termination instructions, called $\mathrm{PGA}_{\mathrm{bt}}$, and define a thread extraction operation for it. $\mathrm{PGLB}_{\mathrm{bt}}$ programs can be translated into closed $\mathrm{PGA}_{\mathrm{bt}}$ terms such that thread extraction for $\mathrm{PGLB}_{\mathrm{bt}}$ yields the same behaviours as translation followed by thread extraction for $\mathrm{PGA}_{\mathrm{bt}}$. In [3], we also introduce an extension of basic thread algebra similar to $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$. In addition to the constants and operators of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$, that extension has a constant (S) for termination without delivery of a Boolean value and an operator ($/$) which is concerned with the effects of service families on threads and therefore produces threads.

References

1. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming **51**(2), 125–156 (2002)
2. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. Journal of Applied Logic **6**(4), 553–563 (2008)
3. Bergstra, J.A., Middelburg, C.A.: Instruction sequence processing operators. Electronic Report PRG0912, Programming Research Group, University of Amsterdam (2009). Available from `http://www.science.uva.nl/research/prog/publications.html`. Also available from `http://arxiv.org/`: `arXiv:0910.5564v1 [cs.PL]`
4. Bergstra, J.A., Ponse, A.: Combining programs and state machines. Journal of Logic and Algebraic Programming **51**(2), 175–192 (2002)
5. Kleene, S.C.: General recursive functions of natural numbers. Mathematische Annalen **112**, 727–742 (1936)
6. Minsky, M.L.: Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines. Annals of Mathematics **74**(3), 437–455 (1961)
7. Ponse, A., van der Zwaag, M.B.: An introduction to program and thread algebra. In: A. Beckmann, et al. (eds.) CiE 2006, *Lecture Notes in Computer Science*, vol. 3988, pp. 445–458. Springer-Verlag (2006)
8. Sannella, D., Tarlecki, A.: Algebraic preliminaries. In: E. Astesiano, H.J. Kreowski, B. Krieg-Brückner (eds.) Algebraic Foundations of Systems Specification, pp. 13–30. Springer-Verlag, Berlin (1999)
9. Shepherdson, J.C., Sturgis, H.E.: Computability of recursive functions. Journal of the ACM **10**(2), 217–255 (1963)
10. Wirsing, M.: Algebraic specification. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 675–788. Elsevier, Amsterdam (1990)

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0912] J.A. Bergstra and C.A. Middelburg, *Instruction Sequence Processing Operators,* Programming Research Group - University of Amsterdam, 2009.

[PRG0911] J.A. Bergstra and C.A. Middelburg, *Partial Komori Fields and Imperative Komori Fields,* Programming Research Group - University of Amsterdam, 2009.

[PRG0910] J.A. Bergstra and C.A. Middelburg, *Indirect Jumps Improve Instruction Sequence Performance,* Programming Research Group - University of Amsterdam, 2009.

[PRG0909] J.A. Bergstra and C.A. Middelburg, *Arithmetical Meadows,* Programming Research Group - University of Amsterdam, 2009.

[PRG0908] B. Diertens, *Software Engineering with Process Algebra: Modelling Client / Server Architecures,* Programming Research Group - University of Amsterdam, 2009.

[PRG0907] J.A. Bergstra and C.A. Middelburg, *Inversive Meadows and Divisive Meadows,* Programming Research Group - University of Amsterdam, 2009.

[PRG0906] J.A. Bergstra and C.A. Middelburg, *Instruction Sequence Notations with Probabilistic Instructions,* Programming Research Group - University of Amsterdam, 2009.

[PRG0905] J.A. Bergstra and C.A. Middelburg, *A Protocol for Instruction Stream Processing,* Programming Research Group - University of Amsterdam, 2009.

[PRG0904] J.A. Bergstra and C.A. Middelburg, *A Process Calculus with Finitary Comprehended Terms,* Programming Research Group - University of Amsterdam, 2009.

[PRG0903] J.A. Bergstra and C.A. Middelburg, *Transmission Protocols for Instruction Streams,* Programming Research Group - University of Amsterdam, 2009.

[PRG0902] J.A. Bergstra and C.A. Middelburg, *Meadow Enriched ACP Process Algebras,* Programming Research Group - University of Amsterdam, 2009.

[PRG0901] J.A. Bergstra and C.A. Middelburg, *Timed Tuplix Calculus and the Wesseling and van den Berg Equation,* Programming Research Group - University of Amsterdam, 2009.

[PRG0814] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences for the Production of Processes,* Programming Research Group - University of Amsterdam, 2008.

[PRG0813] J.A. Bergstra and C.A. Middelburg, *On the Expressiveness of Single-Pass Instruction Sequences,* Programming Research Group - University of Amsterdam, 2008.

[PRG0812] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences and Non-uniform Complexity Theory,* Programming Research Group - University of Amsterdam, 2008.

[PRG0811] D. Staudt, *A Case Study in Software Engineering with PSF: A Domotics Application,* Programming Research Group - University of Amsterdam, 2008.

[PRG0810] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Poly-Threading,* Programming Research Group - University of Amsterdam, 2008.

[PRG0809] J.A. Bergstra and C.A. Middelburg, *Data Linkage Dynamics with Shedding,* Programming Research Group - University of Amsterdam, 2008.

[PRG0808] B. Diertens, *A Process Algebra Software Engineering Environment,* Programming Research Group - University of Amsterdam, 2008.

[PRG0807] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *Tuplix Calculus Specifications of Financial Transfer Networks,* Programming Research Group - University of Amsterdam, 2008.

[PRG0806] J.A. Bergstra and C.A. Middelburg, *Data Linkage Algebra, Data Linkage Dynamics, and Priority Rewriting,* Programming Research Group - University of Amsterdam, 2008.

[PRG0805] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *UvA Budget Allocatie Model,* Programming Research Group - University of Amsterdam, 2008.

[PRG0804] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Sequential Poly-Threading,* Programming Research Group - University of Amsterdam, 2008.

[PRG0803] J.A. Bergstra and C.A. Middelburg, *Thread Extraction for Polyadic Instruction Sequences,* Programming Research Group - University of Amsterdam, 2008.

[PRG0802] A. Barros and T. Hou, *A Constructive Version of AIP Revisited,* Programming Research Group - University of Amsterdam, 2008.

[PRG0801] J.A. Bergstra and C.A. Middelburg, *Programming an Interpreter Using Molecular Dynamics,* Programming Research Group - University of Amsterdam, 2008.

[PRG0713] J.A. Bergstra, A. Ponse, and M.B. van der Zwaag, *Tuplix Calculus,* Programming Research Group - University of Amsterdam, 2007.

[PRG0712] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *Towards a Formalization of Budgets,* Programming Research Group - University of Amsterdam, 2007.

[PRG0711] J.A. Bergstra and C.A. Middelburg, *Program Algebra with a Jump-Shift Instruction,* Programming Research Group - University of Amsterdam, 2007.

[PRG0710] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences with Dynamically Instantiated Instructions,* Programming Research Group - University of Amsterdam, 2007.

[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/