Acta Informatica (2012) 49:139–172
DOI 10.1007/s00236-012-0154-2

ORIGINAL ARTICLE

Instruction sequence processing operators

J. A. Bergstra · C. A. Middelburg

Received: 22 March 2011 / Accepted: 5 March 2012 / Published online: 23 March 2012

Abstract Instruction sequence is a key concept in practice, but it has as yet not come prominently into the picture in theoretical circles. This paper concerns instruction sequences, the behaviours produced by them under execution, the interaction between these behaviours and components of the execution environment, and two issues relating to computability theory. Positioning Turing's result regarding the undecidability of the halting problem as a result about programs rather than machines, and taking instruction sequences as programs, we analyse the autosolvability requirement that a program of a certain kind must solve the halting problem for all programs of that kind. We present novel results concerning this autosolvability requirement. The analysis is streamlined by using the notion of a functional unit, which is an abstract state-based model of a machine. In the case where the behaviours exhibited by a component of an execution environment can be viewed as the behaviours of a machine in its different states, the behaviours concerned are completely determined by a functional unit. The above-mentioned analysis involves functional units whose possible states represent the possible contents of the tapes of Turing machines with a particular tape alphabet. We also investigate functional units whose possible states are the natural numbers. This investigation yields a novel computability result, viz. the existence of a universal computable functional unit for natural numbers.

1 Introduction

The concept of an instruction sequence is a very primitive concept in computing. Instruction sequence execution has always been part of computing because of the fact that it underlies

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

J. A. Bergstra
e-mail: J.A.Bergstra@uva.nl

virtually all past and current generations of computers. It happens that, given a precise definition of an appropriate notion of an instruction sequence, many issues in computer science can be clearly explained in terms of instruction sequences, from issues of a computer-architectural kind to issues of a computation-theoretic kind. A simple yet interesting example is that a program can simply be defined as a text that denotes an instruction sequence. Such a definition corresponds to an appealing empirical perspective found among practitioners.

In theoretical computer science, the meaning of programs usually plays a prominent part in the explanation of many issues concerning programs. Moreover, what is taken for the meaning of programs is mathematical by nature. On the other hand, it is customary that practitioners do not fall back on the mathematical meaning of programs in case explanation of issues concerning programs is needed. They phrase their explanations from an empirical perspective. An appealing empirical perspective is the one that a program is in essence an instruction sequence and an instruction sequence under execution produces a behaviour that is controlled by its execution environment: each step performed actuates the processing of an instruction by the execution environment and a reply returned at completion of the processing determines how the behaviour proceeds.

The work presented in this paper belongs to a line of research which started with an attempt to approach the semantics of programming languages from the perspective mentioned above. The first published paper on this approach is [4]. That paper is superseded by [5] with regard to the groundwork for the approach: program algebra, an algebraic theory of single-pass instruction sequences, and basic thread algebra, an algebraic theory of mathematical objects that represent in a direct way the behaviours produced by instruction sequences under execution.[1] The main advantages of the approach are that it does not require a lot of mathematical background and that it is more appealing to practitioners than the main approaches to programming language semantics: the operational approach, the denotational approach and the axiomatic approach. For an overview of these approaches, see e.g. [32].

As a continuation of the work on a new approach to programming language semantics, the notion of an instruction sequence was subjected to systematic and precise analysis using the groundwork laid earlier. This led among other things to expressiveness results about the instruction sequences considered and variations of the instruction sequences considered (see e.g. [8,17,18,21,37]). Instruction sequences are under discussion for many years in diverse work on computer architecture, as witnessed by e.g. [2,23,24,27,33–35,42,46], but the notion of an instruction sequence has never been subjected to any precise analysis.

As another continuation of the work on a new approach to programming language semantics, selected issues relating to well-known subjects from the theory of computation and the area of computer architecture were rigorously investigated thinking in terms of instruction sequences (see e.g. [7,11,13,16]). The subjects from the theory of computation, namely the halting problem and non-uniform computational complexity, are usually investigated thinking in terms of a common model of computation such as Turing machines and Boolean circuits (see e.g. [1,25,40]). The subjects from the area of computer architecture, namely instruction sequence performance, instruction set architectures and remote instruction processing, are usually not investigated in a rigorous way at all.

This paper concerns among other things an investigation of issues relating to the halting problem thinking in terms of instruction sequences. Positioning Turing's result regarding the undecidability of the halting problem (see e.g. [44]) as a result about programs rather than machines, and taking instruction sequences as programs, we analyse the autosolvability requirement that a program of a certain kind must solve the halting problem for all programs

[1] In [5], basic thread algebra is introduced under the name basic polarized process algebra.

of that kind. We present positive and negative results concerning the autosolvability of the halting problem for programs. To our knowledge, these results are new and unusual.

Most work done in the line of research sketched above requires that basic thread algebra, i.e. the algebraic theory of mathematical objects that represent in a direct way the behaviours produced by instruction sequences under execution, is extended to deal with the interaction between instruction sequences under execution and components of their execution environment concerning the processing of instructions. The first published paper on such an extended theory is [19]. A substantial re-design of the extended theory presented in that paper is presented in the current paper. The changes introduced allow for the material from quite a part of the work done in the line of research sketched above to be streamlined.

Further streamlining is achieved in this paper by introducing and using the notion of a functional unit. In the extended theory, a rather abstract view of the behaviours exhibited by components of execution environments is taken. The view is just sufficiently concrete for the purpose of the theory. A functional unit is an abstract model of a machine. Under the abstract view of the behaviours exhibited by a component of an execution environment, the behaviours concerned are completely determined by a functional unit in the frequently occurring case that they can be viewed as the behaviours of a machine in its different states. The current paper also concerns an investigation of functional units whose possible states are the natural numbers. This investigation yields a computability result that is new and unusual as far as we know, namely the existence of a universal computable functional unit for natural numbers.

The investigations carried out in the line of research sketched above demonstrate that the concept of an instruction sequence offers a novel and useful viewpoint on issues relating to diverse subjects. In view of the very primitive nature of this concept, it is in fact rather surprising that instruction sequences have never been a theme in computer science. A theoretical understanding of issues in terms of instruction sequences will probably become increasingly more important to a growing number of developments in computer science. Among them are for instance the developments with respect to techniques for high-performance program execution on classical or non-classical computers and techniques for estimating execution times of hard real-time systems. For these and other such developments, the abstractions usually made do not allow for all relevant details to be considered.

Some marginal notes are in order. In this paper, we use an extension of a program notation rooted in program algebra instead of an extension of program algebra itself. The program notation in question has been chosen because it turned out to be appropriate. However, in principle any program notation that is as expressive as the closed terms of program algebra would do. The above-mentioned analysis of the autosolvability requirement inherent in Turing's result regarding the undecidability of the halting problem involves functional units whose possible states are objects that represent the possible contents of the tapes of Turing machines with a particular tape alphabet.

Henceforth, objects that represent in a direct way the behaviours produced by instruction sequences under execution are called threads, objects that represent the behaviours exhibited by components of execution environments are called services, and collections of named services are called service families. In order to deal with the different aspects of the interaction between instruction sequences under execution and components of their execution environment concerning the processing of instructions, three operators are added to basic thread algebra. Because these operators are primarily intended to be used to describe and analyse instruction sequence processing, they are loosely referred to by the term instruction sequence processing operators.

This paper is organized as follows. First, we give a survey of the program notation used in this paper (Sect. 2) and define its semantics using basic thread algebra (Sect. 3). Next, we introduce services and a composition operator for services families (Sect. 4), and the three operators that are related to the processing of instructions by a service family (Sect. 5). Then, we propose to comply with conventions that exclude the use of terms that are not really intended to denote anything (Sect. 6). After that, we give an example related to the processing of instructions by a service family (Sect. 7). Further, we present an interesting variant of one of the above-mentioned operators related to the processing of instructions (Sect. 8). Thereafter, we introduce the concept of a functional unit and related concepts (Sect. 9). Subsequently, we investigate functional units for natural numbers (Sect. 10). Then, we define autosolvability and related notions in terms of functional units related to Turing machine tapes (Sect. 11). After that, we discuss the weakness of interpreters when it comes to solving the halting problem (Sect. 12) and give positive and negative results concerning the autosolvability of the halting problem (Sect. 13). Finally, we make some concluding remarks (Sect. 14).

This paper consolidates material from the reports [9,10,12].

2 PGLB with Boolean termination

In this section, we introduce the program notation $\mathrm{PGLB_{bt}}$ (PGLB with Boolean termination). In [5], a hierarchy of program notations rooted in program algebra is presented. One of the program notations that belong to this hierarchy is PGLB (ProGramming Language B). This program notation is close to existing assembly languages and has relative jump instructions. $\mathrm{PGLB_{bt}}$ is PGLB extended with two termination instructions that allow for the execution of instruction sequences to yield a Boolean value at termination. The extension makes it possible to deal naturally with instruction sequences that implement some test, which is relevant throughout the paper.

In $\mathrm{PGLB_{bt}}$, 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 in most instances modifies a state and in all instances produces a reply at its completion. The possible replies are t (standing for true) and f (standing for false), and the actual reply is in most instances state-dependent. Therefore, successive executions of the same basic instruction may produce different replies. The set $\mathfrak{A}$ is the basis for the set of all instructions that may appear in the instruction sequences considered in $\mathrm{PGLB_{bt}}$. These instructions are called primitive instructions.

$\mathrm{PGLB_{bt}}$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 *plain termination instruction* $!$;
- a *positive termination instruction* $!\mathsf{t}$;
- a *negative termination instruction* $!\mathsf{f}$.

$\mathrm{PGLB_{bt}}$ instruction sequences have the form $u_1 \,;\, \ldots \,;\, u_k$, where $u_1, \ldots, u_k$ are primitive instructions of $\mathrm{PGLB_{bt}}$.

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

- 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 instructions 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 instructions 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 instructions to proceed with, deadlock occurs;
- the effect of the plain termination instruction ! is that execution terminates and in doing so does not deliver a value;
- 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 $!\mathsf{f}$ is that execution terminates and in doing so delivers the Boolean value f.

A simple example of a PGLB$_{\mathrm{bt}}$ instruction sequence is

$$+a \; ; \#2 \; ; \backslash\#2 \; ; b \; ; !\mathsf{t}.$$

On execution of this instruction sequence, first the basic instruction a is executed repeatedly until its execution produces the reply t, next the basic instruction b is executed, and after that execution terminates with delivery of the value t.

From Sect. 9, we will use a restricted version of PGLB$_{\mathrm{bt}}$ called PGLB$_{\mathrm{sbt}}$ (PGLB with strict Boolean termination). The primitive instructions of PGLB$_{\mathrm{sbt}}$ are the primitive instructions of PGLB$_{\mathrm{bt}}$ with the exception of the plain termination instruction. Thus, PGLB$_{\mathrm{sbt}}$ instruction sequences are PGLB$_{\mathrm{bt}}$ instruction sequences in which the plain termination instruction does not occur.

In Sect. 7, we will give examples of instruction sequences for which the delivery of a Boolean value at termination of their execution is natural. There, we will write $;_{i=1}^{n} P_i$, where $P_1, \ldots, P_n$ are PGLB$_{\mathrm{bt}}$ instruction sequences, for the PGLB$_{\mathrm{bt}}$ instruction sequence $P_1 \; ; \ldots ; P_n$.

3 Thread extraction

In this section, we make precise in the setting of BTA$_{\mathrm{bt}}$ (Basic Thread Algebra with Boolean termination) which behaviours are exhibited on execution by PGLB$_{\mathrm{bt}}$ instruction sequences. We start by introducing BTA$_{\mathrm{bt}}$. In [5], BPPA (Basic Polarized Process Algebra) is introduced as a setting for modelling the behaviours exhibited by instruction sequences under execution. Later, BPPA has been renamed to BTA (Basic Thread Algebra). BTA$_{\mathrm{bt}}$ is BTA extended with two constants for termination at which a Boolean value is yielded.

In $\mathrm{BTA}_{\mathrm{bt}}$, it is assumed that a fixed but arbitrary non-empty finite set $\mathscr{A}$ of *basic actions*, with $\mathsf{tau} \notin \mathscr{A}$, has been given. We write $\mathscr{A}_{\mathsf{tau}}$ for $\mathscr{A} \cup \{\mathsf{tau}\}$. The members of $\mathscr{A}_{\mathsf{tau}}$ 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 the thread proceeds. The possible replies are the Boolean values t and f. Performing the action tau will always lead 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 Sect. 5. 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 *plain termination* constant $\mathsf{S} : \mathbf{T}$;
- the *positive termination* constant $\mathsf{S{+}} : \mathbf{T}$;
- the *negative termination* constant $\mathsf{S{-}} : \mathbf{T}$;
- for each $a \in \mathscr{A}_{\mathsf{tau}}$, 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 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 thread denoted by D will become inactive, the thread denoted by S will terminate without yielding a value, and the threads denoted by $\mathsf{S{+}}$ and $\mathsf{S{-}}$ will terminate and with that yield the Boolean values t and f, respectively.

A simple example of a closed $\mathrm{BTA}_{\mathrm{bt}}$ term is

$$(b \circ \mathsf{S{+}}) \trianglelefteq a \trianglerighteq (c \circ \mathsf{S{-}}).$$

This term denotes the thread that first performs basic action a, if the reply from the execution environment on performing a is t, next performs the basic action b and after that terminates with delivery of the Boolean value t, and if the reply from the execution environment on performing a is f, next performs the basic action c and after that terminates with delivery of the Boolean value f.

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

Each closed BTA term denotes a finite thread, i.e. a thread with a finite upper bound to the number of actions that it can perform. Infinite threads, i.e. threads without a finite upper bound to the number of actions that it can perform, can be described by guarded recursion.

A *guarded 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, S, $\mathsf{S{+}}$, $\mathsf{S{-}}$ or $t \trianglelefteq a \trianglerighteq t'$ with t and t' that contain only variables from V.

We are only interested in models of $\mathrm{BTA}_{\mathrm{bt}}$ in which guarded recursive specifications have unique solutions, such as the appropriate expansion of the projective limit model of BTA presented in [3].

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
$\pi_{n+1}(\mathsf{S}) = \mathsf{S}$	P1c
$\pi_{n+1}(\mathsf{D}) = \mathsf{D}$	P2
$\pi_{n+1}(x \unlhd a \unrhd y) = \pi_n(x) \unlhd a \unrhd \pi_n(y)$	P3

A simple example of a guarded recursive specification is the one consisting of the following two equations:

$$x = x \unlhd a \unrhd y, \qquad y = y \unlhd b \unrhd \mathsf{S}.$$

The x-component of the solution of this guarded recursive specification is the thread that first performs basic action a repeatedly until the reply from the execution environment on performing a is f, next performs basic action b repeatedly until the reply from the execution environment on performing b is f, and after that terminates without delivery of a Boolean value.

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 $\mathscr{A}_{\mathsf{tau}}$ and n stands for an arbitrary natural number.

We can prove that the projections of solutions of guarded recursive specifications over $\mathrm{BTA}_{\mathrm{bt}}$ are representable by closed $\mathrm{BTA}_{\mathrm{bt}}$ terms of sort $\mathbf{T}$.

Lemma 1 *Let E be a guarded recursive specification over* $\mathrm{BTA}_{\mathrm{bt}}$*, and let x be a variable occurring in E. Then, for all $n \in \mathbb{N}$, there exists a closed* $\mathrm{BTA}_{\mathrm{bt}}$ *term p of sort* $\mathbf{T}$ *such that $\pi_n(x) = p$ is derivable from E and the axioms for the projection operators.*

Proof In the case of BTA, this is proved in [6] as part of the proof of Theorem 1 from that paper. The proof concerned goes through in the case of $\mathrm{BTA}_{\mathrm{bt}}$. □

For example, let E be the guarded recursive specification consisting of the equation $x = x \unlhd a \unrhd \mathsf{S}$ only. Then the projections of x are as follows:

$$\begin{aligned}
\pi_0(x) &= \mathsf{D}, \\
\pi_1(x) &= \mathsf{D} \unlhd a \unrhd \mathsf{S}, \\
\pi_2(x) &= (\mathsf{D} \unlhd a \unrhd \mathsf{S}) \unlhd a \unrhd \mathsf{S}, \\
\pi_3(x) &= ((\mathsf{D} \unlhd a \unrhd \mathsf{S}) \unlhd a \unrhd \mathsf{S}) \unlhd a \unrhd \mathsf{S}, \\
&\vdots
\end{aligned}$$

Henceforth, we will write $\mathrm{BTA}_{\mathrm{bt}}^{+}$ for $\mathrm{BTA}_{\mathrm{bt}}$ extended with the projection operators, the axioms for the projection operators, and AIP.

Table 3 Defining equations for the 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 $u_i = !$
$\lvert i, u_1 ; \ldots ; u_k\rvert = \mathsf{S}+$	if $u_i = !\mathsf{t}$
$\lvert i, u_1 ; \ldots ; u_k\rvert = \mathsf{S}-$	if $u_i = !\mathsf{f}$

The behaviours exhibited on execution by PGLB$_{\mathrm{bt}}$ instruction sequences are considered to be threads, with the basic instructions taken for basic actions. The *thread extraction* operation $\lvert\_\rvert$ defines, for each PGLB$_{\mathrm{bt}}$ instruction sequence, the behaviour exhibited on its execution. The thread extraction operation is defined by $\lvert u_1 ; \ldots ; u_k\rvert = \lvert 1, u_1 ; \ldots ; u_k\rvert$, where $\lvert\_, \_\rvert$ is defined by the equations given in Table 3 (for $a \in \mathfrak{A}$ and $l, i \in \mathbb{N}$)[2]and the rule that $\lvert i, u_1 ; \ldots ; u_k\rvert = \mathsf{D}$ if u_i is the beginning of an infinite jump chain.[3]

If $1 \leq i \leq k$, $\lvert i, u_1 ; \ldots ; u_k\rvert$ can be read as the behaviour exhibited by $u_1 ; \ldots ; u_k$ on execution if execution starts at the ith primitive instruction, i.e. u_i. By default, execution starts at the first primitive instruction.

Some simple examples of thread extraction are

$$\lvert +a ; \#2 ; \#3 ; b ; !\mathsf{t}\rvert = (b \circ \mathsf{S}+) \trianglelefteq a \trianglerighteq \mathsf{D},$$
$$\lvert +a ; -b ; c ; !\rvert = (\mathsf{S} \trianglelefteq b \trianglerighteq (c \circ \mathsf{S})) \trianglelefteq a \trianglerighteq (c \circ \mathsf{S}).$$

The behaviour exhibited on execution may also be an infinite thread. For example,

$$\lvert a ; +b ; \#2 ; \#3 ; c ; \#4 ; -d ; ! ; a ; \backslash\#8\rvert$$

denotes the x-component of the solution of the guarded recursive specification consisting of the following two equations:

$$x = a \circ y, \qquad y = (c \circ y) \trianglelefteq b \trianglerighteq (x \trianglelefteq d \trianglerighteq \mathsf{S}).$$

4 Services and service families

In this section, we introduce service families and a composition operator for service families. We start by introducing services.

It is assumed that a fixed but arbitrary non-empty finite set $\mathcal{M}$ of *methods* has been given. A service is able to process certain methods. The processing of a method may involve a change of the service. At completion of the processing of a method, the service produces a reply value. The set $\mathcal{R}$ of *reply values* is the 3-element set $\{\mathsf{t}, \mathsf{f}, \mathsf{d}\}$. The reply value d stands for divergent.

For example, a service may be able to process methods for pushing a natural number on a stack (`push:`n), testing whether the top of the stack equals a natural number (`topeq:`n), and

[2] As usual, we write $i \dot{-} j$ for the monus of i and j, i.e. $i \dot{-} j = i - j$ if $i \geq j$ and $i \dot{-} j = 0$ otherwise.

[3] This rule can be formalized, cf. [8].

popping the top element from the stack (pop). Processing of a pushing method or a popping method changes the service, because it changes the stack with which it deals, and produces the reply value t if no stack overflow or stack underflow occurs and f otherwise. Processing of a testing method does not change the service, because it does not changes the stack with which it deals, and produces the reply value t if the test succeeds and f otherwise. Attempted processing of a method that the service is not able to process changes the service into one that is not able to process any method and produces the reply d.

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

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

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

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

When a request is made to service S to process method m:

– if $\rho_m(S) \neq \mathsf{d}$, then S processes m, produces the reply $\rho_m(S)$, and next proceeds as $\frac{\partial}{\partial m}(S)$;
– if $\rho_m(S) = \mathsf{d}$, then S is not able to process method m and proceeds as δ.

The empty service δ is unable to process any method.

It is also assumed that a fixed but arbitrary non-empty finite set $\mathscr{F}$ 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 $\Sigma_{\mathscr{S}}$ 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 $f \in \mathscr{F}$, 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 $F \subseteq \mathscr{F}$, the unary *encapsulation* operator $\partial_F : \mathbf{SF} \to \mathbf{SF}$.

We assume that there is a countably infinite set of variables of sort **SF** which includes u, v, w. Terms are built as usual in the many-sorted case (see e.g. [38,45]). We use prefix notation for the singleton service family operators and infix notation for the service family composition operator.

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

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 the service denoted by 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. Thus, the service families denoted by closed terms of the forms $f.H$ and $\partial_{\{f\}}(C)$ do not collapse to an empty service in service family composition.

Using the singleton service family operators and the service family composition operator, any finite number of possibly identical services can be brought together in a service family provided that the services concerned are given different names. In Sect. 7, we will give an example of the use of the singleton service family operators and the service family composition operator. The empty service family constant and the encapsulation operators are primarily meant to axiomatize the operators that are introduced in Sect. 5.

The service family composition operator takes the place of the non-interfering combination operator from [19]. As suggested by the name, service family composition is composition of service families. Non-interfering combination is composition of services. The non-interfering combination of services can process all methods that can be processed by only one of the services. This has the disadvantage that its usefulness is rather limited without an additional renaming mechanism. For example, a finite number of identical services cannot be brought together in a service by means of non-interfering combination.

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

In Sect. 7, we will give an example of the use of the service family composition operator. There, we will write $\bigoplus_{i=1}^{n} C_i$, where $C_1, \ldots, C_n$ are terms of sort **SF**, for the term $C_1 \oplus \ldots \oplus C_n$.

5 Use, apply and reply

A thread may interact with the named services from the service family offered by an execution environment. That is, 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 combine $\mathrm{BTA}^{+}_{\mathrm{bt}}$ with SF and extend the combination with three operators that relate to this kind of interaction between threads and services, resulting in $\mathrm{TA}^{\mathrm{tsi}}_{\mathrm{bt}}$.

The operators in question are called the use operator, the apply operator, and the reply operator. The difference between the use operator and the apply operator is a matter of perspective: the use operator is concerned with the effects of service families on threads and therefore produces threads, whereas 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 effects of service families on the Boolean values that threads possibly deliver at their termination. The use operator and the apply operator introduced here are mainly adaptations of the use operators and the apply operators introduced in [19] to service families. The reply operator has no counterpart in [19].

The reply operator produces special values in the case where no Boolean value is delivered at termination or no termination takes place. Thus, it is accomplished that all terms with occurrences of the reply operator denote something. However, we prefer to use the reply operator only if it is known that termination with delivery of a Boolean value takes place (see also Sect. 6).

For the set $\mathscr{A}$ of basic actions, we take the set $\{f.m \mid f \in \mathscr{F}, m \in \mathscr{M}\}$. All three operators mentioned above are concerned with 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 $\mathsf{t}, \mathsf{f}, \mathsf{d}, \mathsf{m} : \mathbf{R}$;
– the binary *use* operator $\_ \,/\, \_ : \mathbf{T} \times \mathbf{SF} \to \mathbf{T}$;
– 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 use, apply and reply operators.

The thread denoted by a closed term of the form $p \,/\, C$ and the service family denoted by a closed term of the form $p \bullet C$ are the thread and service family, respectively, that result from processing the method of each basic action performed by the thread denoted by p by the service in the service family denoted by C with the focus of the basic action as its name if such a service exists. 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 basic action turns into the internal action tau and the two ways to proceed reduce to one on the basis of the reply value produced by the service. The value denoted by a closed term of the form $p \,!\, C$ is the Boolean value that the thread denoted by p/C delivers at its termination if it terminates and delivers a Boolean value at termination, the value denoted by m (standing for meaningless) if it terminates and does not deliver a Boolean value at termination, and the value denoted by d (standing for divergent) if it does not terminate. We are only interested in models of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ in which the cardinality of the domain associated with the sort $\mathbf{R}$ is 4 and the elements of this domain denoted by the constants t, f, d and m are mutually different.

A simple example of the application of the use operator, the apply operator and the reply operator is

$$((\mathtt{nns.pop} \circ \mathsf{S{+}}) \trianglelefteq \mathtt{nns.topeq{:}0} \trianglerighteq \mathsf{S{-}}) \,/\, \mathtt{nns}.\mathit{NNS}(0\sigma),$$
$$((\mathtt{nns.pop} \circ \mathsf{S{+}}) \trianglelefteq \mathtt{nns.topeq{:}0} \trianglerighteq \mathsf{S{-}}) \bullet \mathtt{nns}.\mathit{NNS}(0\sigma),$$
$$((\mathtt{nns.pop} \circ \mathsf{S{+}}) \trianglelefteq \mathtt{nns.topeq{:}0} \trianglerighteq \mathsf{S{-}}) \,!\, \mathtt{nns}.\mathit{NNS}(0\sigma),$$

where $NNS(\sigma)$ denotes a stack service as described in Sect. 4 dealing with a stack whose content is represented by the sequence σ. The first term denotes the thread that performs tau twice and then terminates with delivery of the Boolean value t. The second term denotes the stack service dealing with a stack whose content is σ, i.e. the content at termination of this thread, and the third term denotes the reply value t, i.e. the reply value delivered at termination of this thread.

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, 6 and 7. In these tables, f stands for an arbitrary focus from $\mathscr{F}$, m stands for an arbitrary method from $\mathscr{M}$, 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 use, apply and reply if inappropriate foci or methods are involved. Axioms A10 and R10 allow for reasoning about infinite threads in the contexts of apply and reply, respectively. The counterpart of A10 and R10 for use, i.e.

$$\bigwedge_{n \geq 0} \pi_n(x) \,/\, u = \pi_n(y) \,/\, v \Rightarrow x \,/\, u = y \,/\, v,$$

follows from AIP and U10.

We can prove that each closed $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ term of sort $\mathbf{T}$ can be reduced to a closed $\mathrm{BTA}_{\mathrm{bt}}$ term of sort $\mathbf{T}$.

Lemma 2 *For all closed* $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ *terms p of sort* $\mathbf{T}$*, there exists a closed* $\mathrm{BTA}_{\mathrm{bt}}$ *term q of sort* $\mathbf{T}$ *such that $p = q$ is derivable from the axioms of* $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$.

Proof In the special case of singleton service families, this is in fact proved in [6] as part of the proof of Theorem 3 from that paper. The proof of the general case follows essentially the same lines. □

In the case of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$, the notion of a guarded recursive specification is somewhat adapted. A *guarded recursive specification* over $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ 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 $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ term of sort $\mathbf{T}$ that can be rewritten, using the axioms of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$, to a term of the form D, S, S+, S− or $t \trianglelefteq a \trianglerighteq t'$ with t and t' that contain only variables from V. We are only interested in models of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ in which guarded recursive specifications have unique solutions.

A thread p in a model $\mathfrak{M}$ of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ in which guarded recursive specifications over $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ have unique solutions is *definable* if it is the solution in $\mathfrak{M}$ of a guarded recursive specification over $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$. It is easy to see that a thread is definable if it is representable by a closed $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ term of sort $\mathbf{T}$.

Henceforth, we assume that a model $\mathfrak{M}$ of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ has been given in which guarded recursive specifications over $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ have unique solutions, all threads are definable, all service families are representable by a closed $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ term of sort $\mathbf{SF}$, and all replies are representable by a closed $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ term of sort $\mathbf{R}$.

Below, we will formulate a proposition about the use, apply and reply operators using the *foci* operation foci defined by the equations in Table 8 (for foci $f \in \mathscr{F}$ and terms H of sort $\mathbf{S}$). The operation foci gives, for each service family, the set of all foci that serve as names of named services belonging to the service family. We will make use of the following properties of foci in the proof of the proposition:

1. $\mathsf{foci}(u) \cap \mathsf{foci}(v) = \emptyset$ iff $f \notin \mathsf{foci}(u)$ or $f \notin \mathsf{foci}(v)$ for all $f \in \mathscr{F}$;
2. $f \notin \mathsf{foci}(u)$ iff $\partial_{\{f\}}(u) = u$.

Table 5 Axioms for the use operator

$\mathsf{S}{+} \,/\, u = \mathsf{S}{+}$		U1
$\mathsf{S}{-} \,/\, u = \mathsf{S}{-}$		U2
$\mathsf{S} \,/\, u = \mathsf{S}$		U3
$\mathsf{D} \,/\, u = \mathsf{D}$		U4
$(\mathsf{tau} \circ x) \,/\, u = \mathsf{tau} \circ (x \,/\, u)$		U5
$(x \unlhd f.m \unrhd y) \,/\, \partial_{\{f\}}(u) = (x \,/\, \partial_{\{f\}}(u)) \unlhd f.m \unrhd (y \,/\, \partial_{\{f\}}(u))$		U6
$(x \unlhd f.m \unrhd y) \,/\, (f.H \oplus \partial_{\{f\}}(u)) = \mathsf{tau} \circ (x \,/\, (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u)))$	if $\rho_m(H) = \mathsf{t}$	U7
$(x \unlhd f.m \unrhd y) \,/\, (f.H \oplus \partial_{\{f\}}(u)) = \mathsf{tau} \circ (y \,/\, (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u)))$	if $\rho_m(H) = \mathsf{f}$	U8
$(x \unlhd f.m \unrhd y) \,/\, (f.H \oplus \partial_{\{f\}}(u)) = \mathsf{D}$	if $\rho_m(H) = \mathsf{d}$	U9
$\pi_n(x \,/\, u) = \pi_n(x) \,/\, u$		U10

Table 6 Axioms for the apply operator

$\mathsf{S}{+} \bullet u = u$		A1
$\mathsf{S}{-} \bullet u = u$		A2
$\mathsf{S} \bullet u = u$		A3
$\mathsf{D} \bullet u = \emptyset$		A4
$(\mathsf{tau} \circ x) \bullet u = x \bullet u$		A5
$(x \unlhd f.m \unrhd y) \bullet \partial_{\{f\}}(u) = \emptyset$		A6
$(x \unlhd f.m \unrhd y) \bullet (f.H \oplus \partial_{\{f\}}(u)) = x \bullet (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$	if $\rho_m(H) = \mathsf{t}$	A7
$(x \unlhd f.m \unrhd y) \bullet (f.H \oplus \partial_{\{f\}}(u)) = y \bullet (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$	if $\rho_m(H) = \mathsf{f}$	A8
$(x \unlhd f.m \unrhd y) \bullet (f.H \oplus \partial_{\{f\}}(u)) = \emptyset$	if $\rho_m(H) = \mathsf{d}$	A9
$\bigwedge_{n \geq 0} \pi_n(x) \bullet u = \pi_n(y) \bullet v \Rightarrow x \bullet u = y \bullet v$		A10

Table 7 Axioms for the reply operator

$\mathsf{S}{+} \,!\, u = \mathsf{t}$		R1
$\mathsf{S}{-} \,!\, u = \mathsf{f}$		R2
$\mathsf{S} \,!\, u = \mathsf{m}$		R3
$\mathsf{D} \,!\, u = \mathsf{d}$		R4
$(\mathsf{tau} \circ x) \,!\, u = x \,!\, u$		R5
$(x \unlhd f.m \unrhd y) \,!\, \partial_{\{f\}}(u) = \mathsf{d}$		R6
$(x \unlhd f.m \unrhd y) \,!\, (f.H \oplus \partial_{\{f\}}(u)) = x \,!\, (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$	if $\rho_m(H) = \mathsf{t}$	R7
$(x \unlhd f.m \unrhd y) \,!\, (f.H \oplus \partial_{\{f\}}(u)) = y \,!\, (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$	if $\rho_m(H) = \mathsf{f}$	R8
$(x \unlhd f.m \unrhd y) \,!\, (f.H \oplus \partial_{\{f\}}(u)) = \mathsf{d}$	if $\rho_m(H) = \mathsf{d}$	R9
$\bigwedge_{n \geq 0} \pi_n(x) \,!\, u = \pi_n(y) \,!\, v \Rightarrow x \,!\, u = y \,!\, v$		R10

Table 8 Defining equations for the foci operation

$\mathsf{foci}(\emptyset) = \emptyset$
$\mathsf{foci}(f.H) = \{f\}$
$\mathsf{foci}(u \oplus v) = \mathsf{foci}(u) \cup \mathsf{foci}(v)$

Proposition 1 *If* $\mathsf{foci}(u) \cap \mathsf{foci}(v) = \emptyset$*, then:*

1. $x \mathbin{/} (u \oplus v) = (x \mathbin{/} u) \mathbin{/} v$;
2. $x \mathbin{!} (u \oplus v) = (x \mathbin{/} u) \mathbin{!} v$;
3. $\partial_{\mathsf{foci}(u)}(x \bullet (u \oplus v)) = (x \mathbin{/} u) \bullet v$.

Proof By the definition of a guarded recursive specification over $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$, Lemmas 1 and 2, and axioms AIP, U10, A10 and R10, it is sufficient to prove for all closed $\mathrm{BTA}_{\mathrm{bt}}$ term p of sort $\mathbf{T}$:

$$p \mathbin{/} (u \oplus v) = (p \mathbin{/} u) \mathbin{/} v;$$
$$p \mathbin{!} (u \oplus v) = (p \mathbin{/} u) \mathbin{!} v;$$
$$\partial_{\mathsf{foci}(u)}(p \bullet (u \oplus v)) = (p \mathbin{/} u) \bullet v.$$

This is straightforward by induction on the structure of p, using the above-mentioned properties of foci. □

Let p and C be $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ terms of sort $\mathbf{T}$ and $\mathbf{SF}$, respectively. Then p *converges on* C, written $p \downarrow C$, is inductively defined by the following clauses:

1. $\mathsf{S} \downarrow u$;
2. $\mathsf{S+} \downarrow u$ and $\mathsf{S-} \downarrow u$;
3. if $x \downarrow u$, then $(\mathsf{tau} \circ x) \downarrow u$;
4. if $\rho_m(H) = \mathsf{t}$ and $x \downarrow (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$, then $(x \trianglelefteq f.m \trianglerighteq y) \downarrow (f.H \oplus \partial_{\{f\}}(u))$;
5. if $\rho_m(H) = \mathsf{f}$ and $y \downarrow (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$, then $(x \trianglelefteq f.m \trianglerighteq y) \downarrow (f.H \oplus \partial_{\{f\}}(u))$;
6. if $\pi_n(x) \downarrow u$, then $x \downarrow u$;

and p *diverges on* C, written $p \uparrow C$, is defined by $p \uparrow C$ iff not $p \downarrow C$. Moreover, p *converges on* C *with Boolean reply*, written $p \downarrow_{\mathbb{B}} C$, is inductively defined by the clauses $2, \ldots, 6$ for $\downarrow$ with everywhere $\downarrow$ replaced by $\downarrow_{\mathbb{B}}$.

The following two propositions concern the connection between convergence and the reply operator.

Proposition 2 *Let* p *be a closed* $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ *term of sort* $\mathbf{T}$*. Then:*

1. *if* $p \downarrow u$, $\mathsf{S+}$ *occurs in* p *and both* $\mathsf{S-}$ *and* S *do not occur in* p*, then* $p \mathbin{!} u = \mathsf{t}$;
2. *if* $p \downarrow u$, $\mathsf{S-}$ *occurs in* p *and both* $\mathsf{S+}$ *and* S *do not occur in* p*, then* $p \mathbin{!} u = \mathsf{f}$;
3. *if* $p \downarrow u$, S *occurs in* p *and both* $\mathsf{S+}$ *and* $\mathsf{S-}$ *do not occur in* p*, then* $p \mathbin{!} u = \mathsf{m}$.

Proof By Lemma 2, it is sufficient to prove it for all closed $\mathrm{BTA}_{\mathrm{bt}}$ terms p of sort $\mathbf{T}$. This is straightforward by induction on the structure of p. □

Proposition 3 *We have that* $x \downarrow u$ *iff* $x \mathbin{!} u = \mathsf{t}$ *or* $x \mathbin{!} u = \mathsf{f}$ *or* $x \mathbin{!} u = \mathsf{m}$.

Proof By the definition of a guarded recursive specification over $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$, the last clause of the inductive definition of $\downarrow$, Lemmas 1 and 2, and axiom R10, it is sufficient to prove $p \downarrow u$ iff $p \mathbin{!} u = \mathsf{t}$ or $p \mathbin{!} u = \mathsf{f}$ or $p \mathbin{!} u = \mathsf{m}$ for all closed $\mathrm{BTA}_{\mathrm{bt}}$ terms p of sort $\mathbf{T}$. This is straightforward by induction on the structure of p. □

Because the use operator, apply operator and reply operator are primarily intended to be used to describe and analyse instruction sequence processing, they are called *instruction sequence processing operators*.

We introduce the apply operator and reply operator in the setting of $\mathrm{PGLB}_{\mathrm{bt}}$ by defining:

$$P \mathbin{/} u = |P| \mathbin{/} u, \quad P \bullet u = |P| \bullet u, \quad P \mathbin{!} u = |P| \mathbin{!} u$$

for all PGLB_{bt} instruction sequences P. Similarly, we introduce convergence in the setting of PGLB_{bt} by defining:

$$P \downarrow u = |P| \downarrow u$$

for all PGLB_{bt} instruction sequences P.

6 Relevant use conventions

In the setting of service families, sets of foci play the role of interfaces. The set of all foci that serve as names of named services in a service family is regarded as the interface of that service family. There are cases in which processing does not terminate or, even worse (because it is statically detectable), interfaces of services families do not match. In the case of non-termination, there is nothing that we intend to denote by a term of the form $p \bullet C$ or $p \,!\, C$. In the case of non-matching services families, there is nothing that we intend to denote by a term of the form $C \oplus D$. Moreover, in the case of termination without a Boolean reply, there is nothing that we intend to denote by a term of the form $p \,!\, C$.

We propose to comply with the following *relevant use conventions*:

- $p \bullet C$ is only used if it is known that $p \downarrow C$;
- $p \,!\, C$ is only used if it is known that $p \downarrow_{\mathbb{B}} C$;
- $C \oplus D$ is only used if it is known that $\mathsf{foci}(C) \cap \mathsf{foci}(D) = \emptyset$.

The condition found in the first convention is justified by the fact that $x \bullet u = \emptyset$ if $x \uparrow u$. We do not have $x \bullet u = \emptyset$ only if $x \uparrow u$. For instance, $\mathsf{S}+ \bullet \emptyset = \emptyset$ whereas $\mathsf{S}+ \downarrow \emptyset$. Similar remarks apply to the condition found in the second convention.

The idea of relevant use conventions is taken from [15], where it plays a central role in an account of the way in which mathematicians usually deal with division by zero in mathematical texts. According to [15], mathematicians deal with this issue by complying with the convention that p/q is only used if it is known that $q \neq 0$. This approach is justified by the fact that there is nothing that mathematicians intend to denote by p/q if $q = 0$. It yields simpler mathematical texts than the popular approach in theoretical computer science, which is characterized by complete formality in definitions, statements and proofs. In this computer science approach, division is considered a partial function and some logic of partial functions is used. In [22], deviating from this, division is considered a total function whose value is zero in all cases of division by zero. It may be imagined that this notion of division is the one with which mathematicians make themselves familiar before they start to read and write mathematical texts professionally.

We think that the idea to comply with conventions that exclude the use of terms that are not really intended to denote anything is not only of importance in mathematics, but also in theoretical computer science. For example, the consequence of adapting Proposition 1 to comply with the relevant use conventions described above, by adding appropriate conditions to the three properties, is that we do not have to consider in the proof of the proposition the equality of terms by which we do not intend to denote anything.

In the sequel, we will comply with the relevant use conventions described above.

We can define the use operators introduced earlier in [8,14],[4] the apply operators introduced earlier in [19], and similar counterparts of the reply operator as follows:

$$x /_f H = x / f.H,$$

[4] The use operators introduced in [19] are counterparts of the abstracting use operator introduced later in Sect. 8.

$$x \bullet_f H = x \bullet f.H,$$
$$x \,!_f\, H = x \,!\, f.H.$$

These definitions give rise to the derived conventions that $p \bullet_f H$ is only used if it is known that $p \downarrow f.H$ and $p \,!_f\, H$ is only used if it is known that $p \downarrow_{\mathbb{B}} f.H$.

7 Example

In this section, we use an implementation of a bounded counter by means of a number of Boolean registers as an example to show that it is easy to bring a number of identical services together in a service family by means of the service family composition operator. Accomplishing something resemblant with the non-interfering service combination operation from [19] is quite involved. We also show in this example that there are cases in which the delivery of a Boolean value at termination of the execution of an instruction sequence is quite natural.

First, we describe services that make up Boolean registers. The Boolean register services are able to process the following methods:

– the *set to true method* set:t;
– the *set to false method* set:f;
– the *get method* get.

It is assumed that set:t, set:f, get $\in \mathscr{M}$.

The methods that Boolean register services are able to process can be explained as follows:

– set:t : the contents of the Boolean register becomes t and the reply is t;
– set:f : the contents of the Boolean register becomes f and the reply is f;
– get : nothing changes and the reply is the contents of the Boolean register.

For the set $\mathscr{S}$ of services, we take the set $\{BR_{\mathsf{t}}, BR_{\mathsf{f}}, BR_{\mathsf{d}}\}$ of *Boolean register services*. For each $m \in \mathscr{M}$, we take the functions $\frac{\partial}{\partial m}$ and ρ_m such that ($b \in \{\mathsf{t}, \mathsf{f}\}$):

$$\begin{array}{ll}
\frac{\partial}{\partial \mathsf{set{:}t}}(BR_b) = BR_{\mathsf{t}}, & \frac{\partial}{\partial m}(BR_b) = BR_{\mathsf{d}} \quad \text{if } m \notin \{\mathsf{set{:}t}, \mathsf{set{:}f}, \mathsf{get}\}, \\
\frac{\partial}{\partial \mathsf{set{:}f}}(BR_b) = BR_{\mathsf{f}}, & \frac{\partial}{\partial m}(BR_{\mathsf{d}}) = BR_{\mathsf{d}}, \\
\frac{\partial}{\partial \mathsf{get}}(BR_b) = BR_b, & \\
\rho_{\mathsf{set{:}t}}(BR_b) = \mathsf{t}, & \rho_m(BR_b) = \mathsf{d} \qquad \text{if } m \notin \{\mathsf{set{:}t}, \mathsf{set{:}f}, \mathsf{get}\}, \\
\rho_{\mathsf{set{:}f}}(BR_b) = \mathsf{f}, & \rho_m(BR_{\mathsf{d}}) = \mathsf{d}. \\
\rho_{\mathsf{get}}(BR_b) = b, &
\end{array}$$

Moreover, we take the names used above to denote the services in $\mathscr{S}$ for constants of sort **S**.

We continue with the implementation of a bounded counter by means of a number of Boolean registers. We consider a counter that can contain a natural number in the interval $[0, 2^n - 1]$ for some $n > 0$. To implement the counter, we represent its content in binary using a collection of n Boolean registers named b:0, …, b:$n-1$. We take t for 0 and f for 1, and we take the bit represented by the content of the Boolean register named b:i for a less significant bit than the bit represented by the content of the Boolean register named b:j if $i < j$.

The following instruction sequences implement set to zero, increment by one, decrement by one, and test on zero, respectively:

$$SETZERO = \mathop{;}_{i=0}^{n-1} (\mathsf{b}{:}i.\mathsf{set}{:}\mathsf{t}) \,;\, !\mathsf{t},$$

$$SUCC = \mathop{;}_{i=0}^{n-1} (-\mathsf{b}{:}i.\mathsf{get} \,;\, \#3 \,;\, \mathsf{b}{:}i.\mathsf{set}{:}\mathsf{f} \,;\, !\mathsf{t} \,;\, \mathsf{b}{:}i.\mathsf{set}{:}\mathsf{t}) \,;\, !\mathsf{f},$$

$$PRED = \mathop{;}_{i=0}^{n-1} (+\mathsf{b}{:}i.\mathsf{get} \,;\, \#3 \,;\, \mathsf{b}{:}i.\mathsf{set}{:}\mathsf{t} \,;\, !\mathsf{t} \,;\, \mathsf{b}{:}i.\mathsf{set}{:}\mathsf{f}) \,;\, !\mathsf{f},$$

$$ISZERO = \mathop{;}_{i=0}^{n-1} (-\mathsf{b}{:}i.\mathsf{get} \,;\, !\mathsf{f}) \,;\, !\mathsf{t}.$$

Concerning the Boolean values delivered at termination of executions of these instruction sequences, we have that:

$$SETZERO \;!\; \Big(\bigoplus_{i=0}^{n-1} \mathsf{b}{:}i.BR_{s_i}\Big) = \mathsf{t},$$

$$SUCC \;!\; \Big(\bigoplus_{i=0}^{n-1} \mathsf{b}{:}i.BR_{s_i}\Big) = \begin{cases} \mathsf{t} \text{ if } \bigvee_{i=0}^{n-1} s_i = \mathsf{t} \\ \mathsf{f} \text{ if } \bigwedge_{i=0}^{n-1} s_i = \mathsf{f}, \end{cases}$$

$$PRED \;!\; \Big(\bigoplus_{i=0}^{n-1} \mathsf{b}{:}i.BR_{s_i}\Big) = \begin{cases} \mathsf{t} \text{ if } \bigvee_{i=0}^{n-1} s_i = \mathsf{f} \\ \mathsf{f} \text{ if } \bigwedge_{i=0}^{n-1} s_i = \mathsf{t}, \end{cases}$$

$$ISZERO \;!\; \Big(\bigoplus_{i=0}^{n-1} \mathsf{b}{:}i.BR_{s_i}\Big) = \begin{cases} \mathsf{t} \text{ if } \bigwedge_{i=0}^{n-1} s_i = \mathsf{t} \\ \mathsf{f} \text{ if } \bigvee_{i=0}^{n-1} s_i = \mathsf{f}. \end{cases}$$

It is obvious that t is delivered at termination of an execution of *SETZERO* and that t or f is delivered at termination of an execution of *ISZERO* depending on whether the content of the counter is zero or not. Increment by one and decrement by one are both modulo 2^n. For that reason, t or f is delivered at termination of an execution of *SUCC* or *PRED* depending on whether the content of the counter is really incremented or decremented by one or not.

8 Abstracting use

With the use operator introduced in Sect. 5, the action tau is left as a trace of a basic action that has led to the processing of a method, like with the use operators on services introduced in e.g. [8,14]. However, with the use operators on services introduced in [19], nothing is left as a trace of a basic action that has led to the processing of a method. Thus, these use operators abstract fully from internal activity. In other words, they are abstracting use operators. For completeness, we introduce an abstracting variant of the use operator introduced in Sect. 5.

That is, we introduce the following additional operator:

– the binary *abstracting use* operator $\_ \,/\!/\, \_ : \mathbf{T} \times \mathbf{SF} \to \mathbf{T}$.

We use infix notation for the abstracting use operator.

Table 9 Axioms for the abstracting use operator

$\mathsf{S+} \mathbin{/\!/} u = \mathsf{S+}$		AU1
$\mathsf{S-} \mathbin{/\!/} u = \mathsf{S-}$		AU2
$\mathsf{S} \mathbin{/\!/} u = \mathsf{S}$		AU3
$\mathsf{D} \mathbin{/\!/} u = \mathsf{D}$		AU4
$(\mathsf{tau} \circ x) \mathbin{/\!/} u = \mathsf{tau} \circ (x \mathbin{/\!/} u)$		AU5
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/\!/} \partial_{\{f\}}(u) = (x \mathbin{/\!/} \partial_{\{f\}}(u)) \trianglelefteq f.m \trianglerighteq (y \mathbin{/\!/} \partial_{\{f\}}(u))$		AU6
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/\!/} (f.H \oplus \partial_{\{f\}}(u)) = x \mathbin{/\!/} (f.\frac{\partial}{\partial m} H \oplus \partial_{\{f\}}(u))$	if $\rho_m(H) = \mathsf{t}$	AU7
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/\!/} (f.H \oplus \partial_{\{f\}}(u)) = y \mathbin{/\!/} (f.\frac{\partial}{\partial m} H \oplus \partial_{\{f\}}(u))$	if $\rho_m(H) = \mathsf{f}$	AU8
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/\!/} (f.H \oplus \partial_{\{f\}}(u)) = \mathsf{D}$	if $\rho_m(H) = \mathsf{d}$	AU9
$\bigwedge_{n \geq 0} \pi_n(x) \mathbin{/\!/} u = \pi_n(y) \mathbin{/\!/} v \Rightarrow x \mathbin{/\!/} u = y \mathbin{/\!/} v$		AU10

The axioms for the abstracting use operator are given in Table 9. Owing to the possible concealment of actions by abstracting use, $\pi_n(x \mathbin{/\!/} u) = \pi_n(x) \mathbin{/\!/} u$ is not a plausible axiom. However, axiom AU10 allows for reasoning about infinite threads in the context of abstracting use.

9 Functional units

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

It is assumed that a non-empty finite or countably infinite 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 $(m, M) \in \mathcal{H}$.

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 $\{(m, M) \in \mathcal{H} \mid m \in I\}$.

Let $\mathcal{H} \in \mathcal{FU}(S)$. Then an *extension* of $\mathcal{H}$ is an $\mathcal{H}' \in \mathcal{FU}(S)$ such that $\mathcal{H} \subseteq \mathcal{H}'$.

The following is a simple illustration of the use of functional units. An unbounded counter can be modelled by a functional unit for $\mathbb{N}$ with method operations for set to zero, increment by one, decrement by one, and test on zero.

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 to 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 PGLB$_{\mathrm{sbt}}$ instruction sequences to derive partial method operations from the method operations of a functional unit. We write $\mathscr{L}(f.I)$, where $I \subseteq \mathscr{M}$, for the set of all PGLB$_{\mathrm{sbt}}$ instruction sequences, taking the set $\{f.m \mid m \in I\}$ 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 service 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 $\mathscr{F}\mathscr{U}(S) \times S$ as the set $\mathscr{S}$ of services. We write $\mathscr{H}(s)$, where $\mathscr{H} \in \mathscr{F}\mathscr{U}(S)$ and $s \in S$, for the service $(\mathscr{H}, s)$. The functions $\frac{\partial}{\partial m}$ and ρ_m are defined as follows:

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

$$\rho_m(\mathscr{H}(s)) = \begin{cases} m^r_{\mathscr{H}}(s) & \text{if } m \in \mathscr{I}(\mathscr{H}) \\ \mathsf{d} & \text{if } m \notin \mathscr{I}(\mathscr{H}), \end{cases}$$

where s' is a fixed but arbitrary state in S. In order to be able to make use of the axioms for the apply operator and the reply operator from Sect. 5 hereafter, we want to use these operators for the services being considered here when making the idea of deriving a partial method operation by means of an instruction sequence precise. Therefore, we assume that there is a constant of sort $\mathbf{S}$ for each $\mathscr{H}(s) \in \mathscr{S}$.[5] In this connection, we use the following notational convention: for each $\mathscr{H}(s) \in \mathscr{S}$, we write $\mathscr{H}(s)$ for the constant of sort $\mathbf{S}$ whose interpretation is $\mathscr{H}(s)$. Note that the service $\emptyset(\sigma')$ is the interpretation of the empty service constant δ.

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

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

where

$$\begin{array}{l} |x|^r_{\mathscr{H}}(s) = x \ ! \ f.\mathscr{H}(s), \\ |x|^e_{\mathscr{H}}(s) = \text{the unique } s' \in S \text{ such that } x \bullet f.\mathscr{H}(s) = f.\mathscr{H}(s'). \end{array}$$

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

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

[5] This may lead to an uncountable number of constants, which is unproblematic and quite normal in model theory.

Theorem 1

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

Proof Property 1: We have to prove that $\mathscr{H} \leq \mathscr{H}'$ and $\mathscr{H}' \leq \mathscr{H}''$ implies $\mathscr{H} \leq \mathscr{H}''$. It is sufficient to show that we can obtain instruction sequences in $\mathscr{L}(f.\mathscr{I}(\mathscr{H}''))$ that produce the method operations of $\mathscr{H}$ from the instruction sequences in $\mathscr{L}(f.\mathscr{I}(\mathscr{H}'))$ that produce the method operations of $\mathscr{H}$ and the instruction sequences in $\mathscr{L}(f.\mathscr{I}(\mathscr{H}''))$ that produce the method operations of $\mathscr{H}'$. 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 $m \in \mathscr{I}(\mathscr{H})$, let M be such that $(m, M) \in \mathscr{H}$, and let $x_m \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}'))$ be such that $M = |x_m|_{\mathscr{H}'}$. Suppose that $\mathscr{I}(\mathscr{H}') = \{m'_1, \ldots, m'_n\}$. For each $i \in [1, n]$, let M'_i be such that $(m'_i, M'_i) \in \mathscr{H}'$ and let $x_{m'_i} = u^i_1 ; \ldots ; u^i_{k_i} ; !\mathsf{t} ; !\mathsf{f} \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}''))$ be such that $M'_i = |x_{m'_i}|_{\mathscr{H}''}$. Consider the $x'_m \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}''))$ obtained from x_m as follows: for each $i \in [1, n]$, (i) first increase each jump over the leftmost occurrence of $+f.m'_i$ in x_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 $M = |x'_m|_{\mathscr{H}''}$.

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

Two functional units $\mathscr{H}$ and $\mathscr{H}'$ belong to the same functional unit degree if and only if $\mathscr{H}$ and $\mathscr{H}'$ have the same derived method operations. A functional unit degree $\mathscr{D}$ is below a functional unit $\mathscr{H}$ if and only if all derived method operations of some member of $\mathscr{D}$ are derived method operations of $\mathscr{H}$.

The binary relation $\leq$ on $\mathscr{FU}(S)$ is reminiscent of the relative computability relation $\leq$ on algebras introduced in [28] because functional units can be looked upon as algebras of a special kind. In the definition of this relative computability relation on algebras, the role of instruction sequences is filled by flow charts. A more striking difference is that the relation allows for algebras with different domains to be related. This corresponds to a relation on functional units that allows for the states from one state space to be represented by the states from another state space. To the best of our knowledge, the work presented in [28] and a few preceding papers of the same authors is the only work on computability that is concerned with a relation comparable to the relation $\leq$ on $\mathscr{FU}(S)$ defined above.

10 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 $\mathscr{FU}(\mathbb{N})$ is an unbounded counter. The method names involved are setzero, succ, pred, and iszero. The method operations involved are the functions $\mathit{Setzero}, \mathit{Succ}, \mathit{Pred}, \mathit{Iszero} : \mathbb{N} \to \mathbb{B} \times \mathbb{N}$ 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:

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

The following proposition shows that there are infinitely many functional units for natural numbers with mutually different sets of derived method operations whose method operations are derived method operations of a major restriction of the functional unit *Counter*.

Proposition 4 *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 $\mathscr{H}_n \in \mathscr{F}\mathscr{U}(\mathbb{N})$ such that $\mathscr{H}_n \leq (\{\mathsf{pred}, \mathsf{iszero}\}, \mathit{Counter})$ as follows:

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

where

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

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 an $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}_m))$ such that $|x|_{\mathscr{H}_m}(m) = (\mathsf{t}, m-n)$ because $\mathit{Pred}{:}m(m) = (\mathsf{t}, 0)$. Hence, $\mathscr{H}_n \not\leq \mathscr{H}_m$ for all $n, m \in \mathbb{N}$ with $n < m$. □

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

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

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

Take an arbitrary $P \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}'))$ such that $|P|_{\mathscr{H}'}$ is a derived method operations of $\mathscr{H}'$. It follows immediately from the definition of thread extraction that $|P|$ is the solution of a finite linear recursive specification over $\mathrm{BTA}_{\mathrm{bt}}$, i.e. a finite guarded recursive specification over $\mathrm{BTA}_{\mathrm{bt}}$ in which the right-hand side of each equation is a $\mathrm{BTA}_{\mathrm{bt}}$ term of the form D, $\mathsf{S}{+}$, $\mathsf{S}{-}$ or $x \trianglelefteq a \trianglerighteq y$ where x and y are variables of sort $\mathbf{T}$. Let E be a finite linear recursive specification over $\mathrm{BTA}_{\mathrm{bt}}$ of which the solution for x_1 is $|P|$. Because $|P|_{\mathscr{H}'}$ 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

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

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 $|P|^e_{\mathscr{H}'}$:

$$\{F_i(s) = F_{l(i)}(m_i{}^e_{\mathscr{H}'}(s)) \cdot \overline{\mathsf{sg}}(\chi_i(s)) + F_{r(i)}(m_i{}^e_{\mathscr{H}'}(s)) \cdot \mathsf{sg}(\chi_i(s)) \mid i \in [1, n]\}$$
$$\cup \{F_{n+1}(s) = s,\, F_{n+2}(s) = s\},$$

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

$$\chi_i(s) = 0 \;\Leftrightarrow\; m_i{}^r_{\mathscr{H}'}(s) = \mathsf{t},$$

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 $m_i{}^e_{\mathscr{H}'}$ 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 $|P|^e_{\mathscr{H}'}$ is defined recursively in the sense of [26]. This means that $|P|^e_{\mathscr{H}'}$ is general recursive, and hence computable.

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

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

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

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

As a corollary of Theorem 10.3 from [39],[6] we have that each computable $M \in \mathscr{M}\mathscr{O}(\mathbb{N})$ can be computed by means of a register machine with six registers, say r0, r1, r2, r3, r4, and r5. The registers are used as follows: r0 as input register; r1 as output register for the output in $\mathbb{B}$; r2 as output register for the output in $\mathbb{N}$; r3, r4 and r5 as auxiliary registers. The content of r1 represents the Boolean output as follows: 0 represents t and all other natural numbers represent f. For each $i \in [0, 5]$, register ri can be incremented by one, decremented by one, and tested for zero by means of instructions ri.succ, ri.pred and ri.iszero, respectively. We write $\mathscr{L}(\mathscr{R}\mathscr{M}_6)$ for the set of all PGLB$_{\mathrm{sbt}}$ instruction sequences, taking the set $\{\mathsf{r}i.\mathsf{succ}, \mathsf{r}i.\mathsf{pred}, \mathsf{r}i.\mathsf{iszero} \mid i \in [0, 5]\}$ as the set $\mathfrak{A}$ of basic instructions. Clearly, $\mathscr{L}(\mathscr{R}\mathscr{M}_6)$ is adequate to represent all register machine programs using six registers.

We define a computable functional unit $\mathscr{U} \in \mathscr{F}\mathscr{U}(\mathbb{N})$ 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:

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

where the method operations are defined as follows:

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

[6] That theorem can be looked upon as a corollary of Theorem Ia from [30].

and, for each $i \in [0, 5]$:[7]

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

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

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

where

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

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

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

Take an arbitrary computable $M \in \mathscr{MO}(\mathbb{N})$. Then there exists an instruction sequence in $\mathscr{L}(\mathscr{RM}_6)$ that computes M. Take an arbitrary $P \in \mathscr{L}(\mathscr{RM}_6)$ that computes M. Then $|\texttt{rml2ful}(P)|_{\mathscr{U}} = M$. Hence, M is a derived method operation of $\mathscr{U}$. □

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

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

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

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

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

7 As usual, we write $x \mid y$ for y is divisible by x.

where the method operations are defined as follows:

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

where

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

We have that $M_i(x) = G3(3^i \cdot 2^x)$ for each $i \in [0, 19]$. Moreover, state $3^i \cdot 2^x$ can be obtained from state x by first applying $G1$ once and next applying $G2\, i$ times. Hence, for each $i \in [0, 19]$, $|f.\mathsf{g1} \,;\, f.\mathsf{g2}^i \,;\, +f.\mathsf{g3} \,;\, !\mathsf{t} \,;\, !\mathsf{f}|_{\mathscr{U}'} = M_i$.[8] This means that $M_0, \ldots, M_{19}$ are derived method operations of $\mathscr{U}'$. □

The universal computable functional unit $\mathscr{U}'$ 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 $\mathscr{H} \in \mathscr{FU}(\mathbb{N})$ with only one method operation that is universal.*

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

Assume that there exists a computable $\mathscr{H} \in \mathscr{FU}(\mathbb{N})$ with one method operation such that $\mathit{Counter} \leq \mathscr{H}$. Let $\mathscr{H}' \in \mathscr{FU}(\mathbb{N})$ be such that $\mathscr{H}'$ has one method operation and $\mathit{Counter} \leq \mathscr{H}'$, and let m be the unique method name such that $\mathscr{I}(\mathscr{H}') = \{m\}$. Take arbitrary $P_1, P_2 \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}'))$ such that $|P_1|_{\mathscr{H}'} = \mathit{Succ}$ and $|P_2|_{\mathscr{H}'} = \mathit{Pred}$. Then $|P_1|_{\mathscr{H}'}(0) = (\mathsf{t}, 1)$ and $|P_2|_{\mathscr{H}'}(1) = (\mathsf{t}, 0)$. Instruction $f.m$ is processed at least once if P_1 is applied to $\mathscr{H}'(0)$ or P_2 is applied to $\mathscr{H}'(1)$. Let k_0 be the number of times that instruction $f.m$ is processed on application of P_1 to $\mathscr{H}'(0)$ and let k_1 be the number of times that instruction $f.m$ is processed on application of P_2 to $\mathscr{H}'(1)$ (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_1 to $\mathscr{H}'(0)$ at most $k_0 + k_1$ different states can be reached. This contradicts with $|P_1|_{\mathscr{H}'} = \mathit{Succ}$. Hence, there does not exist a computable $\mathscr{H} \in \mathscr{FU}(\mathbb{N})$ with one method operation such that $\mathit{Counter} \leq \mathscr{H}$. □

It is an open problem whether two method operations suffice.

To the best of our knowledge, there are no existing results in computability theory directly related to Theorems 3, 4 and 5. We could not even say which existing notion from computability theory corresponds to the universality of a functional unit for natural numbers.

In Sect. 11, we will give a rough sketch of a universal functional unit for a state space whose elements can be understood as the possible contents of the tape of Turing machines with a particular tape alphabet. This universal functional unit corresponds to the common part of all Turing machines with that tape alphabet. The part that differs for different Turing

[8] 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 $u^{n+2} = u \,;\, u^{n+1}$.

machines is what is usually called their "transition function" or "program". In the current setting, the role of that part is filled by an instruction sequence whose instructions correspond to the method operations of the above-mentioned universal functional unit. This means that different instruction sequences are needed for different Turing machines with the tape alphabet concerned, but the same universal functional unit suffices for all of them. In particular, the same universal functional unit suffices for universal Turing machines and non-universal Turing machines.

11 Functional units relating to Turing machine tapes

In this section, we define some notions that have a bearing on the halting problem in the setting of $\mathrm{PGLB_{sbt}}$ and functional units. The notions in question are defined in terms of functional units for the following state space:

$$\mathbb{T} = \{v\hat{\ }w \mid v, w \in \{0, 1, :\}^*\}.$$

The elements of $\mathbb{T}$ can be understood as the possible contents of the tape of a Turing machine whose tape alphabet is $\{0, 1, :\}$, including the position of the tape head. Consider an element $v\hat{\ }w \in \mathbb{T}$. Then v corresponds to the content of the tape to the left of the position of the tape head and w corresponds to the content of the tape from the position of the tape head to the right—the indefinite numbers of padding blanks at both ends are left out. The colon serves as a separator of bit sequences. This is for instance useful if the input of a program consists of another program and an input to the latter program, both encoded as a bit sequences. We could have taken any other tape alphabet whose cardinality is greater than one, but $\{0, 1, :\}$ is extremely handy when dealing with issues relating to the halting problem. In fact, we could first have introduced the general notation $\mathbb{T}_A$, where A stands for a finite set of tape symbols, for the set $\{v\hat{\ }w \mid v, w \in A^*\}$ and then have introduced $\mathbb{T}$ as an abbreviation for $\mathbb{T}_{\{0,1,:\}}$.

Below, we will use a computable injective function $\alpha : \mathbb{T} \to \mathbb{N}$ to encode the members of $\mathbb{T}$ as natural numbers. Because $\mathbb{T}$ is a countably infinite set, we assume that it is understood what is a computable function from $\mathbb{T}$ to $\mathbb{N}$. An obvious instance of a computable injective function $\alpha : \mathbb{T} \to \mathbb{N}$ is the one where $\alpha(a_1 \dots a_n)$ is the natural number represented in the quinary number-system by $a_1 \dots a_n$ if the symbols $0,\ 1,\ :$ and $\hat{\ }$ are taken as digits representing the numbers $1,\ 2,\ 3$ and 4, respectively.

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

Like in the case of $\mathcal{FU}(\mathbb{N})$, a computable $\mathcal{H} \in \mathcal{FU}(\mathbb{T})$ is *universal* if for each computable $\mathcal{H}' \in \mathcal{FU}(\mathbb{T})$, we have $\mathcal{H}' \leq \mathcal{H}$.

An example of a computable functional unit in $\mathcal{FU}(\mathbb{T})$ is the functional unit whose method operations correspond to the basic steps that a Turing machine with tape alphabet $\{0, 1, :\}$ can perform on its tape. It turns out that this functional unit is universal, which can be proved using simple programming in $\mathrm{PGLB_{bt}}$.

It is assumed that, for each $\mathcal{H} \in \mathcal{FU}(\mathbb{T})$, a computable injective function from $\mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ to $\{0, 1\}^*$ with a computable image has been given that yields, for each $x \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}))$, an encoding of x as a bit sequence. If we consider the case where the jump lengths in jump instructions are character strings representing the jump lengths in decimal notation and method names are character strings, such an encoding function can easily be obtained using the ASCII character-encoding. We use the notation $\overline{x}$ to denote the encoding of x as a bit sequence.

Let $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$, and let $I \subseteq \mathscr{I}(\mathscr{H})$. Then:

- $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ produces a *solution of the halting problem* for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$ if:
$$x \downarrow f.\mathscr{H}(v) \text{ for all } v \in \mathbb{T},$$
$$x \mathbin{!} f.\mathscr{H}(\hat{}\,\overline{y}{:}v) = \mathsf{t} \Leftrightarrow y \downarrow f.\mathscr{H}(\hat{}\,v) \text{ for all } y \in \mathscr{L}(f.I) \text{ and } v \in \{0, 1, :\}^* \ ;$$
- $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ produces a *reflexive solution of the halting problem* for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$ if x produces a solution of the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$ and $x \in \mathscr{L}(f.I)$;
- the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$ is *autosolvable* if there exists an $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ such that x produces a reflexive solution of the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$;
- the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$ is *potentially autosolvable* if there exists an extension $\mathscr{H}'$ of $\mathscr{H}$ such that the halting problem for $\mathscr{L}(f.\mathscr{I}(\mathscr{H}'))$ with respect to $\mathscr{H}'$ is autosolvable;
- the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$ is *potentially recursively autosolvable* if there exists an extension $\mathscr{H}'$ of $\mathscr{H}$ such that the halting problem for $\mathscr{L}(f.\mathscr{I}(\mathscr{H}'))$ with respect to $\mathscr{H}'$ is autosolvable and $\mathscr{H}'$ is computable.

These definitions make clear that each combination of an $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ and an $I \subseteq \mathscr{I}(\mathscr{H})$ gives rise to a *halting problem instance*.

In Sects. 12 and 13, we will make use of a method operation $\mathit{Dup} \in \mathscr{M}\mathscr{O}(\mathbb{T})$ for duplicating bit sequences. This method operation is defined as follows:

$$\begin{aligned} \mathit{Dup}(v\,\hat{}\,w) &= \mathit{Dup}(\hat{}\,vw), \\ \mathit{Dup}(\hat{}\,v) &= (\mathsf{t}, \hat{}\,v{:}v) && \text{if } v \in \{0, 1\}^*, \\ \mathit{Dup}(\hat{}\,v{:}w) &= (\mathsf{t}, \hat{}\,v{:}v{:}w) && \text{if } v \in \{0, 1\}^*. \end{aligned}$$

Proposition 5 *Let $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ be such that $(\mathsf{dup}, \mathit{Dup}) \in \mathscr{H}$, let $I \subseteq \mathscr{I}(\mathscr{H})$ be such that $\mathsf{dup} \in I$, let $x \in \mathscr{L}(f.I)$, and let $v \in \{0, 1\}^*$ and $w \in \{0, 1, :\}^*$ be such that $w = v$ or $w = v{:}w'$ for some $w' \in \{0, 1, :\}^*$. Then $(f.\mathsf{dup}\ ;\ x) \mathbin{!} f.\mathscr{H}(\hat{}\,w) = x \mathbin{!} f.\mathscr{H}(\hat{}\,v{:}w)$.*

Proof This follows immediately from the definition of *Dup* and the axioms for !. □

The method operation *Dup* is a derived method operation of the above-mentioned functional unit whose method operations correspond to the basic steps that a Turing machine with tape alphabet {0, 1, :} can perform on its tape. This follows immediately from the computability of *Dup* and the universality of this functional unit.

In Sects. 12 and 13, we will make use of two simple transformations of PGLB$_{\mathrm{sbt}}$ instruction sequences that affect only their termination behaviour on execution and the Boolean value yielded at termination in the case of termination. Here, we introduce notations for those transformations.

Let x be a PGLB$_{\mathrm{sbt}}$ instruction sequence. Then we write $\mathit{swap}(x)$ for x with each occurrence of !t replaced by !f and each occurrence of !f replaced by !t, and we write $\mathit{f2d}(x)$ for x with each occurrence of !f replaced by #0. In the following proposition, the most important properties relating to these transformations are stated.

Proposition 6 *Let x be a* PGLB$_{\mathrm{sbt}}$ *instruction sequence. Then:*

1. *if* $x \mathbin{!} u = \mathsf{t}$ *then* $\mathit{swap}(x) \mathbin{!} u = \mathsf{f}$ *and* $\mathit{f2d}(x) \mathbin{!} u = \mathsf{t}$;
2. *if* $x \mathbin{!} u = \mathsf{f}$ *then* $\mathit{swap}(x) \mathbin{!} u = \mathsf{t}$ *and* $\mathit{f2d}(x) \mathbin{!} u = \mathsf{d}$.

Proof Let p be a closed $\mathrm{BTA}_{\mathrm{bt}}$ term of sort $\mathbf{T}$. Then we write $\mathit{swap}'(p)$ for p with each occurrence of $\mathsf{S}{+}$ replaced by $\mathsf{S}{-}$ and each occurrence of $\mathsf{S}{-}$ replaced by $\mathsf{S}{+}$, and we write $\mathit{f2d}'(p)$ for p with each occurrence of $\mathsf{S}{-}$ replaced by D. It is easy to prove by induction on i that $|i, \mathit{swap}(x)| = \mathit{swap}'(|i, x|)$ and $|i, \mathit{f2d}(x)| = \mathit{f2d}'(|i, x|)$ for all $i \in \mathbb{N}$. By this result, Lemma 1, and axiom R10, it is sufficient to prove the following for each closed $\mathrm{BTA}_{\mathrm{bt}}$ term p of sort $\mathbf{T}$:

if $p \mathbin{!} u = \mathsf{t}$ then $\mathit{swap}'(p) \mathbin{!} u = \mathsf{f}$ and $\mathit{f2d}'(p) \mathbin{!} u = \mathsf{t}$;
if $p \mathbin{!} u = \mathsf{f}$ then $\mathit{swap}'(p) \mathbin{!} u = \mathsf{t}$ and $\mathit{f2d}'(p) \mathbin{!} u = \mathsf{d}$.

This is easy by induction on the structure of p. □

By the use of foci and the introduction of apply and reply operators on service families, we make it possible to deal with cases that remind of multi-tape Turing machines, Turing machines that has random access memory, etc. However, in this paper, we will only consider the case that reminds of single-tape Turing machines. This means that we will use only one focus (f) and only singleton service families.

12 Interpreters

It is often mentioned in textbooks on computability that an interpreter, which is a program for simulating the execution of programs that it is given as input, cannot solve the halting problem because the execution of the interpreter will not terminate if the execution of its input program does not terminate. In this section, we have a look at the termination behaviour of interpreters in the setting of $\mathrm{PGLB}_{\mathrm{sbt}}$ and functional units.

Let $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$, let $I \subseteq \mathscr{I}(\mathscr{H})$, and let $I' \subseteq I$. Then $x \in \mathscr{L}(f.I)$ is an *interpreter* for $\mathscr{L}(f.I')$ with respect to $\mathscr{H}$ if for all $y \in \mathscr{L}(f.I')$ and $v \in \{0, 1, :\}^*$:

$$y \downarrow f.\mathscr{H}(\hat{\ }v) \Rightarrow$$
$$x \downarrow f.\mathscr{H}(\hat{\ }\overline{y}{:}v) \wedge x \bullet f.\mathscr{H}(\hat{\ }\overline{y}{:}v) = y \bullet f.\mathscr{H}(\hat{\ }v) \wedge x \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}{:}v) = y \mathbin{!} f.\mathscr{H}(\hat{\ }v).$$

Moreover, $x \in \mathscr{L}(f.I)$ is a *reflexive interpreter* for $\mathscr{L}(f.I')$ with respect to $\mathscr{H}$ if x is an interpreter for $\mathscr{L}(f.I')$ with respect to $\mathscr{H}$ and $x \in \mathscr{L}(f.I')$.

The following theorem states that a reflexive interpreter that always terminates is impossible in the presence of the method operation *Dup*.

Theorem 6 *Let* $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ *be such that* $(\mathsf{dup}, \mathit{Dup}) \in \mathscr{H}$, *let* $I \subseteq \mathscr{I}(\mathscr{H})$ *be such that* $\mathsf{dup} \in I$, *and let* $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ *be a reflexive interpreter for* $\mathscr{L}(f.I)$ *with respect to* $\mathscr{H}$. *Then there exist an* $y \in \mathscr{L}(f.I)$ *and a* $v \in \{0, 1, :\}^*$ *such that* $x \uparrow f.\mathscr{H}(\hat{\ }\overline{y}{:}v)$.

Proof Assume the contrary. Take $y = f.\mathsf{dup} \,;\, \mathit{swap}(x)$. By the assumption, $x \downarrow f.\mathscr{H}(\hat{\ }\overline{y}{:}\overline{y})$. By Propositions 3 and 6, it follows that $\mathit{swap}(x) \downarrow f.\mathscr{H}(\hat{\ }\overline{y}{:}\overline{y})$ and $\mathit{swap}(x) \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}{:}\overline{y}) \neq x \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}{:}\overline{y})$. By Propositions 3 and 5, it follows that $(f.\mathsf{dup} \,;\, \mathit{swap}(x)) \downarrow f.\mathscr{H}(\hat{\ }\overline{y})$ and $(f.\mathsf{dup} \,;\, \mathit{swap}(x)) \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}) \neq x \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}{:}\overline{y})$. Since $y = f.\mathsf{dup} \,;\, \mathit{swap}(x)$, we have $y \downarrow f.\mathscr{H}(\hat{\ }\overline{y})$ and $y \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}) \neq x \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}{:}\overline{y})$. Because x is a reflexive interpreter, this implies $x \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}{:}\overline{y}) = y \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y})$ and $y \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}) \neq x \mathbin{!} f.\mathscr{H}(\hat{\ }\overline{y}{:}\overline{y})$. This is a contradiction. □

It is easy to see that Theorem 6 goes through for all functional units for $\mathbb{T}$ of which *Dup* is a derived method operation. Recall that the functional units concerned include the afore-mentioned functional unit whose method operations correspond to the basic steps that a Turing machine with tape alphabet {0, 1, :} can perform on its tape.

For each $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$, $m \in \mathscr{I}(\mathscr{H})$, and $v \in \mathbb{T}$, we have $(f.m \ ; \ !\mathsf{t} \ ; \ !\mathsf{f}) \downarrow f.\mathscr{H}(v)$. This leads us to the following corollary of Theorem 6.

Corollary 1 *For all $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ with $(\mathsf{dup}, \mathit{Dup}) \in \mathscr{H}$ and $I \subseteq \mathscr{I}(\mathscr{H})$ with $\mathsf{dup} \in I$, there does not exist an $m \in I$ such that $f.m \ ; \ !\mathsf{t} \ ; \ !\mathsf{f}$ is a reflexive interpreter for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$.*

To the best of our knowledge, there are no existing results in computability theory or elsewhere directly related to Theorem 6. It looks as if the closest to this result are results on termination of particular interpreters for particular logic and functional programming languages.

13 Autosolvability of the halting problem

Because a reflexive interpreter that always terminates is impossible in the presence of the method operation *Dup*, we must conclude that solving the halting problem by means of a reflexive interpreter is out of the question in the presence of the method operation *Dup*. The question arises whether the proviso “by means of a reflexive interpreter” can be dropped. In this section, we answer this question in the affirmative. Before we present this negative result concerning autosolvability of the halting problem, we present a positive result.

Let $M \in \mathscr{M}\mathscr{O}(\mathbb{T})$. Then we say that M *increases the number of colons* if for some $v \in \mathbb{T}$ the number of colons in $M^e(v)$ is greater than the number of colons in v.

Theorem 7 *Let $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ be such that no method operation of $\mathscr{H}$ increases the number of colons. Then there exist an extension $\mathscr{H}'$ of $\mathscr{H}$, an $I' \subseteq \mathscr{I}(\mathscr{H}')$, and an $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}'))$ such that x produces a reflexive solution of the halting problem for $\mathscr{L}(f.I')$ with respect to $\mathscr{H}'$.*

Proof Let $\mathsf{halting} \in \mathscr{M}$ be such that $\mathsf{halting} \notin \mathscr{I}(\mathscr{H})$. Take $I' = \mathscr{I}(\mathscr{H}) \cup \{\mathsf{halting}\}$. Take $\mathscr{H}' = \mathscr{H} \cup \{(\mathsf{halting}, \mathit{Halting})\}$, where $\mathit{Halting} \in \mathscr{M}\mathscr{O}(\mathbb{T})$ is defined as follows:

$$\begin{array}{lll}
\mathit{Halting}(v\,\hat{}\,w) & = \mathit{Halting}(\hat{}\,vw), & \\
\mathit{Halting}(\hat{}\,v) & = (\mathsf{f}, \hat{}\,) & \text{if } v \in \{0,1\}^*, \\
\mathit{Halting}(\hat{}\,v{:}w) & = (\mathsf{f}, \hat{}\,) & \text{if } v \in \{0,1\}^* \wedge \forall x \in \mathscr{L}(f.I') \bullet v \neq \overline{x}, \\
\mathit{Halting}(\hat{}\,\overline{x}{:}w) & = (\mathsf{f}, \hat{}\,) & \text{if } x \in \mathscr{L}(f.I') \wedge x \uparrow f.\mathscr{H}'(w), \\
\mathit{Halting}(\hat{}\,\overline{x}{:}w) & = (\mathsf{t}, \hat{}\,) & \text{if } x \in \mathscr{L}(f.I') \wedge x \downarrow f.\mathscr{H}'(w).
\end{array}$$

Then $+f.\mathsf{halting} \ ; \ !\mathsf{t} \ ; \ !\mathsf{f}$ produces a reflexive solution of the halting problem for $\mathscr{L}(f.I')$ with respect to $\mathscr{H}'$. □

Theorem 7 tells us that there exist functional units $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ with the property that the halting problem is potentially autosolvable for $\mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ with respect to $\mathscr{H}$. Thus, we know that there exist functional units $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ with the property that the halting problem is autosolvable for $\mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ with respect to $\mathscr{H}$.

There exists an $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ for which *Halting* as defined in the proof of Theorem 7 is computable.

Theorem 8 *Let $\mathscr{H} = \emptyset$ and $\mathscr{H}' = \mathscr{H} \cup \{(\mathsf{halting}, \mathit{Halting})\}$, where Halting is as defined in the proof of Theorem 7. Then, Halting is computable.*

Proof It is sufficient to prove for an arbitrary $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}'))$ that, for all $v \in \mathbb{T}$, $x \downarrow f.\mathscr{H}'(v)$ is decidable. We will prove this by induction on the number of colons in v.

The basis step. Because the number of colons in v equals 0, $\mathit{Halting}(v) = (\mathsf{f}, \hat{})$. It follows that $x \downarrow f.\mathscr{H}'(v) \Leftrightarrow x' \downarrow \emptyset$, where x' is x with each occurrence of $f.\mathsf{halting}$ and $-f.\mathsf{halting}$ replaced by #1 and each occurrence of $+f.\mathsf{halting}$ replaced by #2. Because x' is finite, $x' \downarrow \emptyset$ is decidable. Hence, $x \downarrow f.\mathscr{H}'(v)$ is decidable.

The inductive step. Because the number of colons in v is greater than 0, either $\mathit{Halting}(v) = (\mathsf{t}, \hat{})$ or $\mathit{Halting}(v) = (\mathsf{f}, \hat{})$. It follows that $x \downarrow f.\mathscr{H}'(v) \Leftrightarrow x' \downarrow \emptyset$, where x' is x with:

– each occurrence of $f.\mathsf{halting}$ and $+f.\mathsf{halting}$ replaced by #1 if the occurrence leads to the first application of *Halting* and $\mathit{Halting}^r(v) = \mathsf{t}$, and by #2 otherwise;
– each occurrence of $-f.\mathsf{halting}$ replaced by #2 if the occurrence leads to the first application of *Halting* and $\mathit{Halting}^r(v) = \mathsf{t}$, and by #1 otherwise.

An occurrence of $f.\mathsf{halting}$, $+f.\mathsf{halting}$ or $-f.\mathsf{halting}$ in x leads to the first application of *Halting* iff $|1, x| = |i, x|$, where i is the position of the occurrence in x. Because x is finite, it is decidable whether an occurrence of $f.\mathsf{halting}$, $+f.\mathsf{halting}$ or $-f.\mathsf{halting}$ leads to the first processing of $\mathsf{halting}$. Moreover, by the induction hypothesis, it is decidable whether $\mathit{Halting}^r(v) = \mathsf{t}$. Because x' is finite, it follows that $x' \downarrow \emptyset$ is decidable. Hence, $x \downarrow f.\mathscr{H}'(v)$ is decidable. □

Theorems 7 and 8 together tell us that there exists a functional unit $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$, viz. $\emptyset$, with the property that the halting problem is potentially recursively autosolvable for $\mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ with respect to $\mathscr{H}$.

Let $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ be such that all derived method operations of $\mathscr{H}$ are computable and do not increase the number of colons. Then the halting problem is potentially autosolvable for $\mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ with respect to $\mathscr{H}$. However, the halting problem is not always potentially recursively autosolvable for $\mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ with respect to $\mathscr{H}$ because otherwise the halting problem would always be decidable.

The following theorem tells us essentially that potential autosolvability of the halting problem is precluded in the presence of the method operation *Dup*.

Theorem 9 *Let $\mathscr{H} \in \mathscr{F}\mathscr{U}(\mathbb{T})$ be such that $(\mathsf{dup}, \mathit{Dup}) \in \mathscr{H}$, and let $I \subseteq \mathscr{I}(\mathscr{H})$ be such that $\mathsf{dup} \in I$. Then there does not exist an $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ such that x produces a reflexive solution of the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$.*

Proof Assume the contrary. Let $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ be such that x produces a reflexive solution of the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$, and let $y = f.\mathsf{dup} \,;\mathit{f2d}(\mathit{swap}(x))$. Then $x \downarrow f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y})$. By Propositions 3 and 6, it follows that $\mathit{swap}(x) \downarrow f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y})$ and either $\mathit{swap}(x) \,!\, f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y}) = \mathsf{t}$ or $\mathit{swap}(x) \,!\, f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y}) = \mathsf{f}$.

In the case where $\mathit{swap}(x) \,!\, f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y}) = \mathsf{t}$, we have by Proposition 6 that (i) $\mathit{f2d}(\mathit{swap}(x)) \,!\, f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y}) = \mathsf{t}$ and (ii) $x \,!\, f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y}) = \mathsf{f}$. By Proposition 5, it follows from (i) that $(f.\mathsf{dup} \,;\mathit{f2d}(\mathit{swap}(x))) \,!\, f.\mathscr{H}(\hat{}\,\overline{y}) = \mathsf{t}$. Since $y = f.\mathsf{dup} \,;\mathit{f2d}(\mathit{swap}(x))$, we have $y \,!\, f.\mathscr{H}(\hat{}\,\overline{y}) = \mathsf{t}$. On the other hand, because x produces a reflexive solution, it follows from (ii) that $y \uparrow f.\mathscr{H}(\hat{}\,\overline{y})$. By Proposition 3, this contradicts with $y \,!\, f.\mathscr{H}(\hat{}\,\overline{y}) = \mathsf{t}$.

In the case where $\mathit{swap}(x) \,!\, f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y}) = \mathsf{f}$, we have by Proposition 6 that (i) $\mathit{f2d}(\mathit{swap}(x)) \,!\, f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y}) = \mathsf{d}$ and (ii) $x \,!\, f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y}) = \mathsf{t}$. By Proposition 5, it follows from (i) that $(f.\mathsf{dup} \,;\mathit{f2d}(\mathit{swap}(x))) \,!\, f.\mathscr{H}(\hat{}\,\overline{y}) = \mathsf{d}$. Since $y = f.\mathsf{dup} \,;\mathit{f2d}(\mathit{swap}(x))$, we have $y\ !\ f.\mathscr{H}(\hat{}\,\overline{y}) = \mathsf{d}$. On the other hand, because x produces a reflexive solution, it follows from (ii) that $y \downarrow f.\mathscr{H}(\hat{}\,\overline{y})$. By Proposition 3, this contradicts with $y\ !\ f.\mathscr{H}(\hat{}\,\overline{y}) = \mathsf{d}$. □

It is easy to see that Theorem 9 goes through for all functional units for $\mathbb{T}$ of which *Dup* is a derived method operation. Recall that the functional units concerned include the afore-mentioned functional unit whose method operations correspond to the basic steps that a Turing machine with tape alphabet $\{0, 1, :\}$ can perform on its tape. Because of this, the unsolvability of the halting problem for Turing machines can be understood as a corollary of Theorem 9.

Below, we will give an alternative proof of Theorem 9. A case distinction is needed in both proofs, but in the alternative proof it concerns a minor issue. The issue in question is covered by the following lemma.

Lemma 3 *Let $\mathscr{H} \in \mathscr{FU}(\mathbb{T})$, let $I \subseteq \mathscr{I}(\mathscr{H})$, let $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ be such that x produces a reflexive solution of the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$, let $y \in \mathscr{L}(f.I)$, and let $v \in \{0, 1, :\}^*$. Then $y \downarrow f.\mathscr{H}(\hat{}\,v)$ implies $y\ !\ f.\mathscr{H}(\hat{}\,v) = x\ !\ f.\mathscr{H}(\hat{}\,\overline{f2d(y)}{:}v)$.*

Proof By Proposition 3, it follows from $y \downarrow f.\mathscr{H}(\hat{}\,v)$ that either $y\ !\ f.\mathscr{H}(\hat{}\,v) = \mathsf{t}$ or $y\ !\ f.\mathscr{H}(\hat{}\,v) = \mathsf{f}$.

In the case where $y\ !\ f.\mathscr{H}(\hat{}\,v) = \mathsf{t}$, we have by Propositions 3 and 6 that $f2d(y) \downarrow f.\mathscr{H}(\hat{}\,v)$ and so $x\ !\ f.\mathscr{H}(\hat{}\,\overline{f2d(y)}{:}v) = \mathsf{t}$.

In the case where $y\ !\ f.\mathscr{H}(\hat{}\,v) = \mathsf{f}$, we have by Propositions 3 and 6 that $f2d(y) \uparrow f.\mathscr{H}(\hat{}\,v)$ and so $x\ !\ f.\mathscr{H}(\hat{}\,\overline{f2d(y)}{:}v) = \mathsf{f}$. □

Proof (Another proof of Theorem 9.) Assume the contrary. Let $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ be such that x produces a reflexive solution of the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$, and let $y = f2d(swap(f.\mathsf{dup}\ ;\ x))$. Then $x \downarrow f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y})$. By Propositions 3, 5 and 6, it follows that $swap(f.\mathsf{dup}\,;x) \downarrow f.\mathscr{H}(\hat{}\,\overline{y})$. By Lemma 3, it follows that $swap(f.\mathsf{dup}\,;x)\ !\ f.\mathscr{H}(\hat{}\,\overline{y}) = x\ !\ f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y})$. By Proposition 6, it follows that $(f.\mathsf{dup}\ ;\ x)\ !\ f.\mathscr{H}(\hat{}\,\overline{y}) \neq x\ !\ f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y})$. On the other hand, by Proposition 5, we have that $(f.\mathsf{dup}\,;x)\ !\ f.\mathscr{H}(\hat{}\,\overline{y}) = x\ !\ f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y})$. This contradicts with $(f.\mathsf{dup}\ ;\ x)\ !\ f.\mathscr{H}(\hat{}\,\overline{y}) \neq x\ !\ f.\mathscr{H}(\hat{}\,\overline{y}{:}\overline{y})$. □

Both proofs of Theorem 9 given above are diagonalization proofs in disguise.

Now, let $\mathscr{H} = \{(\mathsf{dup}, Dup)\}$. By Theorem 9, the halting problem for $\mathscr{L}(f.\{\mathsf{dup}\})$ with respect to $\mathscr{H}$ is not (potentially) autosolvable. However, it is decidable.

Theorem 10 *Let $\mathscr{H} = \{(\mathsf{dup}, Dup)\}$. Then the halting problem for $\mathscr{L}(f.\{\mathsf{dup}\})$ with respect to $\mathscr{H}$ is decidable.*

Proof Let $x \in \mathscr{L}(f.\{\mathsf{dup}\})$, and let x' be x with each occurrence of $f.\mathsf{dup}$ and $+f.\mathsf{dup}$ replaced by #1 and each occurrence of $-f.\mathsf{dup}$ replaced by #2. For all $v \in \mathbb{T}$, $Dup^r(v) = \mathsf{t}$. Therefore, $x \downarrow f.\mathscr{H}(v) \Leftrightarrow x' \downarrow \emptyset$ for all $v \in \mathbb{T}$. Because x' is finite, $x' \downarrow \emptyset$ is decidable. □

It follows from Theorem 10 that there exists a computable method operation by means of which a solution for the halting problem for $\mathscr{L}(f.\{\mathsf{dup}\})$ can be produced. This leads us to the following corollary of Theorem 10.

Corollary 2 *There exist a computable $\mathscr{H} \in \mathscr{FU}(\mathbb{T})$ with $(\mathsf{dup}, Dup) \in \mathscr{H}$, an $I \subseteq \mathscr{I}(\mathscr{H})$ with $\mathsf{dup} \in I$, and an $x \in \mathscr{L}(f.\mathscr{I}(\mathscr{H}))$ such that x produces a solution of the halting problem for $\mathscr{L}(f.I)$ with respect to $\mathscr{H}$.*

To the best of our knowledge, there are no existing results in computability theory directly related to Theorems 7, 8, 9 and 10. The closest to these results are probably the positive results in the setting of Turing machines that have been obtained with restrictions on the number of states, the minimum of the number of transitions where the tape head moves to the left and the number of transitions where the tape head moves to the right, or the number of different combinations of input symbol, direction of head move, and output symbol occurring in the transitions (see e.g. [29,36]).

14 Concluding remarks

We have presented a re-design of the extension of basic thread algebra that was used in previous work to deal with the interaction between instruction sequences under execution and components of their execution environment concerning the processing of instructions. The changes introduced allow for the material from quite a part of that work to be streamlined. Moreover, we have introduced the notion of a functional unit. Using the resulting setting, we have obtained a novel computability result about functional units for natural numbers and several novel results relating to the autosolvability of the halting problem.

The following remarks may clarify the relationship between the setting that is used in this paper and the setting of Turing machines and the extent to which the results presented in this paper can be transferred to the setting of Turing machines.

Each single-tape Turing machine can be simulated by means of a thread that interacts with a service from a singleton service family. The thread and service correspond to the finite control and tape of the single-tape Turing machine. The threads that correspond to the finite controls of single-tape Turing machines are examples of regular threads, i.e. threads that can only evolve into a finite number of other threads. Similar remarks can be made about multi-tape Turing machines, register machines, multi-stack machines, et cetera.

The results about functional units can probably be transferred to the setting of Turing machines after the notion of a functional unit has been linked with that setting. However, we believe that the setting of Turing machines does not lend itself well to the investigation of the universality of functional units for natural numbers. The results relating to the autosolvability of the halting problem cannot be transferred to the setting of Turing machines because that setting corresponds to a restriction to a single fixed functional unit in our setting. The point is that all Turing machines have the same tape manipulation features. Because of that only the effects of restrictions on the use of these features on the solvability of the halting problem are open for investigation in the setting of Turing machines.

The following remarks touch on closely related previous work on the halting problem and an interesting option for related future work on the halting problem.

The results relating to the autosolvability of the halting problem extend and strengthen the results regarding the halting problem for programs given in [20] in a setting which looks to be more adequate to describe and analyse issues regarding the halting problem for programs. It happens that decidability depends on the halting problem instance considered. This is different in the case of the on-line halting problem for programs, i.e. the problem to forecast during its execution whether a program will eventually terminate (see [20]).

The bounded halting problem for programs is the problem to determine, given a program and an input to the program, whether execution of the program on that input will terminate after no more than a fixed number of steps. An interesting option for future work is to investigate whether we can find a lower bound for the complexity of solving the bounded halting problem for programs using an appropriate functional unit.

The following remarks are miscellaneous ones relating to the material presented in the current paper.

We have proposed three instruction sequence processing operators: the use operator, the apply operator and the reply operator. The apply operator fits in with the viewpoint that programs are state transformers that can be modelled by partial functions. This viewpoint was first taken in the early days of denotational semantics, see e.g. [31,41,43].

Pursuant to [15], we have also proposed to comply with conventions that exclude the use of terms that can be built by means of the proposed operators, but are not really intended to denote anything. The idea to comply with such conventions looks to be more widely applicable in theoretical computer science.

In the 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 $\mathscr{M}\mathscr{O}(\mathbb{B})$. There are in principle 2^{16} different functional units in $\mathscr{F}\mathscr{U}(\mathbb{B})$, 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 $\mathscr{F}\mathscr{U}(\mathbb{B})/\equiv$. However, it is straightforward to show that $\mathscr{F}\mathscr{U}(\mathbb{B})/\equiv$ 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 $\mathscr{F}\mathscr{U}(S_k)$. Already with $k = 3$, it becomes unclear whether the number of functional unit degrees in $\mathscr{F}\mathscr{U}(S_k)$ can be determined manually. Actually, we do not know at the moment whether it can be determined with computer support either.

Acknowledgments We thank two anonymous referees for carefully reading preliminary versions of this paper and for suggesting improvements of the presentation of the paper.

References

1. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press, Cambridge (2009)
2. Baker, H.G.: Precise instruction scheduling without a precise machine model. SIGARCH Comput. Archit. News **19**(6), 4–8 (1991)
3. Bergstra, J.A., Bethke, I.: Polarized process algebra and program equivalence. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) Proceedings 30th ICALP, Lecture Notes in Computer Science, vol. 2719, pp. 1–21. Springer, New York (2003)
4. Bergstra, J.A., Loots, M.E.: Program algebra for component code. Formal Aspects Comput. **12**(1), 1–17 (2000)
5. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. J. Logic Algebraic Programm. **51**(2), 125–156 (2002)
6. Bergstra, J.A., Middelburg, C.A.: A thread algebra with multi-level strategic interleaving. Theory Comput. Syst. **41**(1), 3–32 (2007)
7. Bergstra, J.A., Middelburg, C.A.: Instruction sequences and non-uniform complexity theory. arXiv:0809.0352v3 [cs.CC] (2008)
8. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. J. Appl. Logic **6**(4), 553–563 (2008)
9. Bergstra, J.A., Middelburg, C.A.: Autosolvability of halting problem instances for instruction sequences. arXiv:0911.5018v3 [cs.LO] (2009)
10. Bergstra, J.A., Middelburg, C.A.: Functional units for natural numbers. arXiv:0911.1851v3 [cs.PL] (2009)
11. Bergstra, J.A., Middelburg, C.A.: Indirect jumps improve instruction sequence performance. arXiv:0909.2089v2 [cs.PL] (2009)

12. Bergstra, J.A., Middelburg, C.A.: Instruction sequence processing operators. arXiv:0910.5564v4 [cs.LO] (2009)
13. Bergstra, J.A., Middelburg, C.A.: On the operating unit size of load/store architectures. Math. Struct. Comput. Sci. **20**(3), 395–417 (2010)
14. Bergstra, J.A., Middelburg, C.A.: A thread calculus with molecular dynamics. Inform. Comput. **208**(7), 817–844 (2010)
15. Bergstra, J.A., Middelburg, C.A.: Inversive meadows and divisive meadows. J. Appl. Logic **9**(3), 203–220 (2011)
16. Bergstra, J.A., Middelburg, C.A.: On the behaviours produced by instruction sequences under execution. arXiv:1106.6196v1 [cs.PL] (2011)
17. Bergstra, J.A., Middelburg, C.A.: Thread extraction for polyadic instruction sequences. Sci. Ann. Comput. Sci. **21**(2), 283–310 (2011)
18. Bergstra, J.A., Middelburg, C.A.: On the expressiveness of single-pass instruction sequences. Theory Comput. Syst. **50**(2), 313–328 (2012)
19. Bergstra, J.A., Ponse, A.: Combining programs and state machines. J. Logic Algebr. Programm. **51**(2), 175–192 (2002)
20. Bergstra, J.A., Ponse, A.: Execution architectures for program algebra. J. Appl. Logic **5**(1), 170–192 (2007)
21. Bergstra, J.A., Ponse, A.: An instruction sequence semigroup with involutive anti-automorphisms. Sci. Ann. Comput. Sci. **19**, 57–92 (2009)
22. Bergstra, J.A., Tucker, J.V.: The rational numbers as an abstract data type. J. ACM **54**(2), Article 7 (2007)
23. Brock, C., Hunt, W.A.: Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In: ICCD '97, pp. 31–36 (1997)
24. Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., Gill, J.: MIPS: A microprocessor architecture. In: MICRO '82, pp. 17–22 (1982)
25. Hermes, H.: Enumerability, Decidability, Computability. Springer, Berlin (1965)
26. Kleene, S.C.: General recursive functions of natural numbers. Math. Ann. **112**, 727–742 (1936)
27. Lunde, A.: Empirical evaluation of some features of instruction set processor architectures. Commun. ACM **20**(3), 143–153 (1977)
28. Lynch, N.A., Blum, E.K.: Relative complexity of algebras. Math. Syst. Theory **14**(1), 193–214 (1981)
29. Margenstern, M.: Decidability and undecidability of the halting problem on Turing machines, a survey. In: Adian, S., Nerode, A. (eds.) LFCS'97, Lecture Notes in Computer Science, vol. 1234, pp. 226–236. Springer, New York (1997)
30. Minsky, M.L.: Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines. Ann. Math. **74**(3), 437–455 (1961)
31. Mosses, P.D.: The mathematical semantics of ALGOL 60. Tech. Rep. PRG-12, Programming Research Group, Oxford University (1974)
32. Mosses, P.D.: Formal semantics of programming languages—an overview. Electron. Notes Theor. Comput. Sci. **148**, 41–73 (2006)
33. Nair, R., Hopkins, M.E.: Exploiting instruction level parallelism in processors by caching scheduled groups. SIGARCH Comput. Archit. News **25**(2), 13–25 (1997)
34. Ofelt, D., Hennessy, J.L.: Efficient performance prediction for modern microprocessors. In: SIGMETRICS '00, pp. 229–239 (2000)
35. Patterson, D.A., Ditzel, D.R.: The case for the reduced instruction set computer. SIGARCH Comput. Archit. News **8**(6), 25–33 (1980)
36. Pavlotskaya, L.M.: Solvability of the halting problem for certain classes of Turing machines. Math. Notes **13**(6), 537–541 (1973)
37. Ponse, A., van der Zwaag, M.B.: An introduction to program and thread algebra. In: Beckmann, A. et al. (eds.) CiE 2006, Lecture Notes in Computer Science, vol. 3988, pp. 445–458. Springer, New York (2006)
38. Sannella, D., Tarlecki, A.: Algebraic preliminaries. In: Astesiano, E., Kreowski, H.J., Krieg-Brückner, B. (eds.) Algebraic Foundations of Systems Specification, pp. 13–30. Springer, Berlin (1999)
39. Shepherdson, J.C., Sturgis, H.E.: Computability of recursive functions. J. ACM **10**(2), 217–255 (1963)
40. Sipser, M.: Introduction to the Theory of Computation, 2nd edn. Thomson, Boston (2006)
41. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, Series in Computer Science. MIT Press, Cambridge (1977)
42. Tennenhouse, D.L., Wetherall, D.J.: Towards an active network architecture. SIGCOMM Comput. Commun. Rev. **37**(5), 81–94 (2007)

43. Tennent, R.D.: A denotational definition of the programming language Pascal. Tech. Rep. TR77-47, Department of Computing and Information Sciences, Queen's University, Kingston, Ontario, Canada (1977)
44. Turing, A.M.: On computable numbers, with an application to the Entscheidungs problem. Proc. Lond. Math. Soc. Ser. 2 **42**, 230–265 (1937). Correction: ibid, **43**, 544–546 (1937)
45. Wirsing, M.: Algebraic specification. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol B, pp. 675–788. Elsevier, Amsterdam (1990)
46. Xia, C., Torrellas, J.: Instruction prefetching of systems codes with layout optimized for reduced cache misses. In: ISCA '96, pp. 271–282 (1996)