University of Amsterdam
Programming Research Group

Autosolvability of Halting Problem Instances for Instruction Sequences

J.A. Bergstra
C.A. Middelburg

Report PRG0914 November 2009

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

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

C.A. Middelburg

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

e-mail: kmiddelb@science.uva.nl

Programming Research Group Electronic Report Series

Autosolvability of Halting Problem Instances for Instruction Sequences

J.A. Bergstra and C.A. Middelburg

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

Abstract. We position Turing's result regarding the undecidability of the halting problem as a result about programs rather than machines. The mere requirement that a program of a certain kind must solve the halting problem for all programs of that kind leads to a contradiction in the case of a recent unsolvability result regarding the halting problem for programs. In this paper, we investigate this autosolvability requirement in a setting in which programs take the form of instruction sequences.

Keywords: halting problem, instruction sequence, autosolvability, functional unit.

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

1 Introduction

The halting problem is frequently paraphrased as follows: the halting problem is the problem to determine, given a program and an input to the program, whether execution of the program on that input will eventually terminate. To indicate that this problem might be undecidable, it is often mentioned that an interpreter, which is a program that simulates the execution of programs that it is given as input, cannot solve the halting problem because the interpreter will not terminate if its input program does not terminate. However, Turing's result regarding the undecidability of the halting problem is a result about Turing machines rather than programs. It says that there does not exist a single Turing machine that, given the description of an arbitrary Turing machine and input, will determine whether the computation of that Turing machine applied to that input eventually halts (see e.g. [8]).

Our objective is to position Turing's result regarding the undecidability of the halting problem as a result about programs rather than machines. In the case of the unsolvability result regarding the halting problem for programs presented in [6], the mere requirement that a program of a certain kind must solve the halting problem for all programs of that kind leads to a contradiction. In this paper, we pay closer attention to this autosolvability requirement. Like in [6], we carry out our investigation in a setting in which programs take the form of instruction sequences. The instruction set concerned includes instructions whose processing needs a device that resembles the tape of a Turing machine.

The work presented in this paper belongs to a line of research in which program algebra [1] is the setting used for investigating issues in which instruction sequences are involved. The starting-point of program algebra is the perception of a program as a single-pass instruction sequence, i.e. a finite or infinite sequence of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over. Moreover, basic thread algebra [1] is the setting used for modelling the behaviours exhibited by instruction sequences under execution.[1] In this paper, we use a program notation rooted in program algebra, instead of program algebra itself. The program notation in question was first presented in [4]. In that paper, the concept of a functional unit is introduced and studied. Here, we will model the devices that resemble the tape of a Turing machine by a functional unit.

This paper is organized as follows. First, we give a survey of the program notation used in this paper (Section 2) and define its semantics using basic thread algebra (Section 3). Next, we extend basic thread algebra with operators that are related to the processing of instructions by services (Section 4) and introduce those operator in the setting of the program notation used (Section 5). Then, we introduce the concept of a functional unit (Section 6) and define autosolvability and related notions in terms of functional units related to Turing machine tapes (Section 7). After that, we discuss the weakness of interpreters when it comes to solving the halting problem (Section 8) and give positive and negative results concerning the autosolvability of the halting problem (Section 9). Finally, we make some concluding remarks (Section 10).

2 PGLB with Boolean Termination

In this section, we give a survey of the program notation $\mathrm{PGLB}_{\mathrm{bt}}$. This program notation is a variant of the program notation PGLB, which belongs to a hierarchy of program notations rooted in program algebra (see [1]). $\mathrm{PGLB}_{\mathrm{bt}}$ is PGLB with the Boolean termination instructions $!\mathtt{t}$ and $!\mathtt{f}$ from [5] instead of the termination instruction !. PGLB and $\mathrm{PGLB}_{\mathrm{bt}}$ are close to existing assembly languages and have relative jump instructions.

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

$\mathrm{PGLB}_{\mathrm{bt}}$ has the following primitive instructions:

- for each $a \in \mathfrak{A}$, a *plain basic instruction* a;
- for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$;
- for each $l \in \mathbb{N}$, a *forward jump instruction* $\#l$;
- for each $l \in \mathbb{N}$, a *backward jump instruction* $\backslash\#l$;
- a *positive termination instruction* $!\mathtt{t}$;
- a *negative termination instruction* $!\mathtt{f}$.

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

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

On execution of a PGLB$_{\mathrm{bt}}$ program, 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 l-th 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 l-th previous primitive instruction – if l equals 0 or there is no primitive instructions to proceed with, deadlock occurs;
- the effect of the positive termination instruction $!\mathsf{t}$ is that execution terminates and in doing so delivers the Boolean value t;
- the effect of the negative termination instruction $!\mathsf{t}$ is that execution terminates and in doing so delivers the Boolean value f.

3 Thread Extraction

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

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

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

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

- the *deadlock* constant $\mathsf{D} : \mathbf{T}$;
- the *positive termination* constant $\mathsf{S+} : \mathbf{T}$;
- the *negative termination* constant $\mathsf{S-} : \mathbf{T}$;

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

$x \unlhd \mathsf{tau} \unrhd y = x \unlhd \mathsf{tau} \unrhd x$	T1

Table 2. Approximation induction principle

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

– for each $a \in \mathcal{A}_{\mathsf{tau}}$, the binary *postconditional composition* operator $\_ \unlhd a \unrhd \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

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 $p \unlhd a \unrhd p$.

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

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

Each closed $\mathrm{BTA}_{\mathrm{bt}}$ term of sort $\mathbf{T}$ denotes a thread that will become inactive or terminate after it has performed finitely many actions. Infinite threads can be described by 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, $\mathsf{S+}$, $\mathsf{S-}$ or $t \unlhd a \unrhd t'$ 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. Regular threads, i.e. threads that can only be in a finite number of states, are solutions of finite guarded recursive specifications.

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

Table 3. Defining equations for thread extraction operation

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

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 $E \Rightarrow \pi_n(x) = p$.*

Proof. In the case of BTA, this is proved in [2] as part of the proof of Theorem 1 from that paper. The proof concerned goes through in the case of $\mathrm{BTA}_{\mathrm{bt}}$. □

The behaviours exhibited on execution by $\mathrm{PGLB}_{\mathrm{bt}}$ programs are considered to be regular threads, with the basic instructions taken for basic actions. The *thread extraction* operation $\lvert\_\rvert$ defines, for each $\mathrm{PGLB}_{\mathrm{bt}}$ program, the behaviour exhibited on execution by that $\mathrm{PGLB}_{\mathrm{bt}}$ program. The thread extraction operation is defined by $\lvert u_1 ; \ldots ; u_k \rvert = \lvert 1, u_1 ; \ldots ; u_k \rvert$, where $\lvert\_,\_\rvert$ is defined by the equations given in Table 3 (for $a \in \mathfrak{A}$ and $l, i \in \mathbb{N}$) 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.[2]

4 Interaction between Threads and Services

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

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

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

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

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

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

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

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

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

- the sort $\mathbf{SF}$ of *service families*;

and the following constant and operators:

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

We assume that there is a countably infinite set of variables of sort $\mathbf{SF}$ which includes u, v, w. Terms are built as usual in the many-sorted case (see e.g. [7, 9]). We use prefix notation for the singleton service family operators and infix notation for the service family composition operator.

The service family denoted by $\emptyset$ is the empty service family. The service family denoted by a closed term of the form $f.H$ consists of one named service only, the service concerned is 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.

The axioms of SF are given in Table 4. In this table, f stands for an arbitrary

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

focus from $\mathcal{F}$ and H and H' stand for arbitrary closed terms of sort **S**. The axioms of SF simply formalize the informal explanation given above.

Below we will introduce two operators related to the interaction between threads and services. They are called the apply operator and the reply operator. The apply operator is concerned with the effects of threads on service families and therefore produces service families. The reply operator is concerned with the effects of service families on the Boolean values that threads deliver at their termination. The reply operator does not only produce Boolean values: it produces a special value in cases where no termination takes place.

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

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

– the sort **R** of *replies*;

and the following constants and operators:

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

We use infix notation for the apply and reply operators.

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

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

Table 5. Axioms for apply operator

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

Table 6. Axioms for reply operator

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

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

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

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

Moreover, p *diverges on* C, written $p \uparrow C$, is defined by $p \uparrow C$ iff not $p \downarrow C$.

In the case where $p \uparrow C$, either the processing of methods does not halt or inappropriate foci or methods are involved. In that case, there is nothing that we intend to denote by a term of the form $p \bullet C$ or $p \,!\, C$. 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 $p \downarrow C$.

The condition found in the first convention is justified by the fact that in the intended model of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$, for definable threads x, $x \bullet u = \emptyset$ if $x \uparrow u$ (see [5]). We do not have $x \bullet u = \emptyset$ only if $x \uparrow u$. For instance, $\mathsf{S}+ \bullet \emptyset = \emptyset$ whereas $\mathsf{S}+ \downarrow \emptyset$.

5 Interaction between Programs and Services

In this paper, the apply operator and reply operator are primarily intended to be used in the setting of $\mathrm{PGLB}_{\mathrm{bt}}$. In this section, we introduce the apply operator and reply operator in the setting of $\mathrm{PGLB}_{\mathrm{bt}}$. We also introduce notations for two simple transformations of $\mathrm{PGLB}_{\mathrm{bt}}$ programs that affect only their termination behaviour on execution and the Boolean value yielded at termination in the case of termination. These notations will be used in Sections 8 and 9.

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

$$x \bullet u = |x| \bullet u \;, \quad x \,!\, u = |x| \,!\, u$$

for all $\mathrm{PGLB}_{\mathrm{bt}}$ programs x. Similarly, we introduce convergence in the setting of $\mathrm{PGA}_{\mathrm{bt}}$ by defining:

$$x \downarrow u = |x| \downarrow u$$

for all $\mathrm{PGLB}_{\mathrm{bt}}$ programs x.

The following proposition states that convergence corresponds with termination.

Proposition 1. *Let x be a $\mathrm{PGLB}_{\mathrm{bt}}$ program. Then $x \downarrow u$ iff $x \,!\, u = \mathsf{t}$ or $x \,!\, u = \mathsf{f}$.*

Proof. By the definition of $|\_|$, the last clause of the inductive definition of $\downarrow$, axiom R9, and Lemma 1 it is sufficient to prove $x \downarrow u$ iff $p \,!\, u = \mathsf{t}$ or $p \,!\, u = \mathsf{f}$ for each closed $\mathrm{BTA}_{\mathrm{bt}}$ term p of sort $\mathbf{T}$. This is easy by induction on the structure of p. $\square$

In Sections 8 and 9, we will make use of two simple transformations of $\mathrm{PGLB}_{\mathrm{bt}}$ programs. Here, we introduce notations for those transformations.

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

Proposition 2. *Let x be a $\mathrm{PGLB}_{\mathrm{bt}}$ program. Then:*

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

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, axiom R9, and Lemma 1 it is sufficient to prove the following for each closed $\mathrm{BTA}_{\mathrm{bt}}$ term p of sort $\mathbf{T}$:

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

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

6 Functional Units

In this section, we introduce the concept of a functional unit and related concepts. The concept of a functional unit was first introduced in [4].

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

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

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

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

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

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

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

We assume that each $\mathcal{H}(s) \in \mathcal{S}$ can be denoted by a closed term of sort $\mathbf{S}$. In this connection, we use the following notational convention: for each $\mathcal{H}(s) \in \mathcal{S}$, we write $\mathcal{H}(s)$ for an arbitrary closed term of sort $\mathbf{T}$ that denotes $\mathcal{H}(s)$. The ambiguity thus introduced could be obviated by decorating $\mathcal{H}(s)$ wherever it stands for a closed term. However, in this paper, it is always immediately clear from the context whether it stands for a closed term. Moreover, we believe that the decorations are more often than not distracting. Therefore, we leave it to the reader to make the decorations mentally wherever appropriate.

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

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

where

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

If $|x|_H$ is total, then it is called a *derived method operation* of H.

The binary relation $\leq$ on $\mathcal{FU}(S)$ is defined by $H \leq H'$ iff for all $(m, M) \in H$, M is a derived method operation of H'. The binary relation $\equiv$ on $\mathcal{FU}(S)$ is defined by $H \equiv H'$ iff $H \leq H'$ and $H' \leq H$. In [4], it is proved that $\leq$ is a quasi-order relation and $\equiv$ is an equivalence relation.

7 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 PGLB_{bt} and functional units. The notions in question are defined in terms of functional units for the following state space:

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

The states from V resemble the possible contents of the tape of a Turing machine whose tape alphabet is $\{0, 1, :\}$. Consider a state $v\hat{}w \in V$. 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 seperator 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.

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

In the sequel, we will comply with the relevant use conventions introduced at the end of Section 4.

It is assumed that, for each $\mathcal{H} \in \mathcal{FU}(V)$, an injective function from $\mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ to $\{0, 1\}^*$ has been given that yields for each $x \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}))$, an encoding of x as a bit sequence. We use the notation $\overline{x}$ to denote the encoding of x as a bit sequence.

Let $\mathcal{H} \in \mathcal{FU}(V)$, and let $I \subseteq \mathcal{I}(\mathcal{H})$. Then:

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

These definitions make clear that each combination of an $\mathcal{H} \in \mathcal{FU}(V)$ and an $I \subseteq \mathcal{I}(\mathcal{H})$ gives rise to a *halting problem instance.*

In Section 8 and 9, we will make use of a method operation $\mathit{Dup} \in \mathcal{MO}(V)$ 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 3. *Let $\mathcal{H} \in \mathcal{FU}(V)$ be such that $(\mathsf{dup}, \mathit{Dup}) \in \mathcal{H}$, let $I \subseteq \mathcal{I}(\mathcal{H})$ be such that $\mathsf{dup} \in I$, let $x \in \mathcal{L}(f.I)$, 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 $(f.\mathsf{dup} \mathbin{;} x) \mathbin{!} f.\mathcal{H}(\hat{\ }w) = x \mathbin{!} f.\mathcal{H}(\hat{\ }v{:}w)$.*

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

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.

8 Interpreters

It is often mentioned 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 upon the termination behaviour of interpreters in the setting of $\mathrm{PGLB_{bt}}$ and functional units.

Let $\mathcal{H} \in \mathcal{FU}(V)$, let $I \subseteq \mathcal{I}(\mathcal{H})$, and let $I' \subseteq I$. Then $x \in \mathcal{L}(f.I)$ is an *interpreter* for $\mathcal{L}(f.I')$ with respect to $\mathcal{H}$ if for all $y \in \mathcal{L}(f.I')$ and $v \in \{0,1,:\}^*$:

$$y \downarrow f.\mathcal{H}(\hat{\ }v) \Rightarrow x \downarrow f.\mathcal{H}(\hat{\ }\overline{y}{:}v)\ ,$$
$$x \bullet f.\mathcal{H}(\hat{\ }\overline{y}{:}v) = y \bullet f.\mathcal{H}(\hat{\ }v) \text{ and } x \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}{:}v) = y \mathbin{!} f.\mathcal{H}(\hat{\ }v)\ .$$

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

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

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

Proof. Assume the contrary. Take $y = f.\mathsf{dup} \mathbin{;} \mathit{swap}(x)$. By the assumption, $x \downarrow f.\mathcal{H}(\hat{\ }\overline{y}{:}\overline{y})$. By Propositions 1 and 2, it follows that $\mathit{swap}(x) \downarrow f.\mathcal{H}(\hat{\ }\overline{y}{:}\overline{y})$ and $\mathit{swap}(x) \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}{:}\overline{y}) \neq x \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}{:}\overline{y})$. By Propositions 1 and 3, it follows that $(f.\mathsf{dup} \mathbin{;} \mathit{swap}(x)) \downarrow f.\mathcal{H}(\hat{\ }\overline{y})$ and $(f.\mathsf{dup} \mathbin{;} \mathit{swap}(x)) \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}) \neq x \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}{:}\overline{y})$. Since $y = f.\mathsf{dup} \mathbin{;} \mathit{swap}(x)$, we have $y \downarrow f.\mathcal{H}(\hat{\ }\overline{y})$ and $y \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}) \neq x \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}{:}\overline{y})$. Because x is a reflexive interpreter, this implies $x \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}{:}\overline{y}) = y \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y})$ and $y \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}) \neq x \mathbin{!} f.\mathcal{H}(\hat{\ }\overline{y}{:}\overline{y})$. This is a contradiction. □

In the proof of Theorem 1, the presence of the method operation *Dup* is essential. It is easy to see that the theorem goes through for all functional units for V of which *Dup* is a derived method operation. An example of such a functional unit is the one whose method operations correspond to the basic steps that can be performed on the tape of a Turing machine.

For each $\mathcal{H} \in \mathcal{FU}(V)$, $m \in \mathcal{I}(\mathcal{H})$, and $v \in V$, we have $(f.m \mathbin{;} !\mathsf{t} \mathbin{;} !\mathsf{f}) \downarrow f.\mathcal{H}(v)$. This leads us to the following corollary of Theorem 1.

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

9 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 $M \in \mathcal{MO}(V)$. Then we say that M *increases the number of colons* if for some $v \in V$ the number of colons in $M^e(v)$ is greater than the number of colons in v.

Theorem 2. *Let $\mathcal{H} \in \mathcal{FU}(V)$ be such that no method operation of $\mathcal{H}$ increases the number of colons. Then there exist an extension $\mathcal{H}'$ of $\mathcal{H}$, an $I' \subseteq \mathcal{I}(\mathcal{H}')$, and an $x \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}'))$ such that x produces a reflexive solution of the halting problem for $\mathcal{L}(f.I')$ with respect to $\mathcal{H}'$.*

Proof. Let $\mathsf{halting} \in \mathcal{M}$ be such that $\mathsf{halting} \notin \mathcal{I}(\mathcal{H})$. Take $I' = \mathcal{I}(\mathcal{H}) \cup \{\mathsf{halting}\}$. Take $\mathcal{H}' = \mathcal{H} \cup \{(\mathsf{halting}, \mathit{Halting})\}$, where $\mathit{Halting} \in \mathcal{MO}(V)$ is defined by induction on the number of colons in the argument of *Halting* as follows:

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

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

Theorem 2 tells us that there exist functional units $\mathcal{H} \in \mathcal{FU}(V)$ with the property that the halting problem is potentially autosolvable for $\mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ with respect to $\mathcal{H}$. Thus, we know that there exist functional units $\mathcal{H} \in \mathcal{FU}(V)$ with the property that the halting problem is autosolvable for $\mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ with respect to $\mathcal{H}$.

There exists an $\mathcal{H} \in \mathcal{FU}(V)$ for which *Halting* as defined in the proof of Theorem 2 is computable, and hence recursive.

Theorem 3. *Let $\mathcal{H} = \emptyset$ and $\mathcal{H}' = \mathcal{H} \cup \{(\mathsf{halting}, \mathit{Halting})\}$, where Halting is as defined in the proof of Theorem 2. Then, Halting is computable.*

Proof. It is sufficient to prove for an arbitrary $x \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}'))$ that, for all $v \in V$, $x \downarrow f.\mathcal{H}'(v)$ 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 $x \downarrow f.\mathcal{H}'(v) \Leftrightarrow x' \downarrow \emptyset$, where x' is x with each occurrence of $f.\mathsf{halting}$ and $+f.\mathsf{halting}$ replaced by #2 and each occurrence of $-f.\mathsf{halting}$ replaced by #1. Because x' is finite, $x' \downarrow \emptyset$ is decidable. Hence, $x \downarrow f.\mathcal{H}'(v)$ 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 $x \downarrow f.\mathcal{H}'(v) \Leftrightarrow x' \downarrow \emptyset$, 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 its position 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, $x \downarrow f.\mathcal{H}'(v)$ is decidable. □

Theorems 2 and 3 together tell us that there exists a functional unit $\mathcal{H} \in \mathcal{FU}(V)$, viz. $\emptyset$, with the property that the halting problem is potentially recursively autosolvable for $\mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ with respect to $\mathcal{H}$.

There exist functional units in $\mathcal{FU}(V)$ of which all recursive $M \in \mathcal{MO}(V)$ that do not increase the number of colons are derived method operations. A witness is the functional unit whose method operations correspond to the basic steps that can be performed on the tape of a Turing machine. Let $\mathcal{H} \in \mathcal{FU}(V)$ be such that all recursive $M \in \mathcal{MO}(V)$ that do not increase the number of colons are derived method operations of $\mathcal{H}$. Then the halting problem is potentially autosolvable for $\mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ with respect to $\mathcal{H}$. However, the halting problem is not potentially recursively autosolvable for $\mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ with respect to $\mathcal{H}$ because otherwise the halting problem would 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 4. *Let $\mathcal{H} \in \mathcal{FU}(V)$ be such that $(\mathsf{dup}, Dup) \in \mathcal{H}$, and let $I \subseteq \mathcal{I}(\mathcal{H})$ be such that $\mathsf{dup} \in I$. Then there does not exist an $x \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ such that x produces a reflexive solution of the halting problem for $\mathcal{L}(f.I)$ with respect to $\mathcal{H}$.*

Proof. Assume the contrary. Let $x \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ be such that x produces a reflexive solution of the halting problem for $\mathcal{L}(f.I)$ with respect to $\mathcal{H}$, and let $y = f.\mathsf{dup}\,; \mathit{f2d}(\mathit{swap}(x))$. Then $x \downarrow f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y})$. By Propositions 1 and 2, it follows that $\mathit{swap}(x) \downarrow f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y})$ and either $\mathit{swap}(x) \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y}) = \mathsf{t}$ or $\mathit{swap}(x) \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y}) = \mathsf{f}$.

In the case where $\mathit{swap}(x) \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y}) = \mathsf{t}$, we have by Proposition 2 that (i) $\mathit{f2d}(\mathit{swap}(x)) \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y}) = \mathsf{t}$ and (ii) $x \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y}) = \mathsf{f}$. By Proposition 3, it follows from (i) that $(f.\mathsf{dup}\,; \mathit{f2d}(\mathit{swap}(x))) \,!\, f.\mathcal{H}({}^\frown\overline{y}) = \mathsf{t}$. Since $y = f.\mathsf{dup}\,; \mathit{f2d}(\mathit{swap}(x))$, we have $y \,!\, f.\mathcal{H}({}^\frown\overline{y}) = \mathsf{t}$. On the other hand, because x produces a reflexive solution, it follows from (ii) that $y \uparrow f.\mathcal{H}({}^\frown\overline{y})$. By Proposition 1, this contradicts with $y \,!\, f.\mathcal{H}({}^\frown\overline{y}) = \mathsf{t}$.

In the case where $\mathit{swap}(x) \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y}) = \mathsf{f}$, we have by Proposition 2 that (i) $\mathit{f2d}(\mathit{swap}(x)) \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y}) = \mathsf{d}$ and (ii) $x \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y}) = \mathsf{t}$. By Proposition 3, it follows from (i) that $(f.\mathsf{dup}\,; \mathit{f2d}(\mathit{swap}(x))) \,!\, f.\mathcal{H}({}^\frown\overline{y}) = \mathsf{d}$. Since $y = f.\mathsf{dup}\,; \mathit{f2d}(\mathit{swap}(x))$, we have $y \,!\, f.\mathcal{H}({}^\frown\overline{y}) = \mathsf{d}$. On the other hand, because x produces a reflexive solution, it follows from (ii) that $y \downarrow f.\mathcal{H}({}^\frown\overline{y})$. By Proposition 1, this contradicts with $y \,!\, f.\mathcal{H}({}^\frown\overline{y}) = \mathsf{d}$. □

Below, we will give an alternative proof of Theorem 4. 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 2. *Let $\mathcal{H} \in \mathcal{FU}(V)$, let $I \subseteq \mathcal{I}(\mathcal{H})$, let $x \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ be such that x produces a reflexive solution of the halting problem for $\mathcal{L}(f.I)$ with respect to $\mathcal{H}$, let $y \in \mathcal{L}(f.I)$, and let $v \in \{0, 1, :\}^*$. Then $y \downarrow f.\mathcal{H}({}^\frown v)$ implies $y \,!\, f.\mathcal{H}({}^\frown v) = x \,!\, f.\mathcal{H}({}^\frown\overline{\mathit{f2d}(y)}{:}v)$.*

Proof. By Proposition 1, it follows from $y \downarrow f.\mathcal{H}({}^\frown v)$ that either $y \,!\, f.\mathcal{H}({}^\frown v) = \mathsf{t}$ or $y \,!\, f.\mathcal{H}({}^\frown v) = \mathsf{f}$.

In the case where $y \,!\, f.\mathcal{H}({}^\frown v) = \mathsf{t}$, we have by Propositions 1 and 2 that $\mathit{f2d}(y) \downarrow f.\mathcal{H}({}^\frown v)$ and so $x \,!\, f.\mathcal{H}({}^\frown\overline{\mathit{f2d}(y)}{:}v) = \mathsf{t}$.

In the case where $y \,!\, f.\mathcal{H}({}^\frown v) = \mathsf{f}$, we have by Propositions 1 and 2 that $\mathit{f2d}(y) \uparrow f.\mathcal{H}({}^\frown v)$ and so $x \,!\, f.\mathcal{H}({}^\frown\overline{\mathit{f2d}(y)}{:}v) = \mathsf{f}$. □

Another proof of Theorem 4. Assume the contrary. Let $x \in \mathcal{L}(f.\mathcal{I}(\mathcal{H}))$ be such that x produces a reflexive solution of the halting problem for $\mathcal{L}(f.I)$ with respect to $\mathcal{H}$, and let $y = \mathit{f2d}(\mathit{swap}(f.\mathsf{dup}\,; x))$. Then $x \downarrow f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y})$. By Propositions 1, 2 and 3, it follows that $\mathit{swap}(f.\mathsf{dup}\,; x) \downarrow f.\mathcal{H}({}^\frown\overline{y})$. By Lemma 2, it follows that $\mathit{swap}(f.\mathsf{dup}\,; x) \,!\, f.\mathcal{H}({}^\frown\overline{y}) = x \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y})$. By Proposition 2, it follows that $(f.\mathsf{dup}\,; x) \,!\, f.\mathcal{H}({}^\frown\overline{y}) \neq x \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y})$. On the other hand, by Proposition 3, we have that $(f.\mathsf{dup}\,; x) \,!\, f.\mathcal{H}({}^\frown\overline{y}) = x \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y})$. This contradicts with $(f.\mathsf{dup}\,; x) \,!\, f.\mathcal{H}({}^\frown\overline{y}) \neq x \,!\, f.\mathcal{H}({}^\frown\overline{y}{:}\overline{y})$. □

Let $\mathcal{H} = \{(\mathsf{dup}, \mathit{Dup})\}$. By Theorem 4, the halting problem for $\mathcal{L}(f.\{\mathsf{dup}\})$ with respect to $\mathcal{H}$ is not (potentially) autosolvable. However, it is decidable.

Theorem 5. *Let $\mathcal{H} = \{(\mathsf{dup}, \mathit{Dup})\}$. Then the halting problem for $\mathcal{L}(f.\{\mathsf{dup}\})$ with respect to $\mathcal{H}$ is decidable.*

Proof. Let $x \in \mathcal{L}(f.\{\mathsf{dup}\})$, 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 $v \in V$, $\mathit{Dup}^r(v) = \mathsf{t}$. Therefore, $x \downarrow f.\mathcal{H}(v) \Leftrightarrow x' \downarrow \emptyset$ for all $v \in V$. Because x' is finite, $x' \downarrow \emptyset$ is decidable. □

Both proofs of Theorem 4 given above are diagonalization proofs in disguise. Theorem 5 indicates that diagonalization and decidability are independent so to speak.

10 Concluding Remarks

We have extended and strengthened the results regarding the halting problem for programs given in [6] 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 [6]).

An interesting option for future work is to investigate the bounded halting problem for programs, i.e. the problem to determine, given a program and an input to the program, whether execution of the program on that input will terminate after the execution of no more than a fixed number of basic instructions.

References

1. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming **51**(2), 125–156 (2002)
2. Bergstra, J.A., Middelburg, C.A.: A thread algebra with multi-level strategic interleaving. Theory of Computing Systems **41**(1), 3–32 (2007)
3. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. Journal of Applied Logic **6**(4), 553–563 (2008)
4. Bergstra, J.A., Middelburg, C.A.: Functional units for natural numbers. Electronic Report PRG0913, Programming Research Group, University of Amsterdam (2009). Available from `http://www.science.uva.nl/research/prog/publications.html`. Also available from `http://arxiv.org/`: `arXiv:0911.1851v1 [cs.PL]`
5. Bergstra, J.A., Middelburg, C.A.: Instruction sequence processing operators. Electronic Report PRG0912, Programming Research Group, University of Amsterdam (2009). Available from `http://www.science.uva.nl/research/prog/publications.html`. Also available from `http://arxiv.org/`: `arXiv:0910.5564v2 [cs.LO]`

6. Bergstra, J.A., Ponse, A.: Execution architectures for program algebra. Journal of Applied Logic **5**(1), 170–192 (2007)
7. Sannella, D., Tarlecki, A.: Algebraic preliminaries. In: E. Astesiano, H.J. Kreowski, B. Krieg-Brückner (eds.) Algebraic Foundations of Systems Specification, pp. 13–30. Springer-Verlag, Berlin (1999)
8. Turing, A.M.: On computable numbers, with an application to the Entscheidungs problem. Proceedings of the London Mathematical Society, Series 2 **42**, 230–265 (1937). Correction: *ibid*, 43:544–546, 1937
9. Wirsing, M.: Algebraic specification. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 675–788. Elsevier, Amsterdam (1990)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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