Instruction sequence processing operators

Bergstra, J.A.; Middelburg, C.A.

Publication date
2009

Document Version
Submitted manuscript

Instruction Sequence Processing Operators

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. This paper concerns instruction sequences whose execution involves the processing of instructions by an execution environment that offers a family of services and may yield a Boolean value at termination. We introduce a composition operator for families of services and three operators that have a direct bearing on the processing in question. Together they are simpler and more powerful than the operators proposed for the same purpose in earlier work. Some of the operators allow for terms to be built that are not intended to denote anything. We propose to comply with conventions that exclude the use of such terms.

Keywords: instruction sequence processing, service family, relevant use convention.

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

1 Introduction

In this paper, we take the view that the execution of an instruction sequence involves the processing of instructions by an execution environment that offers a family of services and may yield a Boolean value at termination. We introduce a composition operator for families of services and three operators that have a direct bearing on the processing in question. Although all terms that can be build by means of these operators denote something, some of them are not really intended to denote anything. Therefore, we propose to comply with conventions that exclude the use of such terms.

A slightly different view on the execution of an instruction sequence was taken in [6]. This resulted in different operators: a composition operator for services instead of service families and two kinds of operators, called use operators and apply operators, that have a direct bearing on the processing of instructions but do not cover the possibility that some value is yielded at termination. In subsequent work, the composition operator for services was not used. Moreover, we experienced recently the lack of a way to deal with the possibility that some value is yielded at termination of the execution of an instruction sequence. This state of affairs forms the greater part of our motivation for the work presented in this paper.

One of the operators introduced in the current paper removes the above-mentioned lack. We illustrate the use of that operator, as well as the use the

composition operator for service families, by means of a simple example. The other operators are adaptations of the use and apply operators introduced in [6] to service families.

The work presented in this paper belongs to a line of research whose working hypothesis is that instruction sequence is a central notion of computer science. In this line of research, program algebra [2] is the setting used for investigating issues concerning instruction sequences. The starting-point of program algebra is the perception of a program as a single-pass instruction sequence, i.e. a finite or infinite sequence of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over. This perception is simple, appealing, and links up with practice. Moreover, basic thread algebra [2] is the setting used for modelling the behaviours exhibited by instruction sequences under execution.[1]

The three operators that are related to the processing of instructions will be introduced as operators extending basic thread algebra. Because these operators are primarily intended to be used in the setting of program algebra 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 review program algebra and basic thread algebra (Sections 2 and 3). Next, we introduce service families, the composition operator for service families, and the three operators that are related to the processing of instructions by service families (Sections 4 and 5). Then, we propose to comply with conventions that exclude the use of terms that are not really intended to denote anything (Sections 6). After that, we give an example related to the processing of instructions by service families (Section 7). We also present an interesting variant of one of the above-mentioned operators related to the processing of instructions (Section 8). Finally, we make some concluding remarks (Section 9).

2 Program Algebra

In this section, we review PGA (ProGram Algebra).

In PGA, it is assumed that a fixed but arbitrary 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.

PGA has the following *primitive instructions*:

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

[1] In [2], basic thread algebra is introduced under the name basic polarized process algebra.

Table 1. Axioms of PGA

$(X\,;Y)\,;Z = X\,;(Y\,;Z)$	PGA1
$(X^n)^\omega = X^\omega$	PGA2
$X^\omega\,;Y = X^\omega$	PGA3
$(X\,;Y)^\omega = X\,;(Y\,;X)^\omega$	PGA4

We write $\mathfrak{I}_{\mathrm{PGA}}$ for the set of all primitive instructions of PGA. On execution of an 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 $\mathtt{t}$ 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 $\mathtt{t}$ is produced;
- the effect of a forward jump instruction $\#l$ is that execution proceeds with the l-th next instruction of the program concerned — if l equals 0 or there is no primitive instructions to proceed with, deadlock occurs;
- the effect of the termination instruction ! is that execution terminates.

PGA has the following constants and operators:

- for each $u \in \mathfrak{I}_{\mathrm{PGA}}$, an *instruction* constant u;
- the binary *concatenation* operator $\_\,;\_$;
- the unary *repetition* operator $\_^\omega$.

We assume that there is a countably infinite set of variables which includes X, Y, Z. Terms are built as usual. We use infix notation for concatenation and postfix notation for repetition.

A closed PGA term is considered to denote a non-empty, finite or eventually periodic infinite sequence of primitive instructions.[2] Closed PGA terms are considered equal if they represent the same instruction sequence. The axioms for instruction sequence equivalence are given in Table 1. In this table, n stands for an arbitrary positive natural number. The term X^n is defined by induction on n as follows: $X^1 = X$ and $X^{n+1} = X\,;X^n$. The *unfolding* equation $X^\omega = X\,;X^\omega$ is derivable. Each closed PGA term is derivably equal to a term in *canonical form*, i.e. a term of the form P or $P\,;Q^\omega$, where P and Q are closed PGA terms in which the repetition operator does not occur.

The members of the domain of an initial model of PGA are called *instruction sequences*. This is justified by the fact that any initial model of PGA is isomorphic to the initial model in which:

[2] An eventually periodic infinite sequence is an infinite sequence with only finitely many distinct suffixes.

– the domain is the set of all finite and eventually periodic infinite sequences over the set $\mathfrak{I}_{\mathrm{PGA}}$ of primitive instructions;
– the operation associated with ; is concatenation;
– the operation associated with $^\omega$ is the operation $^{\underline{\omega}}$ defined as follows:
  • if s is a finite sequence over $\mathfrak{I}_{\mathrm{PGA}}$, then $s^{\underline{\omega}}$ is the unique eventually periodic infinite sequence t such that s concatenated n times with itself is a proper prefix of t for each $n \in \mathbb{N}$;
  • if s is an eventually periodic infinite sequence over $\mathfrak{I}_{\mathrm{PGA}}$, then $s^{\underline{\omega}}$ is s.

We experienced recently the lack of a way to deal with the possibility that some value is yielded at termination of the execution of an instruction sequence. This lack is particularly felt with instruction sequences that implement some test. To remedy this lack, we extend PGA with Boolean termination instructions, resulting in $\mathrm{PGA}_{\mathrm{bt}}$.

In $\mathrm{PGA}_{\mathrm{bt}}$, like in PGA, it is assumed that there is a fixed but arbitrary set $\mathfrak{A}$ of basic instructions. $\mathrm{PGA}_{\mathrm{bt}}$ has the primitive instructions of PGA and in addition:

– a *positive termination instruction* $!\mathsf{t}$;
– a *negative termination instruction* $!\mathsf{f}$.

We write $\mathfrak{I}_{\mathrm{PGA}_{\mathrm{bt}}}$ for the set of all primitive instructions of $\mathrm{PGA}_{\mathrm{bt}}$. The effect of the Boolean termination instructions $!\mathsf{t}$ and $!\mathsf{f}$ is that execution terminates and in doing so delivers the Boolean value t and f, respectively.

$\mathrm{PGA}_{\mathrm{bt}}$ has an instruction constant u for each $u \in \mathfrak{I}_{\mathrm{PGA}_{\mathrm{bt}}}$ (instead of $\mathfrak{I}_{\mathrm{PGA}}$). $\mathrm{PGA}_{\mathrm{bt}}$ has the same operators and axioms as PGA.

In Section 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 $\mathop{;}_{i=1}^{n} P_i$, where $P_1, \ldots, P_n$ are $\mathrm{PGA}_{\mathrm{bt}}$ terms, for the term $P_1; \ldots; P_n$.

3 Thread Extraction

In this section, we make precise in the setting of BTA (Basic Thread Algebra) which behaviours are exhibited on execution by the instruction sequences denoted by closed $\mathrm{PGA}_{\mathrm{bt}}$ terms. We start by reviewing BTA.

In BTA, it is assumed that a fixed but arbitrary 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 has one sort: the sort $\mathbf{T}$ of *threads*. We make this sort explicit because we will extend BTA with additional sorts in Section 5. To build terms of sort $\mathbf{T}$, BTA has the following constants and operators:

Table 2. Axiom of BTA

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

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

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 threads denoted by D and S will become inactive and terminate, respectively.

BTA has only one axiom. This axiom is given in Table 2. Using the abbreviation introduced above, axiom T1 can be written as follows: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

Each closed BTA term of sort $\mathbf{T}$ denotes a 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 BTA is a set of recursion equations $E = \{x = t_x \mid x \in V\}$, where V is a set of variables of sort $\mathbf{T}$ and each t_x is a BTA term of the form D, S or $t \trianglelefteq a \trianglerighteq t'$ with t and t' that contain only variables from V. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in [1]. 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 3. In this table, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$ and n stands for an arbitrary natural number.

In Section 2, we extended PGA with constants for Boolean termination instructions. Accordingly, we extend BTA with Boolean termination constants, resulting in $\mathrm{BTA}_{\mathrm{bt}}$. Like in BTA, it is assumed that there is a fixed but arbitrary set $\mathcal{A}$ of basic actions. $\mathrm{BTA}_{\mathrm{bt}}$ has the constants and operators of BTA and in addition the following constants:

- the *positive termination* constant $\mathsf{S}+$;
- the *negative termination* constant $\mathsf{S}-$.

Table 3. 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
$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y) = \pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	P3

Table 4. Defining equations for thread extraction operation

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

$\mathrm{BTA}_{\mathrm{bt}}$ has the same axioms as BTA. The notion of a guarded recursive specification over $\mathrm{BTA}_{\mathrm{bt}}$ is defined like in the case of BTA.

The behaviours exhibited on execution by the instruction sequences denoted by closed $\mathrm{PGA}_{\mathrm{bt}}$ terms are considered to be regular threads, with the basic instructions taken for basic actions. The *thread extraction* operation $\lvert\_\rvert$ defines, for each closed $\mathrm{PGA}_{\mathrm{bt}}$ term, the behaviour exhibited on execution by the instruction sequence denoted by that term. The thread extraction operation is defined by the equations given in Table 4 (for basic instructions $a \in \mathfrak{A}$, natural numbers $l \in \mathbb{N}$, and primitive instructions $i \in \mathfrak{I}_{\mathrm{PGA}_{\mathrm{bt}}}$) and the rule that $\lvert \#l\,;X\rvert = \mathsf{D}$ if $\#l$ is the beginning of an infinite jump chain. This rule is formalized in e.g. [4].

4 Services and Service Families

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

It is assumed that a fixed but arbitrary 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 $\{\mathsf{t}, \mathsf{f}, \mathsf{d}\}$.

A *service* $\mathcal{H}$ consists of

- a set S of *states*;
- an *effect* function $\mathit{eff} : \mathcal{M} \times S \to S$;

– a *yield* function $yld : \mathcal{M} \times S \to \mathcal{R}$;
– an *initial state* $s_0 \in S$;

satisfying the following condition:

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

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

Given a service $\mathcal{H} = (S, \mathit{eff}, yld, s_0)$ and a method $m \in \mathcal{M}$:

– $\frac{\partial}{\partial m}\mathcal{H}$, the *derived service of* $\mathcal{H}$ *after processing* m, is defined by

$$\frac{\partial}{\partial m}\mathcal{H} = (S, \mathit{eff}, yld, \mathit{eff}(m, s_0)) \ ;$$

– $\mathcal{H}(m)$, the *reply of* $\mathcal{H}$ *after processing* m, is defined by

$$\mathcal{H}(m) = yld(m, s_0) \ .$$

When a request is made to service $\mathcal{H}$ to process method m:

– if $\mathcal{H}(m) \neq \mathsf{d}$, then the service processes m, produces the reply $\mathcal{H}(m)$, and next proceeds as $\frac{\partial}{\partial m}\mathcal{H}$;
– if $\mathcal{H}(m) = \mathsf{d}$, then the service rejects the request and proceeds as a service that rejects any request to process a method.

A service $\mathcal{H}$ is an *empty* service if $\mathcal{H}(m) = \mathsf{d}$ for all $m \in \mathcal{M}$. In other words, an empty service is a service that is unable to process any method. For each empty service $\mathcal{H}$, $\frac{\partial}{\partial m}\mathcal{H} = \mathcal{H}$ and $\mathcal{H}(m) = \mathsf{d}$ for all $m \in \mathcal{M}$. For that reason, all empty services are identified.

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

– a set $\mathcal{S}$ of services has been given that is closed under $\frac{\partial}{\partial m}$ for each $m \in \mathcal{M}$;
– 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 empty service;
  • if closed term t denotes service $\mathcal{H}$, then $\frac{\partial}{\partial m}t$ denotes service $\frac{\partial}{\partial m}\mathcal{H}$.

Moreover, it is assumed that a fixed but arbitrary 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:

Table 5. 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 sort **SF** of *service families*;

and the following constant and operators:

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

We assume that there are countably infinite many variables of sort **SF**, including u, v, w. Terms are built as usual in the many-sorted case (see e.g. [9, 12]). We use prefix notation for the singleton service family operators and infix notation for the service family composition operator.

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

The service family composition operator takes the place of the non-interfering combination operator from [6]. As suggested by the name, service family composition is composition of service families. Non-interfering combination is composition of services, which has the disadvantage that its usefulness is rather limited without an additional renaming mechanism.

The axioms of SF are given in Table 5. In this table, f stands for an arbitrary focus from $\mathcal{F}$ and H and H' stand for arbitrary closed terms of sort **S**. The axioms of SF simply formalize the informal explanation given above.

In Section 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 make use of the named services from the service family offered by an execution environment. That is, a thread may perform an 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 $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$.

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 reply operator does not only produce Boolean values: it produces special values in cases where no Boolean value is delivered at termination or no termination takes place.

For the set $\mathcal{A}$ of basic actions, we take the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. 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 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 basic action turns into the internal action tau and the two ways to proceed reduces 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 m (standing for meaningless) if it terminates and does not deliver a Boolean value at termination, and the value d (standing for divergent) if it does not terminate.

Table 6. Axioms for use operator

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

Table 7. Axioms for 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
$(x \trianglelefteq f.m \trianglerighteq y) \bullet \partial_{\{f\}}(u) = \emptyset$		A6
$(x \trianglelefteq f.m \trianglerighteq y) \bullet (f.H \oplus \partial_{\{f\}}(u)) = x \bullet (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$	if $H(m) = \mathsf{t}$	A7
$(x \trianglelefteq f.m \trianglerighteq y) \bullet (f.H \oplus \partial_{\{f\}}(u)) = y \bullet (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$	if $H(m) = \mathsf{f}$	A8
$(x \trianglelefteq f.m \trianglerighteq y) \bullet (f.H \oplus \partial_{\{f\}}(u)) = \emptyset$	if $H(m) = \mathsf{d}$	A9
$\bigwedge_{n \geq 0} \pi_n(x) \bullet u = \pi_n(y) \bullet v \Rightarrow x \bullet u = y \bullet v$		A10

Table 8. Axioms for reply operator

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

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 6, 7 and 8. In these tables, f stands for an arbitrary focus from $\mathcal{F}$, m stands for an arbitrary method from $\mathcal{M}$, H stands for an arbitrary term of sort $\mathbf{S}$, and n stands for an arbitrary natural number. The axioms simply formalize the informal explanation given above and in addition stipulate what is the result of 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.

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

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 1. *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 [3] as part of the proof of Theorem 3 from that paper. The proof of the general case goes essentially the same as the proof concerned. □

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 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 $\mathfrak{A}$ of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ in which guarded recursive specifications have unique solutions is *definable* if it is representable by a closed $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ term or it is the solution in $\mathfrak{A}$ of a guarded recursive specification over $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$.

Below, we will formulate a proposition about the use, apply and reply operators which we will prove using the following lemma about guarded recursive specifications and projection.

Lemma 2. *Let* E *be a guarded recursive specification over* $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$*, and let* x *be a variable occurring in* E*. Then, for all* $n \in \mathbb{N}$*, there exists a closed* $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ *term* p *of sort* $\mathbf{T}$ *such that* $E \Rightarrow \pi_n(x) = p$*.*

Proof. In the case of BTA, this is proved in [3] as part of the proof of Theorem 1 from that paper. By the definition of guarded recursive specification over $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$, the proof concerned goes through in the case of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$. □

The proposition about the use, apply and reply operators is formulated using the *foci* operation foci defined by the equations in Table 9 (for foci $f \in \mathcal{F}$ 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 only make use of the following properties of foci in the proof of the proposition:

Table 9. Defining equations for foci operation

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

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

This means that the proposition could be formulated without using the foci operation, but that would make it less intelligible.

Proposition 1. *If x is a definable thread and $\mathsf{foci}(u) \cap \mathsf{foci}(v) = \emptyset$, 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 Lemmas 1 and 2, axioms AIP, U10, A10 and R10, and the definition of definable thread, it is sufficient to prove that the following equations are derivable for each 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 easy 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 $H(m) = \mathsf{t}$ and $x \downarrow (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$, then $(x \trianglelefteq f.m \trianglerighteq y) \downarrow (f.H \oplus \partial_{\{f\}}(u))$;
5. if $H(m) = \mathsf{f}$ and $y \downarrow (f.\frac{\partial}{\partial m}H \oplus \partial_{\{f\}}(u))$, then $(x \trianglelefteq f.m \trianglerighteq y) \downarrow (f.H \oplus \partial_{\{f\}}(u))$;
6. if $\pi_n(x) \downarrow u$, then $x \downarrow u$.

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 $\downarrow_{\mathbb{B}}$.

Proposition 2. *Let p and C be closed $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$ terms of sort $\mathbf{T}$ and $\mathbf{SF}$, respectively, such that $p \downarrow C$. Then:*

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

Proof. By Lemma 1, it is sufficient to prove it for all closed $\mathrm{BTA}_{\mathrm{bt}}$ terms p of sort $\mathbf{T}$. This is easy by induction on the structure of p. $\square$

Because the use operator, apply operator and reply operator are primarily intended to be used in the setting of $\mathrm{PGA}_{\mathrm{bt}}$ to describe and analyse instruction sequence processing, they are called *instruction sequence processing operators*. We introduce the use operator, apply operator and reply operator in the setting $\mathrm{PGA}_{\mathrm{bt}}$ by defining:

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

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. Unavoidably there are cases in which processing does not halt or, even worse (because it is statically detectable), interfaces do not match. This means that there are cases in which there is nothing that we intend to denote by a term of the form $p \bullet C$, $p \mathbin{!} C$ or $C \oplus D$.

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

- $p \bullet C$ is only used if it is known that $p \downarrow C$;
- $p \mathbin{!} C$ 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 in the projective limit model of $\mathrm{TA}_{\mathrm{bt}}^{\mathrm{tsi}}$, for definable threads x, $x \bullet u = \emptyset$ if not $x \downarrow u$. We do not have $x \bullet u = \emptyset$ only if not $x \downarrow u$. For instance, $\mathsf{S}+ \bullet \emptyset = \emptyset$ whereas $\mathsf{S}+ \downarrow \emptyset$.

The idea of relevant use conventions is taken from [5], 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 [5], mathematicians deal with this issue by complying with the convention that $p \mathbin{/} q$ 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 $p \mathbin{/} q$ 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 [7], division is considered a total function whose value is zero in all cases of division by zero. On the analogy of material implication, which is true in all cases where the antecedent is false, we call this notion of division *material division*. It may be imagined that material division is the notion of division 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.

We can define the use and apply operators introduced earlier in [6], and similar counterparts of the reply operator, as follows:

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

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 there are cases in which the delivery of a Boolean value at termination of the execution of an instruction sequence is quite natural. We also show in this example that it is easy to compose a number of Boolean register services by means of the service family composition operation. Accomplishing this with the non-interfering service combination operation from [6] is quite involved.

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* $\mathsf{set{:}t}$;
- the *set to false method* $\mathsf{set{:}f}$;
- the *get method* get.

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

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

- $\mathsf{set{:}t}$: the contents of the Boolean register becomes t and the reply is t;
- $\mathsf{set{:}f}$: 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.

Let $s \in \{\mathsf{t}, \mathsf{f}, \mathsf{d}\}$. Then the *Boolean register service* with initial state s, written BR_s, is the service $(\{\mathsf{t}, \mathsf{f}, \mathsf{d}\}, \mathit{eff}, \mathit{eff}, s)$, where the function eff is defined as follows ($b \in \{\mathsf{t}, \mathsf{f}\}$):

$$\mathit{eff}(\mathsf{set{:}t}, b) = \mathsf{t} \;, \qquad \mathit{eff}(m, b) = \mathsf{d} \;\text{ if } m \notin \{\mathsf{set{:}t}, \mathsf{set{:}f}, \mathsf{get}\} \;,$$
$$\mathit{eff}(\mathsf{set{:}f}, b) = \mathsf{f} \;, \qquad \mathit{eff}(m, \mathsf{d}) = \mathsf{d} \;.$$
$$\mathit{eff}(\mathsf{get}, b) = b \;,$$

Notice that the effect and yield functions of a Boolean register service are the same. For the set $\mathcal{S}$ of services, we take the set $\{BR_{\mathsf{t}}, BR_{\mathsf{f}}, BR_{\mathsf{d}}\}$. Moreover, we take the names used above to denote the services in $\mathcal{S}$ for constants of sort $\mathbf{S}$.

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 binary using a collection of n Boolean registers named $\mathsf{b}{:}0, \ldots, \mathsf{b}{:}n{-}1$. We take t for 0 and f for 1, and we take the bit represented by the content of the Boolean register named $\mathsf{b}{:}i$ for a less significant bit than the bit represented by the content of the Boolean register named $\mathsf{b}{:}j$ if $i < j$.

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

$$\begin{aligned}
\mathit{Up} &= \mathop{;}_{i=0}^{n-1}(-\mathsf{b}{:}i.\mathsf{get}\,;\,\#3\,;\,\mathsf{b}{:}i.\mathsf{set{:}f}\,;\,!\mathsf{t}\,;\,\mathsf{b}{:}i.\mathsf{set{:}t})\,;\,!\mathsf{f}\;,\\
\mathit{Down} &= \mathop{;}_{i=0}^{n-1}(+\mathsf{b}{:}i.\mathsf{get}\,;\,\#3\,;\,\mathsf{b}{:}i.\mathsf{set{:}t}\,;\,!\mathsf{t}\,;\,\mathsf{b}{:}i.\mathsf{set{:}f})\,;\,!\mathsf{f}\;,\\
\mathit{Zero} &= \mathop{;}_{i=0}^{n-1}(-\mathsf{b}{:}i.\mathsf{get}\,;\,!\mathsf{f})\,;\,!\mathsf{t}\;.
\end{aligned}$$

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

$$\begin{aligned}
\mathit{Up}\,!\,\bigl(\textstyle\bigoplus_{i=0}^{n-1}\mathsf{b}{:}i.BR_{s_i}\bigr) &= \begin{cases}\mathsf{t} \;\text{ if }\; \bigvee_{i=0}^{n-1} s_i = \mathsf{t}\\ \mathsf{f} \;\text{ if }\; \bigwedge_{i=0}^{n-1} s_i = \mathsf{f}\;,\end{cases}\\
\mathit{Down}\,!\,\bigl(\textstyle\bigoplus_{i=0}^{n-1}\mathsf{b}{:}i.BR_{s_i}\bigr) &= \begin{cases}\mathsf{t} \;\text{ if }\; \bigvee_{i=0}^{n-1} s_i = \mathsf{f}\\ \mathsf{f} \;\text{ if }\; \bigwedge_{i=0}^{n-1} s_i = \mathsf{t}\;,\end{cases}\\
\mathit{Zero}\,!\,\bigl(\textstyle\bigoplus_{i=0}^{n-1}\mathsf{b}{:}i.BR_{s_i}\bigr) &= \begin{cases}\mathsf{t} \;\text{ if }\; \bigwedge_{i=0}^{n-1} s_i = \mathsf{t}\\ \mathsf{f} \;\text{ if }\; \bigvee_{i=0}^{n-1} s_i = \mathsf{f}\;.\end{cases}
\end{aligned}$$

It is obvious that t is delivered at termination of an execution of *Zero* if the content of the counter is zero and that f is delivered otherwise. Increment by one and decrement by one are both modulo 2^n. Therefore, t is delivered at termination of an execution of *Up* or *Down* if the content of the counter is really incremented by one or decremented by one, and f is delivered otherwise.

8 Abstracting Use

With the use operator introduced in Section 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. [3]. However, with the use operators on services introduced in [6], 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 Section 5.

Table 10. Axioms for abstracting use operator

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

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.

The axioms for the abstracting use operator are given in Table 10. Owing to the possible concealment of actions by abstracting use, $\pi_n(x /\!\!/ u) = \pi_n(x) /\!\!/ u$ is not a plausible axiom. However, axiom AU10 allows for reasoning about infinite threads in the context of abstracting use.

9 Concluding Remarks

We have taken the view that the execution of an instruction sequence involves the processing of instructions by an execution environment that offers a service family and may yield a Boolean value at termination. We have proposed the service family composition operator, the use operator, the apply operator and the reply operator. The latter three operators are directly related to the processing in question. The apply operator fits in with the viewpoint that programs are state transformers that can be modelled by partial functions. This is the viewpoint taken in the early days of denotational semantics, see e.g. [10, 8, 11]. Pursuant to [5], we have also proposed to comply with conventions that exclude the use of terms that can be build by means of the proposed operators, but are not really intended to denote anything. The idea to comply with such conventions looks to be wider applicable in theoretical computer science.

References

1. Bergstra, J.A., Bethke, I.: Polarized process algebra and program equivalence. In: J.C.M. Baeten, J.K. Lenstra, J. Parrow, G.J. Woeginger (eds.) Proceedings 30th ICALP, *Lecture Notes in Computer Science*, vol. 2719, pp. 1–21. Springer-Verlag (2003)

2. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming **51**(2), 125–156 (2002)
3. Bergstra, J.A., Middelburg, C.A.: A thread algebra with multi-level strategic interleaving. Theory of Computing Systems **41**(1), 3–32 (2007)
4. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. Journal of Applied Logic **6**(4), 553–563 (2008)
5. Bergstra, J.A., Middelburg, C.A.: Partial Komori fields and imperative Komori fields. Electronic Report PRG0911, 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:0909.5271v1 [math.RA]`
6. Bergstra, J.A., Ponse, A.: Combining programs and state machines. Journal of Logic and Algebraic Programming **51**(2), 175–192 (2002)
7. Bergstra, J.A., Tucker, J.V.: The rational numbers as an abstract data type. Journal of the ACM **54**(2), Article 7 (2007)
8. Mosses, P.D.: The mathematical semantics of ALGOL 60. Tech. Rep. PRG-12, Programming Research Group, Oxford University (1974)
9. 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)
10. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Series in Computer Science. MIT Press, Cambridge, MA (1977)
11. Tennent, R.: 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)
12. Wirsing, M.: Algebraic specification. In: J. van Leeuwen (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 675–788. Elsevier, Amsterdam (1990)