ELSEVIER

The Journal of Logic and Algebraic Programming 51 (2002) 175–192

THE JOURNAL OF LOGIC AND ALGEBRAIC PROGRAMMING

www.elsevier.com/locate/jlap

Combining programs and state machines

Jan A. Bergstra[a,b,*], Alban Ponse[a,c]

[a] *Programming Research Group, Faculty of Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

[b] *Applied Logic Group, Department of Philosophy, Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands*

[c] *CWI, Department of Software Engineering, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*

Abstract

State machines consume and process actions complementary to programs issuing actions. State machines maintain a state and reply with a boolean response to each action in their interface. As state machines offer a service to programs, their interface is also called a service interface. State machines can be combined with several natural operators, thus giving rise to a state machine calculus. State machines are used for abstract data type modeling. © 2002 Published by Elsevier Science Inc.

Keywords: Program algebra; State machine; Abstract data type

1. Introduction

In this paper, so-called *state machines* are used to represent data structures and data types in a context focusing on their role within programs. Although programs are given the lead and data types are considered auxiliary in nature, the formalization of data types is still of major importance for program understanding, and data type modeling requires as much care as the analysis of programs and programming concepts. State machines admit several interesting operators, so there is an algebra of state machines as well as there is an algebra of programs. However, we here prefer to introduce a calculus of state machines. The reason for this choice is that identity is defined in terms of set-theoretic constructions rather than directly or indirectly by means of equational reasoning.

In this paper we focus on how state machines and programs (or, more precisely, program *behaviors*) can interact. We distinguish two main approaches: first, a state machine may support a program in its execution, for example as a memory device, yielding a (programmed) behavior. Secondly, a program may transform a state machine into another state machine. We describe programs and their behavior in the setting of program algebra (PGA, [4,5]).

* Corresponding author.
E-mail addresses: janb@science.uva.nl (J.A. Bergstra), alban@science.uva.nl (A. Ponse).

1567-8326/02/$ - see front matter © 2002 Published by Elsevier Science Inc.
PII: S1567-8326(02)00020-6

The paper is set up as follows: in Section 2 we provide an informal introduction to the operational intuition of state machines and programs. Subsequently, in Section 3 we provide some examples of state machines and discuss the calculus of state machines in some detail. In Section 4 we consider two ways of interfacing state machines and program execution. Then, in Section 5 we argue that state machines (or similar memory devices) are indispensable in programming technology. Finally, in Appendix A we briefly elaborate on specification techniques for state machines.

2. State machines, programs, and behaviors

In this section, we first introduce state machines. In order to model the interaction between state machines and programs or program behaviors, we discuss a particular type of primitive instructions and actions in the setting of program algebra. For general information on program algebra we refer to [4,5].

2.1. State machines

A state machine is a pair $\langle \Sigma, F \rangle$ consisting of an interface Σ and a reply function F. The interface of a state machine consists of a collection of so-called *co-actions* (where the prefix "co" emphasizes the cooperative or complementary nature). We use

$$\mathtt{a}(x)$$

as a general notation for co-actions. The bracket pair of a co-action can be empty as in `succ()`. Co-actions that will occur in examples to come are `set(true)`, `println(hello)` and `c1 : isZero()`. An example of an interface of a state machine is

$$\Sigma = \{\mathtt{a}(n), \mathtt{b}(n) \mid n < 1000\}.$$

The reply function F of a state machine $\langle \Sigma, F \rangle$ is a mapping that gives for each finite sequence of co-actions from the interface Σ the reply produced by the state machine. This reply is a boolean value `true` or `false` unless inaction occurs (symbolized by D). We write Σ^+ for the set of non-empty sequences over Σ, and use "," in the textual representation of sequences as a separator between Σ-elements. Furthermore, the reply function should satisfy the following requirement: if for some $\sigma \in \Sigma^+$, $F(\sigma) = \mathsf{D}$, then also $F(\sigma, \mathtt{a}(x)) = \mathsf{D}$ for all $\mathtt{a}(x) \in \Sigma$. It will turn out convenient to assume that $F(\mathtt{a}(x)) = \mathsf{D}$ in case $\mathtt{a}(x) \notin \Sigma$.

The operational intuition of state machines is as follows: a program can make use of the service offered by a state machine. These services are listed as co-actions in the service interface of that state machine, and the state machine is identifiable by its name. Each co-action in the interface of a state machine can be processed: the state machine updates its state and produces a reply in the form of a boolean value which is returned to the program that invoked the service. Furthermore, an error can occur, upon which the state machine replies with the error value D instead of an ordinary boolean. A state machine will not process any subsequent requests after having run into D. It is assumed that the program starts the operation of the state machine with its first request for a service. The state machine always starts from the same initial state.

The role of state machines. State machines emerge first of all as a tool for the description of data structures. Well-known data structures such as memory registers, stacks, queues and

tables can all be modeled as state machines.[1] Technically a state machine represents the behavior of an automaton. If one forgets the use of inaction (D), a state machine characterizes a formal language,[2] perhaps based on an unusual alphabet of symbols. State machines describe abstract data types in such a way that they can be used as part of an abstract machine. There is no implication that a state machine must be realized either in hardware or in software if it is used in a system description (however, it should allow immediate access to the co-actions in its service interface). Its role is restricted to specification. In many cases specifications involving data type state machines can be transformed into implementations in a fully automatic and an algorithmically satisfactory way.[3] For a set Σ, the collection of all state machines with service interface Σ can be considered the semantics of the service interface Σ. We will not discuss the set-theoretic size of this collection. It is quite large but only a fraction of its conceivable elements has some relevance for computer programming.

2.2. Programs

Programs are constructed using various types of primitive instructions, the most fundamental of which is the *void basic instruction*. In this paper, such instructions have more syntactical structure than in e.g. [4,5], in order to be able to provide a specific description of the interaction between the behavior of a program and a state machine. As a general notation for void basic instructions we use

$$\mathtt{f.a}(x)$$

where `f` is the so-called *focus* of the instruction, and $\mathtt{a}(x)$ is its *co-instruction* part (upon execution also called *co-action*; program behaviors are discussed in the next section). The bracket pair of an instruction can be empty or instantiated with some value. E.g., `smbr.set(true)` is an instruction that we will meet again in this paper, containing the focus `smbr` and the co-instruction `set(true)`. Another example of a primitive instruction is `Console.println(hello)`, having focus `Console` and co-instruction `println(hello)`. We write

$$a, b, \ldots$$

for void basic instructions whenever we do not care about their particular focus co-instruction structure.

A basic composition construct in the setting of program algebra is *concatenation*, written ";", and taken to be associative: if X and Y are programs (or 'program terms'), so is $X; Y$. Furthermore, if also Z is a program, then $(X; Y); Z$ and $X; (Y; Z)$ denote the same program, and brackets will not be used in repeated concatenations.

The primitive instructions and their operational intuitions considered in this paper are the following:

Void basic instruction. When executed, a void basic instruction generates a boolean value and the associated behavior may modify a state. After execution, a program has to

[1] In [1] state machines appear under the name of processes, a phrase that is currently used for a far more general kind of behavior. We refer to [2] for a discussion of abstract data type behavior motivating the role of state machines in this paper. An extensive literature on observability in data types relates to very similar notions.

[2] The language consisting of all sequences of co-actions for which the state machine produces the reply `false` (or `true`, depending on conventions regarding initialization and the empy string).

[3] The transformation used for that purpose, however, need not generate a system that has the same or virtually the same architecture as the given specification. Only the externally visible behavior must comply with the specification, all of its internal details in fact being private to the specification.

enact its subsequent instruction. If that instruction fails to exist, termination occurs. The attribute void expresses that execution is not influenced by the returned boolean value.

Termination instruction. The termination instruction ! yields termination of the program. It does not modify a state, and it does not return a boolean value.

Positive test instruction. Associated to each void basic instruction a there is a positive test instruction $+a$. When executed, the state is affected according to a, and in case `true` is returned, the remaining sequence of actions is performed. If there are no remaining instructions, termination occurs. In the case that `false` is returned, the next instruction is skipped and execution proceeds with the instruction following the skipped one. If no such instruction exists, termination occurs.

Negative test instruction. Associated to each void basic instruction a there is a negative test instruction $-a$. When executed, the state is affected according to a, and in case `false` is returned, the remaining sequence of actions is performed. If there are no remaining instructions, termination occurs. In the case that `true` is returned, the next instruction is skipped and execution proceeds with the instruction following the skipped one. If no such instruction exists, termination occurs.

As an example we introduce the basic program notation PGLE (see [5]), which is defined by adding labels and goto's to the instructions mentioned above, and employing concatenation as the only programming construct. The added instructions are these:

Label catch instruction. The label catch instruction has the form `L`k for k some natural number. Upon execution, this instruction is simply passed and cannot modify a state. If there is no subsequent instruction to be executed, termination occurs.

Absolute goto instruction. This instruction takes the form `##L`k for k some natural number, and represents a jump to the leftmost occurrence of the label catch instruction `L`k in the program. If there is no such instruction, termination occurs.

Furthermore, PGLE is characterized by the syntactic restriction that each test instruction in a program is followed by either ! or a `##L`k for some k.

Example 1. The program

```
ProgA = smbr.set(true); L0; +smbr.eq(true); ##L1; ##L2;
        L1; smbr.set(false); Console.println(hello); ##L0;
        L2; smbr.set(true); Console.println(goodbye); ##L0
```

is a particular example of a PGLE program that will be used in the sequel of this paper.

Programs as such do not have an interface, the difficulty being that it is not clear which superset of the collection of actions featuring in a program behavior should be considered the correct interface. Ideally an interface for a program combines two properties:

1. to include the actions mentioned in the program text, and
2. to allow easy communication.

Therefore an interface ought to have a low information content. The information content can become lower by adding 'redundant' elements to the interface. The minimal collection of instructions that are present in the text of a program[4] X is denoted with $\mathbf{\Sigma^{ps}}(X)$, the

[4] A reference to program objects is needed if all formal definitions are to be presented.

collection being called a *pseudo interface*. To see why $\mathbf{\Sigma}^{\mathbf{ps}}(X)$ is a pseudo interface rather than a proper one consider this example:

$$\mathtt{ProgB} = \mathtt{f.a}(124); +\mathtt{g.b}(358); \mathtt{L67}; -\mathtt{f.a}(98); \mathtt{f.a}(358).$$

Now $\mathbf{\Sigma}^{\mathbf{ps}}(\mathtt{ProgB}) = \{\mathtt{f.a}(98), \mathtt{f.a}(124), \mathtt{f.a}(358), \mathtt{g.b}(358)\}$ which is very implausible as an interface. A reasonable program interface might be:

$$\{\mathtt{f.a}(n), \mathtt{g.b}(n) | n < 1000\}.$$

A *program component*[5] is a pair consisting of a program interface and a program such that all basic instructions used by the program are in the interface. This definition is meaningful for all program notations in program algebra. Referring to the program ProgB as given above

$$[\{\mathtt{f.a}(n), \mathtt{g.b}(n) | n < 1000\}, \mathtt{ProgB}]$$

is an example of a program component. (Recall that a state machine has a definite interface, so 'state machine components' can be identified with state machines.)

2.3. Behaviors

Each program can be associated with a *program behavior*, and it is on the level of behaviors, i.e. program execution, that we shall study the interfacing with state machines. Below we provide a brief introduction to program behaviors (we refer to [5] for an extended overview). D is the behavior of an inactive program. The cause of inaction is either the absence of proper termination when needed or a cyclic succession of goto (or jump) instructions. S is the irreducible behavior of the termination instruction. Intuitively, a denotes the atomic behavior describing the execution of this basic instruction. However, we shall only consider atomic behavior in connection with the boolean reply it triggers: if P and Q are program behaviors and a is a boolean-returning action, then

$$P \trianglelefteq a \trianglerighteq Q$$

performs an a subsequently acting like P if a returns true and like Q if it does not. In the special case that $P = Q$, we write

$$a \circ P$$

instead of $P \trianglelefteq a \trianglerighteq P$. A finite program behavior always ends with S or D. Below we provide some examples of (finite and infinite) program behaviors.

In general, we write $|X|$ for the behavior of program X if we do not care in what program notation X is written. It is then simply assumed that $|X|$ yields a behavior as described above. In order to be able to describe some examples in detail, we define below the *behavior extraction* function $|\ldots|_{\mathrm{pgle}}$ on PGLE programs: $|X|_{\mathrm{pgle}}$ yields the behavior of PGLE program X. Let $X = u_1, \ldots, u_k$. Then $|X|_{\mathrm{pgle}} = |1, X|_g$, where the auxiliary function $|\_, \_|_g$ is defined as follows:

$$|j, X|_g = \mathsf{S} \quad \text{if } j > k \text{ or } j < 1 \text{ or } u_j = ! \text{ for } 1 \leqslant j \leqslant k,$$
$$|j, X|_g = a \circ |j + 1, X|_g \quad \text{if } u_j = a,$$

[5] We use 'program component' independently of the phrase 'software component'. There is no implication that a program component contains a program in binary form.

$$|j, X|_g = |j+1, X|_g \trianglelefteq a \trianglerighteq |j+2, X|_g \quad \text{if } u_j = +a,$$
$$|j, X|_g = |j+2, X|_g \trianglelefteq a \trianglerighteq |j+1, X|_g \quad \text{if } u_j = -a,$$
$$|j, X|_g = |j+1, X|_g \quad \text{if } u_j = \mathtt{L}n,$$
$$|j, X|_g = |m, X|_g \quad \text{if } u_j = \#\#\mathtt{L}n \text{ and } m = \mathtt{target}(n, X),$$

and the auxiliary function $\mathtt{target}(n, X)$ yields the smallest m such that $u_m = \mathtt{L}n$, and 0 if there is no such u_m in $X = u_1; ...; u_k$.

In the case that for some j these equations yield a cyclic succession without any atomic behavior, we set $|j, X|_g = \mathsf{D}$.

We provide some simple examples, followed by our 'running example':

$|a|_{\mathrm{pgle}} = a \circ \mathsf{S}$,
$|\mathtt{L0}; a; b; \#\#\mathtt{L0}|_{\mathrm{pgle}} = P$ where P can be defined by $P = a \circ (b \circ P)$,
$|+a; \mathtt{L0}; \#\#\mathtt{L0}; b|_{\mathrm{pgle}} = a \circ \mathsf{D}$.

Example 2. The PGLE program `ProgA` as defined in Example 1, i.e.

```
ProgA = smbr.set(true); L0; +smbr.eq(true); ##L1; ##L2;
        L1; smbr.set(false); Console.println(hello); ##L0;
        L2; smbr.set(true); Console.println(goodbye); ##L0
```

defines the behavior $|\mathtt{ProgA}| = Q$, where Q can be defined by the following equations:

$$Q = \mathtt{smbr.set(true)} \circ M,$$
$$M = L \circ M \trianglelefteq \mathtt{smbr.eq(true)} \trianglerighteq R \circ M,$$
$$L = \mathtt{smbr.set(false)} \circ \mathtt{Console.println(hello)},$$
$$R = \mathtt{smbr.set(true)} \circ \mathtt{Console.println(goodbye)}.$$

adopting the conventions that $\circ$ is right associative and binds stronger than $\_ \trianglelefteq a \trianglerighteq \_$.

The pseudo interface of a program behavior simply contains all actions that the behavior performs. We omit the precise definitions. For a behavior P we denote its pseudo interface with $\mathbf{\Sigma^{ps}}(P)$. If X is a PGLE program with behavior $|X|_{\mathrm{pgle}}$, then $\mathbf{\Sigma^{ps}}(|X|_{\mathrm{pgle}}) \subseteq \mathbf{\Sigma^{ps}}(X)$. A strict inclusion may hold, as for instance in $X = !; a$.[6]

3. Examples of state machines and state machine calculus

In this section a small collection of important state machines is presented as well as the operator set of state machine calculus.

3.1. Four data type state machines

Below we provide some examples of state machines that we consider fundamental and that are certainly useful in a number of circumstances (these examples and their names may

[6] The interface condition for program components can be stated as follows: for $[I, X]$ to be a program component it is required that $\mathbf{\Sigma^{ps}}(|X|) \subseteq I$.

be considered the initial segment of a library of useful state machines). Our specifications of these state machines are informal, but can of course be formalized in much more detail (we return to this matter in Appendix A). We write *NN* for the natural numbers (and `nn` in specific identifiers). The co-actions in the state machines below will be called by actions (instructions) having the same co-action part. Interfacing programs and state machines is discussed in Section 4.

Boolean register. This state machine has name `smbr` (abbreviating state machine boolean register), and is defined by

$$\langle\{\texttt{set(true)}, \texttt{set(false)}, \texttt{eq(true)}, \texttt{eq(false)}\}, \mathrm{F}_{\mathrm{br}}\rangle.$$

where the reply function will reply `false` to the co-actions `eq(true)` and `eq(false)` until the first `set(true)` is called. Thereafter the state of the register is `true`. At each co-action of the form `set(true)` or `set(false)` the reply function replies `true`.[7] The state of the state machine is flipped whenever it receives a 'set co-action' to the opposite truth value. The co-actions `eq(`b`)` do not change the state but will return a reply `true` whenever b is the current value of the register, and `false` otherwise.

NN register. $\texttt{smnnr} = \langle\{\texttt{set}(i), \texttt{eq}(i) \mid i \in NN\}, F_{nr}\rangle$ can contain arbitrary natural numbers. The reply function is self-evident. It generalizes the reply function of `smbr` to the parameter domain *NN* instead of {`true`, `false`}.

NN counter. $\texttt{smnnc} = \langle\{\texttt{succ}(), \texttt{pred}(), \texttt{isZero}()\}, F_{nc}\rangle$. Again the reply function is clear from the mnemonics of the interface actions, as is the convention to reply `true` as a default, under the assumption that the counter always starts with value zero. In particular it is assumed that, having become zero, the action `pred()` leaves its argument at zero.

NN stack. $\texttt{smnns} = \langle\{\texttt{push}(i), \texttt{pop}(), \texttt{topEq}(i) | i \in NN\}, F_{ns}\rangle$. The reply function is clear from the mnemonics of the interface actions, as is the convention to reply `true` as a default, under the assumption that the stack always starts empty. In particular, the action `pop()` leaves the empty stack empty, and pops the top element from the non-empty stack.

As a rule, the co-actions in the above concrete examples will be called by actions (instructions) that carry a plausible name, e.g., `smbr.set(true)` is an action of which the focus characterizes reference to a boolean register `smbr`, while the co-action part `set(true)` models the request to set the value of the register to `true`.

3.2. State machine calculus

Five operators and a constant for state machines are useful in many cases: service interface ($\boldsymbol{\Sigma}_{\mathbf{s}}(H)$), non-interfering combination ($H_1 \oplus H_2$), co-action prefixing ($\mathtt{p} : H$), restriction ($\Sigma \triangle H$), export ($\Sigma \square H$), which is complementary to restriction, and the empty state machine ($\boldsymbol{\emptyset}$).

- If the state machine H consists of the pair $\langle \Sigma, F\rangle$, then the service operator $\boldsymbol{\Sigma}_{\mathbf{s}}(H)$ gives its service interface Σ, thus $\boldsymbol{\Sigma}_{\mathbf{s}}(H) = \Sigma$.
- The non-interfering combination operator ($\oplus$) takes two state machines as operands, say H_1 and H_2. The service interface consists of those co-actions occurring in only one of the two service interfaces:

  $$\boldsymbol{\Sigma}_{\mathbf{s}}(H_1 \oplus H_2) = (\boldsymbol{\Sigma}_{\mathbf{s}}(H_1) \cup \boldsymbol{\Sigma}_{\mathbf{s}}(H_2)) - (\boldsymbol{\Sigma}_{\mathbf{s}}(H_1) \cap \boldsymbol{\Sigma}_{\mathbf{s}}(H_2)).$$

  Let F_i be the reply function for H_i. The reply function F for $H_1 \oplus H_2$ inspects the last instruction of its argument list. If that instruction is from H_i all actions of H_{3-i} (i.e. in

[7] This is an arbitrary default reply for a co-action that only serves to update the register.

$\mathbf{\Sigma_s}(H_{3-i})$ are removed and the reply is computed by means of an application of F_i to the remaining list.

- Co-action prefixing ($\mathtt{p}:H$) prefixes each co-action $\mathtt{a}(x)$ of a state machine (H) with "$\mathtt{p}:$" for some name (or "part") $\mathtt{p}$ and modifies its reply function:

  $$\mathtt{p}:\langle \Sigma, F\rangle = \langle \mathtt{p}:\Sigma, \mathtt{p}:F\rangle$$

  where $\mathtt{p}:\Sigma = \{\mathtt{p}:x \mid x \in \Sigma\}$, and $\mathtt{p}:F(\sigma) = F(\sigma\restriction\Sigma)$ where $\sigma\restriction\Sigma$ is the operation that projects σ into Σ^+ (i.e., strips off all prefixes $\mathtt{p}:$ from the σ-elements).
- The restriction ($\Sigma\triangle H$) of state machine H is obtained by removing all interface actions in Σ from the interface of H and by dropping all input streams, i.e., elements of Σ^+, featuring an action in Σ.
- The export operator ($\Sigma\,\square\, H$) does the converse of restriction: it drops all interface actions outside Σ.
- The empty state machine is denoted with $\emptyset$. This is the unique state machine H with $\mathbf{\Sigma_s}(H) = \emptyset$, playing but a formal role in the calculus of state machines.

Several identities concerning non-interfering combination are valid:

$$\begin{aligned}
H \oplus \emptyset &= H,\\
H \oplus H &= \emptyset,\\
H_1 \oplus H_2 &= H_2 \oplus H_1,\\
\mathtt{p}:(H_1 \oplus H_2) &= (\mathtt{p}:H_1) \oplus (\mathtt{p}:H_2),\\
(H_1 \oplus H_2) \oplus H_3 &= H_1 \oplus (H_2 \oplus H_3) \ \mathtt{if} \quad \mathbf{\Sigma_s}(H_1) \cap \mathbf{\Sigma_s}(H_2) \cap \mathbf{\Sigma_s}(H_3) = \emptyset.
\end{aligned}$$

Remark 3. The non-interfering combination provides a disjoint combination of the facilities of the two state machines, provided their service interfaces are disjoint. If the service interfaces overlap an ambiguity must be avoided and for that reason actions in the overlap are not offered by the non-interfering combination. The facilities of several copies of the same state machine can be combined after preparatory co-action prefixing, for instance:

$$\mathtt{p1}:\mathtt{smbr} \oplus \mathtt{p2}:\mathtt{smbr} \oplus \mathtt{p3}:\mathtt{smbr},$$

where the interface actions of $\mathtt{p1}:\mathtt{smbr}$ are

$$\{\mathtt{p1}:\mathtt{set}(\mathtt{true}), \mathtt{p1}:\mathtt{set}(\mathtt{false}), \mathtt{p1}:\mathtt{eq}(\mathtt{true}), \mathtt{p1}:\mathtt{eq}(\mathtt{false})\}.$$

For restriction and export there are several universally valid identities:
(distribution identities)

$$\begin{aligned}
\Sigma \triangle (H_1 \oplus H_2) &= (\Sigma \triangle H_1) \oplus (\Sigma \triangle H_2),\\
\Sigma \,\square\, (H_1 \oplus H_2) &= (\Sigma \,\square\, H_1) \oplus (\Sigma \,\square\, H_2),
\end{aligned}$$

(special cases)

$$\begin{aligned}
\mathbf{\Sigma_s}(H) \,\square\, H &= H,\\
\emptyset \triangle H &= H,\\
\emptyset \,\square\, H &= \emptyset,\\
\mathbf{\Sigma_s}(H_1) \,\square\, (H_1 \oplus H_2) &= H_1 \quad \mathtt{if}\ \ \mathbf{\Sigma_s}(H_1) \cap \mathbf{\Sigma_s}(H_2) = \emptyset,
\end{aligned}$$

(interaction with the service interface operator)

$$\begin{aligned}
\mathbf{\Sigma_s}(\Sigma \triangle H) &= \mathbf{\Sigma_s}(H) - \Sigma,\\
\mathbf{\Sigma_s}(\Sigma \,\square\, H) &= \Sigma \cap \mathbf{\Sigma_s}(H),\\
\Sigma \triangle H &= (\mathbf{\Sigma_s}(H) - \Sigma) \,\square\, H,\\
\Sigma \,\square\, H &= (\mathbf{\Sigma_s}(H) - \Sigma) \triangle H.
\end{aligned}$$

4. Interfacing programs and state machines

In this section we address the issue of interfacing programs and state machines. The subject of interfacing being open-ended in nature, our discussion focuses on two especially important cases. Then we discuss some applications of the state machine calculus that was defined in the previous section.

4.1. Use and apply

Program components have a program interface and state machines have a (definite) service interface. We consider two ways of interfacing a program (behavior) and a state machine:

Use. In a "use interface" the task of a state machine H is to support program X in its operation, which will express its value by executing actions that are not processed by H. Upon termination of the execution of X, H is forgotten and so is the state it is in.

Apply. In an "apply interface" the task of a program X is to transform the state of a state machine H. Upon termination of the execution of X, the result of the computation materializes in the state of H.

Both cases 'use' and 'apply' are captured by means of special purpose operators with an explicit reference to some focus.

In the case of 'use', the special operator $/_{\mathtt{f}}$ produces the behavior of the program's instructions referenced by focus $\mathtt{f}$ alongside a given interface. More precisely, the *use-operator* $/_{\mathtt{f}}$ combines a program behavior P and a state machine H producing a behavior, written $P/_{\mathtt{f}}H$. The actions in this behavior cannot have focus $\mathtt{f}$. "Use" refers to the fact that the program issues instructions to the state machine only for the sake of getting replies returned. The use-operator simply drops the state machine after program termination or inaction. So $P/_{\mathtt{f}}H$ is meaningful only for program behaviors P that issue actions of the form $\mathtt{g.a}(x)$ for $\mathtt{f} \neq \mathtt{g}$.

In the case of 'apply', the special operator $\bullet_{\mathtt{f}}$ for some focus $\mathtt{f}$ determines the state in which a program leaves a state machine after termination. The *apply-operator* $\bullet_{\mathtt{f}}$ connects a program behavior P and a state machine H, and yields a state machine $P \bullet_{\mathtt{f}} H$. This requires $\mathbf{\Sigma^{ps}}(P) \subseteq \mathtt{f}.\mathbf{\Sigma_s}(H)$, and raises the notorious question how to apply D to a state machine. A simple, though rather unelegant, solution is to introduce a default state machine D with universal interface that replies to each action with the (new) reply D. Then one may assume that D applied to any state machine H (including D) produces D, and P applied to (the default state machine) D for any behavior P also produces D. "Apply" refers to the fact that the program issues instructions to the state machine only for the sake of state machine transformation. So $P \bullet_{\mathtt{f}} H$ is meaningful only for program behaviors P that issue actions of the form $\mathtt{f.a}(x)$ and that terminate.

We shall define the use-operator and the apply-operator on the level of behaviors, although both can be defined on programs rather than behaviors as their first arguments. We

will only use that extension in combination with the behavioral abstraction operator $|-|$. A subscript will then indicate the program notation of the program and the notation works as follows:

$$|X/_{\mathtt{f}}H|_L = |X|_L/_{\mathtt{f}}H,$$

$$(X \bullet_{\mathtt{f}} H)_L = |X|_L \bullet_{\mathtt{f}} H.$$

The difference between the use-operator and the apply-operator is a matter of perspective: the use-operator considers the state machine a transformer of behaviors (or programs) and determines another behavior, whereas the apply-operator considers the program a transformer of state machines and therefore produces a state machine (unless some error occurs). In the perspective of the apply-operator the state machine is input and the modifications that are applied to it during a computation are the essence.

4.2. Semantic equations for use and apply

State machine effect notation. Let P be a program behavior and let H be a state machine. We wish to define $P/_{\mathtt{f}}H$ as the behavior of P using H. An action of P with focus $\mathtt{f}$ should be processed by H. An auxiliary notation has to be introduced beforehand. For a state machine $H = \langle \Sigma, F\rangle$ and a co-action $\mathtt{a}(x)$ in its interface, the state machine $\partial/\partial\mathtt{a}(x)\ H$ is defined by

$$\frac{\partial}{\partial \mathtt{a}(x)} H = \langle \Sigma, F'\rangle$$

with $F'(\sigma) = F(\mathtt{a}(x), \sigma)$ for all interface action sequences σ. The default state is covered by $\partial/\partial\mathtt{a}(x)\ \mathsf{D} = \mathsf{D}$. Furthermore, for $\sigma \in \Sigma^+$ we define $\partial/\partial\sigma H$ as H after having subsequently processed the co-actions in σ, so

$$\frac{\partial}{\partial \rho, \mathtt{a}(x)} H = \frac{\partial}{\partial \mathtt{a}(x)} \left(\frac{\partial}{\partial \rho} H \right).$$

Semantic equations for the use-operator. The defining rules for $P/_{\mathtt{f}}H$ are these (using $H = \langle \Sigma, F\rangle$):

$$\begin{aligned}
\mathsf{S}/_{\mathtt{f}}H &= \mathsf{S},\\
\mathsf{D}/_{\mathtt{f}}H &= \mathsf{D},\\
(P \trianglelefteq \mathtt{g.a}(x) \trianglerighteq Q)/_{\mathtt{f}}H &= (P/_{\mathtt{f}}H) \trianglelefteq \mathtt{g.a}(x) \trianglerighteq (Q/_{\mathtt{f}}H) \quad \text{if } \mathtt{g} \neq \mathtt{f},\\
(P \trianglelefteq \mathtt{f.a}(x) \trianglerighteq Q)/_{\mathtt{f}}H &= P/_{\mathtt{f}}\frac{\partial}{\partial \mathtt{a}(x)} H \quad \text{if } \mathtt{a}(x) \in \Sigma \text{ and } F(\mathtt{a}(x)) = \mathtt{true},\\
(P \trianglelefteq \mathtt{f.a}(x) \trianglerighteq Q)/_{\mathtt{f}}H &= Q/_{\mathtt{f}}\frac{\partial}{\partial \mathtt{a}(x)} H \quad \text{if } \mathtt{a}(x) \in \Sigma \text{ and } F(\mathtt{a}(x)) = \mathtt{false},\\
(P \trianglelefteq \mathtt{f.a}(x) \trianglerighteq Q)/_{\mathtt{f}}H &= \mathsf{D} \quad \text{if } \mathtt{a}(x) \notin \Sigma.
\end{aligned}$$

With the *conditional operator* of [9], which we define for $b \in \{\mathtt{true}, \mathtt{false}, \mathsf{D}\}$ by

$$x \triangleleft b \triangleright y = \begin{cases} x & \text{if } b = \mathtt{true},\\ y & \text{if } b = \mathtt{false},\\ \mathsf{D} & \text{if } b = \mathsf{D}, \end{cases}$$

(cf. [6,7]) we can summarize the last four equations into a single one, provided we adopt the convention that $F(w, \mathtt{a}(x)) = \mathsf{D}$ in case $\mathtt{a}(x) \notin \Sigma$:

$$(P \trianglelefteq \mathtt{f.a}(x) \trianglerighteq Q)/_{\mathtt{f}} H = \left(P/_{\mathtt{f}} \frac{\partial}{\partial \mathtt{a}(x)} H\right) \triangleleft F(\mathtt{a}(x)) \triangleright \left(Q/_{\mathtt{f}} \frac{\partial}{\partial \mathtt{a}(x)} H\right).$$

The equations for the use-operator $/_{\mathtt{f}}$ determine a behavior by means of a step-wise processing of the actions of the program. Only the actions with focus different from f will contribute to the resulting behavior. Below we consider a use-application on the PGLE program considered in Examples 1 and 2.

Example 4. Consider the program

$$\begin{aligned}\mathtt{ProgA} = {} & \mathtt{smbr.set(true)}; \mathtt{L0}; +\mathtt{smbr.eq(true)}; \#\#\mathtt{L1}; \#\#\mathtt{L2};\\ & \mathtt{L1}; \mathtt{smbr.set(false)}; \mathtt{Console.println(hello)}; \#\#\mathtt{L0};\\ & \mathtt{L2}; \mathtt{smbr.set(true)}; \mathtt{Console.println(goodbye)}; \#\#\mathtt{L0}\end{aligned}$$

from Examples 1 and 2 and the state machine smbr defined in Section 3.1. It easily follows that $|\mathtt{ProgA}|_{\mathtt{pgle}}/_{\mathtt{smbr}}\mathtt{smbr}$ traverses the cycle

$$(\mathtt{Console.println(hello)}, \mathtt{Console.println(goodbye)}).$$

Abbreviating $|\mathtt{ProgA}|_{\mathtt{pgle}}/_{\mathtt{smbr}}\mathtt{smbr}$ to P, we may write

$$P = \mathtt{Console.println(hello)} \circ \mathtt{Console.println(goodbye)} \circ P.$$

The use-operator can play a key role in projection semantics. In some cases a projection for a program notation can only be found at the cost of the introduction of an auxiliary state machine which the projected program may use (we return to this matter in Section 5).

Semantic equations for the apply-operator. Using $H = \langle \Sigma, F \rangle$, the definition of $P \bullet_{\mathtt{f}} H$ is as follows:

$$\begin{aligned} S \bullet_{\mathtt{f}} H &= H,\\ \mathsf{D} \bullet_{\mathtt{f}} H &= \mathsf{D},\\ P \bullet_{\mathtt{f}} \mathsf{D} &= \mathsf{D},\\ (P \trianglelefteq \mathtt{g.a}(x) \trianglerighteq Q) \bullet_{\mathtt{f}} H &= \mathsf{D} \quad \text{if } \mathtt{f} \neq \mathtt{g},\\ (P \trianglelefteq \mathtt{f.a}(x) \trianglerighteq Q) \bullet_{\mathtt{f}} H &= \left(P \bullet_{\mathtt{f}} \frac{\partial}{\partial \mathtt{a}(x)} H\right) \triangleleft F(\mathtt{a}(x)) \triangleright \left(Q \bullet_{\mathtt{f}} \frac{\partial}{\partial \mathtt{a}(x)} H\right). \end{aligned}$$

If application of these rules fails to lead to a converging computation, the computation of P on H is said to diverge, which is written as $P \bullet_{\mathtt{f}} H = \mathsf{D}$.

The apply-operator plays a key role in the description of batch processing. In the formalization of batch processing both inputs and outputs of programs are packed in a state machine. A batch program is seen as a state machine transformer, semantically captured by the apply-operator.

4.3. Focus renaming and focused instruction refinement

It is reasonable to repeatedly use the use-operator, as in $(P/_{\mathtt{f}} H_1)/_{\mathtt{g}} H_2$, or more briefly, $P/_{\mathtt{f}} H_1/_{\mathtt{g}} H_2$. Note that by definition, $P/_{\mathtt{f}} H_1/_{\mathtt{f}} H_2 = P/_{\mathtt{f}} H_1$. Also in the case that $\mathtt{f} \neq \mathtt{g}$, the repeated use application in $P/_{\mathtt{f}} H_1/_{\mathtt{g}} H_2$ can be combined into a single one using 'focus renaming' and 'focus-dependent co-action prefixing' (or 'focused instruction prefixing').

Let P be a behavior, then the *focus renaming*

$$[\mathtt{f} \mapsto \mathtt{g}]P$$

is as P, except that all actions of the form $\mathtt{f.a}(x)$ are renamed into $\mathtt{g.a}(x)$. So, $[\mathtt{f} \mapsto \mathtt{g}]\mathsf{S} = \mathsf{S}$, $[\mathtt{f} \mapsto \mathtt{g}]\mathsf{D} = \mathsf{D}$, and

$$[\mathtt{f} \mapsto \mathtt{g}](P \trianglelefteq \mathtt{h.a}(x) \trianglerighteq Q) = \begin{cases} ([\mathtt{f} \mapsto \mathtt{g}]P) \trianglelefteq \mathtt{g.a}(x) \trianglerighteq ([\mathtt{f} \mapsto \mathtt{g}]Q) & \text{if } \mathtt{f} = \mathtt{h}, \\ ([\mathtt{f} \mapsto \mathtt{g}]P) \trianglelefteq \mathtt{h.a}(x) \trianglerighteq ([\mathtt{f} \mapsto \mathtt{g}]Q) & \text{otherwise.} \end{cases}$$

(Of course, in any program algebra notation, focus renaming can be defined straightforwardly and satisfies $|[\mathtt{f} \mapsto \mathtt{g}]X| = [\mathtt{f} \mapsto \mathtt{g}]|X|$.) Observe that $P/_{\mathtt{f}}H = ([\mathtt{f} \mapsto \mathtt{g}]P)/_{\mathtt{g}}H$ if $\mathtt{g}$ is fresh in P. We shall use the notation

$$[\mathtt{f1} \mapsto \mathtt{g1}, \mathtt{f2} \mapsto \mathtt{g2}]P$$

for $[\mathtt{f2} \mapsto \mathtt{g2}]([\mathtt{f1} \mapsto \mathtt{g1}]P)$. Observe that if $\mathtt{f1} \neq \mathtt{f2}$, then

$$[\mathtt{f1} \mapsto \mathtt{g1}, \mathtt{f2} \mapsto \mathtt{g2}]P = [\mathtt{f2} \mapsto \mathtt{g2}, \mathtt{f1} \mapsto \mathtt{g1}]P.$$

Furthermore, one can extend 'co-action prefixing' (see Section 3.2) to *focused co-action prefixing* on behaviors (or programs) by fixing a focus and an interface. This operation is written as $\mathtt{p} : \mathtt{f}.\Sigma$, and

$$\mathtt{p} : \mathtt{f}.\Sigma(P)$$

refines each action $\mathtt{f.a}(x)$ of P with $\mathtt{a}(x) \in \Sigma$ to $\mathtt{f.p} : \mathtt{a(x)}$. So, $\mathtt{p} : \mathtt{f}.\Sigma(\mathsf{S}) = \mathsf{S}$, $\mathtt{p} : \mathtt{f}.\Sigma(\mathsf{D}) = \mathsf{D}$, and

$$\mathtt{p} : \mathtt{f}.\Sigma(P \trianglelefteq \mathtt{g.a}(x) \trianglerighteq Q) = \begin{cases} (\mathtt{p} : \mathtt{f}.\Sigma(P) \trianglelefteq \mathtt{g.p} : \mathtt{a}(x) \trianglerighteq (\mathtt{p} : \mathtt{f}.\Sigma(Q)) & \text{if } \mathtt{f} = \mathtt{g} \text{ and } \mathtt{a(x)} \in \Sigma, \\ (\mathtt{p} : \mathtt{f}.\Sigma(P)) \trianglelefteq \mathtt{g.a}(x) \trianglerighteq (\mathtt{p} : \mathtt{f}.\Sigma(Q)) & \text{otherwise.} \end{cases}$$

(Of course, in any program algebra notation, focused instruction prefixing can be defined straightforwardly and satisfies $|\mathtt{p} : \mathtt{f}.\Sigma(x)| = \mathtt{p} : \mathtt{f}.\Sigma(|X|)$.) We write

$$\phi \cup \psi$$

for the application of focused instruction prefixings ϕ and ψ if their foci differ (the application order is immaterial in this case).

The following result states that repeated applications of use operators can always be combined into a single one.

Theorem 5. *Let $H_i = \langle \Sigma_i, F_i \rangle$ for $i = 1, 2$, and let P be a behavior in which focus $\mathtt{g}$ does not occur. If $\mathtt{f1} \neq \mathtt{f2}$ for foci $\mathtt{f1}$ and $\mathtt{f2}$, then*

$$P/_{\mathtt{f1}}H_1/_{\mathtt{f2}}H_2 = [\mathtt{f1} \mapsto \mathtt{g}, \mathtt{f2} \mapsto \mathtt{g}](\mathtt{f1} : \mathtt{f1}.\Sigma_1 \cup \mathtt{f2} : \mathtt{f2}.\Sigma_2)(P)/_{\mathtt{g}}\mathtt{f1} : H_1 \oplus \mathtt{f2} : H_2.$$

Proof. We have to argue that $L = R$ with $L = P/_{\mathtt{f1}}H_1/_{\mathtt{f2}}H_2$ and $R = [\mathtt{f1} \mapsto \mathtt{g}, \mathtt{f2} \mapsto \mathtt{g}](\mathtt{f1} : \mathtt{f1}.\Sigma_1 \cup \mathtt{f2} : \mathtt{f2}.\Sigma_2)(P)/_{\mathtt{g}}H$ for $H = \mathtt{f1} : H_1 \oplus \mathtt{f2} : H_2$. Clearly, neither L nor R can perform actions with focus $\mathtt{f}i$ or with focus $\mathtt{g}$. So, provided that actions of the form

$\mathtt{g.f}i.\mathtt{a}(x)$ yield the same reply in R as the associated $\mathtt{f}i.\mathtt{a}(x)$ actions in L, it follows that L and R are (stepwise) behaviorally equivalent.

If in R some action $\mathtt{g.f1}:\mathtt{a}(x)$ is processed by $\partial/\partial\sigma H$, then the sequence of $\mathtt{f1}:$-prefixed co-actions in σ and $\mathtt{f1}:\mathtt{a}(x)$ determine the reply of H. Now $\partial/\partial\sigma\restriction\Sigma_1 H_1$ computes by definition of $\oplus$ and co-action prefixing the same reply on $\mathtt{a}(x)$. As H_2 is not addressed in L, it follows that both behaviors coincide in this case.

If in R some action $\mathtt{g.f1}:\mathtt{a}(x)$ is processed by $\partial/\partial\sigma H$ a similar argument applies. □

With the commutativity of focus renaming and focused instruction prefixing for different foci $\mathtt{f1}$ and $\mathtt{f2}$, and the commutativity of $\oplus$, it follows from Theorem 5 above that $P/_{\mathtt{f1}}H_1/_{\mathtt{f2}}H_2 = P/_{\mathtt{f2}}H_2/_{\mathtt{f1}}H_1$.

Similarly a repreated application of the apply-operator is possible. Under strict conditions demanding that ';' represents sequential composition between P_1 and P_2, the following is valid: $P_1 \bullet_{\mathtt{f}} (P_2 \bullet_{\mathtt{f}} H) = (P_2; P_1) \bullet_{\mathtt{f}} H$. We will not further address the issue of repreated apply applications, and finish this section with the observation that $P \bullet_{\mathtt{f}} H = ([\mathtt{f} \mapsto \mathtt{g}]P) \bullet_{\mathtt{g}} H$ if $\mathtt{g}$ is fresh in P.

5. Memory is indispensable

In simple cases it is possible to incorporate the 'use functionality' of a state machine in programs, and to define a projection semantics for programs using co-actions of the state machine service interface. In this section we consider such a projection in detail for PGLE (briefly described in Section 2.2) and the boolean register $\mathtt{smbr}$ (see Section 3.1). From this and another example (involving a state machine with an infinitary structure) we conclude that state machines or similar memory devices are indispensable in programming technology.

5.1. Projection semantics for PGLE with use

Consider a PGLE program X using co-actions from $\mathbf{\Sigma_s}(\mathtt{smbr})$. We can view X as a program in PGLEsmbr, an ad hoc version of PGLE equipped with the following semantics:

$$|X|_{\text{pglesmbr}} = |X|_{\text{pgle}/\mathtt{smbr}}\,\mathtt{smbr}.$$

Alternatively one may ask for a projection $\mathtt{pglesmbr2pgle}$ such that the meaning of PGLEsmbr programs can be determined using the projection semantics without any mention of state machines and their 'use':

$$|X|_{\text{pglesmbr}} = |\mathtt{pglesmbr2pgle}(X)|_{\text{pgle}}.$$

So the question is to design a projection $\mathtt{pglesmbr2pgle}$ satisfying the following constraint. For all X, $|X|_{\text{pgle}/\mathtt{smbr}}\,\mathtt{smbr} = |\mathtt{pglesmbr2pgle}(X)|_{\text{pgle}}$.

Let $X = u_1; \ldots; u_k$. We assume that termination of X takes place only at "!", and that all goto's use labels that occur in label catches in the program. Then we define

$$\mathtt{pglesmbr2pgle}(X) = \psi_1^{\mathtt{false}}(u_1); \ldots; \psi_k^{\mathtt{false}}(u_k); \psi_1^{\mathtt{true}}(u_1); \ldots; \psi_k^{\mathtt{true}}(u_k)$$

with the auxiliary operators ψ_i^b determined by the rewrite rules below. The general idea is that each $\psi_i^b(u)$ is a pair consisting of a label catch $\mathtt{L}i$ or $\mathtt{L}k+i$ that pinpoints the position

of the instruction and the value of b (false, respectively true), and a second instruction. This may yield $\mathtt{L}j;\#\#\mathtt{L}j+1$, where the second instruction clearly is redundant; however, we include such redundant instructions in order to expose the systematics of this projection

$$\psi_i^b(\mathtt{L}j) = (\mathtt{L}k+i;\mathtt{L}2k+2j+1) \trianglelefteq b \trianglerighteq (\mathtt{L}i;\mathtt{L}2k+2j+2),$$
$$\psi_i^b(\#\#\mathtt{L}j) = (\mathtt{L}k+i;\#\#\mathtt{L}2k+2j+1) \trianglelefteq b \trianglerighteq (\mathtt{L}i;\#\#\mathtt{L}2k+2j+2),$$
$$\psi_i^b(\mathtt{smbr.set(true)}) = \mathtt{L}(k+i \trianglelefteq b \trianglerighteq i);\#\#\mathtt{L}i+k+1,$$
$$\psi_i^b(+\mathtt{smbr.set(true)}) = \mathtt{L}(\mathtt{k}+\mathtt{i} \trianglelefteq \mathtt{b} \trianglerighteq \mathtt{i});\#\#\mathtt{Li}+\mathtt{k}+1,$$
$$\psi_i^b(-\mathtt{smbr.set(true)}) = \mathtt{L}(\mathtt{k}+\mathtt{i} \trianglelefteq \mathtt{b} \trianglerighteq \mathtt{i});\#\#\mathtt{Li}+\mathtt{k}+2,$$
$$\psi_i^b(\mathtt{smbr.set(false)}) = \mathtt{L}(k+i \trianglelefteq b \trianglerighteq i);\#\#\mathtt{L}i+1,$$
$$\psi_i^b(+\mathtt{smbr.set(false)}) = \mathtt{L}(\mathtt{k}+\mathtt{i} \trianglelefteq \mathtt{b} \trianglerighteq \mathtt{i});\#\#\mathtt{Li}+2,$$
$$\psi_i^b(-\mathtt{smbr.set(false)}) = \mathtt{L}(\mathtt{k}+\mathtt{i} \trianglelefteq \mathtt{b} \trianglerighteq \mathtt{i});\#\#\mathtt{Li}+1,$$
$$\psi_i^b(\mathtt{smbr.eq(true)}) = (\mathtt{L}k+i;\#\#\mathtt{L}k+i+1) \trianglelefteq b \trianglerighteq (\mathtt{L}i;\#\#\mathtt{L}i+1),$$
$$\psi_i^b(+\mathtt{smbr.eq(true)}) = (\mathtt{Lk}+\mathtt{i};\#\#\mathtt{Lk}+\mathtt{i}+1) \trianglelefteq \mathtt{b} \trianglerighteq (\mathtt{Li};\#\#\mathtt{Li}+2),$$
$$\psi_i^b(-\mathtt{smbr.eq(true)}) = (\mathtt{Lk}+\mathtt{i};\#\#\mathtt{Lk}+\mathtt{i}+2) \trianglelefteq \mathtt{b} \trianglerighteq (\mathtt{Li};\#\#\mathtt{Li}+1),$$
$$\psi_i^b(\mathtt{smbr.eq(false)}) = (\mathtt{L}k+i;\#\#\mathtt{L}k+i+1) \trianglelefteq b \trianglerighteq (\mathtt{L}i;\#\#\mathtt{L}i+1),$$
$$\psi_i^b(+\mathtt{smbr.eq(false)}) = (\mathtt{Lk}+\mathtt{i};\#\#\mathtt{Lk}+\mathtt{i}+2) \trianglelefteq \mathtt{b} \trianglerighteq (\mathtt{Li};\#\#\mathtt{Li}+1),$$
$$\psi_i^b(-\mathtt{smbr.eq(false)}) = (\mathtt{Lk}+\mathtt{i};\#\#\mathtt{Lk}+\mathtt{i}+1) \trianglelefteq \mathtt{b} \trianglerighteq (\mathtt{Li};\#\#\mathtt{Li}+2),$$
$$\psi_i^b(u) = \mathtt{L}(k+i \trianglelefteq b \trianglerighteq i);u \text{ otherwise.}$$

The description of this transformation is disappointingly long. Different strategies can be applied to simplify the projection. Nevertheless we prefer the description in the given form as it clearly demonstrates the 'raw data'.

Example 6. (*Continued*) Recall the program

$$\begin{aligned}\mathtt{ProgA} = {} & \mathtt{smbr.set(true)};\mathtt{L0};+\mathtt{smbr.eq(true)};\#\#\mathtt{L1};\#\#\mathtt{L2};\\ & \mathtt{L1};\mathtt{smbr.set(false)};\mathtt{Console.println(hello)};\#\#\mathtt{L0};\\ & \mathtt{L2};\mathtt{smbr.set(true)};\mathtt{Console.println(goodbye)};\#\#\mathtt{L0}\end{aligned}$$

from Examples 1, 2, 4, where it is stated that for $|\mathtt{ProgA}|_{\mathtt{pgle}/\mathtt{smbr}}\mathtt{smbr} = P$,

$$P = \mathtt{Console.println(hello)} \circ \mathtt{Console.println(goodbye)} \circ P. \tag{1}$$

As ProgA is in PGLEsmbr, we can apply the projection pglesmbr2pgle(ProgA) as defined above. We list the outcome in Fig. 1 in two columns, the left one giving the $\psi_i^{\mathtt{false}}$ values, and the right one the $\psi_i^{\mathtt{true}}$ values. It easily follows that the behavioral characterization (1) holds, which demonstrates the correctness of our projection pglesmbr2pgle(ProgA).

5.2. Memory counts

If a combination of disambiguated boolean register state machines is used via distinctive use applications $/_{\mathtt{smbr1}}, /_{\mathtt{smbr2}}, /_{\mathtt{smbr3}}, \ldots$, the mentioned projection can be applied step by step. At each of these steps the program becomes about four times as long causing an exponential blow-up of the length of the program. From a philosophical point of view this

L1; ##L15;	L14; ##L15;
L2; L28;	L15; L27;
L3; ##L5;	L16; ##L17;
L4; ##L30;	L17; ##L29;
L5; ##L32;	L18; ##L31;
L6; L30;	L19; L29;
L7; ##L8;	L20; ##L8;
L8; Console.println(hello);	L21; Console.println(hello);
L9; ##L28;	L22; ##L27;
L10; L32;	L23; L31;
L11; ##L25;	L24; ##L25;
L12; Console.println(goodbye);	L25; Console.println(goodbye);
L13; ##L28;	L26; ##L27

Fig. 1. pglesmbr2pgle(ProgA).

may be considered no issue. As soon as programs become part of technology, however, the matter is vital. It can be concluded that already in the simplest of conceivable cases (a number of copies of a boolean register state machine), it is totally impractical to avoid the use of a boolean register in favor of a longer program. Although our transformation may be less than optimal, the essential problem is that using a number of different possible state machine states can only be expressed in terms of the program itself if an exponential blow-up of the length of the program is accepted. (In the case of smbr, a solution to this problem is provided by Theorem 5.) Furthermore, a projection semantics without the use of state machines does not exist in the case that a program (a behavior) essentially uses a state machine that has a infinitary internal structure. For example, consider the behavior recursively defined by P for actions a and b:

$$\begin{aligned} P &= a \circ Q_{1,0}, \\ Q_{i+1,j} &= b \circ Q_{i,j+1}, \\ Q_{0,j} &= a \circ Q_{j+1,0} \end{aligned}$$

(so P performs $a \circ b \circ a \circ b^2 \circ a \circ b^3 \circ \ldots$). It is not hard to find a PGLE program, say ProgC, that uses a two-counter $H = \mathtt{c1} : \mathtt{smnnc} \oplus \mathtt{c2} : \mathtt{smnnc}$ (see Section 3 for the definition of the *NN* counter smnnc and the operation $\oplus$) and displays this behavior: assume that the actions a, b have a focus different from smnnc. Then set

$$\begin{aligned} \mathtt{ProgC} = \mathtt{L0};\ & a;\ \mathtt{smnnc.c1} : \mathtt{succ}(); \\ & \mathtt{L1};\ +\mathtt{smnnc.c1} : \mathtt{isZero}();\ \#\#\mathtt{L2}; \\ & \quad b;\ \mathtt{smnnc.c1} : \mathtt{pred}();\ \mathtt{smnnc.c2} : \mathtt{succ}();\ \#\#\mathtt{L1}; \\ & \mathtt{L2};\ +\mathtt{smnnc.c2} : \mathtt{isZero}();\ \#\#\mathtt{L0}; \\ & \quad \mathtt{smnnc.c2} : \mathtt{pred}();\ \mathtt{smnnc.c1} : \mathtt{succ}();\ \#\#\mathtt{L2} \end{aligned}$$

Now $|\mathtt{ProgC}|_{\mathrm{pgle}/\mathtt{smnnc}}H = P$ reaches infinitely many different states. A projection of ProgC into PGLE cannot preserve this property: it is easy to prove that any behavior definable by a PGLE program has a regular (or 'finite') control structure. Here lies a second argument for the use of state machines (or data structures) external to a program.

It can be concluded that programming cannot be understood without the use of program independent memory. Memory outside the program has been formalized using state machines.

Appendix A. Specification techniques for state machines

The simplest way of denoting a program is to write it. Programs are typically *constructed* by means of the successive application of a limited number of construction principles to a limited number of primitives. Another obvious category of objects admitting description by construction are the natural numbers. Numbers also allow *specification*: the construction $n = 97$ corresponds to 'n equals the largest prime number below 100'. The second description is typically a specification. Specification is performed by means of the presentation of a list of properties. (Some other mathematical objects, for instance the structure of the real numbers in classical mathematics, are specified rather than constructed.) As it turns out state machines lack any direct means of construction: state machines must be specified. In general, the specification of a state machine poses two questions at the same time: (1) which specification technique should be used, and (2) how should the technique be used in a particular case.

Below we list some options for state machine specification; for more information we refer to [8].

Informal specification. An informal specification of a state machine (or of a class of state machines) will be definite about the interface of instructions that can be issued to the state machine.[8] The informal part is in the description of the reply function. There is no doubt that the combination of a precise interface description with an intuitively appealing description of a reply function is very useful in practice.

Property-oriented specification. This technique includes many more specialized techniques using restricted logics. Point of departure is the observation that a state machine is itself a three-sorted algebra: a sort (Σ) of interface actions with a constant for each interface action, a sort (C) of extended booleans, equipped with constants for both truth values and for inaction, and a sort (Σ^+) consisting of non-empty sequences of interface actions, the reply function being a function in this algebra. Property-oriented specification comprises writing an axiomatization for this three-sorted structure, preferably disallowing auxiliary operators.

Property-oriented state machine specifications in first-order logic are difficult to find, even for remarkably simple state machines. If auxiliary interface actions are allowed it is always possible to find a direct specification of a state machine using equations only.[9] However, the proof of this fact is long and complicated and the practical implication of the result is not immediately clear. (We refer to [3] for the proof.)

[8] With CORBA IDL industry has produced a description technique for such interfaces that may well be sufficiently strong for many years to come.

[9] It must be assumed that the state machine is computable. The case of a finite state-space is especially intricate.

Temporal logics can be used to provide property-oriented state machine specifications, in some cases with remarkable elegance.

State-space model description. A state-space model for a state machine is obtained by replacing Σ^+ by Σ^* in the sorts mentioned above, and adding another auxiliary sort S. The sort S contains states in which the state machine may be during the execution of a succession of interface actions. For the sort of states a constant and three operations are needed:

(1) the initial state constant $\mathtt{s}_{\mathrm{init}}$ determines in which state the state machine will start its operation upon initialization,
(2) the effect function $\mathtt{E} : S \times \Sigma \to S$ takes a state and an interface action and determines the state in which the state machine will be after processing the interface action,
(3) the yield function $\mathtt{Y} : \Sigma \times S \to C$ where $C = \{\mathtt{true}, \mathtt{false}, \mathsf{D}\}$ determines which reply is produced when state s is reached after processing an interface action,
(4) the cumulative effect function $\mathtt{CE} : \Sigma^* \to S$ determines the state of the state machine after a sequence of interface actions has been processed.

The connection between these operators and the state machine $H = \langle \Sigma, F \rangle$ being specified is then as follows (writing [ ] for the empty sequence):

$$\begin{aligned}
&\mathtt{CE}([\,]) = \mathtt{s}_{\mathrm{init}},\\
&\mathtt{CE}(w, \mathtt{a}(x)) = \mathtt{E}(\mathtt{CE}(w), \mathtt{a}(x)),\\
&F(w, \mathtt{a}(x)) = \mathtt{Y}(\mathtt{a}(x), \mathtt{CE}(w)).
\end{aligned}$$

Obviously the operators CE and F can be derived when E and Y are known. Therefore these techniques usually make no mention of any operators other than E and Y.

Technically a state-space description of a state machine amounts to a specification of the algebra with sorts Σ, S and C and operations E,CE,Y. It turns out that if additional functions are allowed, the use of equations will suffice to obtain appropriate specifications of the state-space model in all relevant cases.

From a certain level of complexity, state-space models are the only option if a formal description of a state machine is needed. Regarding state space model descriptions the following remarks can be made.

Parametrized state machines The notation for state space model descriptions of state machines can be made more explicit, thus obtaining a parametrized family of state machines including the state machine H. Using the notation given above each state $s \in S$ determines a state machine $H_s = \langle \Sigma, F_s \rangle$. The definition of F_s makes use of an auxiliary operation $\mathtt{CE}_s$:

$$\begin{aligned}
&\mathtt{CE}_s([\,]) = s,\\
&\mathtt{CE}_s(w, \mathtt{a}(x)) = \mathtt{E}(\mathtt{CE}_s(w), \mathtt{a}(x)),\\
&F_s(w, \mathtt{a}(x)) = \mathtt{Y}(\mathtt{a}(x), \mathtt{CE}_s(w)).
\end{aligned}$$

The connection between H and this state machine family reads $H = H_{\mathtt{s}_{\mathrm{init}}}$.

Notational conventions for effect functions. In practice the format given here is often only present in disguise. Rather than having an effect function E with a second argument in Σ, a special operation (say $e_{\mathtt{a}(x)}$) will be used for each $\mathtt{a}(x) \in \Sigma$. If $\mathtt{a}(x)$ has parameters these will be taken as additional parameters for $e_{\mathtt{a}(x)}$, usually preceding the state parameter. Of course a more mnemonic notation than $e_{\mathtt{a}(x)}$ is likely to be used. The yield function Y is usually only specified for non-void actions, void actions having the yield true by definition. In most cases non-void actions cannot change the state. Then it suffices to describe these by means of a mapping $y_{\mathtt{a}}$ extracting a boolean value from a state argument. Again a

more mnemonic notation is likely to be used, and $y_{\mathtt{a}}$ may be given the parameters of $\mathtt{a}(x)$ as additional parameters.

References

[1] J.A. Bergstra, Datatypen bezien vanuit de recursietheorie, in: J.C. van Vliet (Ed.), Colloquium Capita Datastructuren, Mathematisch Centrum, Amsterdam, 1978, pp. 157–170 (in Dutch).
[2] J.A. Bergstra, What is an abstract data type? Inf. Proc. Lett. 7 (1) (1978) 42–43.
[3] J.A. Bergstra, J.-J.Ch. Meyer, Equational specification of finite minimal unoids, using unary hidden functions only, Fund. Inf. 5 (2) (1982) 143–170.
[4] J.A. Bergstra, M.E. Loots, Program algebra for component code, Formal Aspects Comput. 12 (1) (2000) 1–17.
[5] J.A. Bergstra, M.E. Loots, Program algebra for sequential code, J. Logic Algebr. Programming 51 (2002) 125–156.
[6] J.A. Bergstra, A. Ponse, Kleene's three-valued logic and process algebra, Inf. Proc. Lett. 67 (2) (1998) 95–103.
[7] J.A. Bergstra, A. Ponse, Process algebra and conditional composition, Inf. Proc. Lett. 80 (1) (2001) 41–49.
[8] L.M.G. Feys, H.B.M. Jonkers, C.A. Middelburg, Notations for Software Design, Springer, Berlin, 1994.
[9] C.A.R. Hoare, I.J. Hayes, He. Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, B.A. Sufrin, Laws of programming, Commun. ACM 30 (8) (1987) 672–686.