Computer Science
University of Manchester

Process-Algebraic Foundations for an Object-Based Design Notation

C. B. Jones

Technical Report UMCS–93–10–1

Process-Algebraic Foundations for an Object-Based Design Notation

C. B. Jones*

Department of Computer Science
University of Manchester
Oxford Rd., Manchester, U.K.
cbj@cs.man.ac.uk

October 4, 1993

Abstract

Earlier papers give examples of the development of concurrent programs using $\pi o\beta\lambda$ which is a design notation employing concepts from object-oriented programming languages. Use is made of constraints on the *object graphs* to limit interference and assertions over such graphs to reason about about interference. This report merges (and corrects minor inconsistencies between) two papers which document the semantics of $\pi o\beta\lambda$ by providing a mapping to the π-calculus and indicate how arguments about the design notation might be based on that semantics. It also reflects some recent work not cited in the earlier papers.

Technical Reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from ftp.cs.man.ac.uk in the directory /pub/TR. The files are stored as PostScript, in compressed form, with the report number as filename. Alternatively, reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, U.K.

Contents

1 Introduction

Development methods which are *compositional* make it possible to justify one step of development before proceeding to subsequent design activity: specifications need to isolate the subcomponents introduced in a development step. Compositional development methods offer scope for improving the productivity of the design process by minimizing the 'scrap and rework' inherent in the late detection of early design errors. Relatively simple specification ideas (e.g. pre and post-conditions) suffice for the compositional development of sequential programs; *interference* makes it difficult to find useful compositional approaches for concurrent programs (an interesting account of this quest is contained in [dR85, HdR86]).

The research reported in this paper is intended to contribute to the quest for compositional development methods for concurrent programs: the use of some notions from object-oriented languages is a means to tame interference rather than a (fashionable) end in itself. Two papers – [Jon93a, Jon93c][1] – indicate that selected features from object-oriented languages might further the quest for compositional development methods (they also provide references which trace the evolution of the ideas). Interference is an issue for both shared-variable and communication-based concurrency; object-based languages offer a compromise between the two extremes by placing control of access to state (i.e. instance variables) in the hands of the developer and supporting ways of controlling the activation of methods.

A way of limiting interference in object-based languages is exploited in [Jon93a] to show how concurrency can be introduced by transformations. Central to the justification of observational equivalence is the use of invariants on the *object graphs* which can arise. Interference can not always be controlled in this way and [Jon93c] shows how a logic notation discussed in [Jon91] can be used to reason about interference over complex object graphs. The design notation used in both of these papers is currently known as $\pi o \beta \lambda$. Rather than being viewed as a contending programming language, it is hoped that $\pi o \beta \lambda$ will be used as a design notation for the development of programs in languages like POOL [Ame89], ABCL [Yon90], Beta [KMMN91], Modula-3 [Nel91] or UFO [Sar92]. But – if sound development methods are required – design notations have to be given semantics. For example, the transformation rules used in [Jon93a] have to be shown to preserve observational equivalence.

A denotational semantics for $\pi o \beta \lambda$ would have to deal with concurrency (relevant papers include [Wol88, AR89, AR92, dBdV91]). In some respects, a semantics based on resumptions fits this sort of parallel object-based language quite well but there are some difficulties and these would even be shared by an operational semantics. The core of the problem – in such a model-oriented semantics – is that it is necessary to describe a very fine level of granularity in order to prove that this level of granularity does not actually make any difference. The approach followed here is to map the constructs of $\pi o \beta \lambda$ to the π-calculus [MPW92, Mil92b]. It would be possible to argue that this semantics is also giving too fine a level of granularity but in this case the π-calculus has an algebra which makes it easy to reason about equivalencies between processes.

The current paper investigates the task of basing arguments on that semantics. One of the conclusions is that the familiar notions of *bi-simulation* etc. are not immediately applicable to the proofs needed here. Although the arguments given below are hopefully convincing, they are not completely formal and the current paper might be viewed as a challenge which could stimulate the development of new approaches to equivalence proofs.

[1] The main content of these two papers are combined in [Jon92] but the appendix there on a possible approach to the semantics of $\pi o \beta \lambda$ is superseded by the current report.

Section 2 below introduces $\pi o\beta\lambda$ by an example. Section 3 describes the basic ideas of mapping $\pi o\beta\lambda$ into the π-calculus; the problem of handling instance variables and the special case of those instance variables containing references is discussed in Section 4. Some detailed problems of the mapping are described in Section 5 and the mapping itself is summarized in Appendix C. The version of the π-calculus used is described in Appendix B. An argument to justify one specific transformation which affects the degree of concurrency is tackled in Section 6 and more general results are addressed in Sections 7 and 8. (There are, of course, many simpler transformation rules than those considered here: for example, rules can be given to insert assignment of expressions to local instance variables.) The paper concludes with a discussion in Section 9.

2 A $\pi o\beta\lambda$ example

The $\pi o\beta\lambda$ class presented in Figure 1 provides a sorting facility; instances of the class are linked into a 'sorting ladder'. Each instance of the class has two instance variables. The variable v contains the natural number which is stored in one element of the class and the variable l – if not nil – contains a reference to another instance of the class. By tracing along this series of references, one can collect the sorted sequence of values from the v variables. The class – and thus each of its instances – has three methods. The *add* method accepts a non-zero natural number and stores it at an appropriate place in the sequence of instances; the *remove* method returns the first (i.e. lowest) element of the sequence; and the *test* method determines whether a given natural number is or is not in the sorted vector (for simplicity, zero is used to mark a nil value). The semantics of $\pi o\beta\lambda$ is such that only one method can be active in any one instance of a class at a time. The code which invokes a method is held in a *rendezvous* until the method being executed reaches a return statement. Notice that the *add* method contains a return as its first statement so, as soon as the parameter has been passed, the caller is released from the *rendezvous*. The remainder of the code of the *add* method does what one might expect: the higher value is passed down a chain of method-calls linked by the l instance variables. Because of the way the returns work, concurrency is possible within a sequence of instances of the *Sort* class. The only other point in *add* is to notice that when the first value is stored in an instance of the class, a new next item is created in the linked-list. The *remove* method is similar, noting only that it destroys the final element of the linked-list of instances when a zero is passed back from further down the list. The idea of obtaining concurrency by making sure that returns are executed as soon as possible would also be useful in the *test* method; but, here the invoking procedure is bound to be held up because a value is required. The effect of the yield statement in *test* is to delegate the task of returning a value but to terminate this instance of the *test* method so that other methods can be invoked on the same instance of the class.

It is worth noting that the specification of the *Sort* class in Figure 1 is by no means trivial. It is not possible to write a conventional pre/post-condition specification because the initial state to which one might expect to relate the final state in a post-condition is in fact a composite of the values stored in the v variables and any activity of *add* and *remove* operations which is still rippling down the list. It would be necessary in order to write such a specification to use some sort of 'ghost variables'. The text presented in Figure 1 is actually developed in [Jon93a] via a sequential (i.e. non-concurrent) version of the same algorithm. The sequential version differs from that presented in Figure 1 by having the return statements placed at the end of *add* and *remove* methods and the yield statement in *test* written as a return. The

```
Sort class
vars v: [ℕ] ← 0; l: private ref(Sort) ← nil
add(x: ℕ₁) method
  return
  if v = 0 then (v ← x; l ← new Sort)
  elif v ≤ x then l!add(x)
  else (l!add(v); v ← x)
  fi
rem() method r: ℕ₁
  return v
  if v ≠ 0 then v ← l!rem()
                if v = 0 then l ← nil fi
  fi
test(x: ℕ₁) method r: 𝔹
  if x = v then return true
  elif v = 0 ∨ x ≤ v then return false
  else yield l!test(x)
  fi
```

Figure 1: Example program *Sort*

sequential program is easy to specify and to develop by normal concepts of data reification and operation decomposition; the final concurrent code presented in Figure 1 is derived by a step of transformation. Of course, it is necessary to know that such transformation rules are correct in the sense that they preserve the intended behaviour of the methods of the class; in other words it is necessary to show under what circumstances it is valid to permute the return statements in methods and to substitute yield statements for return statements.

3 Basic mapping

This section (together with Sections 4 and 5) develops a mapping from $\pi o\beta\lambda$ to the version of the polyadic π-calculus given in Appendix B. In the spirit of Peter Landin's [Lan66] – π-calculus equivalents of increasingly complex $\pi o\beta\lambda$ texts are considered; a complete mapping function is given in Appendix C.

3.1 Representing Boolean values

Just as in the pure untyped λ-calculus, values such as Booleans and integers have to be constructed from the raw π-calculus. Here, processes are given for the Boolean values and expressions. The basic idea is simple – for example true located by some name b can be represented as $b(tf).\overline{t}$. So if v_b then P else Q could be represented as

$$\overline{b}(tf).(t().P + f().Q)$$

The process located at b is, however, *ephemeral* in that it can only be used once; replication can be employed to provide Boolean values.

$$Bool \stackrel{\text{def}}{=} !\, b_t(tf).\overline{t} \mid !\, b_f(tf).\overline{f}$$

This is a process which is run in parallel with the meaning of a family of classes and makes available names (b_t, b_f) for the two Boolean values. These values are *immutable*. Evaluation of (Boolean) expression $[\![e]\!]l$ locates the name of the appropriate Boolean value at l; thus, for example

$$
\begin{array}{l}
[\![\mathsf{true}]\!]l = \bar{l}b_t \\
[\![\mathsf{false}]\!]l = \bar{l}b_f \\
[\![c \wedge d]\!]l = (\boldsymbol{\nu} l_1 l_2)([\![c]\!]l_1 \mid [\![d]\!]l_2 \mid l_1(b_1).\overline{b_1}(tf).(t().l_2(b_2).\bar{l}b_2 + f().\bar{l}b_f))
\end{array}
$$

Notice that this coding relies on the fact that there are a finite number of Boolean values thus allowing (finite) sums to be used to distinguish values. This is one reason for handling natural numbers as $\pi o\beta\lambda$ classes in Section 4.3. (Access to variables, and assignment, are postponed to Section 4.1.)

3.2 Simple classes

Consider the simple $\pi o\beta\lambda$ class definition

c class
$m_1(x)$ method return
$m_2()$ method r: ref return self

The semantics must show that multiple instances of c can be created. This is modelled by replication to provide an *unbounded resource*

$$! \, I_c \tag{1}$$

A private name (u) is passed on a channel named c. Because of the output binding, the name is distinct for each instance.[2]

$$I_c = \bar{c}(u).B_u \tag{2}$$

The creation of new instances of c (new c) can be mapped to

$$c(u).\cdots u(\ldots)\cdots$$

The name u can be seen as a 'capability'[3]; the naming rules of the π-calculus ensure that different instances of c do not interfere with each other; interference can only occur when the owner of the name u shares it between parallel threads. (As in [Mil92b], replication could also be used to model the sort of *recursion* which unfolds a new body as it is required; here, recursion is used directly.) So, the model of each instance is (with B_u used recursively at the end of either method)

$$
\begin{array}{l}
B_u = \cdots . M_u \\
M_u = (\alpha_1(\ldots)\cdots . B_u + \alpha_2(\ldots)\cdots . B_u)
\end{array}
$$

[2] The definition B_u is introduced for reference; here – and in similar cases below – the binding of u includes the body of the definition which is considered as a syntactic expansion rule.

[3] On page 56 of 'Time Sharing Computer Systems' (MacDonald/Elsevier 1968), Maurice Wilkes attributes the term *capability* to Van Horn (1966) in 'Computer design for asynchronously reproducible multiprocessing', MAC-TR-34, M.I.T.

Bit class
vars $v\colon \mathbb{B} \leftarrow$ false
$w(x\colon \mathbb{B})$ method $v \leftarrow x$; return
$r()$ method return v

Figure 2: Example program *Bit*

The selection of methods is handled by passing two private start names (α_1, α_2) to the invoking process so that it can use the appropriate one. The ellipses above are completed as follows.

$$\begin{aligned} B_u &= \overline{u}(\alpha_1\alpha_2).M_u \\ M_u &= (\alpha_1(\omega_1 x).\overline{\omega_1}.B_u + \alpha_2(\omega_2).\overline{\omega_2}u.B_u) \end{aligned} \tag{3}$$

As well as any argument (e.g. x to m_1), each start name carries a termination name (ω_i) which is used to signal the end of the *rendezvous*. So the method call $u!m_1(e)$ is mapped to

$$u(\alpha_1\alpha_2).(\boldsymbol{\nu}\omega_1)(\overline{\alpha_1}\omega_1 e.\omega_1())$$

and $u!m_2()$ is mapped to

$$u(\alpha_1\alpha_2).(\boldsymbol{\nu}\omega_2)(\overline{\alpha_2}\omega_2.\omega_2(u'))$$

The mapping $[\![c]\!]$ gives the semantics for class c in the sense that the behaviour as seen by invoking expressions which are also mapped via $[\![\_]\!]$ must be reproduced by any implementation.

4 Handling instance variables

The preceding section has given the basic idea of the mapping but there is little real interest in classes which contain no local state. This section reviews what changes in the mapping are necessary when instance variables are added to classes and moves on to review the case where these instance variables can contain references.

4.1 Boolean instance variables

Consider the class *Bit* in Figure 2. It has methods which write (w) and read (r) an instance variable (v) which contains Boolean values. This can be modelled as in Equations 1 to 3 with an additional *stateful* process (V_y) for each instance variable. (The instance variable itself is like a class for which only one instance is required; this degenerate class has methods for access (a_v) and setting (s_v) whose interface is simple because they can only be invoked from one place.)

$$[\![Bit]\!] = \;!\, I_{Bit} \tag{4}$$
$$I_{Bit} = (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid \overline{bit}(u).B_u) \tag{5}$$
$$V_y = (\overline{a_v}y.V_y + s_v(z).V_z) \tag{6}$$
$$B_u = \overline{u}(\widetilde{\alpha}).M_u \tag{7}$$
$$M_u = (\alpha_w(\omega_w x).\overline{s_v}x.\overline{\omega_w}.B_u + \alpha_r(\omega_r).a_v(y).\overline{\omega_r}y.B_u) \tag{8}$$

Looking more closely at this use of the mapping function ($[\![\_]\!]$), its result – $!\,I_{Bit}$ – represents the semantics of *Bit* by requiring a particular behaviour over its interface (bit, u, α, ω); the replication provides any number of instances of the class; each such instance is identical (again, this fact is important in the following arguments) but has a unique name associated with it by the binding output $(\overline{bit}(u))$. The result of mapping a class has a process like V_y for each instance variable and one (B_u) for the body of the class; these communicate via strictly local names s_v to set and a_v to access the variable; variables are initialized. The body of the process B_u has one summand per method. Private names for the methods are communicated by $\overline{u}(\widetilde{\alpha})$ from the u instance. The ordering of the statements within each summand of M_u is shown here by prefixing; the mapping in Appendix C has to cope with statements whose mapping can yield a composition (the familiar baton passing trick is used); both forms of sequencing are subsumed below under P before Q. The *rendezvous* with a method terminates when the relevant ω prefix occurs as the mapping of the return statement. The fact that only one method can be active in any instance of *Bit* is regulated by the way the recursion on B_u works.

It is worth recording the SORTs (cf. [Mil92b]) for this definition:

$$bit\colon B, u\colon U, \alpha_w\colon A_w,\ \alpha_r\colon A_r,\ \omega_w\colon \Omega_w,\ \omega_r\colon \Omega_r,\ s_v\colon S_V,\ a_v\colon A_V$$

where

$$\left\{\begin{array}{l} B \mapsto (U),\, U \mapsto (A_w, A_r), \\ A_w \mapsto (\Omega_w, \mathbb{B}),\, A_r \mapsto (\Omega_r),\, \Omega_w \mapsto (),\, \Omega_r \mapsto (\mathbb{B}), \\ S_v \mapsto (\mathbb{B}),\, A_v \mapsto (\mathbb{B}) \end{array}\right\}$$

Code which invokes new *Bit* is mapped to

$$bit(u).\cdots u\cdots$$

Code to invoke $w(e)$ is mapped to prefixes which firstly obtain (private) names for both methods then select α_w and pass a private name for termination indication as well as a mapping of the parameter.

$$u(\widetilde{\alpha}).(\boldsymbol{\nu}\omega_w)(\overline{\alpha_w}\omega_w e.\omega_w())$$

The mapping of an invocation of $r()$ reflects the fact that there is no parameter but there is a value to be returned.

$$u(\widetilde{\alpha}).(\boldsymbol{\nu}\omega_r)(\overline{\alpha_r}\omega_r.\omega_r(y))$$

In general, variable access and assignment are mapped as follows

$$[\![v]\!]l \stackrel{\text{def}}{=} a_v(x).\overline{l}x$$
$$[\![v \leftarrow e]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu}l)([\![e]\!]l \mid l(b).\overline{s_v}b)$$

4.2 Reference values

As indicated above, $\pi o\beta\lambda$'s instance names are mapped to names in the π-calculus. Instance variables which contain references store instance names just as those in Section 4.1 contain the names of Boolean values. The only technicality which has to be addressed is that the nil reference indicates an uninitialized reference. This is handled here by using a completely private name ($(\boldsymbol{\nu} n)(L_n)$) whose use would cause the reductions to fail. (One could of course program some exception handling mechanism for richer languages than $\pi o\beta\lambda$.)

The way in which instance variables which contain references are mapped can be understood by comparing Figure 1 with the following.

$$
\begin{aligned}
&[\![Sort]\!] = !\, I_{Sort} \\
&I_{Sort} = (\boldsymbol{\nu}\tilde{s}\tilde{a})(V_0 \mid (\boldsymbol{\nu} n)(L_n) \mid \overline{sort}(u).B_u) \\
&V_y = (\overline{a_v} y.V_y + s_v(z).V_z) \\
&L_y = (\overline{a_l} y.L_y + s_l(z).L_z) \\
&B_u = \overline{u}(\tilde{\alpha}).M_u \\
&M_u = \left(\begin{array}{l} \alpha_a(\omega_a x).\cdots.a_l(u').u'(\tilde{\alpha'}).(\boldsymbol{\nu}\omega'_a)(\overline{\alpha'_a}\omega'_a x.\omega'_a()).\overline{\omega_a}.B_u \\ + \\ \alpha_r(\omega_r).\cdots \\ + \\ \alpha_t(\omega_t x).\cdots \end{array} \right)
\end{aligned}
$$

4.3 Natural numbers

Rather than write out long π-calculus expressions, the natural numbers are presented by $\pi o\beta\lambda$ classes whose mapping would yield the appropriate meaning.

Zero class
vars p: ref($Zero$); v: ref(Nat) $\leftarrow$ nil
$new(x)$ method $p \leftarrow$ self
$testz()$ method return true
$succ()$ method
  if $v =$ nil then $v \leftarrow$ new Nat(self) fi
  return v
$pred()$ method return p

Nat class
vars p: ref(Nat); v: ref(Nat) $\leftarrow$ nil
$new(x)$ method $p \leftarrow x$
$testz()$ method return false
$succ()$ method
  if $v =$ nil then $v \leftarrow$ new Nat(self) fi
  return v
$pred()$ method return p

Notice that *Zero* and *Nat* have to be regarded as having the same type.

4.4 Yield statements

The yield construct is handled by passing on the name (say, ω_t) to which the method containing the construct was to return its result. Thus the *test* method of *Sort* in Figure 1 (which uses a yield statement) is mapped to

$$M_u = \left(\begin{array}{l} \alpha_a(\omega_a x).\cdots \\ + \\ \alpha_r(\omega_r).\cdots \\ + \\ \alpha_t(\omega_t x).a_v(v).[\![x = v]\!].\left(\begin{array}{l} t().S_1 \\ + \\ f().[\![v = 0 \vee x \leq v]\!].(t().S_2 + f().S_3) \end{array} \right) \end{array} \right)$$

$$\begin{array}{l} S_1 = \overline{\omega_t}[\![\mathsf{true}]\!].B_u \\ S_2 = \overline{\omega_t}[\![\mathsf{false}]\!].B_u \\ S_3 = a_l(u').u'(\tilde{\alpha}').\overline{\alpha'_t}\omega_t x.B_u \end{array}$$

This releases the instance executing *test* before the answer is returned to its invoker. Whereas, *test* with a return statement would be mapped to

$$S'_3 = a_l(u').u'(\tilde{\alpha}').(\boldsymbol{\nu}\omega'_t)(\overline{\alpha'_t}\omega'_t x.\omega'_t(b)).\overline{\omega_t}b.B'_u$$

which causes the instance to wait for the result returned from the call to the l instance.

5 More about composite statements

So far, the sequential order of S_1; S_2 has been modelled by prefixing the π-calculus mapping of S_1 to that of S_2 (cf. Equation 8) but this idea will not cope with the case where S_1 has to be mapped to a composition. In the mapping in Appendix C a completion signal is sent from the π-calculus equivalent of S_1 which is composed with an expression which is guarded by that prefix. So $[\![S_1; S_2]\!]$ signals termination on l by

$$(\boldsymbol{\nu}l')([\![S_1]\!]l' \mid l'().[\![S_2]\!]l)$$

Skip statements are modelled by

$$[\![\mathrm{SKIP}]\!] \stackrel{\mathrm{def}}{=} \bar{l}.\mathbf{0}$$

A conditional statement if e then s_1 else s_2 fi can then be modelled by

$$(\boldsymbol{\nu}l'l_1l_2)(\mathit{BoolEval}(l', l_1, l_2) \mid [\![e]\!]l' \mid (l_1().[\![s_1]\!]l + l_2().[\![s_2]\!]l))$$

where

$$\mathit{BoolEval}(l', l_1, l_2) \stackrel{\mathrm{def}}{=} l'(b).\bar{b}(tf).(t().\overline{l_1} + f().\overline{l_2}) \tag{9}$$

A while statement while e do s od can be modelled by the recursive equation

$$W = (\boldsymbol{\nu}l'l_1l_2l'')(\mathit{BoolEval}(l', l_1, l_2) \mid [\![e]\!]l' \mid (l_1().[\![s]\!]l'' + l_2().\bar{l}) \mid l''().W)$$

In [Jon93c], a parallel construct is used in the $\pi o\beta\lambda$ design notation; this is not used in [Jon93a] and is not considered in Appendix C but it could be mapped in much the way that Milner discusses Section 8.3 of [Mil89]. (Recall that Milner explains there why adding a sequential composition operator to interleaving-based process calculi can complicate the algebra.)

6 A specific transformation

Many simple transformations such as removing temporary variables could be considered; in this paper attention is focused on transformations which affect concurrency. The first example considered is the relatively simple case of moving a return where no references are involved. For *Bit* as in Figure 2 and the result of mapping as in Equations 4–8; suppose the w method were transformed into

$$w(x\colon\mathbb{B})\ \textsf{method return}\ ;\ v \leftarrow x$$

Then it would be mapped to a π-calculus expression which only differs by the inversion of two prefixes: M_u of Equation 8 becomes

$$M'_u = (\alpha_w(\omega_w x).\boxed{\overline{\omega_w}.\overline{s_v}x}.B'_u + \cdots) \tag{10}$$

Consider the need to justify this transformation (i.e. showing that no $\pi o\beta\lambda$ system can detect which version of *Bit* is being used). A standard approach to equivalences in process algebras is to prove bi-similarity. Unfortunately, the two π-calculus expressions from the two versions of the method are not immediately bi-similar since prefixes can occur in different orders unless the context is restricted. Any claim of equivalence needs to reflect the context. The approach here is to argue that the two expressions eventually converge – under reduction – to the same process. Specifically, suppose the using code

$$l \leftarrow \textsf{new}\ Bit;\ l!w(\textsf{true});\ x \leftarrow l!r();\ \cdots$$

is mapped to

$$\begin{aligned}
P_0 &= bit(u).P_1\\
P_1 &= u(\widetilde{\alpha}).P_2\\
P_2 &= (\boldsymbol{\nu}\omega_w)(\overline{\alpha_w}\omega_w\, b_t.\omega_w()).P_3\\
P_3 &= u(\widetilde{\alpha}).P_4\\
P_4 &= (\boldsymbol{\nu}\omega_r)(\overline{\alpha_r}\omega_r.\omega_r(y)).P_5
\end{aligned}$$

where P_5 is not defined and just represents the rest of the process – it is assumed that no free occurrences of u occur in P_5. The names within P_i have been chosen so that they pun the names sent and received; although this is convenient, it must be remembered that the names are actually distinct. In fact, if the puns were not there one would have to use *PAR* steps (see Appendix B).

The fact that the distinction between M_u and M'_u makes no difference to the context can be shown by a series of π-calculus reductions. Using the version of the M_u process from Equation 8 first, the composition of the invocation P_0 with the denotation of the class is

$$P_0 \mid [\![Bit]\!] \mid Q$$

This reduces[4] in a number of $COMM'$ (see Appendix B) steps to

$$(\boldsymbol{\nu}u)\Big(P_1 \mid (\boldsymbol{\nu}s_v\, a_v)(\, V_{b_t} \mid B_u)\Big) \mid\, !\, I_{Bit} \mid Q \tag{11}$$

[4] See Appendix D for the detailed reductions.

where, u is local to $(P_1 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_t} \mid B_u))$ – but Q could create its own instances of *Bit*. Continuing the reductions one arrives at

$$(\boldsymbol{\nu} u)\Big((\boldsymbol{\nu}\omega_w)(\omega_w().P_3 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_t} \mid \overline{s_v} b_t.\overline{\omega_w}.B_u))\Big) \mid !\, I_{Bit} \mid Q \tag{12}$$

This is the point at which the distinction between Equations 8 and 10 becomes visible. Continuing the reductions in this case yields

$$(\boldsymbol{\nu} u)\Big(u(\widetilde{\alpha}).P_4 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_t} \mid (\boldsymbol{\nu}\widetilde{\alpha})(\overline{u}\widetilde{\alpha}.M_u))\Big) \mid !\, I_{Bit} \mid Q \tag{13}$$

Were the definition from Equation 10 used, Equation 12 would become

$$(\boldsymbol{\nu} u)\Big((\boldsymbol{\nu}\omega_w)(\omega_w().P_3 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_t} \mid \overline{\omega_w}.\overline{s_v} b_t.B_u))\Big) \mid !\, I_{Bit} \mid Q$$

which reduces to

$$(\boldsymbol{\nu} u)\Big(u(\widetilde{\alpha}).P_4 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_t} \mid \overline{s_v} b_t.B_u)\Big) \mid !\, I_{Bit} \mid Q \tag{14}$$

Intuitively it is easy to see that no further negative occurrence ($\overline{u}$) of u is available in Equation 14 until after the recursion triggers a further instance of B_u; the reductions converge to Equation 13 except for the original distinction between M_u and M'_u. This shows that the specific transformation preserves equivalence in this specific context. (Notice that $[\![b_t]\!]$ has been assumed to be *immutable*.)

7 Moving return statements

Of course, more general proofs are required. In the development of the example with which the paper began (*Sort* in Figure 1), the return statement is permuted with statements which cause non-trivial concurrent activity via references. The issue addressed in this section is the conditions under which it is valid to commute return statements with statements which invoke methods via (private) references. In order to see that the restriction to private references is crucial, contrast the two following examples (with *Bit* as in Figure 2).

Flipflop class
vars l: private ref $\leftarrow$ new *Bit*
    v: $\mathbb{B}$
$f()$ method $v \leftarrow l!r()$; $l!w(\neg\, v)$; return
$r()$ method return $l!r()$

In this case the reference l is marked private and, if the return statement were moved to the beginning of method f, no invocation would detect the difference: e.g.

$c \leftarrow$ new *Flipflop*; $c!f()$; $x \leftarrow c!r()$

If, however, a new method e were added which exposes the l reference, the situation is entirely different.

*Flipflop*2 class
vars l: shared ref $\leftarrow$ new *Bit*
    v: $\mathbb{B}$
$f()$ method $v \leftarrow l!r()$; $l!w(\neg\, v)$; return
$r()$ method return $l!r()$
$e()$ method return (l)

Firstly, notice that the fact that the reference l is copied forces it to be marked shared. Now, consider

$$c \leftarrow \mathsf{new}\ \mathit{Flipflop2};\ l \leftarrow c!e();\ c!f();\ x \leftarrow l!r()$$

If the return statement is moved to the beginning of method f in *Flipflop*2, the value of x would depend on the relative progress of the invoking and method code. It is therefore necessary to be precise about the conditions under which concurrency can be introduced by repositioning a return statement.

Transformation 1 *The relevant transformation rule is*

$S; \mathsf{return}\ e$ *can be replaced by* $\mathsf{return}\ e; S$

providing

1. *S always terminates;*[5]
2. *e is not affected by S; and*
3. *S only invokes methods reachable by private references.*

The example of a specific transformation is proved to give observational equivalence above by showing that the reductions of the mapped forms of both versions of the class eventually converge (under reduction) to equivalent processes. The argument which follows is more general but employs essentially the same idea which is that certain reductions can not interfere with each other. Notice however that this property is delicate in the π-calculus because it is not sufficient to know that $fn(P) \cap fn(Q) = \{\}$ to conclude that P and Q cannot affect each other's reductions: consider

$$\overline{x}y.y(z).\mathbf{0} \mid x'(y'').\overline{y''}z'$$

Although their free name sets are respectively $\{x, y\}$ and $\{x', z'\}$, these two terms interact if composed with the additional term

$$\cdots \mid x(y').\overline{x'}y'$$

As a basis for general argument, consider the following class.

D class
vars $\cdots$
$m_1(x)$ method S_1; S; return y; S_2
$m_2(x)$ method $\cdots$

[5]Termination is not, of course, a syntactically checkable property but it is in the spirit of the development method envisaged here that termination would anyway be proved. This point does however make it doubtful whether the kind of transformations being considered are suitable for automatic application by a compiler.

There is no loss of generality in considering a return with a variable (rather than an expression e) because a simple transformation rule could be used to assign e to y. The interest is in (non-trivial) activity in S.

The class D is mapped as follows.

$$
\begin{aligned}
[\![D]\!] &= !\,(\boldsymbol{\nu}\tilde{s}\tilde{a})(\cdots \mid \overline{d}(u).B_u)\\
B_u &= \overline{u}(\tilde{\alpha}).M_u\\
M_u &= (\alpha_1(\omega_1 x).[\![S_1]\!] \text{ before } [\![S]\!] \text{ before } a_y(y).\overline{\omega_1}y.[\![S_2]\!] \text{ before } B_u + \cdots)
\end{aligned}
$$

Notice that communication with any instance variables (shown by $\cdots$ in $[\![D]\!]$) is hidden by $(\boldsymbol{\nu}\tilde{s}\tilde{a})$; in fact, $fn([\![D]\!]) = \{d\}$.

It is a key point of arguments about observational equivalence at the $\pi o\beta\lambda$ level that methods are only invoked by π-calculus expressions which also result from the mapping $[\![\_]\!]$. Consider some context C which invokes methods of class D (assume, without loss of generality, that Q has no use of u).

$$C = d(u).P_1.Q \text{ before } P_i$$

where

$$P_i = u(\tilde{\alpha}).(\boldsymbol{\nu}\omega_i)(\overline{\alpha_i}\omega_i x.\omega_i(y))$$

The effect of new D is to obtain a unique name u which, by $COMM'$, becomes hidden from any other R.

$$C \mid [\![D]\!] \mid R \twoheadrightarrow (\boldsymbol{\nu}u)(P_1.Q \text{ before } P_i \mid (\boldsymbol{\nu}\tilde{s}\tilde{a})(\cdots \mid B_u)) \mid R$$

The invocation of m_1 causes further reduction to

$$
\begin{aligned}
\twoheadrightarrow (\boldsymbol{\nu}u)(\boldsymbol{\nu}\omega_1)(&\omega_1(y).Q \text{ before } P_i \mid\\
&(\boldsymbol{\nu}\tilde{s}\tilde{a})(\cdots \mid [\![S_1]\!] \text{ before } [\![S]\!] \text{ before } a_y(y).\overline{\omega_1}y.[\![S_2]\!] \text{ before } B_u)) \mid R
\end{aligned}
$$

Now $[\![S_1]\!]$ can change R (and local variables elided by $\cdots$) but so far the distinction between the two versions of D has not come into play and therefore the common reduction is to

$$
\begin{aligned}
\twoheadrightarrow (\boldsymbol{\nu}u)(\boldsymbol{\nu}\omega_1)(&\omega_1(y).Q \text{ before } P_i \mid\\
&(\boldsymbol{\nu}\tilde{s}\tilde{a})(\cdots \mid [\![S]\!] \text{ before } a_y(y).\overline{\omega_1}y.[\![S_2]\!] \text{ before } B_u)) \mid R'
\end{aligned}
$$

Even if new instances are created and their references stored in instance variables of D, their names are kept local.

At this point, the distinction which comes from commuting $[\![S]\!]$ and return(y) becomes important. The alternative to the preceding equation is as follows.

$$
\begin{aligned}
\twoheadrightarrow (\boldsymbol{\nu}u)(\boldsymbol{\nu}\omega_1)(&\omega_1(y).Q \text{ before } P_i \mid\\
&(\boldsymbol{\nu}\tilde{s}\tilde{a})(\cdots \mid a_y(y).\overline{\omega_1}y.[\![S]\!] \text{ before } [\![S_2]\!] \text{ before } B_u)) \mid R'
\end{aligned}
$$

Now, the second side condition on Transformation 1 ensures that S cannot affect y; therefore $[\![S]\!]$ cannot use $\overline{s_y}z$. The only other possible source of $\overline{s_y}z$ – since s_y and a_y are hidden – is another summand of B_u; but these are not available until a negative u is encountered after B_u recurses. Therefore, y is bound to the same name in Q before P_i whichever version of D is used providing $\overline{\omega_1}y$ occurs. The first side condition of Transformation 1 requires that S terminates so the ω_1 prefix will occur.

The other – more interesting – effect of commuting the return statement is that whereas $[\![S]\!]$ has to complete before Q before P_i, the repositioning of $\overline{\omega_1}y$ allows them to run concurrently. Firstly, notice that $[\![S]\!]$ actually only overlaps with Q since P_i cannot begin until a negative u is available and this only happens after B_u recurses. Secondly, note that the transformation allows all of the earlier reductions: if

$$([\![S]\!] \mid R') \twoheadrightarrow (\mathbf{0} \mid R'')$$

then

$$([\![S]\!] \ \mathsf{before}\ Q \mid R') \twoheadrightarrow (Q \mid R'')$$

but a possible reduction in the concurrent case is

$$([\![S]\!] \mid Q \mid R') \twoheadrightarrow (Q \mid R'')$$

What needs to be checked, of course, is whether any of the extra reductions of $[\![S]\!] \mid Q$ produce different $\pi o\beta\lambda$ behaviour. Several things make an argument by bi-simulation difficult here. Firstly, there is a technical problem in that $[\![S]\!]$ and Q do share names. This can be circumvented since the only names that they can share are those for constants (e.g. b_t) and class models. These are all immutable so there would appear to be no problem. Formally, it is a property of the π-calculus that

$$!\,P = \,!\,P \mid \,!\,P$$

The split replications can then be commuted and rebracketed so that $(\boldsymbol{\nu}\, b_t)$ etc. can be inserted to localize the communication. Notice this should be done for all class models because of possible indirect calls.

The other issue which would have to be addressed to formalize this proof in terms of bi-simulation is that there are still further steps in $[\![S]\!]$ which get merged with those of Q. As observed above, the names $\widetilde{s}, \widetilde{a}$ are local; these can however be used to access names of other processes (e.g. u') stored as references in instance variables. The third side condition of Transformation 1 ensures that no such name is shared. Therefore the u' remains local to the $(\boldsymbol{\nu}\, u')$ which created it: u' is not passed as an object name. The only name which affects the reduction of both invocation and D is then ω_1 and that serves to re-synchronize the reductions. Thus the two differing expressions converge under reduction.

8 Using yield statements

Transformation 2 *The transformation rule needed in developing Figure 1 for the introduction of yield statements is*

$\mathsf{return}\ l!m(x)$ *can be replaced by* $\mathsf{yield}\ l!m(x)$

providing

1. *$l!m()$ terminates; and*

2. l is a private reference and m only invokes methods reachable by private references.

Consider the following class.

E class
vars l: private ref
$m_1()$ method S_1; return $l!m_i()$; S_2
$m_2()$ method $\cdots$

This translates to

$$\begin{aligned}
[\![E]\!] &= !\,(\boldsymbol{\nu} s_l a_l)((\boldsymbol{\nu} n)L_n \mid \overline{e}(u).B_u) \\
L_y &= (\overline{a_l} y.L_y + s_l(z).L_z) \\
B_u &= \overline{u}(\tilde{\alpha}).M_u \\
M_u &= (\alpha_1(\omega_1).[\![S_1]\!] \text{ before } a_l(u').u'(\tilde{\alpha}').\overline{\alpha_i'}(\omega_i').\omega_i'(y).\overline{\omega_1}y.[\![S_2]\!] \text{ before } B_u + \cdots)
\end{aligned}$$

Whereas the translation when a yield statement is substituted for the return is

$$M_u' = (\alpha_1(\omega_1).[\![S_1]\!] \text{ before } a_l(u').u'(\tilde{\alpha}').\overline{\alpha_i'}(\omega_1).[\![S_2]\!] \text{ before } B_u' + \cdots) \tag{15}$$

The value which is returned over ω_1 is the same in both cases because the privacy of the references prevents any interference. This would be formalized as in the previous section by arguing that $\boldsymbol{\nu}$ localizes the immutable references.

Here again, the more interesting effect is that the process invoked by α_i' now runs concurrently with $[\![S_2]\!]$. But it can again be shown that they cannot interfere. It is also true that the recursion on B_u' occurs earlier than that on B_u because, in the former case, the delegation of the task of returning on ω_1 completes the m_1 method. This allows further negative occurrences of u and thus invocations of either m_1 or m_2. The privacy of names again removes the risk of interference.

There is however an additional issue – with yield statements – which becomes clear in the example of Figure 1: different uses of the *test* method can receive their results in orders which are not possible with the sequential code. It is intended that no $\pi o\beta\lambda$ program can be written which is sensitive to this difference but this conjecture has not yet been proved. Walker – in [Wal93a] – constructs a detailed argument for the special case shown in Appendix A.

9 Discussion

9.1 Further work

As discussed in [Jon92] some language issues remain to be resolved in $\pi o\beta\lambda$ itself. One radical alternative which might be considered is to follow the Smalltalk-80 idea of getting by without a while statement and using block statements in the language. This would obviate the need for the while statement itself and in some senses amounts to doing in $\pi o\beta\lambda$ what is already done here in the π-calculus. Another idea for an extension to the $\pi o\beta\lambda$ language is that it would be possible to give path expressions as constraints on the order in which methods can be invoked (for example a w must precede any sequence of w or r in *Bit* of Figure 2). As also discussed in [Jon92], $\pi o\beta\lambda$ needs an extension in order to handle situations where method calls cannot necessarily be accepted. An obvious extension would be to add a conditional method accept construct but it might also be possible to add a suspend idea or indeed the whole problem may be circumvented by some form of exceptions.

Much more work remains to be done. Although the arguments in Sections 7 are hopefully convincing, more formality would permit the use of mechanical proof tools which might be worthwhile as more proofs are needed. Moreover, the challenge of such formalization could yield new insights into notions of behavioural equivalences. As the logic used in [Jon93c] stabilizes, it will be necessary to undertake justification of its interference rules and this will require proofs about the relationship between logical expressions and $\pi o \beta \lambda$ statements. Furthermore, continuing work on general properties of rely/guarantee specifications (notably [Col93, CC93]) could force reconsideration of the approach taken in [Jon93c].

9.2 Related work

The reader is referred to [MPW92, Mil92b] for notes on the evolution of process algebras which permit name passing. Milner discusses useful examples of representing imperative languages in CCS already in [Mil89]. An early version of the mapping contained in this paper was nearly complete when [Wal91] was sent to the current author. In that paper, Walker maps POOL to the monadic version of the π-calculus but the paper had a stimulating effect on the mapping from $\pi o \beta \lambda$ to the polyadic π-calculus and resulted in a number of simplifications. A preliminary sketch of the mapping from $\pi o \beta \lambda$ to the polyadic π-calculus is given in [Jon92]. In [Wal93b], Walker, amongst other things, provides a mapping from POOL to the polyadic π-calculus and this again prompted revision of the mapping which is finally presented in [Jon93b] and repeated in Appendix C here.

Other researchers who have provided semantics for object-oriented languages based on process algebras include Honda and Tokoro [HT91b] (based on [HT91a]) and [Vaa90] (employs the process algebra known as ACP). Davide Sangiorgi's work on the higher-order π-calculus [San93b, San93a] provides strong arguments for using a higher-order calculus; [Wal93c] actually provides a mapping from $\pi o \beta \lambda$ to the higher-order π-calculus; and the proofs in [Wal93a] are based on this notion. The current author's concern is that the extra power of the higher-order calculus may make it more difficult to prove the sort of results which are needed here. As is seen in [Wal93a], it is in fact only necessary to use a second-order version of the π-calculus and even this is only really needed to provide a better model for value passing. It is still a research issue to establish whether the extra power actually makes the sort of proof considered in the current paper easier or not: it is a tenet of object-oriented thinking that everything is passed as a name.

David Walker in [Wal93a] has shown that two specific $\pi o \beta \lambda$ transformations *can* be proved weakly bi-similar; it is hoped to combine the formality of that proof with the generality of arguing about the transformation rules themselves in a future paper.

Another author who has investigated calculi which are suitable for object-oriented programming languages is Oscar Nierstrasz. His 'object calculus' in [Nie92] can be compared to the higher-order π-calculus. Further developments by the group at Keio University (e.g. [HY93]) on the ν-calculus could influence the approach to the required formalization of the proofs; furthermore [VH93] considers the problem of *principal types* in the π-calculus.

Acknowledgements

The author is grateful for financial support from the SERC who fund his Senior Research Fellowship.

Technical acknowledgements to people who have helped with the evolution of $\pi o \beta \lambda$ itself are contained in [Jon93a, Jon93c]. This report is based on two papers: [Jon93b] was specifically influenced by comments from Kohei Honda, Colin Stirling, Robin Milner, Davide Sangiorgi and above all by detailed reviews from David Walker. Valuable comments were received during seminars at IFIP WG 2.3, Oslo, Edinburgh and Geneva Universities. The presentation in [Jon94] was influenced by comments on an earlier version by David Walker and by discussions with Samson Abramsky and Colin Stirling (in Tokyo!). Final amendements were made to this report based on comments from Brian Monahan and discussions with David Walker, Benjamin Pierce and Davide Sangiorgi at the workshop on object-oriented language semantics organized by Erlangen University in September 1993.

References

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4), 1989.

[Ame91] P. America, editor. *ECOOP'91*, volume 612 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[AR89] Pierre America and Jan Rutten. *A Parallel Object-Oriented Language: Design and Semantic Foundations*. PhD thesis, Free University of Amsterdam, 1989.

[AR92] Pierre America and Jan Rutten. A layered semantics for a parallel object-oriented language. *Formal Aspects of Computing*, 4(4):376–408, 1992.

[Bae90] J. C. M. Baeten, editor. *Applications of Process Algebra*. Cambridge University Press, 1990.

[Bes93] E. Best, editor. *CONCUR'93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Bro89] Manfred Broy. On bounded buffers: Modularity, robustness, and reliability in reactive systems. Technical Report MIP-8920, Universitat Passau, Fakultat fur mathematik und Informatik, June 1989.

[BW90] J. C. M. Baeten and W. P. Weijland, editors. *Process Algebra*. Cambridge University Press, 1990.

[CC93] Pierre Collette and Antonio Cau. Parallel composition of assumption-commitment specifications, 1993. private communication.

[Col93] Pierre Collette. Application of the composition principle to Unity-like specifications. In *[GJ93]*, pages 230–242, 1993.

[dBdV91] J. W. de Bakker and E. P. de Vink. Rendez-vous with metric semantics. In *Proceedings of the REX Workshop on Semantics: Foundations and Applications*, volume 506 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[dR85] W. P. de Roever. The quest for compositionality: A survey of assertion-based proof systems for concurrent programs: Part I: Concurrency based on shared variables. In E. J. Neuhold and G. Chroust, editors, *Formal Models in Programming*. North-Holland, 1985.

[GJ93] M-C. Gaudel and J-P. Jouannaud, editors. *TAPSOFT'93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[HdR86] J. Hooman and W. P. de Roever. The quest goes on: a survey of proof systems for partial correctness of CSP. In J.W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, pages 343–395. Springer-Verlag, 1986. LNCS 224.

[Hen90] M. Hennessy. *The Semantics of Programming Languages*. John Wiley, 1990.

[HJ89] C. A. R. Hoare and C. B. Jones. *Essays in Computing Science*. Prentice Hall International, 1989.

[Hoa75] C. A. R. Hoare. Recursive data structures. *International Journal of Computer & Information Sciences*, 4(2):105–132, June 1975. see also, Chapter 14 of [HJ89].

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. see also, Chapter 16 of [HJ89].

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HT91a] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *[Ame91]*, pages 133–147, 1991.

[HT91b] K. Honda and M. Tokoro. A small calculus for concurrent objects. *ACM, OOPS Messenger*, 2(2):50–54, 1991.

[HY93] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics, 1993. private communication.

[IM91] T. Ito and A. R. Meyer, editors. *TACS'91 – Proceedings of the International Conference on Theoretical Aspects of Computer Science, Sendai, Japan*, volume 526 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Jon91] C. B. Jones. Interference resumed. In P. Bailes, editor, *Engineering Safe Software*, pages 31–56. Australian Computer Society, 1991.

[Jon92] C. B. Jones. An object-based design method for concurrent programs. Technical Report UMCS-92-12-1, Manchester University, 1992.

[Jon93a] C. B. Jones. Constraining interference in an object-based design method. In *[GJ93]*, pages 136–150, 1993.

[Jon93b] C. B. Jones. A pi-calculus semantics for an object-based design notation. In *[Bes93]*, pages 158–172, 1993.

[Jon93c] C. B. Jones. Reasoning about interference in an object-based design method. In *[WL93]*, pages 1–18, 1993.

[Jon94] C. B. Jones. Process algebra arguments about an object-based design notation. In A. W. Roscoe, editor, *Essays in Honour of C.A.R. Hoare*, chapter 14. Prentice-Hall, 1994.

[KMMN91] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Object oriented programming in the Beta programming language. Technical report, University of Oslo, September 1991.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[Mil80] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil92a] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mil92b] R. Milner. The polyadic π-calculus: A tutorial. In M. Broy, editor, *Logic and Algebra of Specification.* Springer-Verlag, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[Nel91] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[Nie92] Oscar Nierstrasz. Towards an object calculus. In M. Tokoro, O. Nierstrasz, and P.Wegner, editors, *ECOOP'91*, volume 612 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1992.

[San93a] D. Sangiorgi. From pi-calculus to higher-order pi-calculus – and back. In *[GJ93]*, pages 151–166, 1993.

[San93b] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, Edinburgh University, May 1993. printed as ECS-LFCS-93-266.

[Sar92] J. Sargeant. UFO – united functions and objects draft language description. Technical Report UMCS-92-4-3, Manchester University, 1992.

[Vaa90] F. W. Vaandrager. Process algebra semantics of POOL. In *[Bae90]*, pages 173–236, 1990.

[VH93] V. T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In *CONCUR'93*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[Wal91] D. Walker. π-calculus semantics for object-oriented programming languages. In *[IM91]*, pages 532–547, 1991.

[Wal93a] D. Walker. Algebraic proofs of properties of objects, July 1993. Draft paper, private communication.

[Wal93b] D. Walker. Objects in the π-calculus. *Information and Computation*, 1993. (to appear in).

[Wal93c] D. Walker. Process calculus and parallel object-oriented programming languages. In *International Summer Institute on Parallel Computer Architectures, Languages, and Algorithms, Prague*, 1993.

[WL93] J. C. P. Woodcock and P. G. Larsen, editors. *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[Wol88] Mario I. Wolczko. *Semantics of Object-Oriented Languages*. PhD thesis, Department of Computer Science, University of Manchester, 1988.

[Yon90] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.

A Further example

As indicated in [Wal93a], the situation with *Symtab* (for yield in particular) is more delicate than for the linear object graphs. Figure 3 is the outcome of a development in [Jon93a] from a simple specification describing an *abstract machine* for handling symbol tables which associate *Key*/*Data* pairs: the associations are created or modified by the *insert* method and used by the *search* method. The representation used in Figure 3 is a binary tree whose nodes are instances of the *Symtab* class. Each such instance – as well as storing a local *Key* (k) and *Data* (d) pair – has references to two sub-trees. These (l and r) references are marked private to indicate that they can not be be copied. This immediately ensures that the object-graph has no sharing: it is a tree. (It is interesting to compare this with [Hoa75]).

The semantics of $\pi o \beta \lambda$ require that at most one method can be active in each (object) instance at any one time. An interesting question is how this restriction admits concurrency (see [Ame89] for a review of the options). The intention of the conditional statement in the *insert* method should be obvious. But notice that this is preceded by a return statement. The effect of the return is to release the invoking code from the *rendezvous* and to permit execution of statements following the method call to overlap with that of the body of the *insert* method. Given the restriction on only one method being active per instance, a further invocation might be somewhat delayed. But notice that once the nested invocation (e.g. $l!insert(k', d')$) is released from its *rendezvous*, the first method can complete. In this way, a whole series of *insert* methods can be rippling down a (binary tree representation of a) symbol table. Achieving a similar effect for *search* (and an intermingling of the two activities) requires noticing that the invoking code must be held up until a value can be returned but that – if the task of returning that value is delegated – the instance first called can be made dormant and thus available for other method calls. This is the semantics chosen for the yield statement of $\pi o \beta \lambda$.

The class in Figure 3 is actually developed via the sequential version shown in Figure 4; the final step of the design process in [Jon93a] is to apply given $\pi o \beta \lambda$ transformation rules. One reason for introducing concurrency at the end of the development becomes obvious if the task of specifying – for example – the *insert* method of Figure 3 is considered: a post-condition alone will not suffice unless some form of auxiliary variable is used to fix the methods which are active on sub-trees when the method itself begins execution.[6] In contrast, the design in [Jon93a] is not only based on a simple pre/post condition specification, but even the initial design steps use standard sequential data reification and operation decomposition proofs.

[6] It would, of course, be possible to define the task in terms of streams in the style of [Bro89].

$Symtab$ class
vars k: $Key \leftarrow$ nil; d: $Data \leftarrow$ nil; l: private ref($Symtab$) $\leftarrow$ nil; r: private ref($Symtab$) $\leftarrow$ nil
$insert(k'$: Key, d': $Data)$ method
  return
  if $k =$ nil then $(k \leftarrow k'$; $d \leftarrow d')$
  elif $k' = k$ then $d \leftarrow d'$
  elif $k' < k$ then (if $l =$ nil then $l \leftarrow$ new $Symtab$ fi ; $l!insert(k', d')$)
  else (if $r =$ nil then $r \leftarrow$ new $Symtab$ fi ; $r!insert(k', d')$)
  fi
$search(k'$: $Key)$ method $Data$
  if $k = k'$ then return d
  elif $k' < k$ then yield $l!search(k')$
  else yield $r!search(k')$
  fi

Figure 3: Example program $Symtab$ (concurrent)

$Symtab$ class
vars k: $Key \leftarrow$ nil; d: $Data \leftarrow$ nil; l: private ref($Symtab$) $\leftarrow$ nil; r: private ref($Symtab$) $\leftarrow$ nil
$insert(k'$: Key, d': $Data)$ method
  if $k =$ nil then $(k \leftarrow k'$; $d \leftarrow d')$
  elif $k' = k$ then $d \leftarrow d'$
  elif $k' < k$ then (if $l =$ nil then $l \leftarrow$ new $Symtab$ fi ; $l!insert(k', d')$)
  else (if $r =$ nil then $r \leftarrow$ new $Symtab$ fi ; $r!insert(k', d')$)
  fi
  return
$search(k'$: $Key)$ method $Data$
  if $k = k'$ then return d
  elif $k' < k$ then return $l!search(k')$
  else return $r!search(k')$
  fi

Figure 4: Example program $Symtab$ (sequential)

B The π-calculus

Since the pioneering publications on CSP [Hoa78] and CCS [Mil80], many process algebras have been studied (e.g. [Hen90, BW90, Hoa85, Mil89]). However, the treatment of names in the π-calculus [MPW92] makes it an obvious candidate as a semantic base for object-oriented languages. The version of the π-calculus used here is a minor variant of the (first-order) polyadic π-calculus proposed in [Mil92b]: the only difference is that the decision to identify abstractions and concretions as separate phrases of the language has not been followed: the symmetry in [Mil92b] is pleasing but here there is little benefit in separating concretions – even for abstractions, it appears to fit better with object-oriented thinking to locate everything by name. Unlike [Wal93b, Mil92a], (binary) sums are employed here – the summands are always prefixed so normal processes are identified as a separate class.

The syntax of the calculus is very simple. Processes (typical elements P, Q) can be

$$P ::= N \;\Big|\; P \mid Q \;\Big|\; !\,P \;\Big|\; (\boldsymbol{\nu} x)P$$

Normal processes (typical elements M, N) can be

$$N ::= \pi.P \;\Big|\; \mathbf{0} \;\Big|\; M + N$$

Prefixes (typical element π) are

$$\pi ::= x(\tilde{y}) \;\Big|\; \overline{x}\tilde{y}$$

Typical elements for names here are x, y (names more closely linked to the objects being mapped are used below). Prefix and $\boldsymbol{\nu}$ bind more strongly than composition; sum binds weakest of all.

A number of abbreviations are useful. Trailing stop processes are omitted, so $\pi.\mathbf{0}$ can be written π. Multiple new names are combined, so $(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)$ is written $(\boldsymbol{\nu} xy)$. A sequence of names such as α_1, α_2 is sometimes abbreviated to $\tilde{\alpha}$. Recursive definitions are written with an obvious meaning; they can be viewed as an abbreviation of a 'baton' passing trick with replication. The parentheses on the input prefix $(x(\tilde{y}))$ should remind the reader that this – and of course $\boldsymbol{\nu}$ – serve to bind names whereas the basic output prefix $(\overline{x}\tilde{y})$ does not. There is however a convenient abbreviation with a binding form of the output prefix.[7]

$$\overline{x}(\tilde{y}).P \stackrel{\text{def}}{=} (\boldsymbol{\nu}\tilde{y})(\overline{x}\tilde{y}.P)$$

Structural equivalences can be defined. Alpha-convertible terms are taken to be structurally equivalent. Structural equivalence laws include the following (the first three rules for $+$ $(|)$ can be summarized by saying that $\langle M, +, \mathbf{0}\rangle$ and $\langle P, |, \mathbf{0}\rangle$ are symmetric monoids).

$$\begin{aligned} M + \mathbf{0} &\equiv M \\ M + N &\equiv N + M \\ M_1 + (M_2 + M_3) &\equiv (M_1 + M_2) + M_3 \\ M + M &\equiv M \end{aligned}$$

[7] This is not used in [Jon93b]. In fact, it would make the mapping somewhat simpler to employ finer binding distinctions as in [Wal91].

$$
\begin{array}{rcl}
P \mid \mathbf{0} & \equiv & P \\
P \mid Q & \equiv & Q \mid P \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R \\
!\,P & \equiv & P \mid !\,P \\
(\boldsymbol{\nu} x)\mathbf{0} & \equiv & \mathbf{0} \\
(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P & \equiv & (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P
\end{array}
$$

A function (*fn*) which yields the free names of a process can be defined.

$$
\begin{array}{rcl}
fn(P \mid Q) & = & fn(P) \cup fn(Q) \\
fn(\ !\,P) & = & fn(P) \\
fn((\boldsymbol{\nu} x)P) & = & fn(P) - \{x\} \\
fn(\mathbf{0}) & = & \{\} \\
fn(M + N) & = & fn(M) \cup fn(N) \\
fn(x(\tilde{y}).P) & = & \{x\} \cup (fn(P) - \{\tilde{y}\}) \\
fn(\overline{x}\tilde{y}.P) & = & \{x\} \cup \{\tilde{y}\} \cup fn(P) \\
fn(\overline{x}(\tilde{y}).P) & = & \{x\} \cup (fn(P) - \{\tilde{y}\})
\end{array}
$$

A similar function (bn) for bound names can be defined and $P\{\tilde{x}/\tilde{y}\}$ is the obvious syntactic substitution (with avoidance of accidental capture). The following equivalences also hold.

$$
\begin{array}{rcl}
(\boldsymbol{\nu} x)(P \mid Q) & \equiv & P \mid (\boldsymbol{\nu} x)Q \text{ if } x \notin fn(P) \\
(\boldsymbol{\nu} x)y(\tilde{z}).P & \equiv & y(\tilde{z}).(\boldsymbol{\nu} x)P \text{ where } x \neq y, x \notin \tilde{z} \\
(\boldsymbol{\nu} x)\overline{y}\tilde{z}.P & \equiv & \overline{y}\tilde{z}.(\boldsymbol{\nu} x)P \text{ where } x \neq y, x \notin \tilde{z} \\
(\boldsymbol{\nu} x)\pi.P & \equiv & \mathbf{0} \text{ if } \pi \text{ is } x(\tilde{y}) \text{ or } \overline{x}\tilde{y}
\end{array}
$$

The notion of *reduction* is key to the understanding of further equivalences. Reading $P \rightarrow Q$ as P can immediately reduce to Q, the following rules are taken from [Mil92b].

$$\boxed{COMM}\ \dfrac{}{(\cdots + \overline{x}\tilde{y}.P) \mid (x(\tilde{z}).Q + \cdots) \rightarrow P \mid Q\{\tilde{y}/\tilde{z}\}}$$

$$\boxed{PAR}\ \dfrac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$$

$$\boxed{RES}\ \dfrac{P \rightarrow P'}{(\boldsymbol{\nu} x)P \rightarrow (\boldsymbol{\nu} x)P'}$$

$$\boxed{STRUCT}\ \dfrac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

Notice that reduction is invalid under prefix or sum; there is also no rule given for reduction under replication but its effect can be simulated.

As in CCS, *bi-simulation* can be defined. But it is argued below that this does not hold for the examples in Sections 7 and 8. In arguments there, the reflexive, transitive closure of $\rightarrow$ is discussed: this is written $\twoheadrightarrow$. One rule for this can already be given: from *COMM* and *STRUCT* it follows that

$$\boxed{COMM'}\ \overline{\qquad (\cdots + \overline{x}(\tilde{y}).P) \mid (x(\tilde{z}).Q + \cdots) \twoheadrightarrow (\boldsymbol{\nu}\tilde{y})(P \mid Q\{\tilde{y}/\tilde{z}\})}$$

C Mapping

This appendix contains both an abstract syntax for $\pi o\beta\lambda$ and the formal mapping to the π-calculus. For ease of reading, concrete $\pi o\beta\lambda$ expressions are written as arguments to $[\![\_]\!]$. The abstract syntax of a *System* shows it to be a collection of named class definitions (*Cdef*).

$$System = Id \xrightarrow{m} Cdef$$

Its semantics is given by an $n+1$-fold composition

$$[\![c_1\text{: }\mathsf{class}\ C_1, \cdots, c_n\text{: }\mathsf{class}\ C_n]\!] \stackrel{\text{def}}{=} \prod_{i\in\{1,\ldots,n\}} [\![C_i]\!]c_i \mid Bool$$

Notice that the class name is needed as an argument to the mapping of the class body. (Strictly, there should be an argument – say ρ – which is a one:one mapping from $\{c_1, \ldots, c_n\}$ to a set of π-calculus *Names*; this is avoided here by assuming that c_i are *Names*.) A single class definition contains

$$\begin{array}{lll} Cdef :: & ivars & : Id \xrightarrow{m} Type \\ & mm & : Id \xrightarrow{m} Mdef \end{array}$$

Its semantics is given by

$$[\![\mathsf{vars}\ v_1\text{: }T_1; \cdots; v_n\text{: }T_n; M]\!]c \stackrel{\text{def}}{=} !\,(\boldsymbol{\nu}\tilde{s}\tilde{a})(\prod_{i\in\{1,\ldots,n\}} V_i \mid \overline{c}(u).[\![M]\!]u)$$

Types are defined as follows (notice that NAT is handled as in Section 4.3).

$$Type = \text{LOCALREF} \mid \text{SHAREDREF} \mid \text{BOOL}$$

Method definitions contain

$$\begin{array}{lll} Mdef :: & r & : [Type] \\ & pl & : (Id \times Type)^* \\ & b & : Stmt \end{array}$$

The semantics of the collection of such definitions is given by

$$[\![m_1(x)\ \mathsf{method}\ S_1, \cdots, m_n(x)\ \mathsf{method}\ S_n]\!] \stackrel{\text{def}}{=} B_u$$

which is an n-fold (prefixed) sum

$$B_u = \overline{u}(\tilde{\alpha}).(\boldsymbol{\nu} l)((\sum_{i\in\{1,\ldots,n\}} \alpha_i(\omega_i x).\overline{s_x}x.[\![S_i]\!]lu\omega_i) \mid l().B_u)$$

The mapping of a statement requires a termination label l, the unique name of the current activation u (for self), and an appropriate termination indicator ω_i. The different forms of statement are defined

$$Stmt = Mref \mid Assign \mid Compound \mid \textsc{Skip} \mid If \mid While \mid Return \mid Yield$$

The syntax and mapping of these follow.

$$\begin{aligned} Mref \;::\; & cn \;:\; Expr \\ & mn \;:\; Id \\ & al \;:\; Expr^* \end{aligned}$$

$$\begin{aligned} &[\![e_1!m(e_2)]\!]lu\omega \stackrel{\text{def}}{=} \\ &\quad (\boldsymbol{\nu} l'l'')([\![e_1]\!]l'u \mid [\![e_2]\!]l''u \mid l'(u').u'(\tilde{\alpha}).(\boldsymbol{\nu}\omega_m)(l''(x).\overline{\alpha_m}\omega_m x.\omega_m().\overline{l})) \end{aligned}$$

$$\begin{aligned} Assign \;::\; & lhs \;:\; Id \\ & rhs \;:\; Expr \end{aligned}$$

$$[\![v \leftarrow e]\!]lu\omega \stackrel{\text{def}}{=} (\boldsymbol{\nu} l')([\![e]\!]l'u \mid \overline{s_v}l'.\overline{l})$$

$$Compound \;::\; sl \;:\; Stmt^*$$

$$[\![s_1;\cdots;s_n]\!]lu\omega \stackrel{\text{def}}{=} (\boldsymbol{\nu} l_1\ldots l_{n-1})([\![s_1]\!]l_1 u\omega \mid \cdots \mid l_{n-1}().[\![s_n]\!]lu\omega)$$

$$[\![\textsc{Skip}]\!]lu\omega \stackrel{\text{def}}{=} \overline{l}.\mathbf{0}$$

$$\begin{aligned} If \;::\; & b \;:\; Expr \\ & th \;:\; Stmt \\ & el \;:\; Stmt \end{aligned}$$

$$\begin{aligned} &[\![\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi}]\!]lu\omega \stackrel{\text{def}}{=} \\ &\quad (\boldsymbol{\nu} l'l_1l_2)(BoolEval(l', l_1, l_2) \mid [\![e]\!]l'u \mid (l_1().[\![s_1]\!]lu\omega + l_2().[\![s_2]\!]lu\omega)) \end{aligned}$$

Where $BoolEval$ is as Equation 9 of Section 5.

$$\begin{aligned} While \;::\; & b \;:\; Expr \\ & s \;:\; Stmt \end{aligned}$$

$$\begin{aligned} &[\![\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}]\!]lu\omega \stackrel{\text{def}}{=} W \\ &W = (\boldsymbol{\nu} l'l_1l_2l'')(BoolEval(l', l_1, l_2) \mid [\![e]\!]l'u \mid (l_1().[\![s]\!]l''u\omega + l_2().\overline{l}) \mid l''().W) \end{aligned}$$

$$Return \;::\; r \;:\; [Expr]$$

$$[\![\mathsf{return}\ ]\!] lu\omega \stackrel{\mathrm{def}}{=} \overline{\varpi}.\overline{l}$$

$$[\![\mathsf{return}\ e]\!] lu\omega \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} l')([\![e]\!] l'u \mid l'(r).\overline{\varpi}r.\overline{l})$$

$$\mathit{Yield} :: r : \mathit{Mref}$$

$$[\![\mathsf{yield}\ e_1!m(e_2)]\!] lu\omega \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} l'l'')([\![e_1]\!] l'u \mid [\![e_2]\!] l''u \mid l'(u').u'(\widetilde{\alpha}).l''(x).\overline{\alpha_m}\omega x.\overline{l})$$

Expressions can be of the following forms.

$$\mathit{Expr} = \mathit{New} \mid \mathit{Mref} \mid \textsc{Nil} \mid \textsc{Self} \mid \mathit{Compare} \mid \mathit{And} \mid \mathit{Id} \mid \mathbb{B}$$

The mapping of expressions requires a label l for the name of the result, and the unique name of the current activation u (for self).

$$\begin{aligned} \mathit{New} :: \ & cn : \mathit{Id} \\ & al : \mathit{Expr}^* \end{aligned}$$

The new construct has to invoke the new method of the instance.

$$[\![\mathsf{new}\ c]\!] lu \stackrel{\mathrm{def}}{=} c(u).u(\widetilde{\alpha}).(\boldsymbol{\nu}\omega_n)(\overline{\alpha_n}\omega_n.\omega_n().\overline{l}u)$$

$$\begin{aligned} & [\![e_1!m(e_2)]\!] lu \stackrel{\mathrm{def}}{=} \\ & \quad (\boldsymbol{\nu} l'l'')([\![e_1]\!] l'u \mid [\![e_2]\!] l''u \mid l'(u').u'(\widetilde{\alpha}).(\boldsymbol{\nu}\omega_m)(l''(x).\overline{\alpha_m}\omega_m x.\omega_m(r).\overline{l}r)) \end{aligned}$$

$$[\![\mathsf{nil}]\!] lu \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu} n)(\overline{l}n)$$

$$[\![\mathsf{self}]\!] lu \stackrel{\mathrm{def}}{=} \overline{l}u$$

$$\begin{aligned} \mathit{Compare} :: \ & e1 : \mathit{Expr} \\ & e2 : \mathit{Expr} \end{aligned}$$

$$\begin{aligned} & [\![e_1 = e_2]\!] lu \stackrel{\mathrm{def}}{=} \\ & \quad (\boldsymbol{\nu} l_1 l_2)([\![e_1]\!] l_1 u \mid [\![e_2]\!] l_2 u \mid l_1(b_1).(\boldsymbol{\nu} tf)(\overline{b_1}tf.(t().\cdots + f().\cdots))) \end{aligned}$$

$$\begin{aligned} \mathit{And} :: \ & e1 : \mathit{Expr} \\ & e2 : \mathit{Expr} \end{aligned}$$

$$\begin{aligned} & [\![e_1 \wedge e_2]\!] lu \stackrel{\mathrm{def}}{=} \\ & \quad (\boldsymbol{\nu} l_1 l_2)([\![e_1]\!] l_1 u \mid [\![e_2]\!] l_2 u \mid l_1(b_1).\overline{b_1}(tf).(t().l_2(b_2).\overline{l}b_2) + f().\overline{l}b_f) \end{aligned}$$

$$[\![v]\!] lu \stackrel{\mathrm{def}}{=} a_v(x).\overline{l}x$$

$$[\![\mathsf{true}]\!] l \stackrel{\mathrm{def}}{=} \overline{l}b_t$$
$$[\![\mathsf{false}]\!] l \stackrel{\mathrm{def}}{=} \overline{l}b_f$$

D Detailed reductions of *Bit*

This appendix shows the detail of the reductions in Section 6.

$$
\begin{array}{l}
P_0 \mid [\![Bit]\!] \mid Q \\
\quad \rightarrow P_0, [\![Bit]\!] \\
bit(u).P_1 \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow\, ! \\
bit(u).P_1 \mid I_{Bit} \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow I_{Bit} \\
bit(u).P_1 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid \overline{bit}(u).B_u) \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow COMM'
\end{array}
$$

$$
(\boldsymbol{\nu} u)\Big(P_1 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid B_u)\Big) \mid !\, I_{Bit} \mid Q \tag{16}
$$

In Equation 16, u is local to $(P_1 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid B_u))$ but Q could create its own instances of *Bit*. Continuing the reductions

$$
\begin{array}{l}
\quad \rightarrow P_1, B_u \\
(\boldsymbol{\nu} u)\Big(u(\widetilde{\alpha}).P_2 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid \overline{u}(\widetilde{\alpha}).M_u)\Big) \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow COMM' \\
(\boldsymbol{\nu} u\widetilde{\alpha})\Big(P_2 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid M_u)\Big) \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow P_2, M_u \\
(\boldsymbol{\nu} u\widetilde{\alpha})\Big((\boldsymbol{\nu}\omega_w)(\overline{\alpha_w}\omega_w b_t.\omega_w()).P_3 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid (\alpha_w(\omega_w x).\overline{s_v}x.\overline{\omega_w}.B_u + \cdots))\Big) \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow COMM
\end{array}
$$

$$
(\boldsymbol{\nu} u)\Big((\boldsymbol{\nu}\omega_w)(\omega_w().P_3 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid \overline{s_v}b_t.\overline{\omega_w}.B_u))\Big) \mid !\, I_{Bit} \mid Q \tag{17}
$$

Equation 17 is the point at which the distinction between M_u (of Equation 8) and M'_u (of Equation 15) becomes visible. Continuing the reductions

$$
\begin{array}{l}
\quad \rightarrow V_y \\
(\boldsymbol{\nu} u)\Big((\boldsymbol{\nu}\omega_w)(\omega_w().P_3 \mid (\boldsymbol{\nu} s_v a_v)((\cdots + s_v(z).V_z) \mid \overline{s_v}b_t.\overline{\omega_w}.B_u))\Big) \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow COMM \\
(\boldsymbol{\nu} u)\Big((\boldsymbol{\nu}\omega_w)(\omega_w().P_3 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{t}}} \mid \overline{\omega_w}.B_u))\Big) \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow COMM \\
(\boldsymbol{\nu} u)\Big(P_3 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{t}}} \mid B_u)\Big) \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow P_3, B_u
\end{array}
$$

$$
(\boldsymbol{\nu} u)\Big(u(\widetilde{\alpha}).P_4 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{t}}} \mid \overline{u}(\widetilde{\alpha}).M_u)\Big) \mid !\, I_{Bit} \mid Q \tag{18}
$$

below the reductions are continued (just to give a better feel for the semantics); here return to Equation 17 (where the permuted prefixes are first visible), M'_u would give

$$
\begin{array}{l}
(\boldsymbol{\nu} u)\Big((\boldsymbol{\nu}\omega_w)(\omega_w().P_3 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid \overline{\omega_w}.\overline{s_v}b_t.B'_u))\Big) \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow COMM \\
(\boldsymbol{\nu} u)\Big(P_3 \mid (\boldsymbol{\nu} s_v a_v)(V_{b_{\mathrm{f}}} \mid \overline{s_v}b_t.B'_u)\Big) \mid !\, I_{Bit} \mid Q \\
\quad \rightarrow P_3
\end{array}
$$

$$(\boldsymbol{\nu} u)\Big(u(\widetilde{\alpha}).P_4 \mid (\boldsymbol{\nu} s_v a_v)(\,V_{b_t} \mid \overline{s_v} b_t.B'_u)\Big) \mid !\, I_{Bit} \mid Q \tag{19}$$

Intuitively it is easy to see that no further negative occurrence of u is available in Equation 19 until after the recursion triggers a further instance of B_u; the reductions continue

$$\begin{aligned}
&\quad \to V_y \\
&(\boldsymbol{\nu} u)\Big(u(\widetilde{\alpha}).P_4 \mid (\boldsymbol{\nu} s_v a_v)((\cdots + s_v(z).\,V_z) \mid \overline{s_v} b_t.B'_u)\Big) \mid !\, I_{Bit} \mid Q \\
&\quad \to \mathit{COMM} \\
&(\boldsymbol{\nu} u)\Big(u(\widetilde{\alpha}).P_4 \mid (\boldsymbol{\nu} s_v a_v)(\,V_{b_t} \mid B'_u)\Big) \mid !\, I_{Bit} \mid Q \\
&\quad \to B_u
\end{aligned}$$

$$(\boldsymbol{\nu} u)\Big(u(\widetilde{\alpha}).P_4 \mid (\boldsymbol{\nu} s_v a_v)(\,V_{b_t} \mid \overline{u}(\widetilde{\alpha}).M'_u)\Big) \mid !\, I_{Bit} \mid Q \tag{20}$$

Equations 18 and 20 are identical except for the original distinction between M_u and M'_u and therefore the permutation preserves equivalence.

Notice that $[\![b_t]\!]$ has been assumed to be *immutable*.

Continuing reduction steps from Equation 18 gives a feel for the overall semantics of method invocation.

$$\begin{aligned}
&\quad \to \mathit{COMM}' \\
&(\boldsymbol{\nu} u\widetilde{\alpha})\Big(P_4 \mid (\boldsymbol{\nu} s_v a_v)(\,V_{b_t} \mid M_u)\Big) \mid !\, I_{Bit} \mid Q \\
&\quad \to P_4, M_u \\
&(\boldsymbol{\nu} u\widetilde{\alpha})\Big((\boldsymbol{\nu}\omega_r)(\overline{\alpha_r}\omega_r.\omega_r(y)).P_5 \mid (\boldsymbol{\nu} s_v a_v)(\,V_{b_t} \mid (\cdots + \alpha_r(\omega_r).a_v(y).\overline{\omega_r} y.B_u))\Big) \mid !\, I_{Bit} \mid Q \\
&\quad \to \mathit{COMM} \\
&(\boldsymbol{\nu} u)\Big((\boldsymbol{\nu}\omega_r)(\omega_r(y).P_5 \mid (\boldsymbol{\nu} s_v a_v)(\,V_{b_t} \mid a_v(y).\overline{\omega_r} y.B_u))\Big) \mid !\, I_{Bit} \mid Q \\
&\quad \to V_y \\
&(\boldsymbol{\nu} u)\Big((\boldsymbol{\nu}\omega_r)(\omega_r(y).P_5 \mid (\boldsymbol{\nu} s_v a_v)((\overline{a_v} b_t.\,V_{b_t} + \cdots) \mid a_v(y).\overline{\omega_r} y.B_u))\Big) \mid !\, I_{Bit} \mid Q \\
&\quad \to \mathit{COMM} \\
&(\boldsymbol{\nu} u)\Big((\boldsymbol{\nu}\omega_r)(\omega_r(y).P_5 \mid (\boldsymbol{\nu} s_v a_v)(\,V_{b_t} \mid \overline{\omega_r} b_t.B_u))\Big) \mid !\, I_{Bit} \mid Q \\
&\quad \to \mathit{COMM} \\
&(\boldsymbol{\nu} u)\Big(P_5\{b_t/y\}) \mid (\boldsymbol{\nu} s_v a_v)(\,V_{b_t} \mid B_u)\Big) \mid !\, I_{Bit} \mid Q \\
&\quad \to (\text{because } u \notin \mathit{fn}(P_5))) \\
&P_5\{b_t/y\} \mid !\, I_{Bit} \mid Q
\end{aligned}$$