Department of Computer Science
University of Manchester

Manchester M13 9PL, England

Technical Report Series

UMCS–92–12–1

C. B. Jones

An Object-Based Design Method for Concurrent Programs

An Object-Based Design Method for Concurrent Programs

C. B. Jones*

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

1992-12-04

Abstract

The property of a (formal) development method which gives the development process the potential for productivity is compositionality. Interference is what makes it difficult to find compositional development methods for concurrent systems. This paper is intended to contribute to tractable development methods for concurrent programs. In particular it explores ways in which object-based language concepts can be used to provide a compositional development method for concurrent programs. This text summarizes results from three draft papers. It firstly shows how object-based concepts can be used to provide a designer with control over interference and proposes a transformational style of development (for systems with limited interference) in which concurrency is introduced only in the final stages of design. The essential idea here is to show that certain object graphs limit interference. Secondly, the paper shows how a suitable logic can be used to reason about those systems where interference plays an essential role. Here again, concepts are used in the design notation which are taken from object-oriented languages since they offer control of granularity and way of pinpointing interference. Thirdly, the paper outlines the semantics of the design notation mapping its constructs to Milner's π-calculus.

Technical Reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from m1.cs.man.ac.uk (130.88.13.4) 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

The most difficult aspect of finding tractable development methods for concurrent systems is to provide a useful notion of compositionality which facilitates division of work. Compositionality can be defined as follows (adapted from [Zwi88])

> A development method is *compositional* if the fact that a design step satisfies a given specification can be justified on the basis of the specifications of any constituent components without knowledge of their interior construction

Earlier work on shared-variable concurrency (see [Jon83a] which is significantly extended in [Stø90, Stø91a, Stø91b]) used rely and guarantee conditions both to describe and to reason about *interference*. The fixed format of these specifications was rejected in [Jon91a] in favour of a logic with operators which use predicates of pairs of states (there are similarities with Lamport's TLA [Lam90, Lam91]). But the proofs remain long-winded and the earlier work has been dogged by issues like atomicity (granularity) and questions about where invariants etc. are supposed to hold.

In common with many others, the current author sees language restrictions as a way of constraining concurrency so as to reduce the number of proof obligations in development. The current approach uses concepts of object-oriented languages in order to constrain interference and fix a level of granularity. (The idea to use object-oriented languages was made more tempting by the positive experience of building a theorem proving assistant [JJLM91] in Smalltalk and more recent discussions about exploiting parallel hardware and tackling a multi-user version of *mural*.) It is not, however, the aim to add yet one more language to those claiming to be object-oriented; the development method envisaged here ought be used for programs in languages such as ABCL [Yon90], Modula-3 [Nel91], Beta [KMMN91] or UFO [Sar92]. The claim is that some carefully chosen subset of object-oriented concepts makes the design of concurrent programs more tractable than in arbitrary shared-variable languages (or even languages like CSP). The move to an object-based language has not made the interference logic redundant it has only reduced the need for interference arguments; Sections 4 to 7 explore the situation where interference is essential.

The design notation used in this paper (known as $\pi o\beta\lambda$) is heavily influenced by the programming language 'POOL' (see Appendix A.1 for references and some comparative notes); it also reflects discussions with colleagues at Manchester University. Most of the features of the language are presented by examples. Points of interest include the following. *Classes* have *methods* only one of which may be active at any one time (for a particular instance); invocation of methods is synchronous but methods can return before they complete and this releases the invoking process from the *rendezvous*. Consider the following

```
Priq class
vars m: [ℕ] ← nil; l: private ref(Priq) ← nil
add(e: ℕ) method
  return
  if m = nil then (m ← e; l ← new Priq)
  elif m < e then l!add(e)
  else (l!add(m); m ← e)
  fi
rem() method r: ℕ
  return m
  if m≠nil then m ← l!rem()
                if m = nil then l ← nil
                fi
  fi
```

This can be read as an object-oriented program (which is actually developed from a specification in Section 2). The programming task which is considered concerns sorting: a priority queue delivers – and removes – its smallest value via a remove method (*rem*); new values can be added by another method (*add*). Programs obtain a reference to (an instance of) a priority queue by using a new *Priq* statement. In fact, the created queue can be a linked list of instances of *Priq* but the using program would have no way of detecting this. Each instance has two variables containing a value and a link (possibly nil) to the next element.

In the class *Priq*, the *new* method is implicit; all that happens is that the instance variables (m and l) is initialized. Once created, there are two methods which can be invoked in an instance of the *Priq* class: *add* puts its argument into the queue and *rem* – which takes no arguments – returns the smallest value contained in the queue. Methods are invoked by expressions like $l!add(7)$ (where l is a reference to an instance of *Priq*). The semantics dictates that only one method can be active at any time in a particular instance of *Priq*.[1] Notice that the return statements occur at the the beginning of the *add* and *rem* methods. This releases the user from the *rendezvous* and lets the remaining code run in parallel with other activity of the invoking program. Furthermore, once – say – the call to the next *add* has been released, the method terminates and its instance is available for other method calls. One can picture a whole sequence of *add* and *rem* methods rippling along the linked-list structure. The fact that the activity can never get out of order is important and results from the object graph which is created. Marking the contained references as private makes it easier to establish results about the object graphs. Were $\pi o\beta\lambda$ a programming language, all sorts of concrete syntax details would have to be resolved – here, a rather relaxed syntax is used with line breaks playing a meaningful part. (The abstract syntax of the language used here is given in Appendix A.2.) The reader should remember that $\pi o\beta\lambda$ is intended as a design notation to be used to develop programs in a language where issues like parsing have received due attention.

In addition to the return statement, there is a yield statement which provides a way of delegating the responsibility to answer a method invocation. As in *Priq*, objects (instances of classes) are created by activating new for a class name; in $\pi o\beta\lambda$ explicit methods for new can be written. The language has no inheritance yet (it is tempting to try something like 'theory morphisms' – cf. [JJLM91] – because inheritance is often used to solve too many problems at once).

[1] It can be useful to think of classes as blocks which can be multiply instantiated; each instance has local (instance) variables and procedures (methods); the instance variables can only be accessed or changed by the methods; methods are called (invoked) by sending messages.

In addition to the language presentation herein, it is to some extent true that the search for a development method has been driven by examples: the approach has been to find plausible development steps and then to look for formal rules which justify them. This is largely motivated by the experience which shows that the thing which makes formal development work like mathematics is finding the right steps of development; detailing the proofs of individual steps is less rewarding. One key insight was the realization that assertions (invariants etc.) about the object graphs created by object references are central to the explanation of many algorithms. This first part of this paper looks at two topologies in Sections 2 and 3 which both support a 'promotion' of properties about instances to properties about collections of instances. This can be compared with the way in which an inference rule for a while statement can be used to infer results about a composite statement from properties of its components.

Section 5 also shows the sort of transformational development – usable on simple object graphs – which is discussed above but Section 7 tackles the problem of interference when such simple object graphs do not suffice. Section 6 discusses the logic used.

There are at least two options for giving the semantics: a resumption semantics which fits the way methods work here (cf. [AR89, pp111]; see also [Wol88, AR92]) or mapping to Milner's Polyadic π-calculus [Mil92]. Since the mapping to the π-calculus is quite far advanced, the working name for the design notation is $\pi o\beta\lambda$. (see Appendix C).

2 Linked-lists of objects

The first example illustrates the object-based nature of the programming language and the role that this plays in developing programs. What follows is a step-wise development of a program which stores each element of the queue as a local variable in an instance of an object; these objects are organized into a linked-list. Because the specifications are simpler, the first steps of development assume sequential execution within a queue (there might – however – be other concurrent threads); concurrency within a queue is considered in the final development step where its use is justified by arguing that it provides the same visible behaviour as the sequential implementation.

Specification

As in a Larch [GHW85, GH93] 'interface language', the design notation is used here to provide a framework for the specification which is given as a class definition. The methods are specified by pre- and post-conditions in a style similar to that used in VDM [Jon90].[2] In post-conditions, hooked identifiers refer to the value of the instance variables before execution of the method and undecorated identifiers refer to the values after execution of the method. Thus

$$b = \overleftarrow{b} \cup \{e\}$$

requires that the value of the instance variable b after an invocation of *add* is the bag union of the value of that variable before execution of the method with a unit bag containing the value of the parameter. Notice that *rem* is a partial method and – as in VDM – the post-condition can be undefined if its pre-condition is not satisfied. (The external clauses from VDM are barely necessary in the context of a class but there are places where one really ought note that some variables are read-only.) Values of type bag etc. and operators like $\cup$ are part of the specification language.

[2]The classes here can be compared with modules in VDM-SL [BSI92, Daw91].

$Priq$ class
vars b: $\mathbb{N}$-bag $\leftarrow \{\,\}$
$add(e$: $\mathbb{N})$ method
  post $b = \overleftarrow{b} \cup \{e\}$
$rem()$ method r: $\mathbb{N}$
  pre $b \neq \{\,\}$
  post $r = min(\overleftarrow{b}) \wedge b = \overleftarrow{b} - \{r\}$

Just as in VDM, ‘satisfiability’ proof obligations can be generated for each method specification.

Straightforward data reification

It is possible to undertake a step of data reification of the bag b to an ascending sequence. Such a step is sketched here in order to afford comparison with the reification to a linked-list which follows. The objects concerned are

$AscSeq = \mathbb{N}^*$
inv $(b) \triangleq$ *is-ascending*(b)

The invariant is a restriction on the elements which are in the set *AscSeq* (*is-ascending* – and other simple functions – are taken to be obvious).[3]

The relationship between this representation and the abstract objects is defined

$$retr : AscSeq \to \mathbb{N}\text{-}Bag$$

$$retr(b) \quad \triangleq \quad bagof(b)$$

$$bagof : X^* \to X\text{-}Bag$$

$$bagof(t) \quad \triangleq \quad \{e \mapsto \mathsf{card}\,\{i \in \mathsf{inds}\, t \mid t(i) = e\} \mid e \in \mathsf{elems}\, t\}$$

This representation is ‘adequate’ (there is at least one element of *AscSeq* which corresponds – under *retr* – to each element of $\mathbb{N}$-bag). The methods of *Priq* can be specified on this representation as follows.

$Priq$ class
vars b: $AscSeq \leftarrow [\,]$
$add(e$: $\mathbb{N})$ method
  post $\exists i \in \mathsf{inds}\, b \cdot b(i) = e \wedge del(b, i) = \overleftarrow{b}$
$rem()$ method r: $\mathbb{N}$
  pre $b \neq [\,]$
  post $r = \mathsf{hd}\,\overleftarrow{b} \wedge b = \mathsf{tl}\,\overleftarrow{b}$

$$del(t, i) \quad \triangleq \quad t(1, \ldots, i-1) \frown t(i+1, \ldots, \mathsf{len}\, t)$$

[3]Throughout this paper, VDM notation [Jon90] is used for sequences, maps etc.

The correctness of such a step can be justified by further rules (operation domain/result) of [Jon90].

It is worth taking this opportunity to reflect on where the invariant must hold: a user would presumably accept an implementation of *add* which put new elements at the end of a list and then sorted it. In this view, an invariant does not have to be true mid-operation: it is really a way of abbreviating pre-/post-conditions. It would be possible to develop from here a sequential implementation using decomposition rules to justify the use of while statements etc.

Reification involving class instances

The main line of object-based development is now considered (i.e. the reification to *AscSeq* is ignored and the reference point for this step is the initial specification). Here again, a reification focuses on the development of the data structure and finding an appropriate invariant is a key to the design. This development step employs multiple instances of class *Priq*; their local variables (m) collectively represent b; the instances form a linked-list with the l variable in one instance pointing to the next. The use of references necessitates talking about a global state ($\sigma \in \Sigma$). This is viewed as a mapping from references to instances

$$\Sigma = Ref \xrightarrow{m} Inst$$

and variable names can be applied as selectors to objects of *Inst* (e.g. if p is a reference to an instance of *Priq*, then $m(\sigma(p))$ is a natural number). The state is a Curried argument to functions which depend on the global state. The predicate $\mathit{is\text{-}linked\text{-}list}(p, l)(\sigma)$ is true if the instance pointed to by p (in σ) is the start of a linked-list via the references contained in the l variables of each instance. Although the objective here is to talk about linked-lists etc. without needing to think at the reference level, this predicate can be defined in terms of Σ as follows.[4]

$$\mathit{is\text{-}linked\text{-}list} : Ref \times Name \to \Sigma \to \mathbb{B}$$

$$\begin{array}{l} \mathit{is\text{-}linked\text{-}list}(p, l)(\sigma) \quad \triangleq \\ \quad \exists pl \in Ref^* \cdot \\ \quad\quad pl(1) = p \wedge l(\sigma(pl(\mathsf{len}\, pl))) = \mathsf{nil} \wedge \\ \quad\quad \forall i \in \{1, \ldots, \mathsf{len}\, pl - 1\} \cdot pl(i+1) = l(\sigma(pl(i))) \end{array}$$

Similarly, a function to extract a sequence from a linked list is $\mathit{extract\text{-}seq}(p, l, n)$ which generates a sequence of the (non-nil) n values from instances linked by the l references.

$$\mathit{extract\text{-}seq} : Ref \times Name \times Name \to \Sigma \to X^*$$

$$\begin{array}{l} \mathit{extract\text{-}seq}(p, l, n)(\sigma) \quad \triangleq \\ \quad \mathsf{if}\ p = \mathsf{nil}\ \mathsf{then}\ [\,] \\ \quad \mathsf{elif}\ n(\sigma(p)) = \mathsf{nil}\ \mathsf{then}\ \mathit{extract\text{-}seq}(l(\sigma(p)), l, n)(\sigma) \\ \quad \mathsf{else}\ [n(\sigma(p))] \frown \mathit{extract\text{-}seq}(l(\sigma(p)), l, n)(\sigma) \\ \quad \mathsf{fi} \end{array}$$

This can be used to define the set of references which can be reached from a reference.

$$\mathit{reach} : Ref \times Name \to \Sigma \to X^*$$

$$\mathit{reach}(p, l)(\sigma) \quad \triangleq \quad \mathsf{elems}\, \mathit{extract\text{-}seq}(p, l, l)(\sigma)$$

[4] It would be possible to pass a lambda expression (or simply make l a constant) in order to avoid passing a name to *is-linked-list*.

The data type invariant can then be defined as follows.

$$inv : Ref \to \Sigma \to \mathbb{B}$$

$$\begin{aligned} & inv(p)(\sigma) \quad \triangleq \\ & \quad is\text{-}linked\text{-}list(p, l)(\sigma) \wedge is\text{-}ascending(extract\text{-}seq(p, l, m)(\sigma)) \wedge \\ & \quad \forall r \in reach(p, l)(\sigma) \cdot l(\sigma(r)) = \mathsf{nil} \iff m(\sigma(r)) = \mathsf{nil} \end{aligned}$$

The invariant is considered to be true only between method invocations (rather than during the execution of a method). The retrieve function is as follows.

$$retr : Ref \to \Sigma \to \mathbb{N}\text{-}Bag$$

$$retr(p)(\sigma) \quad \triangleq \quad bagof(extract\text{-}seq(p, l, m)(\sigma))$$

It is now possible to specify *Priq* on the linked-lists.[5]

$$\begin{array}{l} Priq \ \mathsf{class} \\ \mathsf{vars}\ m\colon\ [\mathbb{N}] \leftarrow \mathsf{nil};\ l\colon\ \mathsf{private\ ref}(Priq) \leftarrow \mathsf{nil} \\ add(e\colon\ \mathbb{N})\ \mathsf{method} \\ \quad \mathsf{post\ let}\ \overleftarrow{b} = extract\text{-}seq(\mathsf{self}, l, m)(\overleftarrow{\sigma})\ \ \mathsf{in} \\ \quad\quad \mathsf{let}\ b = extract\text{-}seq(\mathsf{self}, l, m)(\sigma)\ \ \mathsf{in} \\ \quad\quad \exists i \in \mathsf{inds}\, b \cdot b(i) = e \wedge del(b, i) = \overleftarrow{b} \\ rem()\ \mathsf{method}\ r\colon\ \mathbb{N} \\ \quad \mathsf{pre}\ extract\text{-}seq(\mathsf{self}, l, m)(\sigma) \neq [\,] \\ \quad \mathsf{post\ let}\ \overleftarrow{b} = extract\text{-}seq(\mathsf{self}, l, m)(\overleftarrow{\sigma})\ \ \mathsf{in} \\ \quad\quad \mathsf{let}\ b = extract\text{-}seq(\mathsf{self}, l, m)(\sigma)\ \ \mathsf{in} \\ \quad\quad r = \mathsf{hd}\ \overleftarrow{b} \wedge b = \mathsf{tl}\ \overleftarrow{b} \end{array}$$

Any user of a *Priq* would be unaware that the implementation involved multiple instances; since the references are private (cannot be copied) they are invisible and free from danger of interference. In order to state the pre- and post-conditions, the sequences are extracted from the state with a reference to the current instance (self) providing the start of the list. A simple generalization of standard refinement rules will cover such a reification step.

Operation decomposition

The next step of development is to look at code which satisfies the above: the specifications are decomposed into executable statements.

[5]Notice m can contain a VDM-like nil; for the *Ref* type, a nil value is a normal null reference; there is a sort of pun here since a 'real' object-oriented language would anyway make all values into objects.

```
Priq class
vars m: [ℕ] ← nil; l: private ref(Priq) ← nil
add(e: ℕ) method
  if m = nil then (m ← e; l ← new Priq)
  elif m < e then l!add(e)
  else (l!add(m); m ← e)
  fi
  return
rem() method r: ℕ
  t: ℕ
  t ← m
  if t≠nil then m ← l!rem()
             if m = nil then l ← nil
             fi
  fi
  return t
```

The inductive justification of this decomposition relies on rules which promote assumptions on one instance of the class to collections of such instances; the linear reference topology justifies a structural induction argument about the recursive calls to methods. The base case for *add* which starts with b as the empty sequence is straightforward (p and l are both nil). The inductive step assumes that the recursive call to $l!add(m)$ performs according to specification. Notice that *inv* above implies that there can not be a loop in the reference chain which is important since otherwise calls to *add* would deadlock. Notice also that it is not necessary to rely on *pre-rem*: the implementation happens to deliver a nil result if the method is used outside its intended domain.

Equivalent code

As mentioned above, the initial steps of this development have not employed concurrency within a queue: in the preceding code, *add* and *rem* hold the invoking process in a *rendezvous* until they complete and a method call at the head of the list does not complete until all recursive calls terminate. (Recall that only one method can be active in each instance of a method at any one time.) Parallelism can be achieved by letting – for example – *rem* return the local m before it ripples through bringing up values as required; the invoking process is released from the *rendezvous* and can run in parallel with the *Priq* methods. Furthermore, this also applies to the instances of *Priq* within one queue: once *rem* has obtained a value from the next element in the queue, it can terminate making it possible for either of the methods of this instance to be invoked. Because of the linear reference topology controlled by private refs, no other thread of control can interfere with the queue.

The argument for the correctness of this step follows from a transformation which permits moving statements. Essentially

$$S; \text{return } e \quad \rightsquigarrow \quad \text{return } e; S \tag{1}$$

providing e is not affected by S_2 and S_2 only changes (other than its own state) states reachable by private references. Thus the preceding code can be transformed as follows.

```
Priq class
vars m: [ℕ] ← nil; l: private ref(Priq) ← nil
add(e: ℕ) method
   return
   if m = nil then (m ← e; l ← new Priq)
   elif m < e then l!add(e)
   else (l!add(m); m ← e)
   fi
rem() method r: ℕ
   return m
   if m≠nil then m ← l!rem()
                 if m = nil then l ← nil
                 fi
   fi
```

This step uses algebraic laws to re-order code which is an observationally equivalent parallel program to the one which was first specified. Apart from offering what is hopefully an intuitive development route, this has obviated the need to describe post-conditions for the concurrent behaviour of the methods. It is not immediately obvious how to write such post-conditions because at the point at which an execution of a method begins, methods on other instances might still be active (such post-conditions appear to need something like Lamport's 'prophesy variables').

The final code behaves in much the same way as *BUBLAT* (cf. [CLW79]) did in earlier work on 'interference' proofs (e.g. [Stø90]) but there is much less 'mechanism' visible here – further steps of development could bring in the extra variables of the earlier code if so desired.

Alternatives

A couple of general observations can be made even after this simple example. There is a reliance above on the fact that the values (in $\mathbb{N}$) are immutable; while this is taken for granted in non-OO-languages, it is not the norm in the OO-world (cf. open issue 2 in Appendix A.1). If the element values could change, such changes would need to be constrained by interference assertions like those used in Section 7.

It must be conceded that – thus far – it would be possible to use a development method in which objects can be guarded from interference by encapsulation and then to have a compiler generate the actual class instances. The reason for taking the approach of creating the instances and reasoning about (non-)interference is that it prepares for the more general approach below. It is – for example – interesting to consider what would go wrong with the above development if a 'fast path' vector of pointers to every tenth element in the list existed. The sharing of pointers which would result would undermine the equivalence shown in Equation 1 and observational equivalence would not be guaranteed.

3 Tree-like object graphs

The programming task considered in this specification is similar to that in the preceding section but it shows that references defining a tree-like topology of instances can also be used as a basis for reasoning; this development also introduces a new statement of the language.

Specification

The example of building a simple symbol table is used in [Ame89]; its specification is very simple.

$Symtab$ class
vars st: $(Key \xrightarrow{m} Data) \leftarrow \{\,\}$
$insert(k\colon Key,\ d\colon Data)$ method
  post $st = \overleftarrow{st} \dagger \{k \mapsto d\}$
$search(k\colon Key)$ method $res\colon Data$
  pre $k \in \mathrm{dom}\, st$
  post $res = st(k)$

Reification

The first design idea is to represent the mapping as a binary tree.

$$
\begin{array}{lll}
Tree :: & mk & : [Key] \\
& md & : [Data] \\
& l & : [Tree] \\
& r & : [Tree]
\end{array}
$$

$$\mathrm{inv}\ (mk\text{-}Tree(mk, md, l, r)) \triangleq (mk = \mathrm{nil} \iff md = \mathrm{nil}) \wedge (mk = \mathrm{nil} \implies l = r = \mathrm{nil})$$

Over which an invariant might be defined

$$is\text{-}ordered\text{-}tree : Tree \to \mathbb{B}$$

$$
\begin{array}{l}
is\text{-}ordered\text{-}tree(mk\text{-}Tree(mk, md, l, r)) \quad \triangleq \\
\quad \mathrm{if}\ mk = \mathrm{nil} \\
\quad \mathrm{then\ true} \\
\quad \mathrm{else}\ (\forall lk \in coll(l) \cdot lk < mk) \wedge (\forall rk \in coll(r) \cdot mk < rk) \wedge \\
\qquad (l \neq \mathrm{nil} \implies is\text{-}ordered\text{-}tree(l)) \wedge (r \neq \mathrm{nil} \implies is\text{-}ordered\text{-}tree(r)) \\
\quad \mathrm{fi}
\end{array}
$$

where the *coll* function simply collects the set of *Keys*

$$coll : [Tree] \to Key\text{-set}$$

$$
\begin{array}{l}
coll(t) \quad \triangleq \\
\quad \mathrm{cases}\ t\ \mathrm{of} \\
\quad \mathrm{nil} \to \{\,\}, \\
\quad mk\text{-}Tree(\mathrm{nil}, md, l, r) \to \{\,\}, \\
\quad mk\text{-}Tree(mk, md, l, r) \to coll(l) \cup \{mk\} \cup coll(r) \\
\quad \mathrm{end}
\end{array}
$$

Nested objects like *Tree* have, in $\pi o \beta \lambda$, to be represented by structures built with references. An invariant must specify that the reference structure forms a genuine tree (*is-linked-tree*) and that the *Tree* obtained by using *extract-tree* on the instances satisfies *is-ordered-tree*.

$$inv : Ref \to \Sigma \to \mathbb{B}$$

$$inv(p)(\sigma) \quad \triangleq \quad is\text{-}linked\text{-}tree(p, l, r)(\sigma) \wedge is\text{-}ordered\text{-}tree(extract\text{-}tree(p, l, r, mk)(\sigma))$$

The functions *is-linked-tree* and *extract-tree* can be defined in an analogous way to *is-linked-list* above.[6]

The retrieve function follows.

$$retr : Ref \to \Sigma \to (Key \xrightarrow{m} Data)$$

$$retr(p)(\sigma) \quad \triangleq$$
$$\quad retrm(extract\text{-}tree(p, l, r, km)(\sigma))$$

$$retrm : [Tree] \to (Key \xrightarrow{m} Data)$$

$$retrm(t) \quad \triangleq$$

cases t of
nil $\to \{\,\}$,
$mk\text{-}Tree(\text{nil}, md, l, r) \to \{\,\}$,
$mk\text{-}Tree(mk, md, l, r) \to retrm(l) \cup \{mk \mapsto md\} \cup retrm(r)$
end

The methods are respecified as follows.

Symtab class
vars mk: $Key \leftarrow$ nil; md: $Data \leftarrow$ nil; l: private ref($Symtab$) $\leftarrow$ nil; r: private ref($Symtab$) $\leftarrow$ nil
$insert(k$: Key, d: $Data)$ method
  post $retr(extract\text{-}tree(\text{self}, l, r, mk)(\sigma)) = retr(extract\text{-}tree(\text{self}, l, r, mk)(\overleftarrow{\sigma})) \dagger \{k \mapsto d\}$
$search(k$: $Key)$ method res: $Data$
  pre $k \in \text{dom}\, retr(extract\text{-}tree(\text{self}, l, r, mk)(\sigma))$
  post $res = (retr(extract\text{-}tree(\text{self}, l, r, mk)(\sigma)))(k)$

Operation decomposition

It is straightforward to provide code which satisfies the pre-/post-conditions on methods of *Symtab*.

Symtab class
vars mk: $Key \leftarrow$ nil; md: $Data \leftarrow$ nil; l: private ref($Symtab$) $\leftarrow$ nil; r: private ref($Symtab$) $\leftarrow$ nil
$insert(k$: Key, d: $Data)$ method
  if mk = nil then ($mk \leftarrow k$; $md \leftarrow d$)
  elif $mk = k$ then $md \leftarrow d$
  elif $k < mk$ 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 res: $Data$
  pre $k \in \text{dom}\, retr(\text{self})$
  if $k = mk$ then return md
  elif $k < mk$ then return $l!search(k)$
  else return $r!search(k)$
  fi

The argument that this code satisfies its specification uses structural induction over the tree topology.

[6] It might, however, by worth passing lambda expressions rather than names to define the link tracing.

Equivalent code

As in Section 3 the above code is sequential (within one instance of a tree). The transformation in Equation 1 can be used to justify executing the return at the beginning of *insert*. There is, however, a problem with re-ordering the steps of *search*: no result can be available until it has been found so the caller of the method has to be held up. But an instance of *Symtab* can be used by another process if the task of delivering a result is delegated (to another instance). This is exactly the semantics of the yield statement. The equivalence used is

$$\text{return } l!m(x) \quad \rightsquigarrow \quad \text{yield } l!m(x) \tag{2}$$

providing l is a private reference and only references via private references. Thus the above code can be transformed into the following.

```
Symtab class
vars mk: Key ← nil; md: Data ← nil; l: private ref(Symtab) ← nil; r: private ref(Symtab) ← nil
insert(k: Key, d: Data) method
   return
   if mk = nil then (mk ← k; md ← d)
   elif mk = k then md ← d
   elif k < mk then (if l = nil then l ← new Symtab fi  l!insert(k, d))
   else (if r = nil then r ← new Symtab fi  r!insert(k, d))
   fi
search(k: Key) method res: Data
   if k = mk then return md
   elif k < mk then yield l!search(k)
   else yield r!search(k)
   fi
```

4 Interference

There are many aspects of concurrent programs and many different problems; the remainder of this paper focuses on interference. It is argued above that methods of reasoning about concurrent programs must accommodate interference. To provide useful compositionality, development methods must offer help also at the earliest stages of design: proofs at lowest level of detail are of less value than those in earlier design phases. The main idea is to structure the design (record) so as to be provable. In fact, much of the motivation of the ideas presented here has been to offer ways of formalizing steps of development which are intuitively acceptable.

It might appear that interference completely rules out the possibility of compositional development but a number of authors have attempted to tame this dragon by recording facts about interference in specifications. An early attempt is presented in [FP78] but this does not offer compositionality. The interference approach in [Jon81, Jon83a, Jon83b] suggested a compositional approach related to the Owicki/Gries method [Owi75, OG76]: rely and guarantee conditions were used to record acceptable and promised interference; proof obligations were given for operation decomposition including parallel statements. The original rely/guarantee method did not cope with liveness issues but there has recently been a flurry of activity and both Stølen [Stø90, Stø91a, Stø91b] and Xu [XH91, Xu92] have proposed extensions to cover liveness. It was always clear that [Jon81] presented only an existence proof of ways of recording and reasoning about interference and that more research was required to make the ideas useful in practice (but [WD88, GR89], for example, show the method has been used on industrial applications). The attempt to find compositional development methods for parallel programs has influenced others – including some work on temporal logic (see [BKP84, dR85]) and the VVSL specification language [Mid90]; related references include [BK84, Sta85, Sti86, Sti88, Sta88, BM88, Ded89, Bro89, SW91]. But by the time the ideas were being recognised, it had become clear that it was possible to improve on the rather heavy proof rules for rely/guarantee conditions and to replace them by a logic with a more pleasing algebra [Jon91b, Jon91a].

5 Sieve of Eratosthenes

The 'Sieve of Eratosthenes' can be used to determine prime numbers up to some stated maximum. Its justification has been used in the literature to illustrate several ways of reasoning about concurrency. The implementation developed in this section is in the spirit of various programs shown in the POOL literature (versions exist in different dialects in [Ame86, AdB90]) but a *test* function has been added here since, without some 'observer', the POOL specifications were forced to talk about internal states rather than behaviours (an alternative observer would be to add a way of listing the primes).

Section 7 presents an alternative development where a DAG-like object graph allows interference; that development is also shown to satisfy the specification below.

Specification

It is easy to write a specification for a prime number tester; what follows already embodies the use of a sieve since starting at a user-oriented view shows nothing new. The specification could be written in a specification notation like that used in VDM. Here, the operations are described as methods of a class called *Primes*. It is obvious that a *test* method is required; here the *new* method is also given explicitly since it has a parameter. The specifications of the methods are

$Primes$ class
vars max: $\mathbb{N}$; $sieve$: $\mathbb{N}$-set
$new(n\colon \mathbb{N})$ method r: ref($Primes$)
  post $r = \mathsf{self} \wedge max = n \wedge sieve = \{2 \leq i \leq max \mid is\text{-}prime(i)\}$
$test(n\colon \mathbb{N})$ method r: $\mathbb{B}$
  rd $sieve, max$
  pre $n \leq max$
  post $r \iff (n \in sieve)$

Instances of *Primes* are created by new $Primes(n)$ which returns a reference, say p; providing the pre-condition is respected, any process to which p is disclosed can then use $p!test(i)$ to obtain a Boolean value which indicates whether i is composite or prime. Although there may be many concurrent threads, only one method can be active at one time per instance of *Primes* (of course, there can be many instances). It is up to the developer of *Primes* to avoid unwanted interference by keeping control of any internal references.

Reification involving class instances

The route to concurrency adopted in this design is to create one process (here, instances of *Sift*) per prime. This is achieved by an initialization in which each instance of *Sift* sifts out any composites for which its index is a factor; each instances pass (to the next) any potential prime which it does not divide

(m div n). At the end of the list of instances, an uninitialized process receives a number which must be a prime, stores it and sets up a new instance of *Sift*. Testing for primality is similar. Thus the instance variables of *Sift* are

Sift class
vars m: $[\mathbb{N}] \leftarrow$ nil; l: private ref(*Sift*) $\leftarrow$ nil

The initial specification of *Primes* is given in terms of local assertions on each method. In contrast, the invariant which plays a part in this design step concerns the multiple instances of *Sift* which are created. These instances form a linked-list object graph in which the variable l of one instance contains the reference of the next instance (with nil marking the end of the list). The use of such references requires that assertions are couched in terms of a global state ($\sigma \in \Sigma$) which is viewed as a map from references to instances

$$\Sigma = Ref \xrightarrow{m} Inst$$

Variable names are treated as selectors to objects of *Inst* (thus, if p is a reference to an instance of *Sift*, $m(\sigma(p))$ selects the natural number in m). The state is a Curried argument to functions which depend on Σ. The predicate $\textit{is-linked-list}: Ref \times Name \rightarrow \Sigma \rightarrow \mathbb{B}$ and the function $\textit{extract-seq}: Ref \times Name \times Name \rightarrow \Sigma \rightarrow X^*$ are as in Section 2. In terms of these, it is straightforward to define an invariant which limits the object graph of *Sift* instances to a linear list and, furthermore, requires that the m values are in ascending order ($\textit{is-ascending}: \mathbb{N}^* \rightarrow Bool$ is assumed to be obvious).

$$inv : Ref \rightarrow \Sigma \rightarrow \mathbb{B}$$

$$inv(sr)(\sigma) \quad \triangleq \quad \textit{is-linked-list}(sr, l)(\sigma) \wedge \textit{is-ascending}(\textit{extract-seq}(sr, l, m)(\sigma))$$

The intuitive idea that the m values in this linear list represent the *sieve* value in the specification of *Primes* can be formalized by a retrieve function

$$retr : Ref \rightarrow \Sigma \rightarrow \mathbb{N}\text{-set}$$

$$retr(p)(\sigma) \quad \triangleq \quad \text{elems}\ \textit{extract-seq}(p, l, m)(\sigma)$$

Now, still following the general pattern of development steps by data reification in [Jon90], the methods of *Primes* can be specified on this representation as follows.

Primes class
vars max: $\mathbb{N}$; sr: private ref(*Sift*) $\leftarrow$ nil
$new(n$: $\mathbb{N})$ method r: ref(*Primes*)
  post $r = \text{self} \wedge max = n \wedge retr(sr)(\sigma) = \{2 \leq i \leq max \mid \textit{is-prime}(i)\}$
$test(n$: $\mathbb{N})$ method r: $\mathbb{B}$
  rd *sieve, max*
  pre $n \leq max$
  post $r \Leftrightarrow (n \in retr(sr)(\sigma))$

Notice that clauses of the invariant such as *is-linked-list* do not have to be stated in, for example, the post-condition of *new*.

Decomposition

It is a straightforward task to write sequential object-oriented programs which satisfy the specification of *Primes* which has resulted from the reification and which also preserve the invariant. In a fully formal operation decomposition one would need inference rules about the specifically object-oriented statements which supplement those (e.g. in [Jon90]) for iterative statements etc. In the code which follows, an outline proof is adumbrated by assertions.

```
Primes class
vars max: ℕ; sr: private ref(Sift) ← nil
new(n: ℕ) method r: ref(Primes)
  ctr: ℕ
  max ← n
  sr ← new Sift
    {retr(sr)(σ) = { }}
  ctr ← 2
  while ctr ≤ max do
    sr!setup(ctr)
      {retr(sr)(σ) = {i ∈ {2, …, ctr} | is-prime(i)}}
    ctr ← ctr + 1
   od
    {retr(sr)(σ) = {i ∈ {2, …, max} | is-prime(i)}}
  return self
test(n: ℕ) method r: 𝔹
  return sr!test(n)

Sift class
vars m: [ℕ] ← nil; l: private ref(Sift) ← nil
setup(n: ℕ) method
  if m = nil then (m ← n; l ← new Sift)
  elif ¬ m div n then l!setup(n)
  else skip
  fi
  return
test(n: ℕ) method r: 𝔹
  if m = nil ∨ n < m then return false
  elif m = n then return true
  else return l!test(n)
  fi
```

The formal argument would use structural induction over the linked-list structure to promote results about one instance to properties of the whole network.

Equivalent code

As in Section 3, the real interest is how to move from the sequential solution to one which realizes the potential for concurrency which is inherent in the many instances of *Sift*. As the code above stands, any invocation of *setup* of *Sift* will not release its invoker until the effect has travelled all the way along the linked list and the returns have come all of the way back. This delay is unnecessary as can be proved using Equation 1. This justifies moving the return statement to the first position in *setup*. The method now releases its invoker as soon as possible and generates activity further along the list; once the method in one instance terminates, it is open to have further methods invoked even though the activity from the first call is still going on. The point of the rule in Equation 1 is that it preserves observational equivalence. The same transformation can be applied to *new* of *Primes*.

In essence, a similar transformation is required for *test* of *Sift* where the change from return to yield is justified by Equation 2. Thus the final code is as follows.

```
Primes class
vars max: ℕ; sr: private ref(Sift) ← nil
new(n: ℕ) method r: ref(Primes)
   ctr: ℕ
   max ← n
   return self
   sr ← new Sift
   ctr ← 2
   while ctr ≤ max do sr!setup(ctr); ctr ← ctr + 1 od
test(n: ℕ) method r: 𝔹
   return sr!test(n)

Sift class
vars m: [ℕ] ← nil; l: private ref(Sift) ← nil
setup(n: ℕ) method
   return
   if m = nil then (m ← n; l ← new Sift)
   else if ¬ m div n  then l!setup(n)   fi
   fi
test(n: ℕ) method r: 𝔹
   if m = nil ∨ n < m then return false
   elif m = n then return true
   else yield l!test(n)
   fi
```

The development route adopted in this section is to stay with data reification and operation decomposition for sequential programs until the final step of development; concurrency is introduced by transformations which preserve observational equivalence. The validity of the transformations rely on restrictions to the object graphs. Where, as in Section 7, sharing of references occurs this is not an appropriate development method.

6 Global safety assertions

The arguments used in Section 7 use global assertions about the evolution of computations. These assertions are written in a logic which is a development of that presented in [Jon91a]. In addition to predicates of one state $p\colon \Sigma \to \mathbb{B}$ and relations on states $r\colon \Sigma \times \Sigma \to \mathbb{B}$, various modal operators are allowed. The most basic operator for safety reasoning is S links r meaning that any step in the execution of S makes a state transition which satisfies the relation r. Some arguments can be documented more concisely using derived operators. For example

$$\boxed{\text{confirms -defn}}\quad \frac{S \text{ links } (\overleftarrow{p} \;\Rightarrow\; p)}{S \text{ confirms } p}$$

$$\boxed{\text{maintains -defn}}\ \frac{S \text{ links } (\overleftarrow{p} \;\Leftrightarrow\; p)}{S \text{ maintains } p}$$

and for $f : \Sigma \to Val$

$$\boxed{\text{conserves -defn}}\ \frac{S \text{ links } (f = \overleftarrow{f}\,)}{S \text{ conserves } f}$$

An operator which asserts that S does terminate – and that the final state satisfies p – is S fin p. Assertions about the behaviour of an execution under interference ($S \backslash e$) are written – for example

$$e \text{ links } (retr(sr)(\sigma) \subseteq retr(sr)(\overleftarrow{\sigma})) \;\Rightarrow\; \text{new } Rem(i, sr) \backslash e \text{ fin } (retr(sr)(\sigma) \cap mults(i) = \{\,\})$$

In addition, the following two rules are used below

$$\boxed{\|\text{ -links}}\ \frac{\bigwedge_i (S_i \text{ links } r)}{\|_i\, S_i \text{ links } r}$$

$$\boxed{\|\text{ -}I}\ \frac{\begin{array}{c}\bigwedge_i (S_i \text{ links } r)\\ \bigwedge_i (e \text{ links } r \;\Rightarrow\; S_i \backslash e \text{ fin } p_i)\end{array}}{e \text{ links } r \;\Rightarrow\; (\|_i\, S_i) \backslash e \text{ fin } \bigwedge_i p_i}$$

7 Reasoning about interference

This section shows how to cope with interference; both specifications and development steps must be considered. A program is developed which employs concurrency in much the same way as [Jon83a] implements the prime sieve – here, of course, the program is built from multiple instances of classes. The development in this section is based on the initial specification of Section 5. The final program uses an acyclic directed graph (DAG) of objects: references which are shared by several objects bring with them many of the problems of shared variables but the fact that the interface of an object is constrained by the available methods simplifies reasoning about interference. But there is certainly a price to pay for the interference: the DAG object graph can no longer support the form of induction proof used in Section 5.

Reification

A straightforward step of data reification could represent *sieve* of Section 5 as an array of Booleans (giving its characteristic function). A general array is however too flexible in that its elements could be changed by assignment in either direction between the two Boolean values; furthermore, such an array offers no scope for distribution. Given that *sieve* is initialized to a large set and then elements are only ever removed, it is a better design decision to place each Boolean in a separate instance of a class *El* which only has a method which deletes its element; these separate instances provide potential parallelism.[7] The instances are located via a map

$$\mathbb{N} \xrightarrow{m} Ref(El) \tag{3}$$

[7] In $\pi o\beta\lambda$ as it stands, some of this potential is squandered. If it were possible to use the natural numbers themselves as references, the program and its development would be shorter and more parallelism would be available (such a 'program' is given in Appendix B) but the fact that a separate mapping from natural numbers to references is required here makes no difference to the sort of interference proof required. Unfortunately, the mapping does introduce an addressing bottleneck. It would be possible to use multiple copies of *Vector* after its initialization and Pierre America (private communication) has ideas about *pragmas* which would request 'one copy per processor'. (Justification of this split would be trivial.)

The *El* class is simple enough that it is easier to document the design decisions directly in its code than to interpose a specification.

El class
vars b: $\mathbb{B} \leftarrow$ true
test() method r: $\mathbb{B}$
  rd b
  return b
del() method
  $b \leftarrow$ false
  return

Instances of *El* are initialized (to true) when created by a new statement. Notice that there is (after creation) only a *del* method available thus restricting interference. This intuitive idea can be formalised by

$$p \in \mathit{Ref}(\mathit{El}) \;\Rightarrow\; p!\mathit{test}() \text{ links } (b(\sigma(p)) \;\Leftrightarrow\; b(\overleftarrow{\sigma}(p)))$$

It is more convenient to record this as

$$p \in \mathit{Ref}(\mathit{El}) \;\Rightarrow\; p!\mathit{test}() \text{ links } (p!b \;\Leftrightarrow\; \overleftarrow{p!b})$$

or even

$$p \in \mathit{Ref}(\mathit{El}) \;\Rightarrow\; p!\mathit{test}() \text{ maintains } p!b$$

and similarly

$$p \in \mathit{Ref}(\mathit{El}) \;\Rightarrow\; p!\mathit{del}() \text{ confirms } \neg\,(p!b) \tag{4}$$

Since there are only these two methods, any designer can rely on *El*'s contribution to the environment satisfying

$$e \text{ confirms } \neg\,(p!b) \tag{5}$$

Furthermore

$$p \in \mathit{Ref}(\mathit{El}) \;\Rightarrow\; p!\mathit{del}() \text{ fin } (\neg\, p!b) \tag{6}$$

The map of Equation 3 is stored in a variable v of (an instance of) class *Vector* which provides via its method *lu* a way of looking up the reference to *El* for an index. Since the references to *El* are to be returned as results, they must be marked as shared. Thus

Vector class
vars max: $\mathbb{N}$; v: $\mathbb{N} \xrightarrow{m}$ shared ref(El)
new(n: $\mathbb{N}$) method r: ref($Vector$)
  post $r = \text{self} \wedge max = n \wedge \forall i \in \{2, \ldots, max\} \cdot b(\sigma(v(i))) \;\Leftrightarrow\;$ true
lu(n: $\mathbb{N}$) method r: ref(El)
  rd max, v
  pre $n \leq max$
  return $v(n)$

Getting back to the task of re-specifying the methods of *Primes* on this representation, it is necessary to relate the representation to the abstraction (i.e. *sieve*) in the normal way. The function which retrieves *sieve* of the specification is[8]

$$retr : Ref \to \Sigma \to \mathbb{N}\text{-set}$$

$$retr(sr)(\sigma) \quad \triangleq \quad \text{let } m = rmap(v(\sigma(sr)))(\sigma) \text{ in } \{i \in \text{dom } m \mid m(i) \;\Leftrightarrow\; \text{true}\}$$

$$rmap : (\mathbb{N} \xrightarrow{m} Ref) \to \Sigma \to (\mathbb{N} \xrightarrow{m} \mathbb{B})$$

$$rmap(rm)(\sigma) \quad \triangleq \quad \{i \mapsto b(\sigma(rm(i))) \mid i \in \text{dom } rm\}$$

Adequacy etc. can be proved.

The specification of the main part of *Primes* (its *new* method) is

Primes class
vars max: $\mathbb{N}$; sr: shared ref($Vector$)
$new(n\colon \mathbb{N})$ method r: ref($Primes$)
  post $r = \text{self} \wedge max = n \wedge retr(sr)(\sigma) = \{i \mapsto is\text{-}prime(i) \mid i \in \{2, \ldots, max\}\}$
$test(n\colon \mathbb{N})$ method r: $\mathbb{B}$
  return $(sr!lu(n))!test()$

Completing the proof that this is a reification of *Primes* at the beginning of Section 5 is not difficult. The interesting part of the design task is the use of parallelism which now follows.

Decomposition of *Vector* and *Primes*

Developing code to satisfy the specification of the *new* method of *Vector* is an easier job than for *Primes*. Since this is the first exposure to the parallel statement of $\pi o\beta\lambda$ the easier task is tackled first. The post-condition of *new* (of *Vector*) can be satisfied if new *El* is invoked to set up each $v(i)$. This could be achieved by a while statement but here it is possible to use a parallel statement which creates independent threads. Since each $v(i)$ is independent, no interference can arise. Thus the code is

Vector class
vars max: $\mathbb{N}$; v: $\mathbb{N} \xrightarrow{m}$ shared ref(El)
$new(n\colon \mathbb{N})$ method r: ref($Vector$)
  $max \leftarrow (\lceil\sqrt{n}\rceil)^2$

$$\mathop{\Big\|}_{i\in\{2,\ldots,max\}} v(i) \leftarrow \text{new } El$$

  return self
$lu(n\colon \mathbb{N})$ method r: ref(El)
  pre $n \leq max$
  return $v(n)$

But the rule 1 can again be used to make the return come after the first assignment in *new*.

It is now time to turn to the interesting task of designing *Primes* so as to satisfy the specification above. The *new* method has to create an instance of *Vector* (which sets the b in each *El* to true) and then arrange that the b of each composite number is 'deleted' (set to false). This deletion could be

[8] In several places below, $m(i) \;\Leftrightarrow\;$ true is written for emphasis where $m(i)$ would be equivalent.

implemented by nested loops and such a sequential approach would pose no interference problems. Here the design decision is to use parallel instances of a *Rem* process: each *Rem*(*i, sr*) is responsible for sieving out those composites of which *i* is a factor; *sr* gives access to the instance of *Vector*. Given that *sr* is shared by the parallel instances of *Rem* the object graph is a DAG. It is easy to see from the types of the variables containing references that no cycles can be present.

It is then, now essential to face the problem of interference. Fortunately the notation and rules of [Jon91a] cover the needs here with little modification. The designer of *Primes* might choose to make a step in which the *new* method is designed and justified in terms of a specification for *Rem* (postponing its implementation).

In order to obtain an understanding of the specification for the *new* method of *Rem*, the simplification where it is assumed to run in the absence of interference is considered first. An initial stab at a post-condition might be[9]

$$retr(sr)(\overleftarrow{\sigma}) - retr(sr)(\sigma) = mults(i)$$

But, even for isolated instances of *Rem*, this is wrong because (other than the first instance executed) some composite $c \in mults(i)$ – which the *i*th instance of *Rem* would have deleted – might be absent from its initial state because it was removed by some earlier invocation of *Rem* (with an index which is another factor of *c*). The correct post-condition for an isolated version of *Rem* is

$$retr(sr)(\overleftarrow{\sigma}) - retr(sr)(\sigma) = mults(i) \cap retr(sr)(\overleftarrow{\sigma}) \tag{7}$$

If, however, instances of *Rem* are run in parallel, interference can occur and it is possible that this can delete elements which are not multiples of *i* in Equation 7. This suggests focusing on the actions of *Rem*(*i, sr*) by writing a dynamic constraint

$$\mathsf{new}\ Rem(i, sr)\ \mathsf{links}\ (retr(sr)(\overleftarrow{\sigma}) - retr(sr)(\sigma) \subseteq mults(i)) \tag{8}$$

Use of $\|$-links of Section 6 makes it possible to conclude from Equation 8 that

$$\|_{i}\ \mathsf{new}\ Rem(i, sr)\ \mathsf{links}\ (retr(sr)(\overleftarrow{\sigma}) - retr(sr)(\sigma) \subseteq \bigcup_{i} mults(i)) \tag{9}$$

So far so good – but this is not enough for the designer of *Primes* since it is necessary to show that enough elements are removed (Equation 9 is satisfied by skip).

Referring back to Equation 7, what is missing is a constraint that

$$\mathsf{new}\ Rem(i, sr)\backslash e\ \mathsf{fin}\ (retr(sr)(\sigma) \cap mults(i) = \{\ \})$$

But the designer of *Rem* will be unable to construct an implementation which achieves this requirement unless permission is given to rely on

$$e\ \mathsf{links}\ (retr(sr)(\sigma) \subseteq retr(sr)(\overleftarrow{\sigma}))$$

So the contract includes

$$e\ \mathsf{links}\ (retr(sr)(\sigma) \subseteq retr(sr)(\overleftarrow{\sigma}))\ \Rightarrow\ \mathsf{new}\ Rem(i, sr)\backslash e\ \mathsf{fin}\ (retr(sr)(\sigma) \cap mults(i) = \{\ \}) \tag{10}$$

[9]Where $mults: \mathbb{N} \to \mathbb{N}\text{-set}$ yields the set of multiples of *i*.

It is now appropriate to use $\|$-I of Section 6 to conclude

$$\text{new } Primes \text{ fin } retr(sr)(\sigma) \cap \bigcup_i mults(i) = \{\,\}$$

So the class *Primes* (with annotations) is

$Primes$ class
vars max: $\mathbb{N}$; sr: shared ref($Vector$)
$new(n\colon \mathbb{N})$ method r: ref($Primes$)
  $max \leftarrow n$
  $sr \leftarrow$ new $Vector(max)$
    $\left\{\text{let } m = rmap(v(\sigma(sr))(\sigma) \text{ in rng } m = \{\text{true}\}\right\}$

$$\mathop{\|}_{i \in \{2,\ldots,\lceil\sqrt{max}\rceil\}} \text{new } Rem(i, sr)$$

    $\left\{\text{let } m = rmap(v(\sigma(sr))(\sigma) \text{ in } \forall i \in \{2, \ldots, max\} \cdot m(i) \iff is\text{-}prime(i)\right\}$
  return self
$test(n\colon \mathbb{N})$ method r: $\mathbb{B}$
  return $(sr!lu(n))!test()$

Decomposition of *Rem*

The remaining task is to develop code which satisfies the requirements on *Rem* (cf. Equations 9 and 10). It follows by $\|$-links from Equation 4 that

$$\mathop{\|}_{m \in \{2,\ldots,\lceil max/i\rceil\}} (sr!lu(i * m))!del() \text{ links } (retr(sr)(\overleftarrow{\sigma}) - retr(sr)(\sigma) \subseteq mults(i)) \tag{11}$$

from which Equation 8 is a consequence. The post-condition in Equation 10 requires $\|$-I again so *Rem* satisfies the annotations shown in the following.

Rem class
$new(i\colon \mathbb{N},\ sr\colon \text{ref})$ method

$$\mathop{\|}_{m \in \{2,\ldots,\lceil max/i\rceil\}} (sr!lu(i*m))!del()$$

  $\left\{\text{let } m = rmap(v(\sigma(sr))(\sigma) \text{ in } \forall c \in mults(i) \cdot m(c) \iff \text{false}\right\}$
  return

Final code transformation

Finally, *El* can be transformed to

El class
vars b: $\mathbb{B} \leftarrow$ true
test() method r: $\mathbb{B}$
  rd b
  return b
del() method
  return
  $b \leftarrow$ false

8 Discussion

Clearly there is much more work to be done. Apart from considering other examples, the major activity is to complete the appendix which provides a semantics for $\pi o\beta\lambda$. This will be the basis on which the proof obligations are to be justified. Examples of liveness proofs have been undertaken but need polishing. Influences on choice of logical operators include UNITY [CM88] (and Misra's more recent work), as well as Lamport's TLA [Lam90, Lam91] (it might be worth defining the operators of Section 6 on top of TLA.

Acknowledgements

The author is grateful to Mario Wolczko, Carlos Camaroa, Trevor Hopkins, John Sargeant, Michael Fisher and John Gurd for stimulating discussions on topics related to the implementation of object-based languages and machine architectures and to Robin Milner, David Walker, Kohei Honda, Akinora Yonezawa, Ole-Johan Dahl, Ketil Stølen and Manfred Broy for detailed technical discussions. The incentive provided by the discussions with the 'Object-Z' group at the University of Queensland is also remembered. Ketil Stølen prompted the use of predicates like *is-linked-list* during an enjoyable visit to Munich. Anders Ravn made useful comments on a draft of this paper and Kohei Honda provided a detailed criticism of both content and presentation style. Feedback from the 1992 meeting of IFIP WG 2.3 was stimulating as were the questions on a trip to NWPC-4 in Bergen and at a seminar in Oslo. The support of a Senior Fellowship from the SERC is gratefully acknowledged.

References

[AdB90] P. America and F. de Boer. A proof system for process creation. In *[BJ90]*, pages 303–332, 1990.

[Ame86] Pierre America. A proof theory for a sequential version of POOL. Technical Report 0188, Philips Research Laboratories, Philips Research Laboratories, Nederlandse Philips Bedrijven, B.V., September 1986.

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

[Ame91a] P. America, editor. *ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Ame91b] P. America. Formal techniques for parallel object-oriented languages. In *[BG91]*, pages 1–17, 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.

[BF91] J. A. Bergstra and L. M. G. Feijs, editors. *Algebraic Methods II: Theory Tools and Applications*, volume 490 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[BG91] J. C. M. Baeten and J. F. Groote, editors. *CONCUR'91 – Proceedings of the 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[BJ90] M. Broy and C. B. Jones, editors. *Programming Concepts and Methods*. North-Holland, 1990.

[BJM88] R. Bloomfield, R. B. Jones, and L. S. Marshall, editors. *VDM'88: VDM – The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[BK84] H. Barringer and R. Kuiper. Hierachical development of concurrent systems in a temporal logic framework. In *Proceedings of NSF/SERC Seminar on Concurrency*, CMU, Pittsburgh, 1984.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you can compose temporal logic specification. In *Proceedings of 16th ACM STOC*, Washington, May 1984.

[BM88] J. Bruijning and C.A. Middelburg. Esprit project 1283: VIP VDM extensions: Final report. Technical Report 2.0, PTT Research, Neher Laboratories, The Netherlands, 1988.

[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.

[BSI92] BSI. VDM specification language protostandard. Technical Report N-231, BSI IST/5/19, 1992.

[CLW79] K. M. Chung, F. Luccio, and C. K. Wong. A new permutation algorithm for bubble memories. Technical Report RC 7633, IBM Research Division, 1979.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Daw91] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[dB91] Frank S. de Boer. *Reasoning about Dynamically Evolving Process Structure*. PhD thesis, Free University of Amsterdam, 1991.

[Ded89] Frank Dederichs. Zur strukturierung von spezifikationen verteilter systeme, March 1989.

[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.

[Dür92] E. H. H. Dürr. Syntactic description of the VDM++ language. Technical report, Rijksuniveriteit Utrecht, 1992.

[FP78] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, DEC, SRC, July 1985.

[GR89] David Grosvenor and Andy Robinson. An evaluation of rely-guarantee, March 1989. Submitted to Formal Aspects of Computer Science.

[HHJ+87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987. see Corrigenda in Communications of the ACM, 30(9): 770.

[HT91a] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *[Ame91a]*, pages 133–147, 1991.

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

[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.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. mural*: A Formal Development Support System*. Springer-Verlag, 1991.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group Technical Monograph 25.

[Jon83a] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon83b] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon91a] C. B. Jones. Interference resumed. In P. Bailes, editor, *Engineering Safe Software*, pages 31–56. Australian Computer Society, 1991.

[Jon91b] C. B. Jones. Interference revisited. In J. E. Nicholls, editor, *Z User Workshop*, pages 58–73. Springer-Verlag, 1991.

[JPZ91] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *[BG91]*, pages 298–316, 1991.

[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 and others, September 1991.

[Lam90] L. Lamport. A temporal logic of actions. Technical Report 57, Digital Equipment Corporation, Systems Research Center, 1990.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, Digital, SRC, 1991.

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

[Len82] C. Lengauer. *A Methodology for Programming with Concurrency*. PhD thesis, Computer Systems Research Group, University of Toronto, 1982.

[Mid90] C.A. Middelburg. *Syntax and Semantics of VVSL A Language for Structured VDM Specifications*. PhD thesis, PTT Research, Department of Applied Computer Science, September 1990.

[Mil89] R. Milner. Functions as processes. In *MSCS*, 1989. (submitted to).

[Mil92] R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*. Springer-Verlag, 1992.

[MPW91] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. In *[BG91]*, pages 45–60, 1991.

[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.

[OA91] E.-R. Olderog and K. R. Apt. Using transformations to verify parallel programs. In *[BF91]*, pages 55–82, 1991.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975. 75-251.

[PT91] S. Prehn and W. J. Toetenel, editors. *VDM'91 – Formal Software Development Methods. Proceedings of the 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 1991. Vol.1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[RH86] A.W. Roscoe and C.A.R. Hoare. Laws of occam programming. Monograph PRG-53, Oxford University Computing Laboratory, Programming Research Group, February 1986.

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

[Sta85] Eugene W Stark. A proof technique for rely/guarantee properties, August 1985.

[Sta88] Eugene W Stark. Proving entailment between conceptual state specifications. *Theoretical Computer Science*, 56:135–154, 1988.

[Sti86] C. Stirling. A compositional reformulation of Owicki-Gries' partial correctness logic for a concurrent while language. In *ICALP'86*. Springer-Verlag, 1986. LNCS 226.

[Sti88] C. Stirling. A generalisation of Owicki-Gries's Hoare logic for a concurrent while language. *TCS*, 58:347–359, 1988.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. available as UMCS-91-1-1.

[Stø91a] K. Stølen. A Method for the Development of Totally Correct Shared-State Parallel Programs. In *[BG91]*, pages 510–525, 1991.

[Stø91b] K. Stølen. An Attempt to Reason About Shared-State Concurrency in the Style of VDM. In *[PT91]*, pages 324–342, 1991.

[SW91] J. Sa and B. C. Warboys. Specifying concurrent object-based systems using combined specification notations. Technical Report UMCS-91-7-2, Manchester University, 1991.

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

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

[Wal93] D. Walker. Objects in the π-calculus. *Information and Computation*, 1993. (to appear).

[WD88] J. C. P. Woodcock and B. Dickinson. Using VDM with rely and guarantee-conditions: Experiences of a real project. In *[BJM88]*, pages 434–458, 1988.

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

[XH91] Qiwen Xu and Jifeng He. A theory of state-based parallel programming by refinement: Part I. In J. Morris, editor, *Proceedings of The Fourth BCS-FACS Refinement Workshop*. Springer-Verlag, 1991.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming*. PhD thesis, Oxford University, 1992.

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

[Zwi88] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof theories for networks of processes, and their connections*. PhD thesis, Technical University Eindhoven, 1988. available as LNCS 321, Springer-Verlag.

A $\pi o\beta\lambda$

A.1 Relationship of $\pi o\beta\lambda$ to POOL

This section comments on the differences between $\pi o\beta\lambda$ and the language which inspired its creation. A useful overview of the work on POOL is [Ame89]. Pierre America and Jan Rutten produced a combined doctoral thesis [AR89] which contains a collection of papers (some published elsewhere) on the formal aspects of the POOL project including their work on (metric space methods of) denotational semantics. A proof theory for a sequential version of POOL is given in [Ame86], while [AdB90] addresses proofs about process creation in a language called P which is more like CSP or CCS in the way that communication is a single event without any way to return a value. A proof method for the full *rendezvous* mechanism of POOL is given in [dB91]: but this multi-level approach is not compositional in a useful sense.

The main changes from POOL (see [Ame89, Ame91b]) are:

1. In $\pi o\beta\lambda$, methods do not have a body (which, in POOL, is a statement which says – for instances of the class – when a *rendezvous* can occur as well as executing autonomous code between method invocations); the examples here were longer with a body and it rarely did anything interesting; one can simulate the effect of this body by code in methods and switches etc.
2. The new message to a class can be defined by an explicit method in $\pi o\beta\lambda$.
3. Methods in $\pi o\beta\lambda$ which do not return a value are distinguished from those which do.
4. The *yield* statement is new in $\pi o\beta\lambda$.[10]
5. The *Parallel* statement is also new but is an obvious extension.
6. References in $\pi o\beta\lambda$ are typed.
7. POOL has a local call; this could easily be added to $\pi o\beta\lambda$.
8. Clearly, $\pi o\beta\lambda$ needs some way of controlling conditional 'firing' of methods.

The development method presented here is not like any in the POOL literature. The approach illustrated in the current paper is the way that developments can first employ normal sequential reasoning based on pre-/post-conditions and then use transformations to admit concurrency (similar ideas are present in the works of Lengauer [Len82], Zwiers [JPZ91] and Xu/He [XH91, Xu92]; equivalence laws are given in [HHJ$^+$87, RH86]; see [OA91]).

Some open issues in $\pi o\beta\lambda$ are:

1. Methods could be divided into those which have a side-effect and those which are purely functional – this is done in UFO [Sar92].
2. It is not clear whether it would be worth distinguishing mutable values from what are constants in other languages – this affects the need for interference assertions (cf. the infamous *ordered-collection* example).

[10] I suspect it would not be liked by the POOL authors who would deprecate such a form of 'future communication' – but compare Section 3 with [Ame89].

3. So far, $\pi o\beta\lambda$ has not used the (ST) trick of defining operators (e.g. $+, \neg$) as methods; since there are no 'block expressions' (yet?), the option to do the same for while does not exist.

4. Block statements and exceptions might be added (exceptions could be in the style of VDM's exit).

5. There is some case for adding constant (e.g. numeric) channel names (cf. AppendixAC).

A.2 Abstract syntax

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

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

$$Type = \textsc{LocalRef} \mid \textsc{SharedRef} \mid \textsc{Bool} \mid \textsc{Int}$$

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

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

$$\begin{array}{lll} Assign :: & lhs & : Id \\ & rhs & : Expr \end{array}$$

$$Parallel :: m : Index \xrightarrow{m} Stmt$$

$$Compound :: sl : Stmt^*$$

$$\begin{array}{lll} If :: & b & : Expr \\ & th & : Stmt \\ & el & : Stmt \end{array}$$

$$\begin{array}{lll} While :: & b & : Expr \\ & s & : Stmt \end{array}$$

$$Return :: r : [Expr]$$

$$Yield :: r : Expr$$

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

$$\begin{array}{lll} New :: & cn & : Id \\ & al & : Expr^* \end{array}$$

$$\begin{array}{lll} Compare :: & e1 & : Expr \\ & e2 & : Expr \end{array}$$

$$\begin{array}{lll} Mref :: & ob & : Expr \\ & mn & : Id \\ & al & : Expr^* \end{array}$$

B Using constant references

This appendix indicates how constants (in this case natural numbers) could be used as channel names: writing r_i for $i \in \mathbb{N}$.

Primes class
vars max: $\mathbb{N}$
$new(n\colon \mathbb{N})$ method r: ref(*Primes*)
  $max \leftarrow n$

$$\mathop{\Big|\Big|}_{i \in \{2,\ldots,max\}} \text{new } El(i)$$

$$\mathop{\Big|\Big|}_{i \in \{2,\ldots,\lceil\sqrt{max}\rceil\}} Rem(i)$$

  return self
$test(n\colon \mathbb{N})$ method r: $\mathbb{B}$
  return $n!test()$

Rem: class
$new(i\colon \mathbb{N})$ method

$$\mathop{\Big|\Big|}_{m \in \{2,\ldots,\lceil max/i\rceil\}} (r_{i*m})!del();$$

  return

El class
vars b: $\mathbb{B}$
$new(i)$ method r: ref
  $b \leftarrow$ true
  return r_i
$test()$ method r: $\mathbb{B}$
  return b
$del()$ method
  return
  $b \leftarrow$ false

C Semantics

The body of this paper uses object-based notation as a way of recording design decisions; it is certainly not the intention to design a new programming language. It is however still necessary to fix its semantics in much the same way as one would for a programming language since the proof obligations which are proposed for the development method must be justified against some semantic base. My own earlier work on operational and denotational semantics naturally led me to attempt a model-oriented semantics (for alternatives see [Wol88, AR89, AR92, Wal93]) and, in fact, a semantics based on *resumptions* fits some aspects of parallel object-based languages quite well. But there are serious difficulties which appear to arise from not being able to capitalise on the limitations on interference: one ends up describing a detailed level of granularity and then proving that a coarser notion of atomic step would give the same overall result. In contrast, it is possible to map $\pi o\beta\lambda$to Milner's π-calculus [Mil92].

C.1 The π-calculus

This section pins down the version of the polyadic π-calculus used below; the main source is [Mil92].

Syntax

Processes (typical elements P, Q)

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

Normal processes (typical elements M, N)[11]

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

Prefixes (typical element π)[12]

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

Names (typical elements x, y)

Abbreviations

Trailing stop processes are omitted, so $\pi.\mathbf{0}$ is written π.

Multiple new names are combined, so $(\nu x)(\nu y)$ is written (νxy).

[11] Unlike [Wal93, Mil89], (binary) sums are used; the summands are always prefixed.

[12] In [Mil92] abstractions and concretions are identified as separate phrases. Although the symmetry is pleasing, separating concretions appears to achieve little for the purposes here; the question of abstractions is more subtle: for now, the object-oriented position is taken that everything is located by name – this would certainly change if a higher-order calculus were used.

Structural equivalence

Assume functions which yield the free (fn) and bound (bn) names of processes.

Structural equivalence laws include the following[13] (Alpha-convertible terms are taken to be structurally equivalent).

$$
\begin{array}{rcl}
M+\mathbf{0} & \equiv & M \\
M+N & \equiv & N+M \\
M_1+(M_2+M_3) & \equiv & (M_1+M_2)+M_3 \\
M+M & \equiv & M \\
P \mid \mathbf{0} & \equiv & P \\
P \mid Q & \equiv & Q \mid P \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R \\
!\,P & \equiv & P \mid !\,P \\
(\nu x)\mathbf{0} & \equiv & \mathbf{0} \\
(\nu x)(\nu y)P & \equiv & (\nu y)(\nu x)P \\
(\nu x)(P \mid Q) & \equiv & P \mid (\nu x)Q \text{ if } x \notin fn(P) \\
(\nu x)y(\tilde{z}).P & \equiv & y(\tilde{z}).(\nu x)P \text{ where } x \neq y,\, x \notin \tilde{z} \\
(\nu x)\overline{y}\tilde{z}.P & \equiv & \overline{y}\tilde{z}.(\nu x)P \text{ where } x \neq y,\, x \notin \tilde{z} \\
(\nu x)\pi.P & \equiv & \mathbf{0} \text{ if } \pi \text{ is } x(\tilde{y}) \text{ or } \overline{x}\tilde{y}
\end{array}
$$

Reduction

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

$$\boxed{\text{PAR}}\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

$$\boxed{\text{RES}}\ \frac{P \longrightarrow P'}{(\nu x)P \longrightarrow (\nu x)P'}$$

$$\boxed{\text{STRUCT}}\ \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

Notice that there are no reductions under prefix (or replication).

C.2 Representing values

Note[14]

[13]The first three rules for $+$ ($|$) can be summarized by saying that $M/+/\mathbf{0}$ ($P/\mid/\mathbf{0}$) are symmetric monoids.

[14]One might prefer to mirror the OOL idea of any value being an object; strictly, there is a problem here because the π-calculus allows only a finite number of instances; how would one show that – in any program – only a finite number of integers were required?

Booleans

$$[\![\mathsf{true}]\!]_b \triangleq b(tf).\bar{t}$$

$$[\![\mathsf{false}]\!]_b \triangleq b(tf).\bar{f}$$

So if v_l then P else Q can be represented as

$$(\nu tf)\bar{l}tf.(t().P + f().Q)$$

Then

$$Copy(b, c) \triangleq (\nu tf)\bar{b}tf.(t().\,[\![\mathsf{true}]\!]_c + f().\,[\![\mathsf{false}]\!]_c)$$

$$And(b, c, d) \triangleq (\nu tf)\bar{b}tf.(t().Copy(c, d) + f().\,[\![\mathsf{false}]\!]_d)$$

Natural numbers

$$[\![0]\!]_l \triangleq l(zs).\bar{z}$$

$$[\![succ(n)]\!]_l \triangleq (\nu l')(l(zs).\bar{s}l' \mid [\![n]\!]_{l'})$$

So, for example $[\![1]\!]$ is

$$(\nu l')(l(zs).\bar{s}l' \mid l'(zs).\bar{z})$$

Then[15]

$$Copy(l, m) \triangleq (\nu zs)\bar{l}zs.(z().\,[\![0]\!]_m + s(l').(\nu m')(m(zs).\bar{s}m' \mid Copy(l', m')))$$

$$Add(k, l, m) \triangleq (\nu zs)\bar{k}zs.(z().Copy(l, m) + s(k').(\nu m')(m(zs).\bar{s}m' \mid Add(k', l, m')))$$

$$Equal(l, m, b) \triangleq (\nu zs)\bar{l}zs.(z().EqualZ(m, b) + s(l').EqualNZ(l', m, b))$$

$$EqualZ(m, b) \triangleq (\nu zs)\bar{m}zs.(z().\,[\![\mathsf{true}]\!]_b + s(m').\,[\![\mathsf{false}]\!]_b)$$

$$EqualNZ(l', m, b) \triangleq (\nu zs)\bar{m}zs.(z().\,[\![\mathsf{false}]\!]_b + s(m').Equal(l', m, b))$$

C.3 Mapping

This section develops a mapping from $\pi o\beta\lambda$ to the version of the polyadic π-calculus given above. In the spirit of Landin [Lan66] – π-calculus equivalents of increasingly complex programs are shown.

[15] It is – just – worth recording the SORTs here: with $\{N \mapsto (Z, S), Z \mapsto (), S \mapsto (N)\}$, then (including primed versions) k, l, m: N, (for $n \in \mathbb{N}$) n: N, z: Z, s: S.

Classes

Consider the 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: the creation of instances of classes is modelled by replication with a private name (u) being passed out for each instance. (It is assumed that a static association will be made of names like c to class names like C.)

$$!\,(\nu u)\overline{c}u.(\nu\alpha)(\overline{\alpha} \mid G_u) \tag{12}$$

Then the creation of new instances of C (new C) can be modelled by

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

A 'baton' (α) is used to make sure that only one method (within any particular instance) is active at any one time.[16] [17]

So, in outline

$$G_u \stackrel{\text{def}}{=}\ !\,\alpha().\cdots.(s_1(\cdots).\cdots.\overline{\alpha} + s_2(\cdots).\cdots.\overline{\alpha})$$

The selection of method is handled by passing out two names ($s_1 s_2$) so that the invoking process can use the appropriate one.

$$G_u \stackrel{\text{def}}{=}\ !\,\alpha().(\nu s_1 s_2)\overline{u}s_1 s_2.(s_1(f_1 x).\overline{f_1}.\overline{\alpha} + s_2(f_2).\overline{f_2}u.\overline{\alpha}) \tag{13}$$

The method $u!m_1(e)$ is invoked by

$$u(s_1 s_2).(\nu f_1)\overline{s_1}f_1 e.f_1()$$

and $u!m_2()$ is invoked by

$$u(s_1 s_2).(\nu f_2)\overline{s_2}f_2.f_2(u')$$

Instance variables

Consider the class with methods which set (s) and access (a) an instance variable (v) where variables contain names.

C class
vars v: $\mathbb{N} \leftarrow$ nil
$s(x)$ method $v \leftarrow x$; return
$a()$ method return v

[16]This is the coding for recursion given in [Mil92].
[17]This 'mutex' behaviour for methods could be relaxed as in VDM++ [Dür92]: I might even follow their use of Deontic logic to fix the activation possibilities.

This can be modelled as in Equation 12 with an additional process (M_v and baton α_v) for each instance variable.

$$! \, (\nu u)(\overline{c}u.(\nu \alpha \alpha_v v_r v_w)(\overline{\alpha_v}\mathsf{nil} \mid M_v \mid \overline{\alpha} \mid G_u)) \tag{14}$$

The instance variable itself is like a class for which only one instance is required; this degenerate class has methods for read (m_r) and write (m_w) whose interface is simple because they can only be invoked from one place this is modelled by

$$M_v \stackrel{\text{def}}{=} \, ! \, \alpha_v(y).(\overline{v_r}y.\overline{\alpha_v}y + v_w(z).\overline{\alpha_v}z) \tag{15}$$

The definition of G_u is:[18]

$$G_u \stackrel{\text{def}}{=} \, ! \, \alpha().(\nu s_s s_a)\overline{u}s_s s_a.(s_s(f_s x).\overline{v_w}x.\overline{f_s}.\overline{\alpha} + s_a(f_a).v_r(y).\overline{f_a}y.\overline{\alpha}) \tag{16}$$

Code after return statement

Were the s method above written

C class
$\ldots$
$s(x)$ method return ; $v \leftarrow x$
$\ldots$

then the $\overline{v_w}x.\overline{f_s}$ is commutated in Equation 16 to give

$$G_u \stackrel{\text{def}}{=} \, ! \, \alpha().(\nu s_s s_a)\overline{u}s_s s_a.(s_s(f_s x).\overline{f_s}.\overline{v_w}x.\overline{\alpha} + \cdots) \tag{17}$$

Of course, under what circumstances this is equivalent to Equation 16 (or even what this means) is the interesting question (see Section C.4).

Yield statement

The yield construct is handled by passing on the name (say, f) to which the method containing the construct was to return its result. Thus while

C class
vars l: private ref(C)
$f(\ldots)$ method $\cdots$ return $l!f(\cdots)$

is modelled by (cf. Equation 16)

$$! \, \alpha().(\nu s_f)(\overline{u}s_f.(s_f(f_f x). \cdots v_r(u').u'(s_{f'}).(\nu f_{f'})(\overline{s_{f'}}f_{f'}x.f_{f'}(r).\overline{f_f}r.\overline{\alpha})))$$

substituting yield for return gives

$$! \, \alpha().(\nu s_f)(\overline{u}s_f.(s_f(f_f x). \cdots v_r(u').u'(s_{f'}).\overline{s_{f'}}f_f x.\overline{\alpha}))$$

[18] Here it is worth recording the SORTs: with $\{C \mapsto (U), U \mapsto (S_S, S_A), S_S \mapsto (F_S, VAL), S_A \mapsto (F_A), F_S \mapsto (), F_A \mapsto (VAL), A \mapsto (), A_{VAL} \mapsto (VAL) < V_R \mapsto (VAL), V_W \mapsto (VAL)\}$ then c: C, u: U, s_s: S_S, s_a: S_A, f_s: F_S, f_a: F_A, α: A, α_V: A_{VAL}, v_r: V_R, v_w: V_W.

Statement composition

So far, order has been modelled by prefixing (see Equation 13); an alternative is to have a link on which a completion signal is sent and to use composition. So $S_1; S_2$ signals termination on l by

$$(\nu l')(S_1(l') \mid l'().S_2(l))$$

and skip statements are modelled by

$$\bar{l}.\mathbf{0}$$

Conditional statements

A conditional statement $\mathsf{if}\ E\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$ can be modelled by

$$(\nu l' l_1 l_2)(BoolEval(l', l_1, l_2) \mid E(l') \mid (l_1.S_1(l) + l_2.S_2(l))) \tag{18}$$

$$BoolEval(l', l_1, l_2) \stackrel{\mathrm{def}}{=} l'(b).\overline{b}tf \mid (t.\overline{l_1} + f.\overline{l_2}) \tag{19}$$

While statements

A while statement $\mathsf{while}\ E\ \mathsf{do}\ S\ \mathsf{od}$ can be modelled by (using the baton trick)

$$(\nu \alpha_w)(\overline{\alpha_w} \mid !\,\alpha_w().W(l)) \tag{20}$$

$$W(l) \stackrel{\mathrm{def}}{=} (\nu l'' l_1 l_2)(BoolEval(l'', l_1, l_2) \mid E(l'') \mid (l_1.S(\alpha_w) + l_2.\bar{l})) \tag{21}$$

There is here a radical alternative: if block statements were added to $\pi o\beta\lambda$, the ST-80 trick of programming out a while statement could obviate the need for this statement as a primitive. (This probably amounts to doing in $\pi o\beta\lambda$ what is done here in the π-calculus.)

Parallel statement

Just maps to composition!

C.4 Proofs

Basic results

To warm up, prove something like $[\![i + j]\!]_m$ is observationally equivalent to

$$(\nu kl)(Add(k, l, m) \mid [\![i]\!]_k \mid [\![j]\!]_l)$$

Transformation 1

Consider the need to justify repositioning the return statement as in Section C.3 (i.e. showing that no $\pi o\beta\lambda$ system can detect which version of C is being used). The composition of the invocation with the denotation of the class is (where the version of the G_u process corresponding to Equation 16 is used; also we unfold – once each – Equations 15 and 16)

$$
\begin{array}{r}
u(s_s s_a).(\nu f_s)(\overline{s_s} f_s l.f_s()).\cdots u(s_s s_a).(\nu f_a)(\overline{s_a} f_a.f_a(r)) \mid \\
(\nu \alpha \alpha_v v_r v_w)((\nu s_s s_a)(\overline{u} s_s s_a.(s_s(f_s x).\overline{v_w} x.\overline{f_s}.\overline{\alpha} + s_a(f_a).v_r(y).\overline{f_a} y.\overline{\alpha})) \mid \\
!\,\alpha().(\nu s_s s_a)(\overline{u} s_s s_a.(s_s(f_s x).\overline{v_w} x.\overline{f_s}.\overline{\alpha} + s_a(f_a).v_r(y).\overline{f_a} y.\overline{\alpha})) \mid \\
\overline{v_r}\mathsf{nil}.\overline{\alpha_v}\mathsf{nil} + v_w(z).\overline{\alpha_v} z \mid \\
!\,\alpha_v(y).(\overline{v_r} y.\overline{\alpha_v} y + v_w(z).\overline{\alpha_v} z))
\end{array}
$$

(Where the elided expressions from the invocation contain no reference to u but other terms in a composition *could* refer to u.) The set of free names of this whole expression is $\{u\}$; intuitively it is easy to see that no further positive occurrence of u is available until after α triggers a further unfolding; therefore the permutation preserves equivalence. More formally, the first two elements of the composition reduce to

$$
\begin{array}{l}
(\nu f_s).(f_s()).\cdots u(s_s s_a).(\nu f_a)(\overline{s_a} f_a.f_a(r)) \mid \\
\qquad (\nu \alpha \alpha_v v_r v_w)(\nu s_s s_a)(\overline{v_w} l.\overline{f_s}.\overline{\alpha} \mid \cdots)
\end{array}
$$

From this it should be clear that $\overline{v_w} l.\overline{f_s}.\overline{\alpha}$ can be commuted to $\overline{f_s}.\overline{v_w} l.\overline{\alpha}$

But handling non-trivial values will mess up the locality of names – unless values are copied – remember distinguishing immutable values is a problem in OOLs so the difficulty is not the fault of the π-calculus.

Transformation 2

Consider the change from return to yield: following a similar pattern to above

$$
\begin{array}{r}
u(s_f).(\nu f_f)(\overline{s_f} f_f x.f_f(r)).\cdots r \cdots \mid \\
(\nu \alpha \alpha_v v_r v_w)((\nu s_f)(\overline{u} s_f.(s_f(f_f x).\cdots v_r(u').u'(s_{f'}).(\nu f_{f'})(\overline{s_{f'}} f_{f'} x.f_{f'}(r).\overline{f_f} r.\overline{\alpha})) \mid \\
!\,\alpha().(\nu s_f)(\overline{u} s_f.(s_f(f_f x).\cdots v_r(u').u'(s_{f'}).(\nu f_{f'})(\overline{s_{f'}} f_{f'} x.f_{f'}(r).\overline{f_f} r.\overline{\alpha})) \mid \\
(\overline{v_r} u'.\overline{\alpha_{u'}} u' + v_w(z).\overline{\alpha_v} z) \mid \\
!\,\alpha_v(y).(\overline{v_r} y.\overline{\alpha_v} y + v_w(z).\overline{\alpha_v} z)))) \mid \\
(\nu \alpha \alpha_v v_r v_w)((\nu s_{f'})(\overline{u'} s_{f'}.(s_{f'}(f_{f'} x).\overline{f_{f'}} \ldots.\overline{\alpha})) \mid \\
!\,\alpha().((\nu s_{f'})(\overline{u'} s_{f'}.(s_{f'}(f_{f'} x).\overline{f_{f'}} \ldots.\overline{\alpha}))))
\end{array}
$$

the composition reduces to

$$
\begin{array}{r}
(\nu f_f).(f_f(r)).\cdots r \cdots \mid \\
(\nu \alpha \alpha_v v_r v_w)(u'(s_{f'}).(\nu f_{f'})(\overline{s_{f'}} f_{f'} x.f_{f'}(r).\overline{f_f} r.\overline{\alpha}) \mid \cdots) \mid \\
(\nu \alpha \alpha_v v_r v_w)((\nu s_{f'})(\overline{u'} s_{f'}.(s_{f'}(f_{f'} x).\overline{f_{f'}} \ldots.\overline{\alpha})) \mid \cdots)
\end{array}
$$

then changing the second line to

$$u'(s_{f'}).\overline{s_{f'}}f_f x.\overline{\alpha}$$

is OK!

Logic

Need to prove that logical expressions make sense. For now at least, look at idea of adding (models of) methods which expose values.

C.5 Related work

Milner already discusses interesting examples in [MPW92][19] While this author was working on an early version of a mapping to the polyadic π-calculus, [Wal91] was sent to him: this mapping from POOL to the monadic version of the calculus [MPW92, MPW91] had a stimulating effect on the work and resulted in a number of changes. Similarly [Wal93] (which *inter alia* maps POOL to the polyadic calculus) provided useful ideas. Other researchers who have provided (process algebra based) semantics of object-oriented languages include Honda and Tokoro ([HT91b] is based on [HT91a]) and [Vaa90] which employs ACP.

[19] See [MPW92, Mil92] for historical notes on name passing calculi.