Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 155 (2006) 43–60

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

An Approach to Splitting Atoms Safely
Extended Abstract

C. B. Jones[1]

School of Computing Science
The University of Newcastle upon Tyne
Newcastle, UK

Abstract

The intention of this paper is to make a contribution to (compositional) development methods for concurrent programs. The topics touched on include interference, atomicity, observability and granularity. The paper sets out some requirements for an approach to developing systems by "splitting atoms safely".

Keywords: Concurrent programs, atomic action, observability, granularity.

1 Introduction

If an action is executed "atomically", it is assumed that it will not be affected by interference and that the environment will not be able to observe intermediate steps of the action in question. The bugbear of concurrency is that interference must be tolerated. With shared state programs, an action must achieve some required result even though its state can be changed by other interfering processes.

The aim here is to argue that there is a useful method for developing concurrent programs which explicitly uses a "fiction of atomicity" as an abstraction and then allows steps of development which "split atoms safely". This development process might be called "refining atomicity". One of the

[1] cliff.jones@ncl.ac.uk

doi:10.1016/j.entcs.2005.11.051

things which makes the approach interesting is that it is used widely: much of database implementation is about preserving the fiction of atomicity when implementations deliberately overlap the execution of transactions.

What is sought here is a general development method which has the properties of other formal methods for developing programs.

2 Interference

What classifies a programming language as "imperative" is the ability of its programs to change some form of "state". Consider, for example, the (VDM – [11,8]) specification given in Figure 1 which indicates how a priority queue might be specified. Starting with an initial state which contains the empty set, operations *ENQ* and *DEQ* (the latter subject to its pre-condition) can be performed in any order; both operations are shown (ext wr) as changing the state (*queue*). The operation *ENQ* takes an argument but delivers no result whereas *DEQ* takes no argument and delivers a result. The function *mins* is assumed to deliver the minimum value from its (non-empty set) argument. It is important to appreciate that the intention of a specification like that in Figure 1 is that the external behaviour is what is defined. It is *not* required that the internal state is implemented with a set data type. The key point is that it is assumed that the only way of *observing* the behaviour of the priority queue is with the (inputs and) outputs of the stated operations. The term "data reification" is used in VDM for steps of development which choose (more) concrete representations for objects.

The post-conditions of Figure 1 define acceptable final results and, for sequential programs, the issue of atomicity is settled.

VDM uses the phrase "operation decomposition" for the use of proof rules for introducing programming language constructs like while. These rules are like "Hoare axioms" except that they cope with VDM's insistence on post-conditions of two states (initial and final). A post-condition does not require execution in a single step, such execution can only be thought of as atomic in the sense of no interference on –and no visibility of– intermediate states. Development is expected to create a program which executes in many steps; but for sequential programs, we ignore interference during their execution.

Since the priority queue example is used below, it is worth highlighting the point about the range of implementations: an implementer could, for example, arrange for quick response to *ENQ* by adding new elements to an unordered state — *DEQ* would then need to determine the minimum element; at the other extreme, there are implementations in which *ENQ* takes more time to place new elements in an ordered data structure so that *DEQ* responds quickly.

$ENQ\ (new: X)$

ext wr $queue\ :\ X$-set

post $queue = \overleftarrow{queue} \cup \{new\}$

$DEQ\ ()\ r: X$

ext wr $queue\ :\ X$-set

pre $queue \neq \{\,\}$

post $r = mins(\overleftarrow{queue}) \wedge queue = \overleftarrow{queue} - \{r\}$

Fig. 1. A specification for a Priority Queue

Section 5 shows how concurrency can be used to make both operations respond quickly.

Both data reification and operation decomposition are "compositional" in the sense that the specifications define all that is required of an implementation. In other words, one can prove that one step of design is correct and know that –if sub-components are developed according to their specifications– there will not be a need to reject them because they do not fit their context.

Interference makes it difficult to find compositional development methods for concurrent systems.

The "Owicki/Gries method" [17] extended Hoare-like approaches to handle concurrency but the resulting method is non-compositional in the sense that proven developments of independent processes might have to be discarded if they fail a final "interference freedom" property *of their proofs*.

It was striving for compositionality which led this author to look for ways of documenting interference using rely and guarantee conditions. Essentially, a *rely condition* records the interference which an implementation must tolerate on its state and a *guarantee condition* documents a limit on the interference the component can generate. Both rely and guarantee conditions are –like VDM's post-conditions– relations over pairs of states. As with pre-conditions, rely conditions can be thought of as giving permission for the implementer to ignore certain potential deployments of the code to be created. On the other hand, guarantee conditions are like post-conditions in that they record an obligation on the created program.

Papers like [10,21] contain proof rules for showing that a decomposition into parallel processes will be correct if the components are developed so as to satisfy their specifications. The details of particular methods vary and are not important here; [6] contains detailed comparisons of different compositional and non-compositional approaches.

3 Atomicity in SOS

What is of more interest in this paper is the connection between interference and atomicity.

One reason for looking here at techniques for describing language semantics is to indicate that SOS [18] provide a straightforward way to formalize many of the points on granularity etc. A stronger justification for exploring SOS can be seen in Section 6 when the issue of justifying design methods is addressed.

One can think of an SOS description as defining a relation over pairs of program texts and states; thus

$$\xrightarrow{s} : \mathcal{P}((\mathit{Statement} \times \Sigma) \times (\mathit{Statement} \times \Sigma))$$

What is going on with shared variable concurrency is that interference occurs when more than one thread of control can read and/or write to the same portion of a state.

As a simple illustration, consider parallel execution of two sequences of assignments. Ignoring for now the possibility of non-determinism in expression evaluation, use

$$\mathit{eval}: \mathit{Expression} \times \Sigma \to \mathit{Value}$$

The (atomic) execution of the assignment statement from the head of the left sequence is expressed as

$$\frac{v = \mathit{eval}(e, \sigma)}{([x \leftarrow e] \curvearrowright \mathit{restleft} \mid\mid \mathit{right}, \sigma) \xrightarrow{s} (\mathit{restleft} \mid\mid \mathit{right}, \sigma \dagger \{x \mapsto v\})}$$

With the obvious symmetric rule for the right sequence,

$$\frac{v = \mathit{eval}(e, \sigma)}{(\mathit{left} \mid\mid [x \leftarrow e] \curvearrowright \mathit{restright}, \sigma) \xrightarrow{s} (\mathit{left} \mid\mid \mathit{restright}, \sigma \dagger \{x \mapsto v\})}$$

This shows how non-determinism can arise depending on the order in which statements are executed from the two parallel streams. Thus

$$(x \leftarrow x * 2; x \leftarrow x * 3) \mid\mid (x \leftarrow x * 4; x \leftarrow x * 5)$$

will, if the initial value of x is 1, set the final value of x to factorial 5 whatever order the assignment statements interleave. Whereas, when x starts at 1

$$(x \leftarrow x + 1) \mid\mid (x \leftarrow x * 2)$$

can leave x as 3 or 4.

Although the basic points are illustrated here with assignment statements, this should not disguise the fact that the same issues arise at different language levels. Interference could be shown with separate programs accessing shared files or separate "transactions" changing a database. Appendix A outlines an SOS for the concurrent OOL used in Section 5.

4 Granularity and reification

The operational semantics in the previous section shows assignment statements being executed atomically. That is, if the head of *left* (say $x \leftarrow e$) is being executed, there are no state changes made between the beginning of the evaluation of e and the change of x; nor can *right* be observing x whilst it is being changed. For a useful programming language, such an assumption of atomicity is unrealistic in that it would be extremely expensive to implement (in terms of say semaphore setting).

One attempt to avoid the problems posed by two threads referring to shared variables is to say that any assignment statement can use (in either left or right-hand contexts) at most one shared variable. This is sometimes referred to as "Reynolds rule". It has its own obvious disadvantage in that *any* statement of the form

$$x \leftarrow e(x)$$

has to be rewritten as

$$local \leftarrow e(x); x \leftarrow local$$

even where the logic of the program shows that in this context, no other thread could change x.

More seriously, this idea gives no clue as to how one might handle variables which cannot be accessed and/or changed atomically: consider for example array or record assignments (but it is not even safe to assume that numbers can be changed by an indivisible machine operation).

The preceding section explains how rely and guarantee conditions are assertions about interference but the cited papers on this way of developing concurrent programs only hint at the question of "atomicity". In cases like simple (Boolean) switches, a rely condition which states that the environment will never set *myswitch* to false

$$\overleftarrow{myswitch} \;\Rightarrow\; myswitch$$

is safe. Conditions that, say, state a variable is monotonically decreasing are quite often needed in program developments using rely/guarantee conditions and can be more delicate in the sense that realising changes atomically can be difficult in most programming languages.

In many such cases, there is a fascinating interplay with data reification. There is an example in [17] where two (or more) parallel processes are searching for the least index (i) to an array A for which a predicate $p(A(i))$ holds. A compositional development of a generalisation of this example is given in [9]. A key point in the development there states that both processes rely on, and guarantee, that the lowest index t where such a value has been found monotonically decreases ($t \leq \overleftharpoon{t}$). If t were itself a shared variable, even an assignment like $t \leftarrow v$ would not safely decrease t in the case where $v \leq t$ at the start of execution because interference could reduce t to a value less than v. What is actually done in [9] is to reify t into two variables v_1 and v_2 and use the retrieve function

$$t = \mathit{min}(v_1, v_2)$$

(The ith process can write v_i; either process can read both v_j.) The ith process can now reduce t by changing v_i without fear of interference and *without atomicity assumptions* on things like assignment statements.

This observation appears to be important because it is echoed in other examples — some of which are considerably more complicated. In fact, the extremely subtle “Asynchronous Communication Mechanisms” (ACM) like Simpson’s “four slot” algorithm can be understood in this way. The essence of the problem with ACMs is to have a shared variable into which processes can both read and write without ever waiting on any locks. If it is not assumed that read and write of whole variables is atomic, this becomes very difficult. Hugo Simpson has shown that –under the assumption only of atomic update of some 1-bit indicators– this can be achieved with 4 slots for values (see [19] and back references therein). The paper [7] develops this example using data reification.

The same link between data reification and interference can be seen in the development of a concurrent “Sieve of Eratosthenes”. The aim is to arrange that some shared set s is set to exactly the set of primes up to some value n. A sequential sieve would first remove all multiples (2 and above) of 2, then find the next lowest value in s and remove its multiples and so on up to the square root of n. The idea of a parallel version is to use a family of parallel processes $\mathit{Rem}(i)$ –one for each index between 2 and the square root of n– and make them responsible for removing the multiples of their index. The specification of each Rem process (i) cannot define a precise set which will exist after the process completes (because other processes also remove elements); (ii) can specify that all of its multiples should have been removed; (iii) can only achieve this under a rely condition that no one re-inserts deleted values; (iv) must guarantee never to re-insert values itself; (v) must guarantee only to remove composites. Full developments of this example can be found in several of the cited papers.

Here, the interest is on the reification of the set s. Even if one had a programming language which supported variables of type set, an assignment like

$$s := s - \{i * n\}$$

would not satisfy the guarantee condition in the presence of interference (e.g. having accessed s to compute the set difference, another process could remove some composite j which would then be re-inserted by the assignment to s). Again, it is the choice of a data representation for s which makes things work. Essentially, s is represented by a vector of bits; an element of the set can be removed by setting the corresponding bit to false; this operation is assumed to be atomic.

5 Splitting atoms safely

If a development method is to be found for concurrent programs, it is worth looking back at data reification and operation decomposition (under which heading, for now, development of concurrent programs using rely/guarantee conditions is included). These approaches are compositional in a useful sense. From a short specification, key early design decisions can be recorded and justified in the knowledge that further, more detailed, steps will not invalidate the early decisions. Development processes, however formal, which create a whole program which is then subjected to some final "check" leave the danger of massive "scrap and rework". Notice that this applies whether the *post facto* check is testing, model-checking or even proof. This is precisely the problem with Owicki's final "interference freedom" proof: having completely developed separate programs and shown they satisfy their individual post-conditions, they might have to be discarded because a statement in one interferes with a proof step in another.

So we would hope to find methods for developing concurrent programs which are also compositional. What is being suggested here is to design as though things will be atomic and then to allow steps of the processes to overlap. This approach might be called "atomicity refinement".

There are obviously cases where it is trivial and cases where it is invalid. Almost all of our programs are now run on operating systems which share the resources of the hardware; even the physical memory itself is shared via paging schemes; but it is the responsibility of the operating system to keep such programs completely independent from interfering with each other. Mostly, they do this successfully.

Atomicity can be relaxed where there is no danger of interference.

On the other hand, it is not valid to relax the (unrealistic) assumption of atomicity on assignments in the sequence of statements in Section 2 for computing factorial 5.

Once one is aware of the power of the "fiction of atomicity", it becomes clear that it is a very useful way of understanding the intention of a large range of core concurrency ideas. [2]

In spite of the fact that this abstraction appears so useful, there is no general way of arguing that the subsequent splitting of atoms is safe. The database literature (see [2] for references) is interesting but the methods exhibited there are honed to their specific application and would not offer a general development method.

A worked example from another domain is worth considering as an existence proof for a development method.

The example of atomicity refinement in [12] satisfies the properties sought for a development method. It can be applied to the development of the specification given in Figure 1. (This example is simpler than examples in earlier papers; the point here is to draw out the lessons.) In particular some new comments about observability give pointers to the further work sketched in the last section of this paper.

After developing rely/guarantee reasoning, it was seen to be "heavy" and in need of curtailment. What was needed was a clear way to show where interference could not occur and to limit the use of interference proofs to minimum portions of the program.

Object-oriented languages in general –and POOL [1] in particular– provided the inspiration for the next step. Class-based OOLs offer the program designer ways to control the degree of interference: local "instance" variables are only accessible via local methods; the degree to which references (i.e. pointers to instances) are shared between objects governs the degree of interference. In earlier papers a language ($\pi o \beta \lambda$) was explored which required that only one method was active in any object instance at one point in time. Coupled with the identification of private references which (among other restrictions) could not be copied, an intermediate class of interference control was described where

[2] There is a sense in which the implementation of databases is all about offering a fiction of atomicity. It is easy to write an operational semantics for the processing of transactions which shows one transaction being selected (from a set of waiting transactions) and executed atomically. The task of the implementer is to achieve one of the possible (non-deterministic) outcomes described by this semantics. Because transactions can run for a (relatively) long time, an implementation actually needs to overlap their processing. But this must be done in a way which does not introduce any new results (such as losing the bank's money because of contention on a database).

"islands" were immune from interference and could execute in parallel with other processes.

For the priority queue specified in Figure 1, it is a straightforward step of data reification to represent the set by an ordered sequence of values. It is also not difficult to develop (by sequential operation decomposition) the $(\pi o \beta \lambda)$ program in Figure 2[3] which sequentially $i$nserts a value into its correct position.

The idea now is to introduce concurrency into such a sequential program. Figure 3 shows the return statement at the head of the $i$nsert method. In contrast to the sequential program, this version of $i$nsert would release its client from the *rendezvous* immediately and the client could proceed in parallel with the activity within the chained sequence of *Priq* elements. Furthermore, as soon as a call is made to $i$nsert in the l element, activity can ripple down the sequence in parallel.

These transformations depend on the properties of references. A preliminary definition fixes what can(not) be done with private references.

A private reference is defined to be one which is never 'copied' nor which has general (unshared) references passed over it – neither in nor out (since one can't pass private references, this restricts to references to 'immutable' objects).

One equivalence is:

S; return(e) is equivalent to return(e); S

providing

- S contains no return or delegate statements and always terminates;
- e is a simple expression and is not affected by S; and
- every method invoked by S belongs to objects reached by private references.

Not all programs are intended to terminate; even if they are, termination is not a syntactically checkable property; but it is in the spirit of the development method envisaged that termination would be proved for relevant methods. (This point does however make it doubtful whether the kind of equivalences being considered are suitable for automatic application by a compiler.)

Another equivalence is:

return $l.m(x)$ is equivalent to delegate $l.m(x)$

providing

- $l.m(x)$ terminates; and
- l is a unique reference.

[3] Because the names of variables and methods are used in semantic expressions in Appendix B, the opportunity is taken to use single character names.

```
class $Q$
vars $v$: $\mathbb{N} \leftarrow 0$; $l$: private reference $Q \leftarrow$ nil;
$i(n\colon \mathbb{N})$ method
  begin
    if $l =$ nil then ($l \leftarrow$ new $Q$; $v \leftarrow n$)
    elif $v < n$ then $l.i(n)$
    else ($l.i(v)$; $v \leftarrow n$)
    fi;
    return
  end
$r()$ method $r\colon \mathbb{N}$
  ...
$c(n\colon \mathbb{N})$ method $r\colon \mathbb{B}$
  if $l =$ nil then return(false)
  elif $v < n$ then return($l.c(n)$)
  elif $v = n$ then return(true)
  else return(false)
  fi
end $Q$
```

Fig. 2. Priority queue program (sequential)

In conclusion, it is worth commenting that it is very difficult even to specify concurrent $\pi o\beta\lambda$ programs like that in Figure 3. Any attempt to use pre/post-conditions would have to overcome the problem that both the initial and final 'states' are combinations of values and unfinished activity. Such a specification would at least need some form of auxiliary variable. So the approach of introducing parallelism by showing a program which is equivalent to a sequential program (whose specification was simple) has avoided considerable complication.

The equivalences had to be justified against some semantics — two are sketched in Appendices A and B. Unfortunately, it proved non-trivial to justify the equivalence rules of $\pi o\beta\lambda$ via the mapping to the π-calculus (see Appendix B). Attempts include research by David Walker and Davide Sangiorgi (who used "barbed bi-simulation" and introduced the idea of "uniformly receptive processes").

Interestingly, it is *not* true that the behaviour of the sequential and concurrent $i$nsert operations is identical. Of course, we have already observed that operation decomposition introduces extra steps and that data reification changes internal (state) representations. But in both of these cases, there is a clear notion of what is observable "at the interface" (via external types). A

```
class $Q$
vars $v$: $\mathbb{N} \leftarrow 0$; $l$: private reference $Q \leftarrow$ nil;
$i(n: \mathbb{N})$ method
   begin
      return;
      if $l =$ nil then $(l \leftarrow$ new $Q; v \leftarrow n)$
      elif $v < n$ then $l.i(n)$
      else $(l.i(v); v \leftarrow n)$
      fi
   end
$r()$ method $r: \mathbb{N}$
   ...
$c(n: \mathbb{N})$ method $r: \mathbb{B}$
   if $l =$ nil then return(false)
   elif $v < n$ then delegate$(l.c(n))$
   elif $v = n$ then return(true)
   else return(false)
   fi
end $Q$
```

Fig. 3. Priority queue program (parallel)

sufficiently rich observation language *could* observe that in

$$i(2); i(3); i(1)$$

the *completion* of $i(1)$ can precede that of $i(3)$ in the concurrent queue but not in the sequential queue. The point is that $\pi o\beta\lambda$ is (deliberately) expressively too weak to be able to detect such differences. (The situation with $c$heck is more complicated but gives rise to a similar collection of issues.)

The transformational introduction of concurrency by the use of $\pi o\beta\lambda$'s equivalence rules comes close to meeting the properties sought of development methods.

6 Further work

A beginning has been made on formalising notions of "atomicity refinement". But, as nearly always in research, much remains to be done. (One non-issue should be recognised: just switching to a Process Algebra is not a solution to "interference": communication based concurrency just has to deal with interfering communication.)

In looking at more transformations, it will be necessary to define the extent of their language dependence. Their justification will require looking rather broadly at interpreted notions of "context". Observability is clearly crucial.

The ubiquity of informal "atomicity refinement" is both challenging and a pointer to considerable intellectual leverage. (There are of course, many concurrent programs which do not fit the mould of atomicity refinement.)

It is important to recognise other related contributions and to understand the extent to which they might have already solved sub-problems that relate to atomicity refinement. The work on refinement calculi [4,14,3] has led to the notion of "Action Systems".

It will be interesting to see whether embedding SOS definitions like that in Appendix A in "logical frames" offers any purchase on the proof of the methods themselves. If one views $\xrightarrow{s}$ as a relation, it can be embedded directly into a proof tool. The "Plotkin rules" are used directly as the inference rules about programs and the logical frame of the proof tool is enriched to reason about the logic of $\xrightarrow{s}$. (Tobias Nipkow and colleagues have done this for definitions of significant subsets of Java [16]; the idea is also discussed in [13] but appears to originate with [5].)

Acknowledgments

For funding to support my research, I am grateful to the (UK) EPSRC funding for the Dependability IRC (see www.dirc.org.uk) and for the new project "Splitting atoms safely"; and to the EU-IST funding for "Rodin" (see rodin.cs.ncl.ac.uk).

References

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

[2] A-Team. The atomicity manifesto: a story in four quarks, 2005. ACM SIGMOD Record.

[3] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A systematic Introduction.* Springer Verlag, 1998.

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

[5] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, August 1992.

[6] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Cambridge University Press, 2001.

[7] N. Henderson and S. E. Paynter. The formal classification and verification of Simpson's 4-slot asynchronous communication mechanism. In L.-H. Eriksson and P.A Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 350–369. Springer Verlag, 2002.

[8] ISO. VDM-SL. Technical Report Draft International Standard, ISO/IEC JTC1/SC22/WG19 N-20, 1995.

[9] 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.

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

[11] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.

[12] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[13] Cliff Jones. Operational semantics: concepts and their expression. *Information Processing Letters*, (88):27–32, 2003.

[14] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

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

[16] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a java-like language. Manuscript, Munich, 2004.

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

[18] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[19] H. R. Simpson. New algorithms for asynchronous communication. *IEE, Proceedings of Computer Digital Technology*, 144(4):227–231, 1997.

[20] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

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

[22] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes.* Cambrisge University Press, 2001.

A SOS of $\pi o\beta\lambda$

Abbreviations used throughout:

Expr	Expression
Obj	Object
Meth	Method
Stmt	Statement
Var	Variable

Abstract Syntax

A *Program* contains a collection of named *ClassBlock*s; it is also here assumed that execution begins with a single (parameterless) method call.

$$\begin{array}{lll} \mathit{Program} :: & \mathit{classes} & : \mathit{Id} \xrightarrow{m} \mathit{ClassBlock} \\ & \mathit{start\text{-}class} & : \mathit{Id} \\ & \mathit{start\text{-}meth} & : \mathit{Id} \end{array}$$

$$\begin{array}{lll} \mathit{ClassBlock} :: & \mathit{vars} & : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & \mathit{meths} & : \mathit{Id} \xrightarrow{m} \mathit{Meth} \end{array}$$

$$\mathit{Type} = \mathit{ScalarType} \mid \mathit{Id}$$

$$\mathit{ScalarType} = \textsc{IntTp} \mid \textsc{BoolTp}$$

$$\begin{array}{lll} \mathit{Meth} :: & \mathit{returns} & : \mathit{Type} \\ & \mathit{params} & : \mathit{Id}^* \\ & \mathit{paramtps} & : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & \mathit{body} & : \mathit{Stmt}^* \end{array}$$

$$\mathit{Stmt} = \mathit{Assign} \mid \mathit{If} \mid \mathit{New} \mid \mathit{MethCall} \mid \mathit{Release} \mid \mathit{Delegate}$$

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

$$\begin{array}{lll} \mathit{If} :: & \mathit{test} & : \mathit{Expr} \\ & \mathit{th} & : \mathit{Stmt}^* \\ & \mathit{el} & : \mathit{Stmt}^* \end{array}$$

$$\begin{array}{lll} \mathit{New} :: & \mathit{targ} & : \mathit{Id} \\ & \mathit{class} & : \mathit{Id} \end{array}$$

$$\begin{array}{lll} \mathit{MethCall} :: & \mathit{lhs} & : \mathit{Id} \\ & \mathit{obj} & : \mathit{Id} \\ & \mathit{meth} & : \mathit{Id} \\ & \mathit{args} & : \mathit{Id}^* \end{array}$$

$$\mathit{Release} :: \mathit{val} : \mathit{Expr}$$

$$\begin{array}{lll} \mathit{Delegate} :: & \mathit{obj} & : \mathit{Id} \\ & \mathit{meth} & : \mathit{Id} \\ & \mathit{args} & : \mathit{Id}^* \end{array}$$

The definition of *Expr*s is straightforward and is omitted here.
Furthermore, the *Context conditions* for the language are also elided.

Semantic objects

Dynamic information[4] about Objects is stored in:

$$\mathit{ObjMap} = \mathit{Handle} \xrightarrow{m} \mathit{Oinfo}$$

$$\begin{array}{lll} \mathit{Oinfo} :: & \mathit{class} & : \mathit{Id} \\ & \mathit{state} & : \mathit{VarState} \\ & \mathit{status} & : \mathit{Status} \\ & \mathit{remaining} & : \mathit{Stmt}^* \\ & \mathit{client} & : [\mathit{Handle}] \end{array}$$

$$\mathit{VarState} = \mathit{Id} \xrightarrow{m} \mathit{Val}$$

$$\mathit{Val} = \mathit{Handle} \mid \mathbb{Z} \mid \mathbb{B}$$

$$\mathit{Status} = \textsc{active} \mid \textsc{idle} \mid \mathit{Wait}$$

$$\mathit{Wait} :: \mathit{lhs} : \mathit{Id}$$

SOS rules

The types of the semantic relations are

$$\xrightarrow{s} : \mathcal{P}((\mathit{ClassTypes} \times \mathit{ObjMap}) \times \mathit{ObjMap})$$

and

$$\xrightarrow{e} : \mathcal{P}((\mathit{Expr} \times \mathit{VarState}) \times \mathit{Val})$$

For *Assign*

[4] Text can be obtained from *ClassTypes*

$$\frac{\begin{array}{l} O(a) = \textit{mk-Oinfo}(c, \sigma, \textsc{active}, [\textit{mk-Assign}(\textit{lhs}, \textit{rhs})] \curvearrowright \textit{rl}, \textit{cl}) \\ (\textit{rhs}, \sigma) \xrightarrow{e} v \\ \textit{aobj}' = \textit{mk-Oinfo}(c, \sigma \dagger \{\textit{lhs} \mapsto v\}, \textsc{active}, \textit{rl}, \textit{cl}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \textit{aobj}'\}}$$

For *New*

$$\frac{\begin{array}{l} O(a) = \textit{mk-Oinfo}(c, \sigma, \textsc{active}, [\textit{mk-New}(\textit{targ}, c')] \curvearrowright \textit{rl}, \textit{cl}) \\ b \in (\textit{Handle} - \mathsf{dom}\, O) \\ \textit{aobj}' = \textit{mk-Oinfo}(c, \sigma \dagger \{\textit{targ} \mapsto b\}, \textsc{active}, \textit{rl}, \textit{cl}) \\ \sigma_b = \{v \mapsto \ldots \mid \ldots\} \\ \textit{nobj} = \textit{mk-Oinfo}(c', \sigma_b, \textsc{idle}, [\,], \mathsf{nil}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \textit{aobj}', b \mapsto \textit{nobj}\}}$$

For invoking methods; note $\sigma(obj)$ must be quiescent

$$\frac{\begin{array}{l} O(a) = \textit{mk-Oinfo}(c, \sigma, \textsc{active}, [\textit{mk-MethCall}(\textit{lhs}, \textit{obj}, \textit{meth}, \textit{args})] \curvearrowright \textit{rl}, \textit{cl}) \\ O(\sigma(\textit{obj})) = \textit{mk-Oinfo}(c', \sigma', \textsc{idle}, [\,], \textsc{nil}) \\ C(c') = \textit{mk-ClassBlock}(\textit{vars}, \textit{meths}) \\ \textit{aobj}' = \textit{mk-Oinfo}(c, \sigma, \textit{mk-Wait}(\textit{lhs}), \textit{rl}, \textit{cl}) \\ \sigma'' = \sigma' \dagger \{(\textit{meths}(\textit{meth}).\textit{params})(i) \mapsto \sigma(\textit{args}(i)) \mid i \in \mathsf{inds}\, \textit{args}\} \\ \textit{sobj} = \textit{mk-Oinfo}(c', \sigma'', \textsc{active}, \textit{meths}(\textit{meth}).\textit{body}, a) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \textit{aobj}', \sigma(\textit{obj}) \mapsto \textit{sobj}\}}$$

When a method finishes (remember the *Release* can have occurred earlier) it returns to the quiescent status.

$$\frac{\begin{array}{l} O(a) = \textit{mk-Oinfo}(c, \sigma, \textsc{active}, [\,], \textit{cl}) \\ \textit{aobj}' = \textit{mk-Oinfo}(c, \sigma, \textsc{idle}, [\,], \mathsf{nil}) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto \textit{aobj}'\}}$$

Releasing a *rendez-vous*

$$\frac{\begin{array}{l} O(a) = mk\text{-}Oinfo(c, \sigma, \textsc{active}, [mk\text{-}Release(e)] \curvearrowright rl, cl) \\ (e, \sigma) \xrightarrow{e} v \\ O(cl) = mk\text{-}Oinfo(c', \sigma', mk\text{-}Wait(lhs), sl, cl') \\ aobj' = mk\text{-}Oinfo(c, \sigma, \textsc{active}, rl, \mathsf{nil}) \\ clobj' = mk\text{-}Oinfo(c', \sigma' \dagger \{lhs \mapsto v\}, \textsc{active}, sl, cl') \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', cl \mapsto clobj'\}}$$

Delegation

$$\frac{\begin{array}{l} O(a) = mk\text{-}Oinfo(c, \sigma, \textsc{active}, [mk\text{-}Delegate(obj, meth, args)] \curvearrowright rl, cl) \\ O(\sigma(obj)) = mk\text{-}Oinfo(c', \sigma', \textsc{idle}, [\,], \mathsf{nil}) \\ C(c') = mk\text{-}ClassBlock(vars, meths) \\ aobj' = mk\text{-}Oinfo(c, \sigma, \textsc{active}, rl, \mathsf{nil}) \\ \sigma'' = \sigma' \dagger \{(meths(meth).params)(i) \mapsto \sigma(args(i)) \mid i \in \mathsf{inds}\, args\} \\ sobj = mk\text{-}Oinfo(c', \sigma'', \textsc{active}, meths(meth).body, cl) \end{array}}{(C, O) \xrightarrow{s} O \dagger \{a \mapsto aobj', \sigma(obj) \mapsto sobj\}}$$

B Mapping $\pi o\beta\lambda$ to a process algebra

Another way to give the semantics of a language is to map it to a known language. It is desirable that it is relatively easy to prove results in this "known" language. This is the essence of "denotational semantics" (see [20]): imperative (sequential) languages are mapped to the Lambda calculus which has a sound mathematical basis in terms of which proofs can be conducted.

It is shown below that there is a rather natural mapping from $\pi o\beta\lambda$ to the π-calculus [15,22] in the sense that the expansion is linear and there are concepts in this process algebra which nicely capture key facets of concurrent OOLs. Given that there is an algebra of the π-calculus, this presents a good starting point for proofs.

It is not the intention here to present a full mapping function from $\pi o\beta\lambda$ to the π-calculus. Instead, the main points can be illustrated by looking at the π-calculus equivalent of one $\pi o\beta\lambda$ program. An example can be used to illustrate the main points of the mapping. Figure 2 contains a $\pi o\beta\lambda$ class which has methods for inserting (into an ordered sequence); removing the least element from the head of a queue; and checking if a value is in the queue. If the values were priorities and there was some other information associated with each entry, this might be a self-organising priority queue.

The class definition itself maps naturally to π-calculus's replication; this ensures that an arbitrary number of instances can be generated

$$[\![Q]\!] = !\, I_Q$$

After suitable hiding, I_Q emits a unique name for each instance of the class

$$I_Q = \overline{q}u.B_Q$$

Thus a $\mathsf{new}(Q)$ is translated into a receipt of the unique "capability" (or "handle") for the new object

$$q(u).\cdots$$

The hiding involved limits the visibility of the chatter with the processes which correspond to the two internal variables $(v,\ l)$. Thus the mapping (with $\widetilde{s} = [s_v, s_l]$ and $\widetilde{a} = [a_v, a_l]$) is actually

$$I_Q = (\boldsymbol{\nu}\widetilde{s}\widetilde{a})(v_{nil} \mid l_{nil} \mid \overline{q}u.B_Q)$$

We will not be concerned with the (indexed) processes v and l here other than to know that their values are set by the corresponding s (and accessed by the corresponding a) actions.

The B_Q process emits on its private name (u) a sequence $\widetilde{\alpha} = [\alpha_i, \alpha_r, \alpha_c]$ of unique names for its methods

$$B_Q = \overline{u}\widetilde{\alpha}.M_Q$$

The π-calculus corresponding to any method has a similar form – for the i method, which takes a parameter n and delivers no result,

$$\alpha_i(\omega n).[\![body_i]\!].\overline{\omega}.B_Q$$

One can see how ω is used to signal the completion of the method; also how the recursion on B_Q reflects the "single method active" rule of $\pi o\beta\lambda$.

Thus the π-calculus corresponding to an invocation of $i(5)$ is

$$q(s).s(\widetilde{\alpha}).(\boldsymbol{\nu}\omega)(\overline{\alpha_i}\omega 5).\omega().\cdots$$

One can now easily see the effect of commuting the return to the head of the i method

$$\alpha_i(\omega n).\overline{\omega}.[\![body_i]\!].B_Q$$