Electronic Notes in Theoretical Computer Science 1 (1995)
URL: http://www.elsevier.nl/locate/entcs/volume1.html 6 pages

Fixing the Semantics of Some Concurrent Object-Oriented Concepts: Extended Abstract

C. B. Jones

Department of Computer Science,
University of Manchester,
Manchester, U.K.

Abstract

Concurrent object-oriented languages provide a suitable target for a compositional design process that copes with the interference inherent with concurrency. Fixing the semantics of an object-based design language has been undertaken using structured operational semantics and by a mapping to the pi-calculus. These two approaches are outlined and contrasted. In particular, the difficulties in the two approaches of justifying the proof rules of the proposed design method are explained.

The language $\pi o\beta\lambda$ is intended as a design language for concurrent object-oriented programs. A description of the development methods envisaged can be found in [7,9] (the material from both of these conference papers is available electronically as [6]) and is briefly discussed in another extended abstract in these proceedings. Because it is not itself intended as a programming language, $\pi o\beta\lambda$ is relatively small (an even smaller subset of $\pi o\beta\lambda$ is considered in this extended abstract). It is necessary to fix the semantics of $\pi o\beta\lambda$ in order to justify the development steps proposed in [7,9]; for example, [7] uses an equivalence on programs which facilitates an increase in concurrency. (It is important to note that it is not intended that the user of the proposed development method is aware of this presentation of the semantics: such developers use the justified rules only.) Consider the 'program' in Figure 1 which implements a sorted priority queue over a linked list of object instances. Notice that the reference contained in l is marked as unique : such a reference is defined to be one which is never 'copied' nor which has other references passed over it. The equivalence rule which permits the return statement to be commuted to the head of the method (thus releasing the *rendez-vous*) is

S; return e can be replaced by   return e; S

providing

(i) S contains no return statement and always terminates;

(ii) e is not affected by S; and

```
Sort class
vars v: ℕ ← nil; l: unique ref(Sort) ← nil
ins(x: ℕ) method
  begin
  if is-nil(v) then (v ← x; l ← new Sort)
  elif v ≤ x then l!ins(x)
  else (l!ins(v); v ← x)
  fi
  ;
  return
  end
⋮
end Sort
```

Fig. 1. An example $\pi o\beta\lambda$ program

(iii) S only invokes methods reachable by unique references.

To illustrate the main points of the semantics, it is sufficient to consider the following (reduced) abstract syntax

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

$$\mathit{Type} = \textsc{UniqueRef} \mid \textsc{SharedRef} \mid \textsc{Bool}$$

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

$$\mathit{Stmt} = \mathit{New} \mid \mathit{Call} \mid \mathit{Assign} \mid \cdots \mid \mathit{Return}$$

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

$$\begin{array}{lll} \mathit{Call} :: & \mathit{lhs} & : \mathit{Id} \\ & \mathit{call} & : \mathit{Mref} \end{array}$$

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

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

$$\mathit{Return} :: r : [\mathit{Expr}]$$

The structured operational semantics is presented at two levels. For the statement level:

$$\xrightarrow{s} \subseteq (\mathit{Stmt}^* \times \Sigma) \times (\mathit{Stmt}^* \times \Sigma)$$

where the values of instance variables are given by

$$\Sigma = Id \xrightarrow{m} Val$$

For example, the transition rule for assignment statements is:

$$\frac{}{([x \leftarrow e] \curvearrowright l, \sigma) \xrightarrow{s} (l, \sigma \dagger \{x \mapsto [\![e]\!]\sigma\})}$$

Rules for other basic statements are straightforward.

To define the global transitions, one needs

$$O = Oid \xrightarrow{m} (Stmt^* \times \Sigma)$$

$$M = Oid \xrightarrow{m} Id$$

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

The promotion of the statement level transitions is covered by

$$\frac{\begin{array}{c} O(\alpha) = (l, \sigma) \\ (l, \sigma) \xrightarrow{s} (l', \sigma') \end{array}}{C \vdash (O, M) \xrightarrow{g} (O \dagger \{\alpha \mapsto (l', \sigma')\}, M)}$$

The global rules for the remaining statements can now be presented. For the new statement

$$\frac{\begin{array}{c} O(\alpha) = ([x \leftarrow \mathsf{new}\ A] \curvearrowright l, \sigma) \\ \beta \notin \mathsf{dom}\, O \end{array}}{C \vdash (O, M) \xrightarrow{g} \left(O \dagger \left\{\begin{array}{l} \alpha \mapsto (l, \sigma \dagger \{x \mapsto \beta\}), \\ \beta \mapsto ([\,], \{\,\}) \end{array}\right\}, M \cup \{\beta \mapsto A\}\right)}$$

To initiate a method call

$$\frac{\begin{array}{c} O(\alpha) = ([x \leftarrow v!m()] \curvearrowright l, \sigma) \\ \sigma(v) = \beta \\ O(\beta) = ([\,], \sigma') \end{array}}{C \vdash (O, M) \xrightarrow{g} \left(O \dagger \left\{\begin{array}{l} \alpha \mapsto ([\mathsf{wait}(\beta, x)] \curvearrowright l, \sigma), \\ \beta \mapsto (mm(C(M(\beta)))(m), \sigma') \end{array}\right\}, M\right)}$$

To terminate the *rendez-vous* of a call

$$\frac{\begin{array}{c} O(\alpha) = ([\mathsf{wait}(\beta, x)] \curvearrowright l, \sigma) \\ O(\beta) = ([\mathsf{return}(e)] \curvearrowright l', \sigma') \end{array}}{C \vdash (O, M) \xrightarrow{g} \left(O \dagger \left\{\begin{array}{l} \alpha \mapsto (l, \sigma \dagger \{x \mapsto [\![e]\!]\sigma'\}), \\ \beta \mapsto (l', \sigma') \end{array}\right\}, M\right)}$$

This SOS was not the first version written: earlier versions followed more closely the operational semantics of –for example– the π-calculus itself and presented separate statement level transitions for method call and receipt etc.: this made the matching of send/receive pairs more opaque.

But even the SOS above presents hurdles to the proof of $\pi o\beta\lambda$ equivalences. One is forced to present a low level of granularity (to permit interleaving of steps in different objects) only to prove that this is not necessary. This is compounded by the fact that there is no algebra for reasoning about such SOS definitions: one is almost always forced to induction over the computation.

A number of authors have looked at presenting the semantics of imperative languages in general by mapping to process algebras (e.g. [11, Chapter 8]); several authors have extended this idea to tackle object-oriented languages [14,15,8,4,16]. Here, a mapping to the (first-order) polyadic π-calculus [12] is given.

Processes (typical elements P, Q)

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

Normal processes (typical elements M, N)

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

Prefixes (typical element π)

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

The following abbreviation is used

$$\tilde{y} \stackrel{\text{def}}{=} y_1 y_2 \dots y_n$$

It is straightforward to model Boolean values and to mimic the sequencing of composite statements. To illustrate the mapping for simple classes, consider

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

This can be mapped to

$$[\![Bit]\!] = !\,(\boldsymbol{\nu}\tilde{\alpha})(\overline{bit}\tilde{\alpha}.I_{\tilde{\alpha}})$$

$$I_{\tilde{\alpha}} = (\boldsymbol{\nu} sa)(\, V \mid B_{\tilde{\alpha}}))$$

$$V = (\boldsymbol{\nu} t)(\overline{t}\,b_f \mid !\,t(x).(\overline{a}x.\overline{t}x + s(y).\overline{t}y))$$

$$B_{\tilde{\alpha}} = (\alpha_w(\omega x).\overline{s}x.\overline{\omega}.B_{\tilde{\alpha}} + \alpha_r(\omega).a(x).\overline{\omega}x.B_{\tilde{\alpha}})$$

and

$$[\![\text{new } Bit]\!] = bit(\tilde{\alpha}).\cdots$$

$$[\![p!w(\text{true})]\!] = (\boldsymbol{\nu}\omega)(\overline{\alpha_w}\omega\, b_t.\omega().\cdots)$$

$$[\![p!r()]\!] = (\boldsymbol{\nu}\omega)(\overline{\alpha_r}\omega.\omega(x).\cdots)$$

This mapping benefits from the unique name generation of the π-calculus which neatly models passing the method names as a ‘capability’. Furthermore, replication is a perfect model for the way in which a class can be used to generate any number of objects. Most importantly, the result of such a mapping is an expression in a language whose algebra and equivalence notions have been studied. This has enabled David Walker (in [16]) to prove both the specific transformation discussed for Figure 1 and a more complex example involving returning values from a tree representation of a symbol table.

The general proof that the equivalence laws hold in all cases are however more troublesome (see [10] for an outline proof which is not completely formal). Basically one might hope that the interactions with local (models of) instance variables could be hidden in a way which would make it possible to prove bi-simulation. While this is true for the specific proofs, in the general $\pi o\beta\lambda$ commutativity results, one has to argue about statements which are unknown (but satisfy stated conditions); here, what one needs is to find π-calculus conditions that follow from those at the higher level and are useful in the proof. The essence is saying what can't happen. Because unique references cannot be passed at the $\pi o\beta\lambda$ level, one would like to be able to say that (the names corresponding to) references can only occur in subject positions – unfortunately, accessing the names from the local instance variables violates this by passing the name out in an object position.

In fact, the SOS has the advantage that the local state shows precisely the limitation that these communications are intended to be local. This has prompted an experiment with local state indices to processes: the state index idea is really a layer of syntactic sugar which brings the level of the π-calculus closer to $\pi o\beta\lambda$. Using state indices the mapping becomes

$$[\![Bit]\!] = !\,(\boldsymbol{\nu}\tilde{\alpha})(\overline{bit}\tilde{\alpha}.B_{\tilde{\alpha}}\{v \mapsto \mathsf{false}\})$$

$$B_{\tilde{\alpha}}\sigma = (\alpha_w(\omega x).\overline{\omega}.B_{\tilde{\alpha}}(\sigma \dagger \{v \mapsto x\}) + \alpha_r(\omega).\overline{\omega}(\sigma(v)).B_{\tilde{\alpha}}\sigma)$$

It is hoped to complete formal proofs of the equivalences in the near future. There are then plenty of interesting challenges remaining. Most notably, the development rules for rely/guarantee-conditions need to be re-expressed for $\pi o\beta\lambda$ and justified against the semantics.

Acknowledgements

The author was grateful to John Mitchell for the invitation to contribute to the special session on object-oriented languages. The research supported here has been greatly helped (and made much more enjoyable) by collaboration with Haruo Yamaguchi, Kohei Honda, Steve Hodges and Pierre Collette. The author gratefully acknowledges the support of EPSRC funding for his research.

References

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

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

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

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

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

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

[7] C. B. Jones. Constraining interference in an object-based design method. In [3] (1993), p. 136–150.

[8] C. B. Jones. A pi-calculus semantics for an object-based design notation. In [2] (1993), pages 158–172.

[9] C. B. Jones. Reasoning about interference in an object-based design method. In [17] (1993), p. 1–18.

[10] C. B. Jones. Process algebra arguments about an object-based design notation. In [13], Chapter 14, p. 231–246. 1994.

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

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

[13] A. W. Roscoe, editor. *A Classical Mind: Essays in Honour of C. A. R. Hoare.* Prentice-Hall, 1994.

[14] F. W. Vaandrager. Process algebra semantics of POOL. In [1] (1990), p. 173–236.

[15] D. Walker. π-calculus semantics for object-oriented programming languages. In [5] (1991), p. 532–547.

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

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