Some Practical Problems and Their Influence on Semantics

Cliff B Jones

Department of Computer Science, Manchester University
M13 9PL, UK
e-mail: cbj@cs.man.ac.uk

Abstract. This paper offers an assessment of what has been achieved in three decades of work on the semantics of programming languages and pinpoints some practical problems in computing which might stimulate further research. The examples sketched in this paper come from the author's own research on concurrent object oriented languages, from database practice, and from more speculative research on Internet issues.

1 Introduction

The main reason for writing this paper is to attempt to persuade leading researchers in our field to encourage some of their younger colleagues to tackle practical problems with semantic theories which are available. Currently it seems that many publications are aimed at devising refinements of theories which themselves may not be applicable to a useful class of applications. One can be a staunch defender of fundamental research while still being concerned that too few of the strong new generation of theoretical computer scientists value the stimulus of practical computing problems. In some respects, this paper echoes the slightly tongue-in-cheek paper by Knuth [Knu73] in that the comments are offered from a position of broad support for theoretical work.

Work on the formal semantics of programming languages began in the 1960's – a useful early reference is [Ste66] which reports on a conference held in Baden-bei-Wien in 1964. The subsequent literature on formal semantics of sequential languages is extensive. A good state of the art example of a formal definition is that of standard ML (cf. [HMT89]). This definition is written in Structured Operational Semantics (cf. [Plo81]). It is sobering that, after a quarter century of denotational semantics, it is still found more convenient to tackle the semantics of a language like SML in an operational way. Furthermore, it must be clear that it is extremely difficult to get a formal semantics to the stage where it correctly reflects the intuitions about a language. There are some language standards like that for Modula 2 which are actually being written using formal techniques. But overall the situation is that formal semantic definitions are written only by a very small number of highly skilled people.

The situation with recording the formal semantics of concurrent languages is even less well developed. Although there are denotational definitions of CSP-like languages, most people who are considering specifying a programming language which embodies concurrency would turn to SOS. In spite of being an impressive piece of work, the SMoLCS definition of Ada is not a document on which one would choose to base reasoning about an implementation of Ada.

One goal of writing a formal semantics of a programming language is to be able to reason about implementations of the language; another desirable objective is to be able to justify proof rules about constructs of the programming language with respect to an underlying model theoretic semantics. There are almost no practical programming languages which have serious sets of axioms and probably none at all which have a complete set of proof rules. The proof theory even for specially designed languages is hard to apply and in very few cases has been applied to programs of significant industrial size.

It is not the intention to decry works on formal semantics, there are of course many successes. This author has been involved in writing semantic definitions of languages like ALGOL-60, Pascal and PL/I. Furthermore, the background ideas on how to write a model theoretic semantics of a programming language enable one to understand –and sketch the domains of– a language which one wishes to study. In the area of proof theory, the application of proof rules for programming languages has led to a process of rigorous design which provides a way of developing proofs to support the top-down documentation of a program. Moreover, the knowledge of how to write proofs using concepts like invariants and termination arguments influences the thinking process of anyone who has been exposed to those ideas.

Nor is it wish to suggest that researchers interested in theoretical aspects of computing must be prepared to model any messy architecture that has been developed by practitioners. It is far more desirable to tease out the fundamental concepts from –for example– programming languages. The position taken in this paper is, however, that proposed formalisms should be challenged by application to concepts from realistic systems and that the process of extracting key targets should be undertaken on a wider and fresher range of applications than appears to be in use in the current theoretical literature.

Progress in mathematics has frequently come from the invention of more tractable notations (the trivial example of the development of Arabic numerals in preference to Roman numerals is a much cited but nonetheless valid case in point). Theoretical computer scientists have provided a range of notations for documenting semantics and further research *is* required to make them more tractable. But this author submits that the test of tractability has to be applied on realistic programming languages rather than on those that could be regarded as toys. The concern which motivated writing this paper is that too few computer scientists are actively involved in practical experiments with the theories which do exist. It is, of course, clear that both sorts of activities are required but the 'reward structure' of our community appears to be heavily biased towards the presentation of new or refined theories. Basic research is necessary in order to refine theories in the direction of greater or more ready applicability but many of the refinements that are published result from striving solely for mathematical elegance regardless of applicability. More experience in applying formal semantic techniques is required in order to better motivate their improvement.

People who have been involved in computing for a reasonable portion of its relatively short history must be impressed with the fact that systems *do* actually work. Today it is possible to achieve a good user interface on top of interface managers like X-windows at a vastly lower investment than was thinkable even a decade ago. Furthermore the excitement about Internet and WWW in a significant portion of the community at large is evidence that computing is beginning to serve the all-important purpose of an information provider rather than simply a computation device (but see comments on WWW under Section 3 below). An essential part of the ability to build new software is the sensible design of interfaces on top of which people can design their own systems.

Formal methods are not a significant factor in the creation of most of the everyday systems on which we work: formal methods are in fact applied almost solely on safety critical systems. Tony Hoare has tried to tease out some of the reasons that system design without formal methods has been successful in [Hoa96]. It is, however, important to remember that many things that were at one stage regarded as parts of a formalist's tool kit have now been absorbed into everyday computing. An obvious example here is the role that context free grammars have played both in the description of programming languages and in the design of tools to analyse and process such languages. Type structure is another important example. But in all humility formalists must ask themselves whether they would really do better than the designers of software like `emacs`.

One of the key arguments for testing formal approaches on realistic applications is that it would provide scientists with the experience of extracting their own abstractions from the messy detail of realistic systems. It is not enough to go on working with abstractions like 'stack' and the 'dining philosophers problem' which were abstracted many years ago. But these are exactly the sorts of examples which are seen over and over again in the papers presented at conferences like ESOP and MFPS. Rarely do we see the best minds applying themselves to exercise theories on new applications.

The approach which is commended in this paper is that authors who wish to explore the applicability of a formalism should be prepared to tackle a new application problem of their own. In this way they will gain the experience of developing abstractions and perhaps find that the 'devil is in the detail' precisely in the process of developing this abstraction. This proposal could sometimes lead to a theory which works in that it gives some purchase on the problem in hand but is 'ugly'. Hardy wrote in his 'Mathematicians Apology' [Har67] that there is no permanent place for ugly mathematics. But it is sometimes necessary to proceed through a period of less than elegant mathematics in order to understand what the real problems are. This author –for example– originated a set of proof obligations for programs which use post conditions of two states. Peter Aczel approved of this step in [Acz82] and wrote 'It is familiar that the specification [using a post-condition of the final state only] does not exactly express all that we have in mind ....' But went on to describe the rules published in [Jon80] in the following way '...his [CBJ] rules appear elaborate and unmemorable.' The revised rules which were later employed in [Jon90] are however comparable with those for single state post-conditions and do address relating the initial and final states.

There are many examples of where a mathematically inelegant result has been of use. Even from this author's experience the fact that the initial set of data refinement rules used in VDM were known to be incomplete did not inhibit their being extremely effective in developing a variety of systems (see [Jon89] for details of this story and the role played by Tobias Nipkow's paper [Nip86]).

Starting from a set of rules which do 'work' but which are inelegant, one can seek the refinement of a theory in a way which should yield some confidence as to its applicability to realistic problems. The presentation of increasingly refined systems which have themselves only been shown to suffice for extremely simple examples seems less likely to yield applicable formal methods.

It is of course true that pure research has its own value and that finding the right framework can make a system vastly more tractable but one also has to be aware that there is a cost/benefit trade-off for esoteric theories in terms of the difficulty of communicating them to engineers who are presumably expected to be the ultimate users.

The plan of this paper is to use the task of describing the semantics of concurrent object oriented languages as a major and solid example of what seems to lay just beyond the scope of our semantic tools at this time and then to sketch in less detail some increasingly tentative research areas which might invite attempts to hone formal methods.

2 An example: Concurrent Object-Oriented Languages

This –the main example given in this paper– is not claimed to have fundamental significance; it is presented because it has provided the stimulus to a sequence of papers and comes from a practical problem. It is revealing that the standard notions of bisimulation do not immediately apply to the example; the challenge here is to offer semantic descriptions which facilitate reasoning.

It would be reasonable to view the challenge of providing the semantics of concurrent object oriented languages as an end in its own right. In fact, the work described here fits into a larger programme of work outlined in [Jon96]. That paper describes why some concepts from object oriented languages are seen as a useful way to control the interference which is inherent with concurrency; it describes how an idea similar to Hoare's 'recursive data structures' (cf. [Hoa75]) can be realised by representing tree-like abstractions as collections of objects; it also describes the role of equivalences which can increase concurrency in programs; lastly it describes how difficult forms of interference can be reasoned about using a variant of rely-/guarantee-conditions. This paper draws on the task of proving the concurrency-enhancing equivalences in order to motivate the need for a tractable semantics.

The language used in [Jon96] is known as $\pi o \beta \lambda$. The essential points of the semantics can be illustrated by a reduced language whose abstract syntax is given in Appendix A.

Figures 1 and 2 illustrate two versions of a program which might be written in $\pi o\beta\lambda$. Both implement a sorting vector in which each object of class *Sort* contains one value and an optional reference to a further instance of the class. Figure 1 is sequential in the sense that an initial *rendezvous* with the *insert* method will stay in that *rendezvous* until the value has trickled all the way down the sorting vector to the appropriate place of insert and returns have traced their way back up this vector. The program in Figure 2 is claimed to have the same observable behaviour but is concurrent in the sense that as soon as the parameters have been transferred to the first instance of *insert* a return is executed and the client is free to execute in parallel with the server. Furthermore, as soon as the nested calls to *insert* have completed –because of the premature return– the earlier members of the sorting vector are available to accept calls from other clients. This is true in spite of the fact that $\pi o\beta\lambda$ has the restriction that only one method can be active in an object at any one time.

```
Sort class
vars $v\colon \mathbf{N} \leftarrow 0$; $l$: unique ref(Sort) ← nil
insert($x\colon \mathbf{N}$) method
  begin
    if is-nil($l$) then ($v \leftarrow x$; $l \leftarrow$ new Sort)
    elif $v \le x$ then $l$.insert($x$)
    else ($l$.insert($v$); $v \leftarrow x$)
    fi
    return
  end
test($x\colon \mathbf{N}$) method : $\mathbf{B}$
  ...
```

Fig. 1. Sequential implementation of *Sort*

The development of the sequential program in Figure 1 from a specification is straightforward. The equivalence which is claimed to justify substitution of the more parallel program in Figure 2 can be described as follows.

Equivalence 1 S; return e *is equivalent to* return e; S *providing*

- *S contains no return or delegate statements;*
- *S always terminates;*
- *e is not affected by S; and*
- *any method invoked by S belongs to an object reached by a unique reference.*

Notice that the reference to l in both programs is marked as being a unique reference.

```
$Sort$ class
vars $v$: $\mathbf{N} \leftarrow 0$; $l$: unique ref($Sort$) $\leftarrow$ nil
$insert(x: \mathbf{N})$ method
  begin
    return;
    if $is\text{-}nil(l)$ then ($v \leftarrow x$; $l \leftarrow$ new $Sort$)
    elif $v \leq x$ then $l.insert(x)$
    else ($l.insert(v)$; $v \leftarrow x$)
    fi
  end
$test(x: \mathbf{N})$ method : $\mathbf{B}$
  ...
```

Fig. 2. Concurrent implementation of *Sort*

Definition 2 *A unique reference must never be copied nor have references to mutable objects passed over it – neither in nor out.*

Definition 3 *An immutable object derives from a class whose methods have no side-effects: thus, once initialized, an immutable object's state remains unchanged.*

There are of course other equivalence laws which can be considered but this one will serve to illustrate the task in hand.

2.1 SOS Semantics of $\pi o\beta\lambda$

A number of people have considered the semantics of object oriented languages (see [HJ96] for a list of references). Most notable among these is David Walker who has provided both SOS semantics and mappings to the π-calculus for both POOL [Ame89] and for $\pi o\beta\lambda$. But David Walker's proofs have so far only tackled specific examples of the equivalence like those envisaged in the example above; what is required is a proof that the general equivalence in Equation 1 holds.

The goal of providing a semantic description for $\pi o\beta\lambda$ does not suggest that it is itself a programming language (final programs designed using $\pi o\beta\lambda$ might be written in languages like MODULA-3) but the semantics need be such that it is possible to justify equivalences of the sort considered. It is important that the equivalences are tackled in general so that it is not necessary for the person who is developing programs using the $\pi o\beta\lambda$ design notation to reason about this underlying semantics: it is orders of magnitude easier for engineers to apply equivalence laws.

In order to pin down the semantics of this subset of $\pi o\beta\lambda$ a structured operational semantics is first given (this follows that in [HJ96]). The object level transitions are defined around states

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

$Val = \mathbf{B} \mid \mathbf{N} \mid Oid$

The low level transitions are relations on $(Stmt^* \times \Sigma)$ (see Appendix A for $Stmt$). A few example rules follow.

$$\boxed{compound} \quad \frac{}{(\langle sl\rangle :: l, \sigma) \xrightarrow{s} (sl \frown l, \sigma)}$$

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

Where $[\![e]\!]\sigma$ is the valuation of e in state σ.

$$\boxed{\mathsf{if}_t} \quad \frac{[\![e]\!]\sigma = \mathsf{true}}{((\mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f) :: l, \sigma) \xrightarrow{s} (S_t :: l, \sigma)}$$

$$\boxed{\mathsf{if}_f} \quad \frac{[\![e]\!]\sigma = \mathsf{false}}{((\mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f) :: l, \sigma) \xrightarrow{s} (S_f :: l, \sigma)}$$

Global transitions require information about all objects which have been created.

$Omap = Oid \xrightarrow{m} Oinfo$

where

$$\begin{array}{lll} Oinfo :: & cn & : Id \\ & act & : \{\text{AVAIL}, \text{WAIT}\} \\ & rest & : Stmt^* \\ & state & : \Sigma \\ & client & : [Oid] \end{array}$$

These higher level transitions also require the class definitions.

$Cmap = Id \xrightarrow{m} Cdef$

They are a relation over $\xrightarrow{g} \subseteq (Cmap \times Omap \times Omap)$. The initial $Omap$ must match the $Cmap$ in an obvious way.

The promotion of low to high-level transitions is handled by the following rule.

$$\boxed{s \leadsto g} \quad \frac{\begin{array}{l} O(\alpha) = (c, \text{AVAIL}, l, \sigma, \omega) \\ (l, \sigma) \xrightarrow{s} (l', \sigma') \end{array}}{C \vdash O \xrightarrow{g} O \dagger \{\alpha \mapsto (c, \text{AVAIL}, l', \sigma', \omega)\}}$$

The transitions which involve more than one object are defined as follows.

$$\boxed{\mathsf{new}} \quad \frac{\begin{array}{l} O(\alpha) = (c_\alpha, \text{AVAIL}, (v \leftarrow \mathsf{new}\ c :: l), \sigma, \omega) \\ \beta \notin \mathsf{dom}\ O \end{array}}{C \vdash O \xrightarrow{g} O \dagger \left\{\begin{array}{l} \alpha \mapsto (c_\alpha, \text{AVAIL}, l, \sigma \dagger \{v \mapsto \beta\}, \omega), \\ \beta \mapsto (c, \text{AVAIL}, [\,], init(C(c)), \mathsf{nil}) \end{array}\right\}}$$

$$\boxed{\text{call}}\ \frac{\begin{array}{l} O(\alpha) = (c_\alpha, \text{AVAIL}, (r \leftarrow v.m() :: l_\alpha), \sigma_\alpha, \omega) \\ \sigma_\alpha(v) = \beta \\ O(\beta) = (c_\beta, \text{AVAIL}, [\,], \sigma_\beta, \text{nil}) \end{array}}{C \vdash O \xrightarrow{g} O \dagger \left\{ \begin{array}{l} \alpha \mapsto (c_\alpha, \text{WAIT}, (r \leftarrow v.m() :: l_\alpha), \sigma_\alpha, \omega), \\ \beta \mapsto (c_\beta, \text{AVAIL}, b(mm(C(c_\beta))(m)), \sigma_\beta, \alpha) \end{array} \right\}}$$

$$\boxed{\text{ret}}\ \frac{\begin{array}{l} O(\alpha) = (c_\alpha, \text{WAIT}, (r \leftarrow v.m() :: l_\alpha), \sigma_\alpha, \omega) \\ O(\beta) = (c_\beta, \text{AVAIL}, (\text{return } (e) :: l_\beta), \sigma_\beta, \alpha) \end{array}}{C \vdash O \xrightarrow{g} O \dagger \left\{ \begin{array}{l} \alpha \mapsto (c_\alpha, \text{AVAIL}, l_\alpha, \sigma_\alpha \dagger \{r \mapsto [\![e]\!]\sigma_\beta\}, \omega), \\ \beta \mapsto (c_\beta, \text{AVAIL}, l_\beta, \sigma_\beta, \text{nil}) \end{array} \right\}}$$

Constructing such an operational semantics is not in itself difficult; tuning it in such a way to make it convenient for proofs does take a considerable amount of experimentation; and –it must be pointed out– there is little practical advice in the literature.

There are a number of negative aspects of such a semantics which seem to be inherent in the operational method. The most obvious comment is that the natural proof strategy is to perform induction over the computation or reduction steps of the SOS. Allied to this is the problem that the semantics has to be documented at a very low level of granularity in the sense that the individual steps in every object must be mergeable. This is somewhat distressing in the cases where one wishes to prove precisely that such merging of the object steps has no influence on the overall behaviour. The essential difficulty here is that the low level of granularity is not easy to reason about because there is no natural algebra of such SOS definitions. Furthermore, it is necessary to make a number of decisions about how to hand-craft the communication links between different objects. On the other hand there are clearly some positive aspects of writing an SOS definition. One of the foremost of these is that it provides a test-bed on which to experiment with ideas of reasoning. One particular advantage of an SOS definition over the sort of mapping to process algebras which is considered below is that it is easy to state which things can *not* happen in logic.

Such a definition has been used to attack both the equivalence rule in Equation 1 above and the more delicate proof rule concerned with $\pi o\beta\lambda$'s delegate statement. Both of these proofs have been tackled in [HJ96] for the general equivalence rather than for specific examples. One essential point of these proofs is that one can reason about what interference can *not* occur. The main innovation in the proof is to partition the state in a way which shows that computation within a particular 'island' cannot affect any objects outside the island.

Similar comments to those made about SOS definitions would of course apply to a denotational semantics definition and in the case of the language under discussion it would be necessary to use power domains [Plo76].

2.2 Mapping $\pi o\beta\lambda$ to the π-calculus

The first attempt (cf. [Jon93a]) to write a semantics for $\pi o\beta\lambda$ was undertaken by mapping it to the polyadic first order π-calculus (cf. [MPW92]). This mapping is pleasingly direct. It is necessary to build some basic data types like Booleans and to work out how to code certain tricks like the sequential composition of statements but the following mapping from a simple *Bit* class to its π-calculus equivalent indicates that the way π-calculus creates new names works ideally for the object identifiers which have to be created as 'capabilities' for each new object; that replication works perfectly for the multiple instances of a class; and that the expansion factor of the mapping is linear. Although it could again be argued that the semantics given is at a very low level of granularity, one can see that the algebra of the π-calculus provides a way of reasoning about equivalent terms. One disadvantage is that there are perhaps too many equivalence notions for process algebras!

```
Bit class
vars $v$: $\mathbf{B} \leftarrow$ false
$w(x: \mathbf{B})$ method $v \leftarrow x$; return
$r()$ method return $v$
```

Gets mapped to

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

$$I_{\widetilde{\alpha}} = (\nu sa)(V \mid B_{\widetilde{\alpha}})$$

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

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

and

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

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

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

On the negative side, it is important to notice that –in this mapping to the basic calculus– everything has to be done by communication. This includes access to the instance variables which are modelled by parallel composed processes. This point becomes more pressing in an example where reference values are concerned. The following is the mapping of the $\pi o\beta\lambda$ code provided in Figure 1.

$$[\![Sort]\!] = \,!\,(\nu u)(\overline{sort}u.I_u)$$

$$I_u = (\nu\widetilde{s}\widetilde{a})(V \mid L \mid B_u)$$

$$V = (\nu t)((\nu n)(\overline{t}n) \mid \,!\,t(x).(\overline{a_v}x.\overline{t}x + s_v(y).\overline{t}y))$$

$$L = (\nu t)((\nu n)(\overline{t}n) \mid \,!\,t(x).(\overline{a_l}x.\overline{t}x + s_l(y).\overline{t}y))$$

$$B_u = \overline{u}\widetilde{\alpha}.\left(\begin{array}{l}\alpha_i(\omega x).a_v(v).\\ \quad \text{if } v = \text{nil then } s_v(x).sort(u').\overline{s_l}u'.\overline{\omega}.B_u\\ \quad \text{elif } v \le x \text{ then } a_l(u').u'(\widetilde{\beta}).(\nu\psi)(\overline{\beta_i}\psi x.\psi()).\overline{\omega}.B_u\\ \quad \text{else } a_l(u').u'(\widetilde{\beta}).(\nu\psi)(\overline{\beta_i}\psi v.\psi()).\overline{s_v}x.\overline{\omega}.B_u\\ +\\ \alpha_t(\omega x).a_l(l).a_v(v).\\ \quad \text{if } l = \text{nil} \lor x \le v \text{ then } \overline{\omega}b_f.B_u\\ \quad \text{elif } x = v \text{ then } \overline{\omega}b_t.B_u\\ \quad \text{else } a_l(u').u'(\widetilde{\beta}).(\nu\psi)(\overline{\beta_t}\psi x.\psi(y).\overline{\omega}y).B_u\end{array}\right)$$

and

$$[\![\text{new } Sort]\!] = sort(u).\cdots$$

$$[\![p!i(n)]\!] = u(\widetilde{\alpha}).(\nu\omega)(\overline{\alpha_i}\omega n.\omega())$$

$$[\![p!t(n)]\!] = u(\widetilde{\alpha}).(\nu\omega)(\overline{\alpha_t}\omega n.\omega(b).\cdots)$$

Here the access to the instance variable l results in communications for which it is unfortunately not true that the names of unique objects never appear in object positions: they are not –in this mapping– ‘uniquely handled’. It is, however, possible to use the idea of indexing process definitions in a way which brings a notion similar to local states into the π-calculus. For example, the *Bit* class above could be coded as

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

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

The *Sort* class above could be coded as follows:

$$[\![Sort]\!] = \,!\,(\nu\widetilde{\alpha})(\overline{sort}\widetilde{\alpha}.B_{\widetilde{\alpha}}\{v \mapsto \text{nil}, l \mapsto \text{nil}\})$$

$$B_{\widetilde{\alpha}}\sigma = \left(\begin{array}{l}\alpha_i(\omega x).\\ \quad \text{if } \sigma(v) = \text{nil then } sort(\widetilde{\beta}).\overline{\omega}.B_{\widetilde{\alpha}}(\sigma \dagger \{v \mapsto x, l \mapsto \widetilde{\beta}\})\\ \quad \text{elif } \sigma(v) \le x \text{ then (let } \widetilde{\beta} = \sigma(l) \text{ in } (\nu\psi)(\overline{\beta_i}\psi x.\psi()).\overline{\omega}.B_{\widetilde{\alpha}}\sigma)\\ \quad \text{else (let } \widetilde{\beta} = \sigma(l) \text{ in } (\nu\psi)(\overline{\beta_i}\psi\sigma(v).\psi()).\overline{\omega}.B_{\widetilde{\alpha}}(\sigma \dagger \{v \mapsto x\}))\\ +\\ \alpha_t(\omega x).\cdots\end{array}\right)$$

The parallel version in Figure 2 gets mapped to

$$B'_{\widetilde{\alpha}}\sigma = \left(\begin{array}{l}\alpha_i(\omega x).\overline{\omega}.\\ \quad \text{if } \sigma(v) = \text{nil then } sort(\widetilde{\beta}).B'_{\widetilde{\alpha}}(\sigma \dagger \{v \mapsto x, l \mapsto \widetilde{\beta}\})\\ \quad \text{elif } \sigma(v) \le x \text{ then (let } \widetilde{\beta} = \sigma(l) \text{ in } (\nu\psi)(\overline{\beta_i}\psi x.\psi()).B'_{\widetilde{\alpha}}\sigma)\\ \quad \text{else (let } \widetilde{\beta} = \sigma(l) \text{ in } (\nu\psi)(\overline{\beta_i}\psi\sigma(v).\psi()).B'_{\widetilde{\alpha}}(\sigma \dagger \{v \mapsto x\}))\\ +\\ \alpha_t(\omega x).\cdots\end{array}\right)$$

As indicated above, the SOS definitions have frequently served as a stimulus to ways in which one might reason in the mapped form of semantics. The use of state indices to process definitions is one example. It is straightforward to see how one could use the notion of islands as a way of dividing a large composition into sub-terms which do not interact with one another. Immutable classes can be made local by alpha converting their mapped versions.

A number of researchers (cf. [Vaa90, Wal91, Jon93b, HT91, Wal93]) have noted the possibility of mapping object oriented languages in general –and concurrent object oriented languages in particular– into process algebras. But even Walker who has expended most effort on the attempt to prove the transformations correct has so far only been able to get out proofs of specific examples of the equivalences. One of the stumbling blocks is the ability to state negative properties: it is not obvious how to say what can *not* happen.

3 Further Challenges

There is an extensive literature on the problems of handling parallel transactions in databases (see for example [Dat94]). There are even several books on formal presentations of the need to verify such systems (cf. [B+87, L+94]). Most of this literature relies on presentations of semantics which would not be familiar to people who have worked on programming language semantics. It is interesting to investigate the extent to which more traditional semantic approaches might give tractable ways of reasoning about the correctness of database systems.

It seems that this is an important area on which researchers could work if only because the database community has created a large number of approaches to transaction scheduling which *do* work. Furthermore, their descriptions appear to have provided ways of reasoning about the correctness of such systems at least to the extent that there is a repertoire of algorithms which can be employed in database systems. Furthermore, the formal methods community may well have something to learn, either from the challenge of reformulating these descriptions in more traditional semantic approaches or by realising that these semantic approaches are not in fact the most apposite for database problems. If we cannot easily verify the algorithms, it could stimulate the development of new ways for reasoning about other classes of concurrent systems. Who knows, the formal methods people may even spot some new database algorithms!

The basic approach in the bulk of the database literature is to argue about 'serialisability' of transactions. In other words, it is permissible for a database system to merge the actions within transactions providing a result is achieved which could have come about by executing the separate transactions in *some* sequential non-interfering order.

A simplified form of the problem can be relatively easily presented based around the following abstract syntax.

$$Pgm = Tid \xrightarrow{m} Trans$$

$$Trans = STrans \mid ATrans$$

$STrans :: Act^*$

$ATrans :: Act^*$

$Act = Rd \mid Wr$

$$Rd :: \begin{array}{l} Temp \\ Var \end{array}$$

$$Wr :: \begin{array}{l} Var \\ Expr \end{array}$$

The basic correctness notion can be provided by a structured operational semantics which processes transactions in a non-deterministic but serial order.

Individual action steps ($\xrightarrow{a}$) are relations on $(\mathbf{N} \times P \times \Sigma)$.

σ: $Var \xrightarrow{m} Val$ $\qquad\qquad$ ρ: $Temp \xrightarrow{m} Val$

$$\boxed{read} \; \frac{\begin{array}{c} i < \mathsf{len}\, t \\ t(i+1) = mk\text{-}Rd(l, v) \end{array}}{t \vdash (i, \rho, \sigma) \xrightarrow{a} (i+1, \rho \dagger \{l \mapsto \sigma(v)\}, \sigma)}$$

$$\boxed{write} \; \frac{\begin{array}{c} i < \mathsf{len}\, t \\ t(i+1) = mk\text{-}Wr(v, e) \end{array}}{t \vdash (i, \rho, \sigma) \xrightarrow{a} (i+1, \rho, \sigma \dagger \{v \mapsto [\![e]\!]_\rho\})}$$

Transaction level transitions ($\xrightarrow{t}$) for a given program are presented as relations on $(Tid\text{-set} \times \Sigma)$.

$$\boxed{succ} \; \frac{\begin{array}{l} tid \in pend \\ p(tid) \in STrans \\ p(tid) \vdash (0, \{\,\}, \sigma) \xrightarrow{a} (\mathsf{len}\,(p(tid)), \rho', \sigma') \end{array}}{p \vdash (pend, \sigma) \xrightarrow{t} (pend - \{tid\}, \sigma')}$$

$$\boxed{u\text{-}abort} \; \frac{\begin{array}{c} tid \in pend \\ p(tid) \in ATrans \end{array}}{p \vdash (pend, \sigma) \xrightarrow{t} (pend - \{tid\}, \sigma)}$$

Then the input/output relation of a program p is given by

$$(\sigma, \sigma') \in [\![p]\!] \text{ iff } p \vdash (\mathsf{dom}\, p, \sigma) \xrightarrow{t} (\{\,\}, \sigma')$$

Various interleaving semantics can be presented as SOS rules. It is normally easy to see that all (σ, σ') transitions of the serial semantics can be reproduced; the interesting issue is whether there are too many transitions. For the simple approaches to a concurrent implementation it does look as though an argument in terms of SOS would be possible. More subtle ways of merging transactions look increasingly interesting. The challenge, of course, is to tackle algorithms which employ, for example, locking on subparts of the database to be able to show that they are correct with respect to the sequential semantics.

There is, in fact, an interesting series of articles stimulated by Lamport's 'Lake Arrowhead' example. These are slanted more towards reasoning about assertions but warrant comparison with what is sketched here. As in the preceding section, it would also be interesting to investigate whether a treatment in terms of process algebra would provide purchase on these problems.

There are other problems in this class of refining atomicity where it is convenient to provide an overall specification with large granularity and to show that implementations which refine that granularity and permit substeps to be interleaved are correct with respect to the original specification. Potential examples include caching [B$^+$94], pipelining in computer architectures and some approaches to fault tolerance.

There are many other areas which would appear to be in need of some formal analysis. One from this author's own experience (cf. [JJLM91]) is the desire to build general purpose theorem proving assistants which have user interfaces tempting to users other than the originators of the system. One of the goals here must be to be able to describe not just 'logic frames' but to be able to deal with 'method frames' in the way outlined by the DEVA group [WSL93]. One would even wish to be able to specify things like version control of specifications and proofs and record information about which test cases had been run against which versions of the system.

If the argument of the formal methods community that their techniques provide convenient ways of designing and thinking about systems are to be justified, it would seem essential that methods are applied not only to existing problems like those outlined above but to areas where new systems are only just beginning to evolve. An example –again taken from this author's own experience– is a desire to understand what it would mean to design a useful 'global yet personal information system'. The starting observation here is that –in spite of its usefulness– World Wide Web can hardly be classed as a global *information* system. URLs are the world's worst pointers! Furthermore, what is actually available on the Internet at the moment seems to be just sequences of bytes rather than structured information. The paper [GJ96] looks at the challenge of designing a genuine information system which would be distributed on a global basis. No magic answers are provided but the paper makes clear that there is a major challenge in designing such systems. This appears to be exactly the sort of challenge which an abstract model with an appropriate notation might well enable the architects to think about more clearly than if they just proceed by designing programs in an *ad hoc* way.

Another very speculative area is that of multi-media systems. These are clearly becoming important in practice and it would be desirable if formalism could be applied before the systems become too unstructured for that to be an appetising possibility.

4 Point of this sermon

It appears that the 'reward structure' of computer science encourages the development of deep theories and under values the process of establishing the usefulness of existing theories. It is certainly not the intention here to argue that people who –for example– undertake a PhD should be invited to apply an established method to a routine problem. But, in spite of the many first generation formal methods books, there are relatively few attempts to take formal methods and apply them to established or evolving areas of computer science. It would certainly be considered desirable to see more research publications on the application of established formal methods to novel practical problems.

It is worrying that some computer scientists who do choose to develop systems to support their own research seem reluctant to employ formal methods during that process. It seems obvious that the persistent application of a formal method to increasingly challenging problems will force us to refine the methods and make them more tractable; to not do so seems an abdication of responsibility.

Many of us feel privileged that we are present during the development of a new science. The argument here is that we must make sure that this subject is *computing science* rather than a branch of mathematics which can be shown to apply to only trivial problems which have some connection with computing.

Senior members of our field do and should continue to choose their own research agendas. When advising younger members of the community they should perhaps put more emphasis on the application of methods rather than just the development of new methods. We must avoid the danger of 'corrupting the young' and having to take the Hemlock (cf. [Pla54])!

Acknowledgements

This paper was written whilst the author was on sabbatical from Manchester University at the Isaac Newton Institute of Mathematical Sciences. The paper evolved out of a talk given at the Institute. The author is also grateful to Gonville and Caius college for their hospitality during his Visiting Fellowship there. Tony Hoare provided insightful comments on a draft of the paper. The author's research is supported by a grant from the Engineering and Physical Sciences Research Committee.

References

[Acz82] P. Aczel. A note on program verification. manuscript, January 1982.

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

[B$^+$87] P. A. Bernstein et al. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[B$^+$94] Ed Brinksma et al. Verifying sequential consistet memory. Technical Report Project P6021, ESPRIT, August 1994.

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

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

[Dat94] C J Date. *An Introduction to Database Systems*. Addison-Wesley, sixth edition, 1994.

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

[GJ96] J. R. Gurd and C. B. Jones. The global-yet-personal information system. In *Computing Tomorrow*. Cambridge University Press, 1996.

[Har67] G. H. Hardy. *A Mathematician's Apology*. Cambridge University Press, 1967.

[HJ96] S. J. Hodges and C. B. Jones. Non-interference properties of a concurrent object-based language: Proofs based on an operational semantics. In C. Lengauer, editor, *to be published*. Kluwer, 1996.

[HMT89] R. Harper, R. Milner, and M. Tofte. The definition of standard ML – Version 3. Technical Report ECS-LFCS-89-81, University of Edinburgh, LFCS, Department of Computer Science, University of Edinburgh, The Kings Buildings, Edinburgh, 1989.

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

[Hoa96] C. A. R. Hoare. How did software get so reliable without proof? In M-C Gaudel, editor, *Proceedings FME'96*. Springer-Verlag, 1996. Lecture Notes in Computer Science.

[HT91] 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. ISBN 3-540-19651-X.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980. ISBN 0-13-821884-6.

[Jon89] C. B. Jones. Data reification. In J. A. McDermid, editor, *The Theory and Practice of Refinement*, pages 79–89. Butterworths, 1989.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.

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

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

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, 1996.

[Knu73] Donald E Knuth. The dangers of computer-science theory. In *Logic, Methodology and Philosophy of Science IV*, pages 189–195. North-Holland, 1973.

[L$^+$94] Nancy Lynch et al. *Atomic Transactions*. MIT Press, 1994.

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

[Nip86] T. Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22:629–661, 1986.

[Pla54] Plato. *The Last Days of Socrates.* Penguin Classics, 1954.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, 5(3), 1976.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[Ste66] T. B. Steel. *Formal Language Description Languages for Computer Programming.* North-Holland, 1966.

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

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

[Wal93] D. Walker. Process calculus and parallel object-oriented programming languages. In *In T. Casavant (ed), Parallel Computers: Theory and Practice.* Computer Society Press, to appear, 1993.

[WSL93] M. Weber, M. Simons, and Ch. Lafontaine. *The Generic Development Language Deva: Presentation and Case Studies*, volume 738 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993. ISBN 3-540-57335-6.

A Abstract syntax

This is a reduced version of the $\pi o\beta\lambda$ language.

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

$$\mathit{Id} = \cdots$$

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

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

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

$$\mathit{Stmt} = \mathit{Compound} \mid \mathit{Assign} \mid \mathit{If} \mid \mathit{New} \mid \mathit{Cal} \mid \mathit{Delegate} \mid \mathit{Return}$$

$$\mathit{Compound} :: \mathit{sl} : \mathit{Stmt}^*$$

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

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

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

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

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

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

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