Some Mistakes I Have and What I Have Learned from Them

Cliff B Jones

Harlequin plc
Queens Court, Wilmslow Road, Alderley Edge, Cheshire, SK9 7QD
e-mail: cbj@harlequin.com
Department of Computer Science, Manchester University, M13 9PL
e-mail: cbj@cs.man.ac.uk

Abstract. The purpose of this paper is to make a number of points about the selection of topics, research style, and dissemination of ideas. The writing style chosen is to present past personal decisions which might be regarded as technical or strategic mistakes and to indicate what positive messages can be derived from the experiences.

Introduction

An invited contribution is an opportunity to do something different: for a start, I write in the first person singular. In a sense, I intend to preach: to argue that some formal methods research is going in a direction which has little chance of making an impact on computing practise; to try to persuade senior researchers to direct students –or anyone who is susceptible to influence– to look more seriously at new applications and less at polishing ways of treating problems which have come to be seen as classical.

Just preaching is not likely to sway people so I have chosen to don a "hair shirt" and describe some decisions that I have made which could be regarded as mistakes. At the end of each of my four "confessions", I try to draw lessons from the story. Of course, there is a degree of self-justification in these lessons. Furthermore, I should concede that I could probably not have presented some of these lessons without a gap of (in some cases, many) years. What follows certainly does not present the only examples I could have drawn from about thirty years of trying to develop and disseminate what have become known as "formal methods". But I believe that each of these four enable me to make a positive point.

1 Handling partial functions in proofs

I first met the problem of how to reason about partial functions in the IBM Vienna Laboratory in the late 1960s; I remember an intense discussion with Peter Lucas during the period that he was writing [Luc69]. The early Vienna (operational semantics) VDL work had used McCarthy's conditional expression interpretation of logical operators and I had found that these presented a number of difficulties which went beyond solving the problem that initially presented itself. The reason for my first stay in Vienna was to experiment with using the VDL (cf. [LW69]) language definitions in proofs about implementation correctness of compiling algorithms and I suspect that I encountered difficulties earlier than many precisely because I was one of the first people to make such use of the definitions.

I have had several tries at this problem since and have –I guess– to view all but the last as mistakes (and to continue to question the last!). Most important is that the criteria for success have been motivated by efficacy in the construction of proofs.

Problem

The difficulty of reasoning about partial functions is apparent to anyone who has faced other than the most trivial of specification tasks. Terms arise in logical expressions which for certain values of their free variables fail to denote an (obvious) value. One source of this problem is operators on basic data types such as sequences: taking the head of an empty sequence does not denote an obvious value so an expression like $\mathsf{hd}\, t$ might or might not denote a value. In

$$\forall t \in \mathbb{N}^* \cdot t = [\,] \vee t = \mathit{append}(\mathsf{hd}\, t, \mathsf{tl}\, t)$$

it is obviously intended that a value be denoted but perhaps less obvious as to how to explain in a compositional way why it is.

The more troublesome terms come from partial functions. In large specifications, one needs many auxiliary functions and they are often defined by recursion. What makes their handling more delicate than monadic operators is that the domain over which they are defined can depend on relationships between their arguments. Examples of moderate difficulty could be cited from work on databases etc. A simple example which I have used in several papers is intentionally just complex enough to illustrate the problem.

$$\mathit{diff} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

$$\mathit{diff}(i,j) \triangleq \mathsf{if}\ i = j\ \mathsf{then}\ 0\ \mathsf{else}\ \mathit{diff}(i, j+1) + 1\ \mathsf{fi}$$

is a perverse definition of subtraction over the integers; its intended domain appears to be summarised by

$$\forall i,j:\mathbb{Z} \cdot i \geq j \;\Rightarrow\; \mathit{diff}(i,j) = i - j \tag{1}$$

But here again a denotational understanding founders on the need to provide a value for terms like $\mathit{diff}(2,3)$.

The conditional interpretation of the logical operators outlined in [McC63] extends their meaning to cover undefined operands so that the implication in Eq. 1 evaluates to true with false as its left operand and undefined as its right. As indicated above, the conditional expression approach works but is less than ideal in proofs: the familiar commutativity property of conjunction and disjunction are lost and –in the case in hand– the contrapositive does not hold.

$$\forall i,j:\mathbb{Z} \cdot \mathit{diff}(i,j) \neq i - j \;\Rightarrow\; i < j \tag{2}$$

Moreover, although less likely to arise, there is no reason why the following should not be true

$$\forall i,j:\mathbb{Z} \cdot \mathit{diff}(i,j) = i - j \vee \mathit{diff}(j,i) = j - i \tag{3}$$

but this cannot hold in the conditional interpretation.

The real rub with the conditional view is that every operator is forced to be understood via its conditional expression meaning whether or not this is necessary. This observation has prompted a number of computer scientists (e.g. [Jon72,Dij76,Gri81]) to experiment with mixed sets of classical and conditional operators. The difficulty then is that the number of rules for manipulating the double set of operators is far greater and less intuitive than one would wish. None of the above cited contributions provide a full set of rules and surprises like right distribution of "conditional and" over the "conditional or" are unintuitive.

I made a further attempt to stay with classical logic in [Jon80] by using quantifiers to bound variables so as to make any term only meet values of its free variables for which it is defined. This experiment was also unsuccessful in proofs although it is broadly what is much more systematically worked out in *Order-Sorted Algebras* – see [GM92].

One of my earliest exposures to classical predicate calculus came from reading [Kle52] and I am sure that I had retained a memory of Łukasiewicz's "three-valued" logical operators. These are presented in Kleene's "blue book" only by truth tables but a student of Peter Aczel had provided an axiomatisation in [Kol76] and I misused the absence of his real supervisor to get Jen Cheng interested in the challenge of giving a natural deduction proof style for such operators (this led to [BCJ84,Che86,CJ91]). This logic has become known as the *Logic of Partial Function* (or LPF) and its use in [Jon86] and subsequent publications on VDM have convinced me that this provides a usable proof system. (As Kees Middelburg pointed out, Cheng had only provided a justification for an untyped version of the logic whereas VDM proofs typically used a typed version – this hole was closed in [JM94].) LPF is not classical logic, but the only

significant casualty is the "law of the excluded middle" – which with expressions such as

$$\mathsf{hd}\,[\,] = 5 \vee \mathsf{hd}\,[\,] \neq 5 \tag{4}$$

is a loss I can tolerate.

There are of course other approaches to the thorny problem of reasoning about partial functions – see [JM94,CJ91,Jon95] for further references.

Lessons

The lessons which I believe it is important to draw from my series of attempts to find a satisfactory way to reason about partial functions include the fact the a real difficulty should be faced rather than ignored (I can accept any approach to this problem much more readily than pretending it does not exist). But the main lesson is that the mathematics that we require to handle computer science problems may not be classical; it might or might not exist in textbooks. Kline wrote

> More than anything else Mathematics is a method

Which I take to indicate that a mathematical approach exists and that there is not some sacrosanct body of results within which we must expect to find the tools to handle all (computing) problems.

2 Operation decomposition rules

An earlier (but yet to be published – see [Jon92] for a preprint) paper, traces one view of a history of work on program verification. One interesting observation that one could add to those made in the history is that all the way from Turing, through Floyd to Hoare and Dijkstra, there has been a reliance on proving programs satisfy specifications in terms of predicates of a single state. Since the purposes of a program are likely to be some sort of input/output relation this restriction being applied to post-conditions is surprising. In effect this odd decision causes people to invent a number of auxiliary tricks such as extra "variables" which cannot be changed by the program but have to be employed to remember the initial state.

In contrast, the earliest work (post my first stay in Vienna) on what was to become VDM [Jon73] used post-conditions which directly related initial and final states. This section relates the discovery of usable operation decomposition rules for such post-conditions.

Problem

In [Jon92], Hoare's seminal contribution of [Hoa69] is taken as a fulcrum around which earlier and subsequent efforts can be conveniently surveyed. The rule

(axiom) there which facilitates reasoning about partial correctness of repetitive while constructs is both well known and a model of clarity.

$$\boxed{\textit{Hoare}}\ \frac{\{P \wedge b\}\ S\ \{P\}}{\{P\}\ \textsf{while}\ b\ \textsf{do}\ S\ \textsf{end}\ \{P \wedge \neg b\}}$$

It is the reward of the single state view of post-conditions that such concise rules are available. (One can see this perhaps even more clearly with Dijkstra's weakest pre-condition work – see [DS90] and the considerable literature that this has spawned; but the Hoare rules provide a better comparison with the VDM work.)

In [Jon80] there are a number of rules which make it possible to establish results about iterative constructs. Interestingly, the first rules presented are for initialised while constructs. There is some virtue in this decision since the rules were intended to be used in program development and it is a fact that most useful iterative constructs have to be preceded by initialisation: the combined rules in some sense included the statement composition with the initialisation in a way which prompted reasonable design decisions. But even comparing the rules in [Hoa69] and [Jon80] for simple iteration the latter look heavy: they are presented as two separate domain conditions, two conditions about the relational meaning of the components and desired post-condition in addition to separate clauses about termination.

Peter Aczel wrote in [Acz82]

> But a more flexible and powerful approach has been advocated by Cliff Jones ... His approach is to allow the post-condition of a specification to depend on the starting state of a computation ...

but went on to add that

> His [CBJ] rules appear elaborate and unmemorable compared with the original rules for partial correctness of Hoare.

which is a considerably more polite commentary than they deserved. Aczel's unpublished note went on to show how a form of rule to cover post-conditions of two states could be formulated neatly (and be memorable). In the following, P is a predicate (truth-valued function) of a single state and R is a (transitive, well-founded) relation over state pairs. The rule

$$\boxed{\textit{AczelJones}}\ \frac{\{P \wedge b\}\ S\ \{P \wedge R\}}{\{P\}\ \textsf{while}\ b\ \textsf{do}\ S\ \textsf{end}\ \{R^r \wedge \neg b\}}\quad R\ \text{twf}$$

captures termination which was treated separately in the Hoare rule. (As a further confession here, [Jon80] presents the stupid compromise of using a termination function whose domain was a single state as in Hoare's rule).

Lesson

The lesson that I wish to draw from this story is that it is sometimes necessary to use a somewhat inelegant formulation until a better solution can be found. The path to the improvement might need to look at "simplifications" (such as Hoare's choice to rely on post-conditions of a single state) but one must also be prepared to look at the untidy solution to see what needs simplifying.

Hardy wrote in [Har67]

> there is no permanent place in the world for ugly mathematics

but there might be times when an ugly formulation is the best we have and its use is more honest than ducking the problem. Perhaps if we all had Hardy's skill and taste, we should always have clean formulations; I suspect not; I am sure that mere mortals can make a contribution be presenting something on which others can develop improvements.

Just before concluding this section, it is worth pointing out that the elegant formulation in the *AczelJones* rule above does actually lose something which was present in [Jon80]. In the original rules –quite apart from the question of including composed initialisation– there was a distinction between forward and backwards composition of the overall loop relation for the repetitive construct with the relation for the loop body. In [Jon80] they were called "up versus down" loops. Essentially the difference is whether the intermediate results of the loop are best understood in terms of a relation from the initial state or a relation with the end state. Use of the Hoare-like axioms tends to force non-overwriting of initial state information and can mostly be viewed as relations with the initial state; programs which compute the same result by destroying their inputs (e.g. computing n factorial whilst subtracting 1 from the variable containing n at each iteration) are often better understood via relations to the final state. The [Jon80] rules reflected this directly; there is an open piece of work to show how to do this with the new rule.

3 A challenge from parallel object-based languages

The events related in this section are much more recent than those discussed above and are to some extent are still unfolding. I developed a compositional way of developing some parallel (shared variable) programs during my work in Oxford: rely and guarantee-conditions were proposed in [Jon81,Jon83] as a way of recording and reasoning about interference; the work then lay somewhat dormant (a significant exception is the transfer to Temporal Logic in [BKP84]) until picked up and significantly developed in [Stø90,Xu92,Col94]. I returned to the challenge of finding usable methods for the design of interfering programs during a Research Council Fellowship from 1988-93. Although the developments of Ketil Stølen and Pierre Collette had made interference reasoning more useful, it was still clear that the proof work involved would be unlikely to find favour with working engineers who were already difficult to wean away from

testing to reasoning about sequential programs. Ideas like rely and guarantee-conditions had shown that it was possible to specify interference in a way which facilitated constructing proofs about compositional developments but there were simply too many things to be proved. (Furthermore, it took considerable experience to juggle items between the various predicates – this is a point reviewed in [CJ98].) Late on in my Fellowship I realised that parallel object-oriented (or perhaps object-based since inheritance is a quagmire) languages offered a marvelous way to constrain interference and thus to put in the hands of the program designer a way to indicate precisely where interference was –and was not– an issue; POOL [AR89,Ame89] was the major inspiration.

The research led to the development of a design language which became known as $\pi o \beta \lambda$ in which there are three levels at which interference can be controlled

- all instance variables are strictly local to the object in which they are contained
- objects which can only be reached via local references form islands within which no interference can be experienced
- objects which can be reached by general references are subject (via their methods) to interference from elsewhere

There was then a linguistic framework in which decisions about interference could be expressed and complex reasoning with rely and guarantee-conditions could be restricted to those areas where the designer made a conscious decision that intimate interference between two processes was necessary. An unexpected bonus of the move to object-based languages was that there was a clear way of introducing some forms of concurrency by transforming sequential programs into ones which were observationally equivalent at the input/output level but could run faster if there were sufficiently many processors available (and a few scheduling problems were resolved).

Problem

A simple example of two allegedly equivalent $\pi o \beta \lambda$ programs can be given in the context of a linked list implementation of a sorting vector: Figures 1 and 2 show two $\pi o \beta \lambda$ programs which ought be equivalent.

A sequence of integers is represented by a linked-list of *Sort* objects. The first object behaves as a server containing the whole queue but, in fact, each object holds a single element of the sequence (in v) and a unique reference to the next object in the list (in l). The method *insert* places its argument such that the resulting sequence is in ascending order; *test* searches the sequence for its argument. The implementation of both of these methods is sequential: at most one object is active at any one time. In Figure 2 concurrency has been introduced by applying two equivalences. The *insert* method given in Figure 1 is sequential: its client is held in a "rendezvous" until the effect of the insert has passed down the list structure to the appropriate point and the return statements have been

executed in every object on the way back up the list. If the return statement of *insert* is commuted to the beginning of the method as in Figure 2, the client is able to continue its computation concurrently with the activity of the insertion. Furthermore, as the insertion progresses down the list, objects "up stream" of the operation are free to accept further method calls. One can thus imagine a whole series of *insert* operations trickling concurrently down the list structure.

```
Sort class
vars v: ℕ ← 0; l: unique ref(Sort) ← nil
insert(x: ℕ) method
  begin
    if is-nil(l) then (v ← x; l ← new Sort)
    elif v ≤ x then l.insert(x)
    else (l.insert(v); v ← x)
    fi;
    return
  end
test(x: ℕ) method : 𝔹
  if is-nil(l) ∨ x ≤ v then return false
  elif x = v then return true
  else return l.test(x)
  fi
```

Fig. 1. Example Program *Sort* – sequential

One task facing the $\pi o \beta \lambda$ research was then to have a justified set of equivalence –or transformation– rules (I rejected the suggestion that I just put down the transformation rules as the (only) language definition both because I did not initially see the prospect of getting a complete set of rules and because this would have simply shifted the burden onto anyone interested in implementing such a language).

Well, this sounded like a familiar challenge — one for which I had been equipped by over two decades of work on language definition topics. One writes a model-oriented (operational or denotational) description of the language and proves the putative equivalences are consistent with the model-oriented semantics. I dismissed the idea of writing a denotational definition in terms of power domains (my views on the real advantages of denotational semantics are somewhat heretical) and opted to write an SOS definition. Although I realised when reviewing the work of some PhD students who picked up this line of research that there is little real guidance in the literature as to how to formulate a clear operational semantics definition, the task is not difficult for someone who has written several definitions. But then the problems start: one could say that there is no natural algebra for SOS definitions. Initial attempts at proving the putative equivalences sound with respect to the SOS definition were cumbersome and unrevealing.

```
Sort class
vars v: ℕ ← 0; l: unique ref(Sort) ← nil
insert(x: ℕ) method
  begin
    return;
    if is-nil(l) then (v ← x; l ← new Sort)
    elif v ≤ x then l.insert(x)
    else (l.insert(v); v ← x)
    fi
  end
test(x: ℕ) method : 𝔹
  if is-nil(l) ∨ x ≤ v then return false
  elif x = v then return true
  else delegate l.test(x)
  fi
```

Fig. 2. The concurrent implementation of *Sort*

As well as stumbling on POOL at what appears to have been the right time, I also read the early papers (cf. [Mil92,MPW92]) on the π-calculus at about this point. Independently of David Walker, I decided to give the semantics of my parallel object based language by mapping it to the π-calculus (David was actually there first and, when Ito-sensei kindly sent me a copy of [IM91], I saw [Wal91] as a confirmation of what I was attempting). A mapping was not difficult to write although here there was less experience on which to base the style and I think it is fair to say that David and I have influenced each other in subsequent choices. There are clear notions of equivalences for process algebras so I was naively optimistic that it was just a case of choosing the most appropriate to the context of $\pi o\beta\lambda$ and it would be straightforward to justify the putative equivalences. Well, there are certainly no shortage of notions of bi-similarity! In fact I conjecture –based on my search– that there are far more notions than there are examples of proofs. It quickly became clear that the process expressions which resulted from my mapping of the $\pi o\beta\lambda$ programs of Figures 1 and 2 to the π-calculus resulted in process expressions which are not bi-similar in any obvious sense. The full story of the search for proofs has yet to be written. I was flattered that scientists like David Walker, Davide Sangiorgi and Benjamin Pierce found the problem interesting (cf. [Wal93,Wal94]). Suffice it to say that the task of proving particular cases of the equivalences (but one related to what I termed in [HJ96] as delegation is more delicate than the one illustrated here) was tractable but prompted new variants of bi-similarity; the task of justifying the general equivalences via the mapping to the π-calculus has proved far more taxing; [HJ96] does contain arguments via SOS for the general case. (It would also be interesting to trace how the SOS proof attempts and those via the mapping have yielded insights which have influenced each other.)

Lesson

I will risk offending some colleagues by maintaining that the notions of process algebraic equivalences have proceeded as an end in themselves rather than being clearly motivated by applications. The lesson that I would wish to draw from the $\pi o \beta \lambda$ story is that a clear need for notions of equivalence might be a better guiding light for research than a pure mathematical taxonomy of variants.

In [Jon96] and in my invited talk at ICFEM in Hiroshima, I argued strongly for looking at new application areas far more exotic than the parallel object-oriented languages discussed in the previous section; examples such as Virtual Reality modelling languages, CORBA, Java etc. seem to me much more potentially stimulating than honing formal description techniques on well worn abstractions of older computing paradigms. I hope the the $\pi o \beta \lambda$ story illustrates the potential payoff.

4 Deploying formal methods

This section relates a strategic rather than a technical mistake. It prepares the way for a comment in the summary. To make clear my starting point here, I assume that the purpose of developing formal methods is to influence practical engineering of computer systems (whether hardware or software). It is a measure of my unease with some research in the area of computer science that I feel it necessary to state this fact.

At one stage of my career, I spent a lot of time trying to transfer technology into practical computing environments. Most notably this was associated with VDM in IBM but there were many other contacts and it is perhaps less well known that I was also involved as a consultant in the Z work for CICS. During the late 1970s and early 80s, I consistently advised that the only way to get formal methods into real use was to insist that everyone in a team became familiar with their use. This advice was the result of several earlier rather negative experiences. Firstly I had repeatedly seen groups of "architects" design systems and record their work in natural language which was passed to a group of formalists who attempted to build a model from their understanding of the English. Anyone who has experienced this process for a significant system will know the upshot: streams of questions and contradictions are generated in the formalisation; the direct reward for passing these back to the architects was a further stream of English (with sometimes a grudging acknowledgement that these formalists were asking very interesting questions). As way of arriving at a coherent model of a system, this left much to be desired. It seemed to me that the only way forward was for everyone to work on the same (formal) model. Furthermore, in teaching VDM, I always insisted on teaching proof concepts because I felt that –even if not used– they deepened people's understanding of the formalism. I had also seen –in IBM Hursley as it so happens– the waste that occurred when one or two people in a large group went off to learn some formal method and came back into a group where even the notation was a complete barrier to interchange. In

contrast, I had had some positive experiences where we were able to brainwash whole project groups at the same time.

Whether these experiences really justified my austere advice is not really the most important point here. It was some time before I saw that not only are there are some engineers who question the need for formal notation but there are some people who find it impossible to extract useful abstractions from the level of detail with which they normally work.

In more recent pronouncements on the application of formal methods I have modified my position considerably. I now recommend that something like Operations Research groups are formed where different members bring different skills. Thus there might be domain experts, implementors and formalists all involved in an architecture group.

Towards the end of the time when I was teaching at Manchester, I also taught a (Master's level) course on defining models of systems which put minimal emphasis on the notation itself and none on formal proof.

The lesson I draw from the above is that computer scientists have to think about how there ideas might be deployed; this might include addressing tool support before expecting users to adopt a new method; it almost certainly involves tackling a significant range of examples; my advice would also be to work together with real engineers (not just students) before thinking that one has the "silver bullet" for which industry has been waiting.

Summary

The above four personal stories are certainly not the only ones that I could have used to illustrate my themes: I could have chosen examples such as the decision in VDM language definitions to use an "exit combinator" rather than continuations, or the risk of deliberately using a data reification rule which was known to be incomplete. I could perhaps have chosen a catalogue of mistakes that I believe have been made by other scientists. But the examples chosen do serve to illustrate a number of points that I feel are in danger of being ignored by some researchers today.

Before reiterating the points of this sermon, I should make one thing absolutely clear. Nothing in what I have to say argues against the search for fundamental concepts which really do change the way we think about key concepts. With Algol-like languages we were lucky enough to find a ready concept for their denotations; for parallel languages, the search has been much harder and has not really yielded a universally agreed result. It is clear that finding the right concept here could make considerably more difference than detailed differences between one notation or another. Nor do I underestimate the importance of notation. Hoare's major supplement to Floyd's work was notational but it bought about a complete change of emphasis from operational reasoning to compositional design. But it must be remembered that significant steps in science are likely to come from long experimentation.

To take my points in the reverse of the order above: if we claim that we are doing research for the practising engineer we must make sure that the ideas proposed have at least some chance of being deployed in a way in which those building systems will actually be able to use them. Every esoteric mathematical concept must really be worthwhile (or carefully hidden in the way that my good friend Michael Jackson did so successfully in his design methods).

If we only look for mathematical elegance without clear applicability, we should be honest enough to list ourselves as (pure) mathematicians and not rely on a spurious contact with some simplified computing problems to justify our research.

We must look at today's applications and learn from them. Much has happened in computing since the "stack" and the problem of the "Dining Philosophers" were first taken as important paradigms on which to test formal approaches. Whatever the disadvantages of modern software (and I know many of them), significant systems are now constructed on top of a flexible and general interfaces to packages which handle much of the detail of –for example– the precise presentation on the screen. Attention has turned from closed systems which compute a particular input/output function to reactive systems.

We must not expect to find solutions to all of the problems presented by building computer systems in standard mathematics. Nor –unless we are unbelievably fortunate– will we always find beautiful mathematical solutions first time; but publishing an attempt which does solve a problem could spur others to show the way to a cleaner formulation. In any case, this is a more honest approach than ignoring all aspects of a problem which do not fit our current formalism. We should perhaps avoid massaging known problems: don't spend too much time on esoteric mathematics unless you're convinced it can all be hidden from engineers — remember that for formal methods to be used they must be usable by engineers.

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.

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

[BCJ84] H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

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

[Che86] J.H. Cheng. *A Logic for Partial Functions.* PhD thesis, University of Manchester, 1986.

[CJ91] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.

[CJ98] P. Collette and C. B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In G. D. Plotkin, editor, *to be published.* MIT Press, 1998.

[Col94] Pierre Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications – Application to UNITY.* PhD thesis, Louvain-la-Neuve, June 1994.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DS90] Edsger W Dijkstra and Carel S Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990. ISBN 0-387-96957-8, 3-540-96957-8.

[GM92] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[Gri81] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

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

[HJ96] Steve J. Hodges and Cliff B. Jones. Non-interference properties of a concurrent object-based language: Proofs based on an operational semantics. In Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *Oject Orientation with Parallelism and Persistence*, pages 1–22. Kluwer Academic Publishers, 1996.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[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.

[JM94] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[Jon72] C. B. Jones. Formal development of correct algorithms: an example based on Earley's recogniser. In *SIGPLAN Notices, Volume 7 Number 1*, pages 150–169. ACM, January 1972.

[Jon73] C. B. Jones. Formal development of programs. Technical Report 12.117, IBM Laboratory Hursley, June 1973.

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

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

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, 1986.

[Jon92] C. B. Jones. The search for tractable ways of reasoning about programs. Technical Report UMCS-92-4-4, Manchester University, 1992.

[Jon95] C.B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.

[Jon96] C. B. Jones. Some practical problems and their influence on semantics. In *ESOP'96*, volume 1058 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1996.

[Kle52] S. C. Kleene. *Introduction to Metamathematics.* Van Nostrad, 1952.

[Kol76] G. Koletsos. Sequent calculus and partial logic. Master's thesis, Manchester University, 1976.

[Luc69] P. Lucas. Note on strong meanings of logical operators. Technical Report LN 25.3.051, IBM Laboratory Vienna, 1969.

[LW69] P. Lucas and K. Walk. *On The Formal Description of PL/I*, volume 6 of *Annual Review in Automatic Programming Part 3.* Pergamon Press, 1969.

[McC63] J. McCarthy. Predicate calculus with 'undefined' as a truth-value. Technical Report AI Memo 1, Stanford Artificial Intelligence Project, March 22nd 1963.

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

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

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

[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.

[Wal94] D. Walker. Algebraic proofs of properties of objects, 1994. Proceedings of ESOP'94.

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