Philosophical Issues in Computer Science

Ralph Kopperman[1], Steve Matthews[2], and Homeira Pajoohesh[3]

[1] Department of Mathematics, The City College of New York,
138th St & Convent Avenue, New York, NY 10031, USA.
rdkcc@att.net
[2] Department of Computer Science,
University of Warwick, Coventry CV4 7AL, UK.
Steve.Matthews@warwick.ac.uk
[3] CEOL, Unit 2200, Cork Airport Business Park,
Kinsale Road, Co. Cork, Ireland.
homeiraaa@yahoo.com

Abstract. The traditional overlap between computer science and philosophy centres upon the issue of in what sense a computer may be said to *think*. A lesser known issue of potential common interest is the ontology of the semantics computer science assigns to programming languages such as Pascal or Java. We argue that the usual purely mathematical approach of denotational semantics necessarily excludes important semantic notions such as *program reliability*. By studying the ontology of determinism versus non determinism we deduce the presence of *vagueness* in programming language semantics, this then being compared to the *Sorites paradox*. Parallels are drawn between Williamson's treatment of the paradox using *ignorance* and generalised metric topology.

Introduction

Computer science has well established itself as an academic discipline throughout universities of the world. Often a separate department will exist specifically for the large number of undergraduates who wish to major in the subject they loosely term 'computers'. The power which computers bring both to the individual and society to change the world, for better or for worse, is self evident. What is less clear is the extent to which the use of computers changes our comprehension of the world. While internet communication makes the world appear smaller it does not mean we have created a new world as such. However, popular culture can clearly visualise the existence of inorganic life forms such as Commander Data in Star Trek. As there is no conceptual problem for *android* life forms can we in the academic discipline of computer science not provide substance to the notion of a computer *existing*? Usually this question is reduced to, can a computer *think*? If so, it is considered to exist, otherwise not. But this begs the question, what is *intelligence*? While being an interesting question, it does not address matters of computer existence which, while successfully employed in computer science, bear no obvious relation to intelligence.

Due to Gödel's incompleteness theorems computers necessarily have to process incomplete data, thus necessarily giving rise to consideration of the semantic

concept of *partial information*. Mathematics before the 1960s had no ready made answers on how to model information which in a well defined sense had bits missing. Computer scientists wishing to understand the semantics of programming languages had to find ways to *denote* information which could, quite legitimately & usefully, be incomplete. An analogy can be drawn from mathematics where there are many applications requiring us to have a real number i such that $i^2 = -1$. As clearly no such real number exists we creatively extend the real numbers to include appropriately termed *imaginary* numbers, one of which is to be $i = \sqrt{-1}$. The mathematical solution for denoting *partiality* accords a pivotal role to *nothing*, that information totally devoid of all content. In what sense then can nothing be said to exist? Posed as an oxymoron, what is the sound of silence? Fortunately we have available to us the inspired mathematics developed by, largely one person, Dana Scott to resolve the potential logical paradox of self reference embodied in any non trivial deterministic programming language. Roughly speaking a computer program is *deterministic* if each time it is executed the same output necessarily results. What are the implicit philosophical premises upon the nature of information assumed in Scott's ground breaking technical work? And what do these premises tell us about why his work has great difficulty in generalising to *non deterministic* programs, those where one execution may legitimately yield different results from another. In this paper we take a non deterministic program to be one where at one or more points in the program's execution the computer may choose any one from a finite set of commands specified in the program to execute next. A non deterministic program thus specifies a possibly infinite number of possible execution sequences. The work in this paper is an attempt to identify the key premises upon the fundamental nature of information underpinning Scott's work, and to demonstrate how computer science has successfully generalised the ontology of information inherited from mathematics. This paper is intended to promote research into ontological backdrops for existing practical work on the epistemology of reasoning, knowledge representation, and knowledge acquisition using computers. A clearer understanding of such ontology could make existing highly technical work on the semantics of programming languages more accessible to programmers. For philosophy, this work is intended to promote debate upon how we might ultimately develop a *theory of everything* to reconcile the essence of mathematics with that of computer science.

Premises for partiality

We now attempt to identify the implicit premises in Scott's approach to modelling partial information in the semantics of programming languages.

> **Premise 1 : programming language semantics is certain knowledge.**

Assigning a semantics to a program is the problem of determining and expressing all of its *certain* properties, these being properties which are provable in an appropriate logic of programs. By implication, there is no room here for approximation, lack of clarity, or ignorance of program properties. A program is

understood to be exactly what it is, neither in part of what it is, nor what it might be.

Premise 2 : a program's semantics is mathematically denotable.

This premise asserts that a mathematical model can be constructed in which each program can, for its meaning, be assigned a single value in that model.

Premise 3 : the semantics of a program is its observable behaviour.

The semantics of a deterministic program is the totality of effects which can be observed and recorded (by a human or another program) in the one, and only possible, execution sequence. At the risk of using an anthropomorphism, we may say that a deterministic program *is* what it *does*[4]. In contrast a non deterministic program may, when executed, produce one of many possible execution sequences. Thus, what may we say a non deterministic program *does*? Attempts have been made to generalise Scott's approach to non deterministic programs, so providing, in accordance with premise 2, a denotation for non determinism. By premise 1 we require certain knowledge of all properties of each non deterministic program. Thus, the meaning of a non deterministic program has to entail all the certain properties of each and every possible execution sequence. Thus, non determinism, it is reasoned by some, can be captured as a set of determinisms, a *determinism* being the meaning assigned to a deterministic program. And so, by premise 2, we need a mathematical model in which a value is a set of determinisms. And so, if S is the set of all possible determinisms for deterministic programs, then all we should need is a so-called *power domain* 2^S of subsets of S, each such subset to be the meaning assigned to a non deterministic program. The power domain approach presumes non determinism to be a set of determinisms. Non determinism is defined in terms of the primitive notion of *determinism*, and as such is a kind of, what we may say, *multiple determinism*.

Process calculi such as Milner's *Calculus of Communicating Systems* (*CCS*) [Mi89] have presumed non determinism to be a primitive notion, rather than a derivative of determinism. The process $P + Q$ (pronounced P *or* Q) is the process which can either behave like process P or behave like process Q. '+' is introduced as a primitive notion of *choice*. While the program may specify what choices there are, such as a coin has the sides *head* and *tail*, it is the computer executing the program, as in the tossing of a coin, that will make the choice. *CCS* reverses the approach of power domains, by asserting choice to be a core primitive notion, and that determinism is a process having just one choice. In other words, determinism is a choice of one. And so, in accordance with premise 2, what is the denotation of choice? An equivalence relation[5], termed

[4] The use of the terms *does* and *can do* in this paper are common parlance in computer science. They have no anthropomorphic connotations, they merely refer to the underlying capabilities of the computer system used to execute programs.

[5] A binary relation $\equiv$ over a set A is an *equivalence* if it has the following properties for all a, b, c in A. $a \equiv a$. If $a \equiv b$ then $b \equiv a$. If $a \equiv b$ and $b \equiv c$ then $a \equiv c$. The *equivalence class* for a is the set of all those members of A equivalent to a.

a *bisimulation* (pronounced *bi simulation*), on processes whose semantics are to be regarded as logically equivalent is introduced. The denotation of a process is its equivalence class.

Premise 4 : the semantics of a program is precisely that information which can be discovered by means of Turing computation.

This premise asserts that we can know whatever the computer can reveal to us by way of (Turing) computation, but no more. It is thus, by presumption, not permitted to enhance our understanding of the program's meaning by use of clever reasoning which would go beyond the Church-Turing hypothesis universally accepted in computer science as the definition of what is computable. The semantics of a program is thus necessarily taken to be precisely the computation(s) it describes. For a deterministic program we observe what it *does*, as this word was used earlier, in its one and only possible computation. For a non deterministic program, such as a process in *CCS*, we need to observe both what choices are available and what each such choice (when chosen) *does*. The required notion of *observation* is thus twofold, what a process *can do* and what it *does*. As a deterministic program only *can do* what it *does*, the notion of *can do* is safely dropped from consideration in its semantics.

Premise 5 : nothing is observable.

The problem of how to assign a denotation for a non terminating loop in a deterministic program written in a typical Pascal-like language loosely corresponds to the problem of assigning a semantics for recursive function theory, which in turn is similar to the problem of how to demonstrate the logical consistency of self applications such as $x(x)$ in the lambda calculus. The initial and instrumental technical concept in Scott's work is to utilise the notion of $\perp$ (pronounced *bottom*). $\perp$ can be understood for Pascal-like languages as a non terminating program which, while remaining alive, never progresses in any observable sense of producing further output. For example, the following Pascal code will run forever, but never produce any output.

```
while true do begin end ;
writeln("hello") ;
```

In other words, silence is the 'output' of the above code that never outputs anything. As with a person in a coma with irreparable brain damage, this program will remain *alive* indefinitely when executed, yet will never usefully progress in the sense that the `writeln` command is never executed, and so cannot produce the observable output `hello`. $\perp$ is introduced as a denotation for that which is undefinable. We can know all there is to know about $\perp$ up to the extent to which $\perp$ is defined. In summary, Scott cleverly arranges in his work that any entity will exist only up to the extent to which we can know, in accordance with Premise 4, its properties by means of computation. Consequently the meaning of something defined in terms of itself is that information which can be computed

from that definition, and no more. In summary, $\perp$ creatively introduces the intriguing oxymoron, *the sound of silence* as the starting point for computing a meaning for self reference.

So, what is *nothing* in a non deterministic language such as a process calculus? Hoare's *Co-operating Sequential Processes* (*CSP*)[Ro98] has a denotational semantics, a refinement of the power domain approach. If, as power domains do, we can take sets of determinisms, why not take sets of partial determinisms as such partiality is understood in Scott's work. *CSP* does just this, it uses, what we may term, *multiple partiality* to construct a Scott-like semantics for non determinism. Just as power domains define non determinism in terms of determinism, so *CSP* understands non determinism to be defined in terms of partial determinism. Power domains and *CSP* thus share the notion of non determinism as being derived from a notion of determinism, the former from totally defined determinisms, the latter from partial determinisms. This is not a reconciliation of determinism with (say) the primitive notion of *choice* as used in *CCS*, but a wishfully simplistic reduction of non determinism. Such a reconciliation is, we argue here, necessary if, in accordance with premise 5, the nothingness which may be either or both of *does* $\perp$ and the (primitive) choice of *can do* $\perp$ is to be truly observable.

A technical fix for an ontological problem?

Bisimulation, undoubtedly ingenious[6], is ultimately a mathematical device for avoiding an ontological problem. We have denotational approaches which can model non determinism in terms of total (i.e. power domains) or partial (i.e. *CSP*) determinism, and a non denotational approach (i.e. *CCS*) having choice as a primitive.

Our suggested premises trace the initial steps in Scott's mathematical constructions. More such premises would be needed for a complete treatment of his work, but the subject of this paper is to discuss philosophical issues surrounding concepts such as $\perp$, *partiality*, and *can do* involving less than certain knowledge of programs. In premises 1 through 5 we have established the key concept of $\perp$. $\perp$ is the starting point for a comprehensive mathematical theory to provide a denotation for each program in a deterministic programming language[AJ94]. $\perp$ is at first sight contradictory, it is the value defined for that which is totally undefined, an ingenious mathematical device to reason about that which is, in a well defined sense, unknown. There is no problem of self reference as the theory is carefully configured to ensure that we know partiality up to, but no more, the extent to which it is known with certainty by means of computation. Mathematically this works just fine, however, it will not generalise to include *choice* as a primitive notion. In contrast the process calculus *CCS* can elegantly handle

[6] The importance of bisimulation, created by Park & Milner for *CCS*, was clearly demonstrated when subsequently the mathematician Peter Aczel defined a theory of non-well-founded sets[Ac88], allowing for a set to have an infinite nesting of subsets. For example, the recursive definition $A = \{A\}$ defines a non-well-founded set $\{\{\{\ldots\}\}\}$, but this would not be a legitimate set in a well-founded set theory such as Zermelo-Frankel.

non determinism using a bisimulation relation, but, for this to work, excludes partial objects such as $\perp$. Computer science semanticists, studying either determinism or non determinism are agreed upon the validity of premises 1, 2, & 3, thus leading to a stark choice of either a denotation of programs having an overly simplistic reduction of non determinism to determinism, or a relation upon programs without the deterministic $\perp$. Is there no model of reconciliation? Is there no model in classical logic having *referents* for certain knowledge of programs such as deterministic behaviour, yet can accommodate the inherent lack of certainty which is non determinism? There is a well known problem in philosophy which strongly suggests that such searching is in vain. In addition, research into this problem indicates an alternative way forward, one which leads us to challenge the validity of the *certain knowledge* of premise 1.

Vagueness

Williamson[Wi96] describes the *Sorites paradox* as being one of seven puzzles proposed by the logician Eubulides of Miletus. No one disputes that that one grain of sand does not constitute a *heap*, and likewise that a trillion does make a heap. So, how many grains are needed to constitute a heap? Williamson traces the development of this perplexing problem from ancient Greece to fuzzy logic[Kos94], concluding that *"... none of the alternative approaches has given a satisfying account of vagueness without falling back upon classical logic"*. In one way or another there is a presumption that we can obtain with certainty a knowledgeable solution to the paradox, and so the inevitable recourse to the only certain language which is logic. Programming language semantics makes the same assumption, that each program can be all known, hence our premise 1. In contrast the serious programmer, such as any of our computer science undergraduates here at Warwick, 'know' that all the 'horrible' mathematics taught in our semantics course is even more complicated than the task of programming itself! They 'know' that one designs good software to be *reliable*, that is, to have a very high chance of producing desired results. They 'know' that producing a *totally correct* program in accordance with a semantics given as a specification (in an appropriate system of mathematics or logic) of what it can or should do is in practice, if not in theory, usually beyond their reach. The notion of *reliability* to a programmer is as a *heap* is to a philosopher. A program that always crashes when executed is clearly not reliable, while one that has run numerous times without a problem clearly is reliable. So, what is *reliability*? Computer science semanticists are sadly disdaining of the notion of reliability, only willing to discuss the certainty of total correctness. Philosophers have to their credit struggled with the notion of *heap*, while computer scientists have, through premise 1, had to reject reliability, a common sense notion used by all accomplished programmers. Reliability is a notion of vagueness that programming language semantics has yet to embrace, as the Sorites paradox has been embraced in philosophy.

The serious programmer does not expect to 'know' everything about their program, accepting that many of its properties cannot, for reasons they care not of, be known. The argument in this paper is that the insistence upon certain knowledge of premise 1 forces us to seek certain models of uncertain situations

such as non determinism. In contrast, fuzzy logic asserts that an imprecise truth value can be modelled by a precise real number between 0 and 1. Following Scott's example in the characterisation of $\perp$, we argue for the following position, contrary to those of both premise 1 and fuzzy logic.

Premise of necessary uncertainty : the partial can at best be known up to the extent to which it is partial.

This premise implies the existence of *ignorance* in programming language semantics, the inclusion of *necessary uncertainty*. Not only may our knowledge of programs be partial, as in the case of $\perp$, but in addition our ability *to know* may be partial as in the case of the actual choices made during the execution of a non deterministic program. A computer program to simulate the tossing of a coin can specify in its code the two possibilities of *head* and *tail*, what the program *can do*. What the program *does* when executed cannot be known from the program's semantics. Yet, paradoxically, current denotational models of non determinism, such as those for *CSP*, define *can do* in terms of *does*, when no one can know what *does* happen until it has happened. The result is that current work on semantics is really *can-do-semantics*, the only knowledge that can possibly conform to premise 1, and as such is of little use to the serious programmer whose common sense mind visualises what the program *does*. The prevailing idealistic culture of mathematical certainty in programming language semantics severely inhibits communication with programmers whose need is for a usable model of reliability. Program reliability is not a notion in semantics that classical mathematics can model as it is not a matter of certainty; mathematics needs to work with vagueness.

Williamson's thesis is that vagueness is, *"... an epistemic phenomenon, a kind of ignorance: there really is a specific grain of sand whose removal turns the heap into a non-heap, but that we cannot know which one it is"* He makes the thoroughly realist point that, *"... even the truth about the boundaries of our concepts can be beyond our capacity to know it"*

Conclusions and further work

This paper has studied a philosophical issue in computer science unrelated to the traditional problem of whether or not a computer can *think*. The usual differing technical approaches to determinism and non determinism in the semantics of programming languages have been studied as an example of a misrepresented ontological issue, and subsequently compared to the problem of vagueness in philosophy. Following Williamson's treatment of the Sorites paradox, we have argued that a reconciliation of determinism and choice has to relax the traditional presumption for the certainty of knowledge of program properties.

Scott's notion of partiality is traditionally modelled in a (point set) topology by weakening the Hausdorff (i.e. T_2) notion of separability, usually assumed in mainstream mathematics, to T_0 separability. Such T_0 topologies have subsequently been modelled using a form of generalised metric topology[Ma95], leading naturally to the study of bi-topology[Kop04]. A *bi-topology* is a pair of related topologies over the same universe of points, thus suggesting the inadequacy of

a single topology to model necessary uncertainty encountered in programming language semantics.

The possibility of a necessary separation of a topological space from our knowledge of it is unknown in the reductionism of classical mathematics, a school of thought to which denotational semantics has always been strongly affiliated. This has left semantics isolated from the potential benefits of 'inherent uncertainty' that have been embraced by chaos theory and quantum computing. Computer science, quantum physics, and philosophy, each in their own distinct way, suggest that a separation of *object* from *knowledge* can usefully, perhaps necessarily, be drawn between the object of study and our capacity to know it. A bi-topology could serve to model one topology defining the object of our study, and a second to tell us what we may be permitted to know in reasoning about the first.

The authors' researches so far[Ma95,Ma02,Kop04], observing as they do the doctrine of premise 1, have nonetheless made useful progress in advancing our technical understanding of *partiality*. But, further progress appears to require a definitive separation of *object* from *knowledge*. Ours, and any other related work, needs to accommodate the possibility of necessary uncertainty. Both *partial metric topology*[Ma95] and Williamson's *logic of clarity*[Wi96, Appendix] achieve such accommodation by quantifying the extent of partialness. At a more fundamental level philosophers need to engage with computer science and mathematics on re-interpreting the classical puzzle of the Sorites paradox as an epistemological problem of necessary uncertainty in computing.

References

[AJ94] S. Abramsky & A. Jung, *Domain Theory*, in *Handbook of Logic in Computer Science*, Vol. 3, Clarendon Press, pp.1-168, 1994.

[Ac88] P. Aczel, *Non-well-found Sets*, CSLI/SRI International, Menlo Park, CA94025, 1988.

[Kos94] B. Kosko, *Fuzzy thinking: the new science of fuzzy logic*, Flamingo, London 1994.

[Kop04] K. Kopperman, S. Matthews, & H. Pajoohesh, *Partial metrizability in value quantales*, Applied General Topology, to appear 2004.

[Ma02] S.G. Matthews, *Pixel geometry*, Electronic Notes in Theoretical Computer Science, vol.40, `www1.elsevier.com/gej-ng/31/29/23/show` , 2002.

[Ma95] S.G. Matthews, *Partial metric topology*, in *Papers on general topology and applications*, Proc. Eighth Summer Conference at Queen's College, Annals of the New York Academy of Sciences vol.728, pp.183-197, 1995.

[Mi89] R. Milner, *Communication and concurrency*, Prentice Hall, London 1989.

[Ro98] A.W. Roscoe, *The theory and practice of concurrency*, Prentice Hall Series in Computer Science 1998.

[Wi96] T. Williamson, *Vagueness*, *The Problems of Philosophy* Series, Routledge 1996.