

COPYRIGHT NOTICE: The original publication is available at www.springerlink.com. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

THE PHILOSOPHY OF COMPUTER SCIENCE: INTRODUCTION TO THE SPECIAL ISSUE

The Discipline of Computer Science

Computer scientists write programs. But so do economists, physicists, chemists, mathematicians, and engineers of all flavours—including software ones; the writing of programs does not distinguish the computer scientist from these other creators of software. Consequently, it does not distinguish the discipline of computer science. So what does? Well, one thing that does is its preoccupation with the meta-activity associated with programming i.e., with the design and development of tools and methodologies that facilitate and aid the specification, development, implementation and analysis of computing systems. For example, one concern of computer science is centered upon the invention of new programming languages that are generated by the need for greater levels of abstraction; abstractions that move us away from the architectures of physical machines and closer to the way that we wish to articulate and represent aspects of the worlds that we are modelling. We are currently in the age of object-oriented programming, but new, revolutionary computational models are already on the horizon which challenge the hegemony of von-Neumann architecture. For instance, genetic and evolutionary programming, quantum computing, and DNA computing have the potential to change not only the content and style of programming but also the underlying model of computation.

There are of course many other preoccupations of the computer scientist, but we have probably said enough to provide some orientation for the papers in the journal. But be aware, these are not straightforwardly computer science papers. Rather, they are concerned with philosophical issues that arise from reflection upon the nature and practice of the discipline. This is what we have in mind when we use the phrase *The Philosophy of Computer Science*.

Mathematics, Engineering, and Science

It seems evident that design is a central aspect of every branch of engineering be it chemical, electrical, or civil engineering. Moreover, almost all engineers employ mathematical models, and almost all of them test their creations. Much the same is true of the computer scientist: she designs new languages and tools, tests them and even provides mathematical models to explore their properties. So, evidently, much of what computer scientists do is, in this broad sense, engineering. But does this simple engineering picture really capture all there is to computer science? We suspect few would think so.

The first paper in the volume (2) more carefully characterises the nature of the discipline. It does so by introducing three caricaturing paradigms for computer science: the mathematical ('rationalist'), the engineering ('technocratic'), and the scientific. Put crudely, the first paradigm takes computer science to be a branch of

mathematics, the second takes it to be a scientific discipline and the last largely as an engineering endeavour that is stripped from any scientific and mathematical underpinnings. Each paradigm holds a different epistemological position concerning the question whether a priori or a posteriori knowledge about programs is attainable. With regards to ontological questions, each paradigm offers a different categorial account of existence for computer programs. These philosophical questions provide the backdrop for a more careful analysis of the nature of the discipline, and so set the scene for much of what follows.

Abstraction in Computer Science

By reference to the differing nature and roles of *abstraction* in mathematics and computer science, in (3), the perspective that computer science is a purely mathematical endeavour is implicitly challenged.

Abstraction, in one form or another, is crucial to any mathematical activity. To select an example close to home, generalised recursion theory subsumes the concrete theory of computation over the natural numbers and generalises it to more abstract mathematical structures. This kind of abstraction in mathematics suppresses details; in this case the specific properties of the natural numbers that are not needed for computability theory to bite. In line with this notion of mathematical abstraction, (3) characterises abstraction in mathematics as involving the *suppression of information* ('information neglect').

However, the authors argue, with the use of a rich variety of examples, that abstraction in computer science is quite different: whereas abstraction in mathematics abstracts away from details (information neglect or suppression), computer science only hides the underlying details (information hiding)—presumably so as they can be handled by different programs. Interestingly, this kind of information hiding is similar to that found in Bishop's constructive mathematics.

Proofs and Computations

Are proofs of program correctness genuine mathematical proofs, i.e. are such proofs on a par with standard mathematical ones? The main philosophical issue here is their epistemological status. The fact that they largely consist of very long chains of formal reasoning, leads one to ponder their *graspable* status. Moreover, with the use of theorem provers, at least some of the steps in the derivations are computer generated and, as such, are not even meant to be read by humans.

The authors of (1) investigate the general epistemological status of such computer-generated proofs. Initially, they set out to provide a sense in which ordinary mathematical proofs may be taken to be a priori. They characterise this in terms of an ideal mathematician who has at her disposal an unbounded amount of time, pencil and paper. If such a mathematician checks a given proof, and consequently takes the proof to be sound, then she may be taken to possess a priori knowledge about the truth of the proven proposition. The authors extend this characterisation to computer-assisted proofs by insisting that the ideal mathematician is able to verify the soundness of an

associated proof of correctness. More generally, they introduce a notion of *computational a priori knowledge* whereby knowledge obtained via a computation may be taken as a priori as long as our ideal mathematician can prove the correctness of the underlying program. Finally, they argue and stress that the amount of confidence that we have in simple proof-checkers is comparable to that we have in moderately complicated proofs that have been checked by humans.

The Semantics of Programming Languages

What are the conceptual issues that underpin the formal semantics of programming languages? (6) discusses the philosophical differences between different approaches to semantics. The author suggests that the semantics of programming languages provides a new branch of the platonist/formalist debate. In particular, he argues that platonism and formalism in the philosophy of mathematics do not line up with the operational/denotational divide in programming language semantics and that one could hold either philosophical position with respect to any of the so-called operational or denotational accounts.

The Gandy-Church-Turing Thesis

Some of the philosophical issues that surround new notions of computation are brought to the fore in (4). Much of computer science takes it for granted that the Church-Turing thesis characterises and prescribes actual physical computation. For example, all currently implemented programming languages are Turing-complete in the sense that they contain exactly the control constructs necessary to simulate a universal Turing machine. This underlying assumption is given its justification in a well known paper by Gandy. He provides four principles that are intended to characterise computation by machine that is as general and convincing as that provided by Turing for computation by humans. He shows that such machines exactly agree with Turing's characterisation (Gandy's Theorem).

In (4), the authors argue that Gandy's account of a discrete deterministic mechanical device is too narrow and that, Gandy's theorem withstanding, there are examples of such (ideal) machines that reach beyond the class of Turing computable functions. The inconsistency of this with Gandy's theorem is resolved by the fact that the proof implicitly employs a notion of *determinism* which some of these counter-examples do not satisfy. Other counter examples rely on Newtonian laws of physics, which are not permitted by Gandy's theory.

Quantum Computing and Complexity

The relationship between the Church-Turing thesis and quantum computing is discussed in (5). The author observes that every Turing computable function is quantum computable and that, to date, no quantum algorithm has been shown to compute a non-Turing computable function. However, when complexity issues are introduced, matters become more delicate. In particular, the class of algorithms that succumb to quantum speed-up is a proper subclass of those that can be simulated on a

Turing machine. For example, the universal Turing Machine is not in this subclass. In the case of NP-hard problems, the picture is less clear. Indeed, in general, it is unclear which quantum mechanical features are responsible for speed-up. However, he predicts that, if classically intractable problems are shown to be in the favoured subclass, they will be hardwired solutions. Potentially, this will give physical machines a more prominent role in any future computational theory.

Conclusion

There are of course many other philosophical issues and questions that surround computer science. But, from these papers, one thing is clear: not only does computer science provide some new twists and questions for other branches of philosophy (e.g. the philosophies of language, science and mathematics) but it also hints at new philosophical questions arising from within the discipline. Of course, which questions will ultimately form the core of *The Philosophy of Computer Science* is still up for grabs.

Raymond Turner, Amnon H. Eden
Colchester, April 2007

References

Konstantine Arkoudas, Selmer Bringsjord. "Computers, justification, and mathematical knowledge." *Minds and Machines*, this volume.

Timothy Colburn, Gary Shute. "Abstraction in Computer Science." *Minds and Machines*, this volume.

B. Jack Copeland, Oron Shagrir. "Physical Computation: How General are Gandy's Principles for Mechanisms?" *Minds and Machines*, this volume.

Amnon H. Eden. "Three Paradigms of Computer Science." *Minds and Machines*, this volume.

Amit Hagar. "Quantum Algorithms—Philosophical Lessons." *Minds and Machines*, this volume.

Raymond Turner. "Understanding Programming Languages." *Minds and Machines*, this volume.