

What is Philosophy of Computer Science?

(Extended Abstract)

Timothy Colburn
Department of Computer Science
University of Minnesota, Duluth

Abstract

This is a personal/historical account of what I regard philosophy of computer science (PCS) to be. I first contrast PCS with the theoretical foundations of computer science. I then contrast contributions made in the other direction, namely by the sciences to traditional problems of philosophy. Finally, I consider what makes a “philosophy of” something, and appeal for the role of abstraction in computer science as a primary source of questions in PCS.

In the 1970s I was an undergraduate and graduate student of philosophy. My undergraduate school did not provide computer facilities for students, and although I was a mathematics minor I never touched a computer in college. My graduate school provided computer time on an IBM 360 mainframe, and a computer science professor there wrote a file retrieval and editing system that allowed some of the more adventurous of us students to enter our term papers into the computer and edit them using paper teletype machines that printed their output on 11”x14” computer paper using daisy wheels. Thus for me the first marriage of philosophy and computer science was the use of a million dollar machine to do what would come to be called word processing on my philosophy papers.

Then, in a graduate seminar, we had to read a book by Boolos and Jeffrey called *Computability and Logic*. Of course, I had already studied symbolic logic, knew the predicate calculus, and even had a rudimentary grasp of Gödel’s incompleteness results, but this was the first philosophical treatment I’d seen of the notion of computation, or of the connection between computability and logic, such as proving the undecidability of first-order logic on the basis of the unsolvability of the halting problem. It was also the first I’d heard of Church’s thesis, the speculation that any plausible, precise characterization of mechanical computation could be shown to be equivalent, as far as the set of functions computable, to that particular characterization known as a Turing machine. Since, as I would later find out, the concept of a universal Turing machine would be the computer scientist’s abstract model of any actual physical computer, the study of computability and logic would be my first foray into the philosophy of computer science, although I would not have called it that at the time.

Today, study of the models of computation falls within the area computer scientists call the theoretical foundations of computer science, or FCS. The sub-areas of FCS are wide-ranging and encompass myriad topics ranging from automata and formal languages, to algorithms and data structures, to logic and complexity theory. If FCS includes computability and logic, and if these topics spawned Church’s thesis, which I call an example of philosophy of computer science, or

PCS, are PCS and FCS then the same? I would say not, for the deliverances of FCS stand to the practitioners of computer science in the same way that the deliverances of chemistry, say, stand to the practitioners of chemical engineering, and those who do chemistry are not necessarily doing the philosophy of chemistry. Suppose a chemist discovers a compound with certain novel catalytic properties and publishes the results. A chemical engineer working on the problem of storing hydrogen for onboard fuel cells then uses these results to come up with a way to store hydrogen in a non-volatile way. Similarly, say a computer scientist working in FCS comes up with a data structure and associated algorithms for sharing files in a network. He or she then publishes the algorithms and their efficiency results, and practitioners, who are applied computer scientists or perhaps software engineers, use the algorithms to solve problems in some application. The point of this is that FCS, despite the “Foundations” descriptor, is simply “pure” computer science, without the pressing constraints of applications. To the extent that the activity of pure science is not to be identified with the philosophy of that science, FCS is not PCS.

But before I had any inkling as to what pure computer science was all about, I wrote a philosophy dissertation in epistemology in which I presented and defended a particular theory of epistemic justification known as foundationalism. Along the way, I became acquainted, around 1977, with Pollock’s theory of defeasible reasoning, and I used it in support of my own theory of justification. After completing the dissertation I began teaching philosophy, and to supplement a senior seminar in metaphysics I read a 1964 anthology of papers edited by A. R. Anderson called *Minds and Machines* in which I first came across Turing’s classic “Computing Machinery and Intelligence.” In it, Turing, who can be regarded as an early philosopher of computer science, was, arguably, posing a theory in the philosophy of mind, namely the functional theory of mind championed by many modern cognitive scientists.

In later years, I realized that when computer scientists advance theories such as that underlying the Turing Test, they are not so much as doing PCS as they are posing answers to classic philosophical questions. This became very clear to me when, around 1980, I saw that computer scientists such as Reiter, McDermott and Doyle, and McCarthy were offering philosophical theories of defeasible reasoning when advancing their accounts of default reasoning, nonmonotonic reasoning, and circumscription, respectively. In other words, they were doing philosophy, in this case epistemology and logic, in the classic sense. In the grand philosophical scheme of things, these forays by computer scientists into philosophy coincided with a movement in the mid to late 20th century toward the “naturalization” of philosophy, or the willingness to entertain the deliverances of empirical investigation of traditionally *a priori* subjects, for example, the analysis of knowledge or the definition of consciousness. Philosophical problems, in other words, might, for the first time, be illuminated by looking at scientific results in the investigation of computational, psychological, or perhaps neurophysiological topics.

In the 1980s I left the teaching of philosophy and dedicated myself to computer science. I worked in the area of applied artificial intelligence, where I came across the work of computer scientists which seemed to my mind to be attempts to do philosophy. Having feet in both disciplines, I did not feel myself drawn to one side or another of a “turf battle.” I felt instead that in this case both sides could benefit from contributions from the other, in the best tradition of multidisciplinary cooperation. Missing academia, I returned to teaching in 1988, when I met Jim Fetzer, a philosopher of science who became intensely interested in verification of program correctness, or the attempt by computer scientists to prove that a program meets its specification. This seemingly innocuous aspect of computer science methodology, under Fetzer’s analytical lens, became, almost overnight,

a contentious battlefield when Fetzer made, in a flagship journal of computer science, the (to my mind) tame observation that a formal program verification can, at best, prove the correctness only of an abstract algorithm embodied by a program, and not the running program itself. I regarded this observation, though not, I thought, a terribly shocking one, as a paradigm example of philosophy of computer science, because it subjected a foundational aspect of a given science, namely program verification, to a characteristically philosophical treatment, namely the analysis of concepts of that science and the critical evaluation of its beliefs.

In observing the firestorm and debate that ensued following Fetzer's article, it occurred to me that the controversy seemed to result from the seeming incompatibility of two points of view regarding computer science: one looking at computer science as an experimental science, situated in an uncertain world where people, their behavior, and the vaguely described problems they have to solve hold sway; and one looking at computer science as solution engineering, where problems are understood and described completely in advance of their solution, which amounts to the application of perhaps formal procedures in the accomplishing of that solution, in much the same way as mathematical reasoning, and perhaps discovery, proceeds. I attempted to reconcile these points of view by stepping back and seeing them both in the purview of computer science "in the large." I was helped in this endeavor by my familiarity with defeasible (non-monotonic) reasoning, which I regarded as fundamental to the reasoning process of formal verificationists. So, curiously, in my meagre attempt to do PCS I was helped by the work of both philosophers, like Pollock, and computer scientists, like Reiter, etc.

The cross-fertilization of philosophy and computer science continued for me as I investigated the relationship between heuristic search in computer science and epistemic justification in philosophy. So then, ten years ago I began writing a book on the intersection of philosophy and computer science. I was careful to use the conjunction "and" and not the preposition "of" in relating the disciplines, because I was struck by the fact that the two disciplines seemed to be making contributions to one another. Still, I devoted the final third of the book to the "of" aspect of the relationship, and now I find myself part of perhaps the first conference track ever devoted to PCS, in which we are considering the question, "What is PCS?"

For the purposes of my book, I tried to answer this by considering the two questions, "What is philosophy?" and "What is a philosophy of X?", where X is a particular domain. After all, work in various "philosophy of"s has proceeded apace for years. My own view is that philosophy, at its core, is concerned with the concept of knowledge (epistemology), reasoning (logic), existence (ontology), and value (ethics), and that a "philosophy of X" addresses these concepts within the domain and conceptual framework of X. So, PCS is to be defined not so much intensionally as extensionally, by pointing at work and saying, "That's PCS." For example, a discussion of the ethics of digital rights management (thwarting illegal distribution of copyrighted works) through the deployment of rootkits (hidden programs and files that communicate externally unbeknownst to a computer's user) is certainly about value, and certainly involves computer science, but it would not be PCS *per sé* because it is not about the *concept* of value within computer science. On the other hand, discussions such as those by James Moor concerning whether computer ethics in general is, due somehow to the unique nature of computers themselves, a new species of ethics, rather than old-fashioned ethics in a new guise — these discussions would constitute PCS, since they are about both computer science and ethics-in-itself, so it were.

If this is right, and PCS is to be defined extensionally, then your favorite philosophical problem of the day, couched in the domain of computer science, would constitute PCS. The array of questions

is large: Can computers think? Can a program know things? What is the ontological status of a virtual machine? Is Church's thesis correct? Do androids have rights? etc.

In my book, I also attempted to characterize a "philosophy of" as encompassing "central questions" about the nature of a discipline. A philosophy of art might discuss whether "performance art" really is art, if the performance involves, for example, a pianist sitting at a piano and doing nothing for an hour. The criterion for being a "central question," of course, would change over time, as debates come and go, and previous questions become answered to a group's satisfaction.

What, then, might be a central question for PCS? Does computer science, for example, suffer from any kind of identity crisis? The controversy surrounding Fetzer's paper 16 years ago seemed to point to friction between those who embrace a mathematical paradigm for computer science and those who see computer science as more of an empirical discipline. In my experience, both developing and teaching software for the last 25 years during the so-called and much ballyhooed "software crisis," I have come to regard computer science as a discipline non-subservient to mathematics but dependent upon it to no more or less an extent than any other natural science is in the construction of needed mathematical models. Gary Shute and I have thought about the kinds of abstraction that occur in computer science and mathematics and found them strikingly different. Since mathematics and its methods have been well ensconced for centuries, then perhaps the abstraction methods that are used in software development are philosophically novel? And if not, have these abstraction methods been employed before in other areas of inquiry? These, at least for Shute and I, are current central questions in PCS.

We are interested in these partly because we have seen the participants in computation progress from machine-oriented data types to domain-oriented objects. This march of progress coincides directly with measured (and by no means complete) success in dealing with the aforementioned software crisis, and with explosive growth in both software complexity and software demand. What accounts for this co-incidence? Is it due to the latest and greatest programming paradigm — object orientation? If so, how do we characterize object orientation's abstraction capabilities, and is it possible to improve on them?

Finally, what is the relationship between abstraction and ontology in software development? What is the ontological status of an object in a running object-oriented program? What kinds of objects are involved in a computational process? Do they depend on the kinds of objects described in a source program? Nearly 60 years ago, right around the dawn of the age of digital electronic computation, the philosopher W. V. O. Quine made a radical claim for the time. In trying to deal with conundrums involving existence, and in adjudicating among various philosophies of mathematics, he said "To be is to be the value of a bound variable." His point was that one's language has much to do with one's ontology. He was quick to say that such a formula cannot say what exists, but only what a given theory *says* exists. As object-oriented programmers, our language says that things as various as shopping carts, chat rooms, and network sockets exist, in some sense, in our computational processes. Have we, in 60 years, come any closer to saying what there is in a computational process? Do our powers of abstraction as programmers have any effect on what there is? I believe that PCS should have something to say about this.