Chapter 7.4
Knowledge, Truth, and Values in Computer Science

Timothy Colburn
University of Minnesota, USA

Gary Shute
University of Minnesota, USA

ABSTRACT

Among empirical disciplines, computer science and the engineering fields share the distinction of creating their own subject matter, raising questions about the kinds of knowledge they engender. The authors argue that knowledge acquisition in computer science fits models as diverse as those proposed by Piaget and Lakatos. However, contrary to natural science, the knowledge acquired by computer science is not knowledge of objective truth, but of values.

INTRODUCTION

Computer science, insofar as it is concerned with the creation of software, shares with mathematics the distinction of creating its own subject matter in the guise of formal abstractions. We have argued (Colburn & Shute, 2007), however, that the nature of computer science abstraction lies in the modeling of interaction patterns, while the nature of mathematical abstraction lies in the modeling of inference structures. In this regard, computer science shares as much with empirical science as it does with mathematics.

But computer science and mathematics are not alone among disciplines that create their own subject matter; the engineering disciplines share this feature as well. For example, although the process of creating road bridges is certainly supported by activities involving mathematical and software modeling, the subject matter of the civil engineer is primarily the bridges themselves, and secondarily the abstractions they use to think about them.

Engineers are also concerned, as are computer scientists, with interaction patterns among aspects

DOI: 10.4018/978-1-61350-456-7.ch7.4

of the objects they study. The bridge engineer studies the interaction of forces at work on bridge superstructure. The automotive engineer studies the interaction of motions inside a motor. But the interaction patterns studied by the engineer take place in a physical environment, while those studied by the software-oriented computer scientist take place in a world of computational abstractions. Near the machine level, these interactions involve registers, memory locations, and subroutines. At a slightly higher level, these interactions involve variables, functions, and pointers. By grouping these entities into arrays, records, and structures, the interactions created can be more complex and can model real world, passive data objects like phone books, dictionaries, and file cabinets. At a higher level still, the interactions can involve objects that actively communicate with one another and are as various as menus, shopping carts, and chat rooms.

So computer science shares with mathematics a concern for formal abstractions, but it parts with mathematics in being more concerned with interaction patterns and less concerned with inference structures. And computer science shares with engineering a concern for studying interaction patterns, but it parts with engineering in that the interaction patterns studied are not physical. Left out of these comparisons is the obvious one suggested by computer science's very name: What does computer science share with empirical *science*? In this chapter we will investigate this question, along with the related question: What is the nature of computer science knowledge?

METAPHOR AND LAW

We were led to these questions, interestingly, when, in our study of abstraction in computer science, we found ourselves considering the role of *metaphor* in computer science (Colburn & Shute, 2008). Computer science abounds in physical metaphors, particularly those centering around *flow* and *motion*. Talk of flow and motion in computer science is largely metaphorical, since when you look inside of a running computer the only things moving are the cooling fan and disk drives (which are probably on the verge of becoming quaint anachronisms). Still, although bits of information do not ``flow'' in the way that continuous fluids do, it helps immeasurably to ``pretend'' as though they do, because it allows network scientists to formulate precise mathematical conditions on information throughput and to design programs and devices that exploit them. The flow metaphor is pervasive and finds its way into systems programming, as programmers find and plug ``memory leaks'' and fastidiously ``flush'' data buffers. But the flow metaphor is itself a special case of a more general metaphor of ``motion'' that is even more pervasive in computer science. Descriptions of the abstract worlds of computer scientists are replete with references to motion, from program jumps and exits, to exception throws and catches, to memory stores and retrievals, to control loops and branches. This is to be expected, of course, since the subject matter of computer science is *interaction* patterns.

The ubiquitous presence of motion metaphors in computer science prompted us to consider whether there is an analogue in computer science to the concern in natural science with the discovery of natural laws. I.e., if computer science is concerned with motion, albeit in a metaphorical sense, are there laws of computational motion, just as there are laws of physical motion? We concluded (Colburn & Shute, 2010) that there are, but they are laws of programmers' own making, and therefore prescriptive, rather than descriptive in the case of natural science. These prescriptive laws are the programming invariants that programmers must first identify and then enforce in order to bring about and control computational processes so that they are predictable and correct for their purposes. The fact that these laws prescribe computational reality rather than describe natural reality is in keeping with computer science's special status,