The Philosophy of Computer Science

First published Fri Dec 12, 2008

The Philosophy of Computer Science (**PCS**) is concerned with philosophical issues that arise from reflection upon the nature and practice of the academic discipline of computer science. But what is the latter? It is certainly not just programming. After all, many people who write programs are not computer scientists. For example, physicists, accountants and chemists do. Indeed, computer science would be better described as being concerned with the meta-activity that is associated with programming. More generally, and more precisely, it is occupied with the design, development and investigation of the concepts and methodologies that facilitate and aid the specification, development, implementation and analysis of computational systems. Examples of this activity might include the design and analysis of programming, specification and architectural description languages; the construction and optimisation of compilers, interpreters, theorem provers and type inference systems; the invention of logical frameworks and the design of embedded systems, and much more. Many of the central philosophical questions of computer science surround and underpin these activities, and many of them centre upon the logical, ontological and epistemological issues that concern it. However, in the end, computer science is what computer scientists do, and no exact formulaic definition can act as more than a guide to the discussion that follows. Indeed, the hope is that **PCS** will eventually contribute to a deeper understanding of the nature of computer science.

But mapping out the philosophical landscape of computer science is no easy task. Fortunately, traditional branches of philosophy can provide intellectual and structural guidance. For example, in the philosophies of mathematics and physics, there are central questions concerning the nature of the objects dealt with, what constitutes knowledge and the means of obtaining that knowledge. The philosophy of language raises questions about the content and form of a semantic theory for natural language. It brings to the fore the underlying ontological and epistemological assumptions of the semantic enterprise. Ontology indicates the kinds of things there are, how to individuate them and their role in framing our conceptual schemes. The philosophy of logic provides an account and analysis

of different kinds of logical systems and their role in everyday and specialized discourse. Analogies and similarities from these and other branches of philosophy should prove helpful in identifying and clarifying some of the central philosophical concerns of computer science. The existing influence of these disciplines on **PCS** will emerge as we proceed. In particular, the second, third and fourth sections will reflect the impact of ontology and the philosophies of language and mathematics.

---

1. Some Central Issues

To begin with we shall enumerate what we take to be some of the central issues and questions. This will provide the reader with a quick outline of matters that will supplement the more detailed discussion to come. Although many of them have not been directly addressed in the literature and are in need of some clarification, these questions illustrate the kinds of issues that we take the **PCS** to be concerned with.

1. What kinds of things are programs? Are they abstract or concrete? (Moor 1978; Colburn 2004)
2. What are the differences between programs and algorithms? (Rapaport 2005a)
3. What is a specification? And what is being specified? (Smith 1985; Turner 2005)
4. Are specifications fundamentally different from programs? (Smith 1985)
5. What is an implementation? (Rapaport 2005b)
6. What distinguishes hardware from software? Do programs exist in both physical and symbolic forms? (Moor 1978; Colburn 2004)
7. What kinds of things are digital objects? Do we need a new ontological category to house them? (Allison et al. 2005)
8. What are the objectives of the various semantic theories of programming languages? (White 2004; Turner 2007)
9. How do questions in the philosophy of programming languages relate to parallel ones in the philosophy of language? (White 2004)
10. Does the principle of modularity (e.g., Dijkstra 1968) relate to the conceptual issues of full-abstraction and compositionality?
11. What are the underlying conceptual differences between the following programming paradigms: structured, functional, logic, and object-oriented programming?
12. What are the roles of types in Computer Science? (Barandregt 1992; Pierce 2002)
13. What is the difference between operational and denotational semantics? (Turner 2007)
14. What does it mean for a program to be correct? What is the epistemological status of correctness proofs? Are they fundamentally different from proofs in mathematics? (DeMillo et al. 1979; Smith 1985)
15. What do correctness proofs establish? (Fetzer 1988; Fetzer 1999; Colburn 2004)
16. What is abstraction in computer science? How is it related to abstraction in mathematics? (Colburn & Shute 2007; Fine 2008; Hale & Wright 2001)
17. What are formal methods? What is formal about formal methods? What is the difference between a formal method and informal one? (Bowen & Hinchey 2005; Bowen & Hinchey 1995)
18. What kind of discipline is computer science? What are the roles of mathematical modelling and experimentation? (Minsky 1970; Denning 1980; Denning 1981; Denning et al. 1989; Denning 1985; Denning 1980b; Hartmanis 1994; Hartmanis1993; Hartmanis 1981; Colburn 2004; Eden 2007)
19. Should programs be considered as scientific theories? (Rapaport 2005a)
20. How is mathematics used in computer science? Are mathematical models used in a descriptive or normative way? (White 2004; Turner 2007)
21. Does the Church-Turing thesis capture the mathematical notion of an *effective* or *mechanical* method in logic and mathematics? Does it capture the computations that can be performed by a human? Does its scope apply to physical machines? (Copeland

2004; Copeland 2007; Hodges 2006)
22. Can the notion of *computational thinking* withstand philosophical scrutiny? (Wing 2006)
23. What is the appropriate logic with which to reason about program correctness and termination? (Hoare 1969; Feferman 1992) How is the logic dependent upon the underlying programming language?
24. What is information? (Floridi 2004; Floridi 2005) Does this notion throw light on some of the questions listed here?
25. Why are there so many programming languages and programming paradigms? (Krishnamurthi 2003)
26. Do programming languages (and paradigms) have the nature of scientific theories? What causes a programming paradigm shift? (Kuhn 1970)
27. Does software engineering raise any philosophical issues? (Eden 2007)

In what follows we shall put some flesh on a few these questions.

2. Existence and Identity

How do we categorize and individuate the entities and concepts of computer science? What kind of things are they and what determines their identity? For example, some are clearly concrete physical objects (e.g. chips, routers, laptops, graphics cards) and some are not (e.g. formal grammars, abstract machines, theorem provers, logical frameworks, process algebras, abstract data types). But the characterization of some of the central notions such as programs and data has been more problematic. In particular, the ontological status of programs has been taken not to be entirely straightforward, and nor has the issue of their criteria of identity.

2.1 The Dual Nature of Programs

Many authors (Moor 1978; Rapaport 2005b; Colburn 2004) discuss the so-called *dual* nature of programs. On the face of it, a program appears to have both a textual and a mechanical or process-like guise. As text, a program can be edited. But its manifestation on a machine-readable disk seems to have quite different properties. In particular, it can be executed on a physical machine. So according to the principle of the indiscernibility of identicals (§3.3), the two guises cannot be the same entity. Of course, anyone persuaded by this duality is under an obligation to say something about the relationship between these two apparent forms of existence.

One immediate suggestion is that one manifestation of a program is an *implementation* of the other i.e., the physical manifestation is an implementation of the textual one. However, even within the confines of computer science, it is not immediately clear that the word

implementation refers to just one notion. Often it is used to refer to the result of a compilation process where a program in a high-level language (the source code) is transformed into machine language (the object code). But equally often it is used to refer to the process where the source code is somehow directly realized in hardware (e.g. a *concrete* implementation in semiconductors). And presumably, this is the relevant notion. But without a more detailed philosophical analysis of the notion of implementation (§3.2) itself (Rapaport 2005b), it is unclear how this advances the discussion; we seem only to have named the relationship between the two apparent forms of existence. In a similar vein, others have described the relationship between the program-text and the program-process as being similar to that between a plan and its manifestation as a series of physical actions. But this does not seem to be quite analogous to the program-process pairing: we are not tempted to refer to the plan and the physical process as being different manifestations of the same thing. For example, are we tempted to think of a plan to go for a walk and the actual walk as different facets of the same thing?

Perhaps matters are best described by saying that programs, as textual objects, *cause* mechanical processes? The idea seems to be that somehow the textual object physically causes the mechanical process. But this would seem to demand some rather careful analysis of the nature of such a causal relation. Colburn (2004) denies that the symbolic text has the causal effect; it is its physical manifestation (the thing on the disk) that has such an effect. Software is a *concrete abstraction* that has a medium of description (the text, the *abstraction*) and a medium of execution (e.g., a *concrete* implementation in semiconductors).

A slightly different perspective on these issues starts from the question of program identity. When are two programs taken to be the same? Such issues arise for example in attempts to determine the legal identity of a piece of software. If we identify a program with its textual manifestation then the identity of a program is sensitive to changes in its appearance (e.g. changing the font). Evidently, it is not the text alone that provides us with any philosophically interesting notion of program identity. Rather, to reach an informed criterion of identity we need to take more account of semantics and implementation. We shall return to this subject in §3 and §6.

2.2 Programs and Algorithms

Whatever view we take of programs, the algorithm-program distinction is also in need of further conceptual clarification. Algorithms are often taken to be mathematical objects. If this is true then many of the philosophical issues concerning them also belong to the philosophy of mathematics. However, algorithms are arguably more central to computer science than to mathematics and deserve more philosophical attention than they have been given. While there has been some considerable mathematical study of algorithms in

theoretical computer science and mathematical logic (e.g., Moschovakis 1997; Blass & Gurevich 2003) there has not been a great deal of philosophical discussion that is centred upon the nature of algorithms and the differences between algorithms and programs.

Is it that algorithms are abstract objects, in the sense offered by Rosen (2001), whereas programs are concrete? More precisely, are algorithms the abstract mathematical counterpart of a textual object that is the program? This picture naturally lends itself to a form of ontological Platonism (Shapiro 1997) where algorithms have ontological priority and programs supply the linguistic means of getting at them. On this view, algorithms might be taken to furnish the semantics (§3) of programming languages. Of course, this picture inherits all the advantages and problems with such a Platonic perspective (Shapiro 1997).

A less Platonic view has it that algorithms contain the *ideas* expressed in a program. In law this has been taken to be the reason that algorithms, as opposed to programs, are not copyrightable (§7.1). Of course, the term *idea* requires further philosophical analysis. Indeed, it could be argued that the bare notion of algorithm is in much less need of clarification than the standard account of ideas and its associated notions of abstraction (Rosen 2001).

Finally, it is almost a folklore view that Turing machines provide us with a formal analysis of our notion of algorithm. But does this fit the contemporary notion that is employed in modern computer science with its sophisticated notions of representation and control? Moschovakis (1997) offers an analysis that does somewhat better.

2.3 Programs and Specifications

Another popular distinction that ought to be the topic of some critical analysis occurs with respect to programs and specifications. What are specifications and how are they different from programs? While there is little direct discussion of this issue in the philosophical literature (but see Smith 1985), the nature of specifications is a fundamental issue for the conceptual foundations of computer science.

One view, commonly found in textbooks on formal specification, is that programs contain detailed machine instructions whereas (functional) specifications only describe the relationship between the input and output. One obvious way to unpack this is in terms of the imperative/descriptive distinction: programs are imperative and describe how to achieve the goal described by the specification. Certainly, in the imperative programming paradigm, this seems to capture a substantive difference. But it is not appropriate for all. For example, logic, functional, and object-oriented programming languages are not obviously governed by it: taken at face value, programs encoded in such languages consist

of sequences of definitions, not 'instructions'. Furthermore, non-functional specifications cannot be articulated as statements about the relation between input and output because they impose requirements on the design and on the kind of instructions that may be included in any program.

Another view insists that the difference between specifications and programs is to be located in terms of the notion of implementation, i.e., can it be compiled and executed? But what is meant by that? Is it meant in the sense of having an existing compiler? This interpretation is rather shallow because it offers not a conceptual criterion of distinction but a contingent one. For example, during the first five generations of programming languages (2nd half of the 20th century), recursive, modular, functional, and object-oriented specifications of one generation have come to be articulated as programs in the next, i.e., today's specification languages frequently become tomorrow's programming languages.

Another view suggests that programming languages are those languages that have an implementation *in principle* whereas specification languages are those that cannot. And presumably, the reason that they cannot is that specification languages permit one to express notions that are not Turing computable. This distinction is in keeping with many existing specification languages that are based upon Zermelo-Fraenkel set theory and higher-order logic. However, it seems odd that what should characterize a specification language is the fact that it can express non-computable properties and relations. Are any of these non-computable demands really necessary in practice (Jones & Hayes 1990; Fuchs 1994)?

The diversity of these views suggests that the traditional, binary divide between specifications and programs is an example of an issue in **PCS** that deserves more attention, not only for conceptual clarification but also because it might have implications for the design of future programming and specification languages.

3. Semantics

The grammar of a programming language only determines what is syntactically legitimate; it does not inform us about the intended meaning of its constructs. Thus the grammar of a programming language does not, by itself, determine the *thing* that people program in. Instead, it is the grammar enriched with a semantic account (formal or informal) that is taken to do so. The semantics is meant to inform the programmer, the compiler writer and the theoretician interested in exploring the properties of the language. Indeed, it is often claimed that to meet the different requirements of the programmer and compiler writer, different semantic accounts, at different levels of abstraction, are required. And the job of the theoretician is to explore their relationship.

This is the standard picture that emerges in the semantic literature. But much of this is in need of conceptual clarification. In this section we consider a just few of the issues that arise from this activity.

3.1 Denotational and Operational Semantics

One of the most important distinctions in programming languages semantics centres upon the distinction between operational and denotational semantics. An operational semantics (Landin 1964; Plotkin 1981) provides an interpretation of a programming language in terms of some abstract machine. More precisely, it is a translation of expressions in the programming language into the instructions or programs of the abstract machine. For example, Program 1 would be unpacked into a sequence of abstract machine operations such as "$a \leftarrow 0$" and *push*. Operational semantics can also be conceived as algorithmic semantics especially when the underlying machine is aimed at characterizing the very notion of algorithm (e.g., Moschovakis 1997).

In contrast, a denotational semantics (Milne & Strachey 1977) provides an interpretation into mathematical structures such as sets or categories. For example, in the classical approach, sets—in the form of complete lattices and continuous functions on them—provide the mathematical framework.

But is there any significant conceptual difference between them? Is it that denotational semantics, being explicitly based upon mathematical structures such as sets, is mathematical whereas operational semantics is not? Turner (2007) argues not: they all provide mathematical interpretations.

Or is it that operational semantics is more *machine-like*, in the sense of constituting an abstract machine, whereas with denotational semantics, which is given in set-theoretic terms, there is no hint of an abstract machine? Such distinctions however have not proven conceptually significant because denotational semantic-accounts can all be seen as structures that constitute an abstract machine with states and operations operating on them. Nor are operational accounts closer to implementation: denotational approaches (Milne & Strachey 1977) are also very flexible and are able to reflect various levels of implementation detail.

Another possible distinction concerns the compositional (or otherwise) nature of the semantics. Loosely speaking, a semantics is taken to be *compositional* (Szabó 2007) if the semantic value of a complex expression is a function of the semantic values of its parts. Compositionality is taken to be a crucial criterion of semantics since is seems required to explain the productivity of our linguistic understanding: it is said to explain how we understand and construct complex programs. But does it provide us with a wedge to

separate operational and denotational semantics? Unfortunately, it seems not to do so: while denotational definitions are designed to be compositional, it is certainly not the case that all operational semantics are not compositional.

Finally, some versions of semantics differ concerning the existence of a recursive model, i.e., an interpretation in Turing machines or Gandy machines (§5.1). However, even this does not exactly line up with the traditional operational/denotational divide. Some denotational definitions have a recursive model and some do not.

It seems very hard to pin this distinction down. On the face of it, there appears no sharp conceptual distinction between operational and denotational semantics.

3.2 Implementation and Semantic Interpretation

What is the difference between a semantic interpretation and an implementation? For example, what is the conceptual difference between compiling a program into machine code and giving it a denotational semantics? According to Rapaport (2005b), an implementation is best viewed as semantic interpretation, where the latter is characterized via a mapping between two domains: a syntactic one and a semantic one. And both of these are determined by *rules* of some description. For example, the compiled code (in combination with the rules that govern its semantics) is the semantic account of the source code.

In a common understanding of the term ‘implementation’, the semantic domain is supplied by a physical machine. In other words, the physical machine itself (the ‘implementation’) determines what the program means. For example, in programming languages this is equivalent to the claim that the semantics for the C++ programming language are determined by Bjarne's computer running his C++ compiler. But this explanation is obviously inadequate: If we assume that Bjarne's machine determines the meaning of C++ programs then there is no notion of malfunction or misinterpretation: whatever Bjarne's computer does is, *ipso facto*, the meaning of the program. But surely an electric storm can cause the machine to *go wrong*. But what could we mean by *going wrong*? Presumably, that the (malfunctioning) machine does not embody the *intended* meaning. But, in turn, we seem only to be able to understand this phrase on the basis of some machine-independent characterization of the meaning. And at some level, this has to be given via some independent semantic description. This suggests that the notion of a bare implementation does not offer an adequate notion of semantics. (Compare with: Kripke 1982; Wittgenstein 1953).

3.3 Semantics, Equality and Identity

We concluded our discussion of program equality (§2.1) with a promise to bring semantics into the picture. Every semantic account of a programming language determines a notion of equality for programs, namely, two programs are taken to be equal if they have the same semantic value, i.e.,

$$P = Q \text{ iff } ||P|| = ||Q||$$

where $||P||$ is the semantic value of the program P. So in this sense, every semantic account determines a criterion of equality. For example, one version of denotational semantics would abstract away from all computational steps and equate programs that in some sense compute the same mathematical function. For example, the following two programs would be deemed equal by this criterion:

```
function Factorial(n: integer): integer
begin
    if n = 0
        then Factorial := 1;
    else
        Factorial := (n) * Factorial(n-1);
end;
```

Program 1

```
function Factorial(n: integer): integer
var
    x, y: integer;
begin
    y := 1;
    x := 0;
    while x<n do begin
        x := x+1;
        y := y*x;
    end;
    Factorial := y;
end;
```

Program 2

On the other hand, a more operational view, which makes reference to the steps of the computation, would *not* take Program 1 and Program 2 to be equal. Indeed, in light of §3.1

we can devise semantic accounts that reflect any level of implementation detail. Different semantic accounts determine different notions of equality that can serve different conceptual and practical ends. But then which one should be taken to determine the language? Fortunately, there are some desiderata that can be applied; we can cut down the options: some semantic accounts provide us with a logically satisfactory notion of identity, and some do not.

The **indiscernibility of identicals** is a principle built into ordinary predicate logic. It states that if two objects are equal then they share all properties. The converse principle, the **identity of indiscernibles** states that if, for every property F, object x has F if and only if object y has F, then x is identical to y. The identity of indiscernibles implies that if x and y are distinct, then there is at least one property that x has and y does not. Sometimes the conjunction of both principles is known as Leibniz's Law (Forrest 2006). Leibniz's Law is often taken to be essential for any notion of equality. These laws are usually formulated in logical theories such as second order logic. But we shall be most interested in their ability to discriminate between different kinds of programming language semantics. Indeed, Leibniz' law is one of the central notions of modern semantic theory. The criterion of identity is fleshed out in terms of *observational equivalence*.

Two programs M and N are defined to be **observationally equivalent** if and only if, in all contexts C[…] where C[M] is a valid program, it is the case that C[N] is also a valid program with the same semantic value. For example, we say that Oracle and DB2 (programs manipulating relational databases) are observationally equivalent according to some semantic account of operations on relational databases if and only if executing them in "same" context (operating system, machine architecture, input, etc.) yields the "same" database. Of course, the term *observationally equivalent* needs to be taken with a pinch of salt. We clearly cannot observe the behaviour of a program in all contexts. However, observational equivalence does reflect an underlying conceptual demand that emanates from the principles of indiscernibility of identicals and from the identity of indiscernibles.

In semantics, if all observably distinct programs have distinct semantic values, the semantics is said to be **sound**. Consequently, a sound semantics satisfies the following principle:

$$||P|| = ||Q|| \text{ implies that for all contexts } C, ||C[P]|| = ||C[Q]||$$

It should be clear that the notion of identity induced by a sound semantics satisfies the *indiscernibility of identicals*.

A semantics is said to be **complete** if any two programs with distinct semantic values are observably distinct. More precisely, a complete semantics satisfies the following:

For all contexts C, $||C[P]|| = ||C[Q]||$ implies $||P|| = ||Q||$

Again, it should be evident that a complete semantics satisfies the principle of *identity of indiscernibles*.

Finally, semantics is said to be **fully abstract** if it is sound and complete. Consequently, a fully abstract semantics satisfies Leibniz's Law.

This logical background provides the philosophical justification for the development of fully abstract semantics. It thus offers us a way of selecting semantic accounts that provide philosophically acceptable notions of equality. It does not of course determine any single notion. It only provides a tool for rejecting those that cannot deliver a conceptually acceptable one. Many so-called denotational semantics are not fully abstract, whereas many operational ones are. Indeed, one of the central topics in the recent history of semantics has involved the search for fully abstract definitions that are cast within the class of semantic definitional techniques that are taken to deliver a denotational semantics.

Semantics plays a normative or defining role in computer science. Without semantic definitions, languages and structures have no content over and above that supplied by their syntactic descriptions. And the latter are hardly sufficient for practical or philosophical purposes. While we have made a start on the analysis of the central concerns, we have only scratched the surface.

4. Proofs and Programs

Specifications (§2.3) induce a particular notion of *correctness*. According to the abstract interpretation of this notion, a program is taken to be correct relative to a (functional) specification if the relation it carves out between the input and output satisfies the one laid down by the specification. More precisely, if p is a program, then it meets a specification R, which is taken to be a relation over the input type I and the output type O, if the following holds:

1. For all inputs, i of type I, the pair $(i, p(i))$, satisfies the relation R.

where $p(i)$ is the result of running the program p on the input i. Here R is expressed in some specification language and p in some programming language. The former is usually some variant of (typed) predicate logic and proofs of correctness (i.e. that statement (1) holds) are carried out in the proof system of the logic. For example, Hoare logic (Hoare 1969) is frequently employed, in which Proofs of correctness take the form of inferences between triples, written

$B\{P\}A$

where *P* is a program and *B* and *A* are assertions (the 'before' and 'after' states of the program) expressed in some version of predicate logic with features that facilitate the expression of the values attached to the program variables.

One philosophical controversy that surrounds the issue of correctness centres upon the nature of such proofs; the other challenges what such proofs deliver.

4.1 Proofs in Computer Science

Are proofs of program correctness genuine mathematical proofs, i.e., are such proofs on a par with standard mathematical ones? DeMillo et al. (1979) claim that because correctness proofs are long and mathematically shallow, they are unlike proofs in mathematics that are conceptually interesting, compelling and attract the attention of other mathematicians who want to study and build upon them. For example, a proof in Hoare logic to the effect that Program 2 computes the factorial function would contain details about the underlying notion of state, employ an inductive argument, and involve reasoning about loop invariance.

But such proofs would be much longer than the program itself. Furthermore, the level at which the reasoning is encoded in Hoare logic would require the expression and representation of many details that would normally be left implicit. It would be a tedious and, in the case of most programs, conceptually trivial.

This argument, parallels the graspability arguments made in the philosophy of mathematics (e.g., Tymoczko 1979; Burge 1988). At its heart are epistemological worries: proofs that are too long, cumbersome and uninteresting cannot be the bearers of the kind of certainty that is attributed to standard mathematical proofs. The nature of the knowledge obtained from correctness proofs is claimed to be different to the knowledge that may be gleaned from proofs in mathematics[1].

One also has to distinguish this essentially sociological perspective on proofs from the one that maintains that proofs are right or wrong in a way that is independent of such epistemological judgements. It is possible to hold on to the more realist position according to which any given proof is either correct or incorrect without giving up the demand that proofs, in order to be taken in and validated, need to graspable.

One might attempt to gain some ground by advocating that correctness proofs should be checked by a computer rather than a human. But of course the proof-checker is itself in need of checking. Arkoudas and Bringsjord (2007) argue that if there is only one correctness proof that needs to be checked, namely that of the proof checker itself, then the possibility of mistakes is significantly reduced.

4.2 Proofs in Mathematics

Mathematical proofs such as the proof of Gödel's incompleteness theorem are also long and complicated. But what renders them transparent, interesting and graspable ('surveyable') by the mathematical community is the use of modularity techniques (e.g. lemmas) and the use of *abstraction* in the act of mathematical creation. The introduction of new concepts enables a proof to be constructed gradually, thereby making the proofs more graspable. Mathematics progresses by inventing new mathematical concepts that enable the construction of higher level and more general proofs that would be far more complex and even impossible without them. For example, the exponent notation makes it possible to carry out computation beyond the complexity of multiplication—and argue about the results. At the other extreme, the invention of category theory facilitated the statement and proof of very general results about algebraic structures that automatically apply to a whole range of such. Mathematics is not just about proof; it also involves the abstraction and creation of new concepts and notation. On the face of it, formal correctness proofs do not, in general, employ the creation of new concepts or get involved in the process of mathematical abstraction. In contrast, abstraction in computer science (§6.1) is concentrated in the notions needed for program design. But how are these two notions of abstraction related? We shall say a little more about this later.

4.3 Physical and Abstract Correctness

Even if we put aside these epistemological worries, a second and seemingly more devastating criticism of correctness proofs questions what is actually established by them. Seemingly, a proof of correctness provides correctness only up to the textual representation of the program. No amount of formal work can get us past the abstract/physical barrier: we can never guarantee that any particular execution of the program on a physical machine will actually proceed as expected (Fetzer 1988; Fetzer 1999; Colburn 2004).

But what does it mean for program *p* to be correct? Assume that we have a specification of the program—and that it can be formal or informal. Then, suppose we carry out a series of test runs to verify that the program meets its specification. If they succeed, we have empirical evidence that the *physical* counterpart of the *textual* program is indeed correct because it functions according to the specification. On this view, it is the physical counterpart that was being tested; not the textual program.

This analysis suggests that there is a duality in the notion of correctness of programs. Consistent with the dual nature of programs, we might say that the textual program is subject to mathematical correctness, while its physical counterpart is subject to empirical verification.

5. Computability

Computability is one of the oldest topics that can be labelled as **PCS**. However, it is the subject of several SEP entries (e.g., Barker-Plummer 2004) and so we shall only mention a few topics and their connections with the rest of the present entry.

5.1 The Church-Turing Thesis

One of the central issues is the Church-Turing Thesis. And here there are two disputes, one historical and one empirical. They centre on the following two possible interpretations of the thesis:

I. Turing machines can do anything that could be described as "rule of thumb" or "purely mechanical".
II. Whatever can be calculated by a machine (working on finite data in accordance with a finite program of instructions) is Turing machine-computable.

Interpretation I is intended to capture the notion of an *effective* or *mechanical* method in logic and mathematics. It is meant to reflect the informal notion of algorithm implicit in mathematics and brought to the fore by the Hilbert's program. Interpretation II is meant to govern physical machines. Indeed, (Gandy 1980) can be seen as a further unpacking of II. Gandy proposes four principles that are intended to characterise computation by a physical machine. He shows that such machines exactly agree with Turing's characterisation (Gandy's Theorem). In connection with our discussion of different semantic paradigms, it is clear that many of the machines that underlie denotational semantics (§3.1) do not qualify as Gandy machines. They most often operate with extensional higher order functions spaces, and these cannot be taken to constitute finite data and do not satisfy Gandy's conditions.

Some claim (Copeland 2004; Copeland 2008) that the thesis as proposed by Church and Turing only concerns interpretation I and does not set a limit on machines in general. Hodges (2007) disagrees. He argues that Church and Turing did not distinguish between the two interpretations. This is the historical dispute.

The physical dispute concerns the capabilities of actual machines (interpretation II.) Many take it for granted that the Church-Turing thesis characterises and prescribes actual physical computation. For example, this seems to be the implicit assumption in mainstream computer science. It is certainly the case that every program written in an existing implemented programming language is Turing computable and conversely, that all general-purpose programming languages are Turing complete, i.e., they contain all the control constructs necessary to simulate a universal Turing machine.

Copeland (2007) argues that Gandy's characterisation of a discrete deterministic mechanical device is too narrow, and consequently there are examples of possible physical machines whose capabilities go beyond the class of Turing computable functions. Many of these require *infinite speedup*, whereby an infinite number of computations can be physically carried out in a finite time. Quantum computing has been cited as a possible example for such machines, but this has been disputed (Hodges 2007; Hagar 2007).

Hodges is also concerned with the applicability of standard mathematical argumentation in physics to those cases where infinite precision is involved. This suggests that this dispute is not a simple empirical one. Indeed, there are those who question whether it is physically possible to complete an infinite number of tasks in a finite time. Dummett (2006) questions whether the very notion of an infinite task that is to be carried out in the physical realm is not only a physical impossibility but also a conceptual one. So the dispute is not just an empirical one but goes to the heart of our understanding of the relationship between our mathematical models and physical reality.

6. Programming and Programming Languages

The design of programs and programming languages is one of the traditional activities of computer science. A host of conceptual questions surrounds them (§1), many of which have received no philosophical attention. Here we shall briefly review two of these problems.

6.1 Abstraction

Abstraction is one of the conceptual cornerstones of computer science. It is an integral part of program design and construction, and forms a core methodology for the design of programming languages. Indeed, it drives the creation of new programming paradigms. It underlies the invention of notions such as procedural and functional abstraction, polymorphism, data abstraction, objects and classes, design patterns, architectural styles, subtyping, and inheritance. Many branches of software engineering (e.g. software modelling, program comprehension, program visualization, reverse- and re-engineering) are primarily concerned with the investigation of appropriate mechanisms for program abstraction. And much of the progress of software engineering has been achieved due to the introduction of new abstraction mechanisms.

But what is the nature of abstraction in computer science? What is its underlying philosophical explanation? Unfortunately, in general, the very idea of abstraction is philosophically problematic. According to the traditional view, that has its origins in philosophical psychology, abstraction is a mental process in which new conceptions are formed by considering several objects or ideas and omitting the features that distinguish

them. (Rosen 2001). But, this approach has few, if any, contemporary philosophical advocates.

A more logical approach to the analysis of abstraction, that does have some strong advocacy (Wright 1983; Hale 1987). But it is unclear whether these ideas, which were developed for mathematical abstraction, are applicable to computer science. Clearly, some of the notions of abstraction in computer science were either inspired by or investigated by means of abstractions in mathematics. But what is the conceptual relationship between abstraction in these disciplines? Are they fundamentally different? Unfortunately, while there is substantial literature on the philosophical foundations of mathematical abstraction (see Wright 1983; Hale 1987; Fine 2002), the conceptual investigation of abstraction in computer science is in its infancy. Colburn (2007) suggests that the distinction between abstraction in mathematics and abstraction in computer science lies in the fact that in mathematics abstraction is *information neglect* whereas in computer science it is *information hiding*. That is, abstractions in mathematics ignore what is judged to be irrelevant (e.g. the colour of similar triangles). In contrast, in computer science, any details that are ignored at one level of abstraction (e.g. Java programmers need not worry about the precise location in memory associated with a particular variable) must not be ignored by one of the lower levels (e.g. the virtual machine handles all memory allocations).

But is this based upon too simplistic a notion of abstraction in mathematics? Is there just one kind of notion? For example, the information hiding in Bishop's analysis (Bishop 1970) is quite different to Wright's notion of abstraction—indeed, is quite similar to the computer science one.

6.2 Types and Ontology

Programming languages are mostly typed languages, where the modern notion of type has its origins in Frege and Russell and in particular in Russell's simple theory of types (Irvine 2003). Of course, Russell was motivated by the logical and semantic paradoxes and this feature is not central to the application of types to computer science. On the other hand, for Russell types carve up the universe of discourse in ways that have both grammatical and semantic import. And this idea has carried over to computer science. Indeed, Type theories have been inspired and enriched by computer science. For example, Russell's theory of types, although mathematically powerful, is somewhat impoverished in its expressive power compared with the type theories of modern computer languages (Coquand 2006; Pierce 2002). Apart from a range of basic types such as numbers and Booleans, programming languages contain a collection of type constructors (ways of building new types from old ones). For example, these include the ability to form some kind of Cartesian product and finite sets. In many object-oriented programming languages, types (classes) can import (and override) operations from other types and offer more sophisticated

constructors that support the formation of abstract data types and various forms of polymorphism.

In computer science, types play a role that is half way between syntax and semantics. Firstly, they extend our normal notion of context free grammar. Some language features, especially those that allow the type of a variable to be fixed by context (i.e., declarations of the language) require a form of grammar that is more flexible than the standard one. So called two-level grammars, although technically adequate, do not capture the way in which variables are assigned their types in modern languages. And they are very clumsy to use. Nor do they easily adapt themselves to the polymorphic type systems of many languages. Modern type systems do better: variables are assigned their types via declarations, e.g., x:Boolean. Subsequently, a compiler can type-check the program, e.g., it can ensure that the occurrence of a variable in subsequent statements (e.g. $x<y$) is consistent with its declaration. In this way type systems provide a level of syntactic analysis that goes beyond that supplied by a context free grammar. But this still seems like a grammatical role.

But types also play a correctness role that would normally not be described in syntactic terms. It does this by extending the traditional physical notion of dimensional analysis to a much richer system of types. Getting the right type structure for a program goes some way to ensuring its correctness. And this is determined by the structure that types impose on a language. Types fix the kinds of things there are in a programming language. So for example, any programming language that admits numbers, products and classes, and nothing else, imposes a conceptual framework on the programmer that she must work within. Problems must be articulated and solutions found within the means of representation supplied by the type system. Once the type structure of a programming language has been set out, much of its ontological setting has been fixed.

Or has it? Perhaps we need to step back and first ask how the ontological commitments of a language are to be determined. Is it the semantics that determines matters (Turner & Eden 2007)? A long tradition that emanates from Frege would suggest so (Dummett 1991). Assuming the analogy with natural languages is legitimate, the ontology of a language is determined by the structures required to furnish its semantic domains. But which semantic theory should we adopt? While any semantics must take types into account, the semantically determined ontology would go beyond them and reflect the semantic domains involved, and these would reflect the implementation detail that is included in the semantics. On this account, as with equality, different semantic interpretations determine different ontologies. It follows that it is not sensible to talk about *the* ontology of a programming language but about multiple ontologies that depend upon the level of abstraction involved in the semantics. For example, the ontology may also be partially determined by the programming paradigm.

7. Legal and Ethical Questions

Some issues in computer ethics belong to the **PCS** in that the activity of building and using software raises ethical questions. However, many are not specific to computer science in the narrow sense of this entry; they impinge upon the whole of information technology and computer applications (Bynum 2001). Consequently, we shall only mention two that seem central to computer science.

7.1 Copyrights, Patents and Identity

Copyrights provide some protection for software, but they are unable to protect its semantic core. And we take it that the latter is to be determined by a semantic account (§3) of the programming language in which the program is written. Presumably, the essence of this issue concerns the problem of program identity (§3.3). But if there are many possible semantic notions of identity, which one is appropriate for legal application?

One informal semantic account that is often cited in law identifies the program with the *ideas* expressed therein, most often taken to be its underlying algorithm. But not only is it often hard to say exactly what this algorithm is, but also, as with mathematical theorems, algorithms cannot be copyrighted. And much the same fate awaits any formal semantic account since any such would be determined by some mathematical notion, be it algorithms or some notion of operation or mathematical function.

But even if we could locate a semantic account that would pass muster with the copyright laws, the legal picture would not be complete. Copyright infringement often hinges not just on some account of identity but on whether it is plausible to suppose that someone would come up with the very same program. So that intentional considerations enter the frame. In other words, even where two programs are taken to be equivalent according to our semantic criterion, if it could be seen as plausible that they were constructed independently, there would be no copyright infringement.

Patents (specifically utility patents) fare no better. They are even harder to get for software since one cannot patent mental processes, abstract ideas and algorithms. And it is algorithms rather than the source code that often contains the new ideas. But once again, it is the identification and identity issues that are the central philosophical concern. And these involve both semantic and intentional considerations.

7.2 Correctness and Responsibility

Is it right that software is sold with little guarantee of fitness for purpose? (Coleman 2008) is devoted to this question. This is an especially pertinent question for safety-critical

systems, e.g., systems that monitor medical conditions, operate nuclear power plants and communicate with space shuttles. Here enforcing more rigorous testing and proofs of correctness would seem essential. But in ethical terms, is the case of a programmer who fails to analyse and test his program any different to that of a civil engineer who fails to carry out the required mathematical modelling and tests on a building construction? The moral obligations seem similar.

One way in which they might be different pertains to the complexity of software (Brooks 1987) which exceeds the complexity of any other kind of human artefact by orders of magnitude. Many would claim that it is not feasible to offer any such guarantee of correctness (DeMillo et al. 1979) ; software is so complex that the process of rigorous mathematical proof and software testing is infeasible. And, presumably, one can only have a (moral or legal) obligation to carry out a feasible process.

But how does one balance the proving and testing aspects of software development against the intended use of the software? Should software developed for entertainment be subject to the same degree of rigorous proof and testing as software that is safety critical? Presumably not, but we might still be tempted to ask, are these new ethical problems or do they just furnish us with further case studies of existing ethical dilemmas? For example, even security glitches in software used in the entertainment industry can carry financial penalties.

8. New Twists or New Issues?

Even this rather brief overview of **PCS** should convince the reader that computer science raises interesting and demanding philosophical issues. Indeed, one of the overriding impressions is that it has substantial links with most of the traditional branches of philosophy. There are clear connections with ontology, ethics, epistemology, and the philosophies of mathematics, physics and language. Indeed, our initial list of questions raises many more themes that connect with other areas of philosophy. In particular, there is a substantial literature on the applications of computer science. Artificial intelligence and cognitive science yield philosophical questions that belong to the philosophy of mind (McLaughlin 2004). Of course, much of this emanates from Turing (1950). Other applications of computer science to traditional areas of science, so called computational science, create issues for the philosophy of science: what is the epistemological impact of computer simulations, especially where these are the only viable form of experimentation? The computational turn in ontology brings new techniques to bear upon the structure of any kind of conceptual ontology. The philosophy of logic is enriched by a mass of material: large numbers of new logical systems have emerged for the purposes of representation and reasoning about computational systems.

While it is clear that computer science raises many significant twists to traditional philosophical concerns, what is less clear is whether it generates any genuinely new philosophical concerns: are there questions in **PCS** that have no parallel in any other branch of philosophy?

Bibliography

- Allison, A., Currall, J., Moss, M. and Stuart, S., 2005, "Digital identity matters", *Journal of American Society Information Science and Technology* 56(4): 364–372.
- Arkoudas, K. and Bringsjord, S., 2007, "Computers, Justification, and Mathematical Knowledge", *Minds and Machines* 17(2): 185–202.
- Barendregt, H.P., 1993, "Lambda calculi with types", in: *Handbook of logic in computer science*, Vol. 2, New York, NY: Oxford University Press Inc.
- Barker-Plummer, D., 2008, "Turing Machines", *The Stanford Encyclopedia of Philosophy* (Fall 2008 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/fall2008/entries/turing-machine/>.
- Bishop, Errett, 1977, *Foundations of constructive analysis*, McGraw-Hill.
- Blass, Andreas and Gurevich, Yuri, 2003, "Algorithms: A Quest for Absolute Definitions", *Bulletin of the European Association for Theoretical Computer Science (EATCS)* No.81: 195-225.
- Bowen, J.P. and Hinchey, M.G., 1995, "Ten Commandments of Formal Methods", *IEEE Computer* 28(4): 56–63.
- Bowen, J.P. and Hinchey, M.G., 2005, "Ten Commandments of Formal Methods: Ten Years Later", *IEEE Computer* 39(1): 40–48.
- Brooks, F. P., 1987, "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer* 20(4): 10-19.
- Burge, T., 1998, "Computer Proof, A Priori Knowledge, and Other Minds", *Philosophical Perspectives* 12: 1–37.
- Bynum, T., 2001, "Computer Ethics: Basic Concepts and Historical Overview", *The Stanford Encyclopaedia of Philosophy* (Winter 2001 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/win2001/entries/ethics-computer/>
- Colburn, T., 2004, "Methodology of Computer Science", *The Blackwell Guide to the Philosophy of Computing and Information*, Luciano Floridi (ed.), Malden: Blackwell, pp. 318–326.
- Colburn, T., and Shute, G., 2007, "Abstraction in Computer Science", *Minds and Machines* 17(2): 169–184.
- Coleman, K.G., 2008, "Computing and Moral Responsibility", *The Stanford Encyclopedia of Philosophy* (Fall 2008 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/fall2008/entries/computing-responsibility/>.
- Copeland, B. Jack, 2008, "The Church-Turing Thesis", *The Stanford Encyclopedia of*

Philosophy (Fall 2008 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/fall2008/entries/church-turing/>.

- Copeland, B. Jack, 2004, "Computation", *The Blackwell Guide to the Philosophy of Computing and Information*, Luciano Floridi (ed.), Malden: Blackwell, pp. 3–17.
- Coquand, Thierry, 2006, "Type Theory", *The Stanford Encyclopedia of Philosophy* (Winter 2006 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/win2006/entries/type-theory/>.
- DeMillo, R.A., Lipton, R.J. and Perlis, A.J., 1979, "Social Processes and Proofs of Theorems and Programs", *Communications of the ACM* 22(5): 271–280.
- Denning, P.J., 1980, "On Folk Theorems and Folk Myths", *Communications of the ACM* 23(9): 493–494.
- Denning, P.J., 1980b, "What is Experimental Computer Science?" *Communications of the ACM* 23(10): 534–544.
- Denning, P.J., 1981, "Performance Analysis: Experimental Computer Science as its Best", *Communications of the ACM* 24(11): 725–727.
- Denning, P.J., 1985, "The Science of Computing: What is computer science?" *American Scientist* 73(1): 16–19.
- Denning, P.J. (ed.), et al., 1989, "Computing as a Discipline", *Communications of the ACM* 32(1): 9–23.
- Dijkstra, E., 1968. "Go To Statement Considered Harmful", *Communications of the ACM* 11(3): 147–148.
- Dummett, M., 1991, "The Logical Basis of Metaphysics", Harvard University Press.
- Dummett, M., 2006, "Thought and Reality", Oxford University Press.
- Eden, Amnon, 2007, "Three Paradigms in Computer Science", *Minds and Machines* 17(2): 135–167.
- Feferman, S., 1992, "Logics for termination and correctness of functional programs", *Logic for Computer Science*: 95–127, MSRI Pubs. vol. 21, New York, NY: Springer-Verlag.
- Fetzer, J.H., 1988, "Program Verification: The Very Idea", *Communications of the ACM* 31(9): 1048–1063.
- Fetzer, J.H., 1999, "The Role of Models in Computer Science", *The Monist* 82(1): 20–36.
- Fine, K., 2008, "The Limits of Abstraction". Oxford: Oxford University Press.
- Floridi, Luciano, 2004. "Information", *The Blackwell Guide to the Philosophy of Computing and Information*, Luciano Floridi (ed.), Malden: Blackwell, pp. 40–62.
- Floridi, Luciano 2007, "Semantic Conceptions of Information", *The Stanford Encyclopedia of Philosophy* (Spring 2007 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/spr2007/entries/information-semantic/>.
- Forrest, P., 2006, "The Identity of Indiscernibles", *The Stanford Encyclopedia of Philosophy* (Fall 2008 Edition), Edward N. Zalta (ed.), forthcoming URL =

<http://plato.stanford.edu/archives/fall2008/entries/identity-indiscernible/>. >.
- Fuchs, N.E., 1992, "Specifications Are (Preferably) Executable". *Software Engineering Journal* 7(5): 323–334.
- Gandy, R., 1980, "Church's thesis and principles for mechanisms", *The Kleene symposium*, Barwise, J., Keisler, H. J. and Kunen, K. (eds.), Amsterdam: North-Holland.
- Hagar, Amit, 2007, "Quantum Algorithms: Philosophical Lessons", *Minds and Machines* 17(2): 233–247.
- Hale, B. and Wright, C., 2001, "The Reason's Proper Study: Essays towards Neo-Fregean Philosophy of Mathematics", *Oxford Scholarships on Line*, Oxford: Oxford University Press.
- Hartmanis, J., 1993, "Some Observations about the Nature of Computer Science", *Lecture Notes in Computer Science* 761, Shyamasundar, R.K. (ed.): 1–12.
- Hartmanis, J., 1994, "Turing Award Lecture: On Computational Complexity and the Nature of Computer Science", *Communications of the ACM* 37(10): 37–43.
- Hoare, C.A.R., 1969, "An axiomatic basis for computer programming". *Communications of the ACM* 12(10):576–585. [Reprint available online]
- Hodges, A., 2006, "Did Church and Turing have a thesis about machines?", *Church's Thesis after 70 years* Olszewski, Adam (ed.)
- Hodges, A., 2007, "Can quantum computing solve classically unsolvable problems?"
- Horsten, L., 2008, "Philosophy of Mathematics", *The Stanford Encyclopedia of Philosophy* (Fall 2008 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/fall2008/entries/philosophy-mathematics/>.
- Immerman, N., 2006, "Computability and Complexity", *The Stanford Encyclopedia of Philosophy* (Fall 2006 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/fall2006/entries/computability/ >.
- Irvine, A.D., 2003, "Russell's Paradox", *The Stanford Encyclopedia of Philosophy* (Fall 2006 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/fall2006/entries/russell-paradox/>
- Jones, C.B. and Hayes, I.J., 1990, "Specifications Are Not (necessarily) Harmful", *Software Engineering Journal* 4(6): 330–339.
- Krishnamurthi, S., 2003. *Programming Languages: Application and Interpretation*,
- Kreisel, G., Gandy, R. O., 1975, "Some Reasons for Generalizing Recursion Theory." *The Journal of Symbolic Logic* 40(2): 230–232.
- Kripke, S., 1982, *Wittgenstein on Rules and Private Language*. Harvard University Press.
- Kuhn, T.S., 1970, *The Structure of Scientific Revolutions*, 2nd. ed., Chicago: Univ. of Chicago Press.
- Landin, P.J., 1964, "The mechanical evaluation of expressions", *Computer Journal* 6(4): 308–320.

- Milne, R. and Strachey, C., 1977, *A Theory of Programming Language Semantics*, New York, NY: Halsted Press.
- McLaughlin, B., 2004, "Computationalism, Connectionism, and the Philosophy of Mind", *The Blackwell Guide to Philosophy of Computing and Information*, Floridi, Luciano (ed.) Malden: Blackwell, pp. 135–152.
- Minsky, M., 1970, "ACM Turing Lecture: Form and Content in Computer Science", *Journal of the Association for Computing Machinery* 17(2): 197–215.
- Moor, J.H., 1978, "Three Myths of Computer Science", *The British Journal for the Philosophy of Science* 29(3): 213–222.
- Moschovakis, Y.N., 1998, "On founding the theory of algorithms", *Truth in Mathematics* Dales, Harold G. and Oliveri, Gianluigi (eds.), Oxford: Oxford University Press.
- Pierce, Benjamin C., 2002, *Types and Programming Languages*, Cambridge, MA: MIT Press.
- Plotkin, G.D., 1981, "*A Structural Approach to Operational Semantics"*, Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark.
- Rapaport, W.J., 2005a, "Philosophy of Computer Science: An Introductory Course", *Teaching Philosophy* 28(4): 319–341.
- Rapaport, W.J., 2005b, "Implementation is Semantic Interpretation: Further Thoughts." *Journal of Experimental and Theoretical Artificial Intelligence* 17(4): 385–417.
- Rosen, Gideon, 2001. "Abstract Objects", *The Stanford Encyclopedia of Philosophy* (Fall 2001 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/fall2001/entries/abstract-objects/>.
- Shapiro, S., 1997, *Philosophy of Mathematics: Structure and Ontology*, Oxford: Oxford University Press.
- Sieg, Wilfried, 2008, "Church without Dogma: Axioms for Computability", *New Computational Paradigms*, Lowe, B., Sorbi, A. and Cooper, B. (eds.), Springer-Verlag, 139–152.
- Smith, B.C., 1996, "Limits of Correctness in Computers", *Computerization and Controversy*, Kling, R. (ed.), Morgan Kaufman, pp. 810–825.
- Szabó, Z.G., 2007, "Compositionality", *The Stanford Encyclopedia of Philosophy* (Spring 2007 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/spr2007/entries/compositionality/>.
- Thomason, R., 2005, "Logic and Artificial Intelligence", *The Stanford Encyclopedia of Philosophy* (Summer 2005 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/sum2005/entries/logic-ai/>.
- Turner, Raymond and Eden, Amnon H., 2007, "Towards Programming Language Ontology", *Computation, Information, Cognition—The Nexus and the Liminal*,

Dodig-Crnkovic, Gordana and Stuart, Susan (eds.), Cambridge, UK: Cambridge Scholars Press, pp. 147–159.
- Turner, Raymond, 2005, "The Foundations of Specification", *Journal of Logic Computation* 15: 623–662.
- Turner, Raymond, 2007, "Understanding Programming Languages". *Minds and Machines* 17(2): 129-133
- Tymoczko, T., 1979, "The Four-Color Problem and Its Philosophical Significance", *Journal of Philosophy* 76(2): 57–83.
- White, G., 2004, "The Philosophy of Computer Languages", *The Blackwell Guide to the Philosophy of Computing and Information*, Floridi, Luciano (ed.), Malden: Blackwell, pp. 318–326.
- Wing, J.M., 2006, "Computational Thinking", *Communications of the ACM*, 49(3): 33–35.
- Wittgenstein, L., 1953. *Philosophical Investigations*. Blackwell Publishing.
- Wright, Crispin, 1983, *Frege's Conception of Numbers as Objects*, Aberdeen University Press.

Other Internet Resources

- Philosophy of Computer Science, at Essex University
- International Association for Computing and Philosophy

Related Entries

artificial intelligence: logic and | Church-Turing Thesis | computability and complexity | computer and information ethics | mind: computational theory of