Computer Science as Immaterial Formal Logic

Selmer Bringsjord

Philosophy & Technology

ISSN 2210-5433
Volume 33
Number 2

Philos. Technol. (2020) 33:339-347
DOI 10.1007/s13347-019-00366-7

Springer

Philosophy & Technology (2020) 33:339–347
https://doi.org/10.1007/s13347-019-00366-7

COMMENTARY

Computer Science as Immaterial Formal Logic

Selmer Bringsjord[1,2,3,4]

Received: 25 June 2019 / Accepted: 8 July 2019 / Published online: 5 August 2019

Abstract

I critically review Raymond Turner's *Computational Artifacts – Towards a Philosophy of Computer Science* by placing beside his position a rather different one, according to which computer science is a branch of, and is therefore subsumed by, immaterial formal logic.

Keywords Formal logic · Logic machines · Computer science

Consider the elementary natural-deduction[1] inference schema *universal elimination*, customarily denoted by such text as the following:

$$\frac{\forall x \phi(x)}{\phi\left(\frac{a}{x}\right)} I_1^{\forall E}$$

This schema often appears in particular specifications of first-order logic (= $\mathscr{L}_1$), as most readers will well know.[2] I have just used text to denote something that is immaterial. You cannot lift the schema in question; nor can you weigh it, or destroy it, or spoon it into a box and hand that box as a gift to someone not fortunate enough

[1] Crisply used by Turner himself in the presentation of his logic TPL (§12.2).

[2] Turner will readily agree that $I_1^{\forall E}$ and the larger specification of which it is usually a part when $\mathscr{L}_1$ is presented makes use of formal languages, which are a crucial part of the seminal philosophy of computer science he gives and defends in (Turner 2018) (see in particular §7.2 therein). Details regarding the particular languages I use in this review are out of scope due to space constraints.

✉ Selmer Bringsjord
Selmer.Bringsjord@gmail.com

1 Rensselaer AI & Reasoning (RAIR) Laboratory, Rensselaer Polytechnic Institute (RPI), Troy, NY 12180, USA

2 Department of Cognitive Science, Rensselaer Polytechnic Institute (RPI), Troy, NY 12180, USA

3 Department of Computer Science, Rensselaer Polytechnic Institute (RPI), Troy, NY 12180, USA

4 Lally School of Management, Rensselaer Polytechnic Institute (RPI), Troy, NY 12180, USA

to be familiar with any of the proof theories $\mathcal{I}_1^1, \mathcal{I}_1^2, \mathcal{I}_1^3, \ldots$ for $\mathscr{L}_1$.[3] In general, as Ross (1992) explains, inference schemata, including ones with which all philosophers will be familiar (e.g., hypothetical syllogism[4]), are non-physical. The realm of the immaterial contains as well other categories of abstract objects that compose those things that, by the lights of both Turner and my own, are at the heart of what computer science is, and therefore at the heart of what genuine computer scientists do. For example, ϕ in $I_1^{\forall E}$ must be a well-formed formula, which means that it must be in accord with some formal grammar G_1 that includes some alphabet A_1. The pair $\langle G_1, A_1 \rangle$ is the formal language $\mathcal{L}_1$ of $\mathscr{L}_1$, and this language inherits immateriality from the two immaterial things that compose it.

Now, the immaterial nature of the familiar things I have just referred to stands in stark contrast to Turner's conception of the nature of those things that everyone agrees are (at least in name) central to computer science. This is easy to confirm. After all, a spoon is an artifact; so is a box; and both are as such quite material/physical. Turner happily embraces these facts early on (see, e.g., p. 25). In fact, one of the refreshing aspects of his book is that he is crystal clear at its outset as to how his philosophy of computer science will in general be erected:

> Technical artifacts are taken to include all the common objects of everyday life, such as chairs, televisions, paper clips, telephones, smartphones and dog collars. They are **material objects**, the engineered things of the world that have been intentionally produced by humans in order to fulfill a practical function. In this regard, they differ from naturally occurring entities such as stars and rivers. A central part of the analytic philosophy of technology aims to understand their nature, design, and construction, and **it is from this perspective that we approach the artifacts of computer science**. (Turner 2018, p. 25; emphasis by bold text mine)

Alas, I doubt very much that there *are* any artifacts of computer science. The reason is that the core elements of computer science are logicist, and as such are immaterial. As to computer *engineering*, well, yes, that might be a rather different story, but it is one we ought to ignore: we are discussing not philosophy of computer engineering, but of computer *science*. Perhaps there is no small irony in the biographical fact that Turner himself, theoretical computer scientist that he is, has made precious little use of, say, electrical engineering, in the creation of his truly impressive oeuvre.[5] It is interesting along this line to further note that Turner is the co-author

[3] In addition to natural deduction, which is seldom used in automated (deductive) reasoning (but see the seminal Pollock 1995), we e.g. have proof theories based on resolution (which is at the heart of Prolog). For classic coverage of resolution for automated reasoning, see (Wos et al. 1992).

[4] Formalized as follows, and, note, part of both zero-order ($\mathscr{L}_0$) and first-order ($\mathscr{L}_1$) logic:

$$\frac{\phi \rightarrow \psi; \psi \rightarrow \gamma}{\phi \rightarrow \gamma} I_{0/1}^{HS}$$

[5] I suspect that Turner would say that he has in fact been occupied at times with what he calls *program artifacts*, defined in Chap. 5, p. 52. Unsurprisingly, I see program artifacts as having one foot in computer science, and another in computer engineering.

of an authoritative overview of philosophy of computer science (Turner and Angius 2017), in which the computer-science-as-mathematics view is presented (§9.1).[6]

Someone might object to what I have so far said, and the line I have started and am obviously on, in the following way:

Objection 1

"What you're saying is really a bit silly, to be frank. You are a fan of what might be called a platonic conception of formal logic. So what? In this regard your views are quite beside the point, for the simple reason that formal logic is formal logic; it's not computer science. The former comes in quite handy, and is indeed in Turner's philosophy of computer science crucial, but just because a spoon comes in handy when eating doesn't mean that designing, making, and using flatware *is* eating."

Making flatware is not eating: quite right. But the analogical inference underlying the objection is invalid in the present case, for the simple reason that, once you have in hand a suitably rigorous and robust conception of what a *logic machine* is, it is easy to see that the computer-science-as-immaterial-formal-logic view holds, or at least makes a great deal of sense, and can be worked out in detail, comprehensively, to include for instance a total subsumption of computer programs and programming.[7]

[6]This view is not to be identified with the view I adumbrate herein, but my view is clearly in the same *spirit* as the view Turner encapsulates in his overview. (My view can also be viewed as a broadening and abstraction of what Turner (in his Chap. 1 of *Computational Artifacts*) calls "the Dijkstra-Hoare core.") One thing that prohibits identification is the fact that while my view of what computer science is falls *generally* in line with the computer-science-as-mathematics position as described by Turner, I am a thoroughgoing immaterialist (some might say a "Platonist," but since Plato had no idea what, say, an infinite ordinal such as even ω is, the label would surely be inaccurate), and it is not clear that those Turner classifies under computer-science-as-mathematics (e.g., Dijkstra) are. For a nice treatment of the main ways of viewing what computer science fundamentally is, see a paper Turner himself draws from: (Eden 2007). It may be, by the way, that different views of computer science can be developed in terms of the relevant notion of *information* at the various levels of abstraction that computer science as a discipline requires (see Primiero 2016).

[7]A variant of Objection 1, pointed out to me by Henri Salha (I confessedly express the objection with a bit of a harder edge than what Henri wrote to me in personal communication), is as follows:

Objection 1′

"Your logicist view of computer science is U.S./U.K.-based, alas. In Continental Europe, e.g. France, *informatique théorique* is often reserved to denote the scientific, theoretical (and small) part of all human activities related to — in the most general sense — computers. In the case of just *informatique*, what is referred to is the full set of such activities. Given this context, you simply disagree with Turner only on the scope of what we should count as computer science."

In reply, yes, I certainly do disagree with Turner (and apparently with the U.S./U.K. computing establishment) about the scope of what should count as true computer science. I am happy to admit that on the Continent, and perhaps elsewhere, my *contra*-Turner position might not have full traction—but I confess that I doubt that is the case. One reason is that I doubt *informatique théorique* is all and only thoroughgoingly logicist in e.g. France. (Theoretical computer science (as it is called) is incidentally not purely logicist in the U.S./U.K.) I also doubt that the generic *informatique* space includes substantial application-oriented activity that is logicist. Of course, I may be wrong: I am a lifelong New Yorker. But of this I am sure: The teaching of computer programming, especially in the US (where "computational thinking" is ill-advisedly taken to consist in thinking procedurally) K–12 system, is pretty much nothing more than to teach procedural programming, with a dash of object-oriented spice. This is e.g. revealed in (Rapaport 2019), recently reviewed in (Bringsjord 2018b).

Turner himself provides a chapter on what he calls "logic machines" (Ch. 4). But a fatal problem rears up in his title for this chapter: "Logic Machines as Technical Artifacts." Since, as we have noted, a technical artifact is a physical thing (despite the fact that during its design by a mind all sorts of seemingly non-physical factors arise, such as beliefs by the designer about the mental states of those who will use the artifact in question), Turner's logic machines certainly are not mine. Rather, they are the standard devices spread now across our planet, made of digital circuits. There is logic here only in the sense that Boolean and arithmetic operations are performed. If one builds a view of what computer science is atop logic machines in this sense, then one will be forever imprisoned, because everything that then comes later must do nothing more than drive the circuits in question, and not only that, but what is driven is physical, and not in the realm of thought.[8]

What then *is* a logic machine, in the computer-science-as-immaterial-formal-logic (= c-s-a-i-f-l) view? Historically considered, this question is easy to start to answer, informally: a logic machine is the sort of thing pointed to by Gardner (1958): viz., machines that automatically reason in accordance with some associated collection of inference schemata in some associated logic/s, in order to answer queries. Gardner's (1958) final chapter is especially helpful in providing a bridge from the anemic physical logical machines covered early his book to a future in which automated reasoning is a reality—a future the reader is lucky enough to be living in.

Of course, one can certainly seek to physicalize logic machines, and doing so, as Gardner recounts, is exactly what many of those working on logic machines did—but nonetheless, the machines themselves are immaterial. By leveraging some of the formal content introduced in the foregoing, we can quickly denote some of these logic machines. For instance, the logic machine $\mathfrak{M}_1$ is the quintuple

$$\langle \mathbb{P}^*, q^*, \mathscr{L}_1, \mathbb{R}, \mathbb{C} \rangle.$$

The third element, $\mathscr{L}_1$, is familiar from above, and comprises the aforementioned formal language $\mathcal{L}_1$, as well as another ingredient introduced above: viz. $\mathcal{I}_1^i$, a collection of inference schemata. As to the new elements in $\mathfrak{M}_1$,

– $\mathbb{P}^*$ is the space of allowable programs ($\mathbb{P}$, with suitable sub/superscripts, denotes a particular one);

[8]This may be a good place to bring to the reader's attention that De Mol and Primiero (2015) provide a treatment of links between logic and the history/philosophy of computer science—but the role they see for logic is Turneresque. They e.g. write:

> ... logic and technology work together, from the lowest hardware level, governed by Boolean circuits and arithmetical operations in the stack memory; through the structure of assignment, sequencing, branching and iteration operations defining modern high-level programming languages; up to the equivalent abstract formulations of recursive definitions for algorithms. (De Mol & Primiero 2015, p. 196)

Following on this comes a lucid treatment of some of the roles of logic in computer science, but by now the reader will be able to predict that since my view, distilled, is that computer science is a branch of logic, and hence the role of logic is—put starkly—that not of contributor but rather ruler, De Mol and Primiero's essay is, like Turner's, ultimately at odds with my position, at least to a significant degree.

- q^* is the space of allowable queries (q, with suitable sub-/superscripts, denotes a particular one);
- $\mathbb{R}$ is an automated reasoner; and
- $\mathbb{C}$ is the checker of the reasoning found and provided by $\mathbb{R}$, and by other automated reasoners in other logic machines.

The automated reasoner's job is to provide answers to queries (i.e., to q), and to discover proofs or arguments that justify these answers. Further explanation of $\mathbb{R}$ is beyond the scope of the present critical review.[9] The checker's job (one that in my experience is undervalued in computer science and AI) unsurprisingly, is to either certify or reject the proofs/arguments supplied by $\mathbb{R}$. (For a foundational treatment of the distinction between proof discovery and proof checking, see (Arkoudas and Bringsjord 2007).) What about the program, $\mathbb{P}$? A program here is simply a set of declarative statements expressed as formulae in the language $\mathcal{L}_1$. This entails that $\mathfrak{M}_1$ subsumes and is a superset of standard logic programming in terms of Horn-clause logic, which is nicely summarized by Turner in §8.3 "Logical Languages." Here, he accurately reports that the core principles of standard logic programming are the following, and I quote: "Programs are collections of logical assertions, and computation is inference" (p. 74). (These principles hold as well for all logic machines as I define them, even for those logic machines that eschew deduction (in favor of non-deductive reasoning).) Turner then goes on to write:

> Prolog is the main language to emerge from this paradigm. Strangely, this paradigm has generated fewer languages than have the other paradigms. Nor has it [= the paradigm] contributed much to type theory. However, it does attempt to raise programming to the level of specification. (Turner 2018, p. 74)

What is said here is understandable, given Turner's experience and type-theoretic orientation, but note that every logic machine $\mathfrak{M}$ corresponds to some programming language, obviously. (The language $\mathcal{L}_{\mathfrak{M}}$ is just in the third element in the tuple that is $\mathfrak{M}$, and recall that every program $\mathbb{P} \in \mathbb{P}^*$ is simply a collection of wffs in this language). Since there are clearly more than $\aleph_0$ logic machines, we are talking about rather a lot of programming languages. The overall programming paradigm comprised by the availability of all these programming languages is **pure general logic programming**, or just PGLP for short. PGLP was first publicly introduced in (Bringsjord 2018a).

Turner might retort that this is an artificially inflated count, because he is talking of "in-use" programming languages. But the c-s-a-i-f-l view is replete with multiple in-use programming languages under the fold of automated reasoning. For example, such languages are concretely operative for $\mathscr{L}_0$ and fragments thereof (e.g., the

[9] For an impressive implementation of $\mathbb{R}$, the reader can obtain and study Govindarajulu's (2016) ShadowProver.

propositional calculus), for $\mathscr{L}_1$ (and for fragments thereof[10]), for $\mathscr{L}_2$, for quantified modal logics (see , e.g., Govindarajulu and Bringsjord 2017), and for higher-order logics (e.g. see Benzmüller and Paleo 2014). I do not have an exact count, but once automated reasoning is allowed to be the basis of computing, and programming languages for this basis bloom as the mode and type of automated reasoning varies, a plethora of programming languages is clearly immediately with us. Even the count of in-use languages of the relevant sort seems to exceed, for instance, those in both the imperative and functional folds.

Please allow me to point out that logic machines connect in some ways directly to computability theory. For example, we have the following:

Theorem 1: Let f be a μ-recursive function (from $\mathbb{N}$ to $\mathbb{N}$). Some Turing machine M can compute f if and only if $\mathfrak{M}_1$ can compute f.

Theorem 1′: Let f be a μ-recursive function (from $\mathbb{N}$ to $\mathbb{N}$). Some Prolog program P can compute f if and only if $\mathfrak{M}_1$ can compute f.

Of course, we have here a family of theorems, each member of which refers to some Turing-level manner of computing the functions in question. However, the flip side of the coin is that the vast majority of logic machines exceed the power of Turing machines.[11] To mention just the first step in the separation between Turing machines and logic machines, simply note that Turing machines (and their equivalents) max out at Δ_1 in the Arithmetic Hierarchy, while $\mathfrak{M}_1$ fully handles Σ_1. Readers unfamiliar with such matters can nonetheless see clearly how the separation works, as follows.

A Turing machine M can be fully captured by a corresponding finite set Φ_M of formulae in $\mathcal{L}_1$; this is well-known and easily understood from reductions of the *Entscheidungsproblem* to the Halting Problem for standard Turing machines (see, e.g., Boolos et al. 2003). We can accordingly show that M halts after having started work on some input i if a formula ϕ_i that expresses the placement of i on M's tape initially, conjoined with Φ_M, entails by some standard proof theory $\mathcal{I}_1^i$ that a formula h expressing halting holds. Since the query

$$\Phi_M \cup \{\phi_i\} \vdash h?$$

can be answered by logic machine $\mathfrak{M}_1$, we've moved from mere semi-decidability to full decidability.[12] Once the separation between Turing-level machines and logic machines is thus established, a formal paradise quickly appears, in the form of the

[10]One longstanding class of "fragment" logic machines (not, I confess, recognized as such until now; they fell under Gardner's (1958) naïve sense of *logic machine*) corresponds to syllogistic reasoning. For example, a member of the class was physicalized by Marquand (1885). Formally, with more space available, we could specify and consider the logic machine $\mathfrak{M}_1^{Org}$, which computes Aristotle's tiny fragment $\mathscr{L}_1^{Org}$ of $\mathscr{L}_1$, set out in his *Organon*. The machine I have in mind has an inferential theory that includes from above $I_1^{\forall E}$ and $I_{0/1}^{HS}$, and $I_1^{\forall I}$ (universal introduction), which, e.g., suffice for this machine's automated reasoner to answer that Yes the syllogism Barbara is valid, and supply a confirming proof. See (Smith 2017) for an overview of Aristotle's logic, and turn to (McKeon 1941) to study the primary source.

[11]Logic machines are obviously in most cases Delphic, and relate therefore analogically to the use of oracles in charting relative computability.

[12]Of course, how one might go about building the automated reasoner for $\mathfrak{M}_1$ is a separate, and challenging, question.

logic-machines hierarchy $\mathfrak{LM}$ (out of scope here). To mention just one example, we have the logic machine $\mathfrak{M}_{\omega_1\omega}$, which corresponds to the "small" and "well-behaved" infinitary logic $\mathscr{L}_{\omega_1\omega}$, which allows countably infinite disjunctions and conjunctions, but only finite quantification.[13]

Shifting now to program correctness, Turner's discussion of the topic (Chap. 25) is efficient, and his concern for such correctness commendable, but the c-s-a-i-f-l paradigm quickly provides everything he needs, in short order, and should allay his fears. As mentioned above, in Turner's conception of computer science, there is on the one hand a computer program, and on the other the program's specification, against which proofs of correctness are to be sought. But when one sees computer science as the invention, crafting, and use of logic machines, computer programs *are* specifications. Furthermore, the specific worries Turner has evaporate. For example, consider the worries he expresses here regarding program verification when outside the paradigm I champion:

> When theorem-provers are employed [for program verification], correctness proofs are derived by representing the programs as axiomatic theories, and their specifications are deduced as consequences of those theories. More sophisticated systems employ model checking, where a proof of correctness is obtained by an algorithm that checks whether the program is a model of the specification. While this may reduce the correctness problem to that of a single program, it still means that we are left with the correctness problem for a program. So, we have replaced the correctness problem for one program by another, and we have the beginning of an infinite regress. (Turner 2018, pp. 207–208)

When we turn to formal logic as our sufficient basis for computer science, there is not even the whiff of a regress, for we can employ what is set out and recommended in (Arkoudas and Bringsjord 2007) and (Bringsjord 2015). In barbarically broad strokes, the execution of a program $\mathbb{P}$ is triggered by the issuance of a query q, and consists in the activity of the automated reasoner $\mathbb{R}$, which yields both an answer and an associated proof (or formal argument) π (α).[14] Once the checker $\mathbb{C}$ certifies that π is valid, the sole remaining issue is the verification of $\mathbb{C}$. But note that there is only a single checker $\mathbb{C}$ for the entire space of logic programs, and the code is short and trivial, so can be classically verified once and for all, easily.

Allow me to end by responding, very briefly, to an inevitable objection to both the logicist paradigm I have outlined, *and* Turner's philosophy of computer science, to wit:

Objection 2

"The two of you, reviewer Bringsjord and author Turner, will doubtless believe yourselves to be miles apart when it comes to your respective views of what

[13] A very brief but sharp overview of $\mathscr{L}_{\omega_1\omega}$ is given in Chap. IX §2 of (Ebbinghaus et al. 1994).

[14] Of course, sometimes no proof will be found, but e.g. a timeout notification will be supplied. For further details, see (Bringsjord 2018a). As wisely noted by Dowek (2015), there is no reason why an automated reasoner working in connection with Turing-undecidable logics can not be allowed to return a timeout message.

> computer science is, but with some sadness I must report that you are actually in lockstep in being utterly out of tune with what computer science in the 21st century is, as a matter of empirical fact. These days, the lion's share of jobs in computer science, both in industry and in the Academy, are in machine learning — and I here specifically must of course emphasize *deep* learning. This is certainly true of the States; but I suspect that in large measure it's true as well of e.g. Europe. Yet, astoundingly, I observe that in *Computational Artifacts* there isn't an iota regarding what a computer scientist does these days in seeing to it that an artificial neural network comes to learn a function, and on the strength of that recognize objects in images, or drive automobiles, etc. As to Bringsjord's c-s-a-i-f-l paradigm, well, machine learning is likewise entirely absent. To the extent, by the way, that both of you venerate theoretical computer science and recursion theory, you both on this score alone find yourself far away from the current wave of — as it's acronymically known — ML."

As to what's in Turner's book in this regard, the critic is entirely correct—but that is a *virtue* of the book. Turner, despite his conflation of computer engineering with computer science, presents in remarkably lucid and economical fashion much of the latter, and I am in fact busily directing my own students to his excellent book (and to others; see Turner 2009) in no small part for this reason. To be clear, it is doubtful that today's ML even qualifies as engineering in any accurate academic sense of that term employed since the advent of the differential and integral calculus. It would be more accurate to regard this beastly departure from the rigorous theoretical structures so nicely covered by Turner in *Computational Artifacts* as alchemy, a claim I have defended at length elsewhere (Bringsjord et al. 2018).

Acknowledgments I am indebted to France's ANR, and specifically to the PROGRAMme project it supports, for the opportunity to interact with that project's brilliant investigators (including special-editor Giuseppe Primiero) regarding the nature of computer science. Among that group, Henri Salha's comments were particularly enlightening and stimulating, and I thank him. PROGRAMme is led by the sagacious and indefatigable Liesbeth De Mol, who in particular has taught me much. Another sponsor, AFOSR (Grant FA9550-17-1-0191), has made it possible for me to explore (with colleagues) high levels of computational intelligence, and the corresponding logic-machines hierarchy $\mathfrak{LM}$ described herein; I am very grateful. I thank special-editor Giuseppe Primiero for having envisioned the forum of which this critical paper is a part, and for overseeing it masterfully. While his fundamental position is not one I embrace, the opportunity to discuss directly with Ray Turner has been invaluable, and perhaps additional oenologically spiced discussions will serve to convince him that the glorious non-physical universe of formal logic suffices for all he himself deeply values in computer science.

References

Arkoudas, K., & Bringsjord, S. (2007). Computers, Justification, and Mathematical Knowledge. *Minds and Machines*, *17*(2), 185–202. http://kryten.mm.rpi.edu/ka_sb_proofs_offprint.pdf.

Benzmüller, C., & Paleo, B.W. (2014). Automating Gödel's Ontological Proof of God's Existence with Higher-order Automated Theorem Provers. In Schaub, T., Friedrich, G., O'Sullivan, B. (Eds.) *Proceedings of the European Conference on Artificial Intelligence 2014 (ECAI 2014)* (pp. 93–98). Amsterdam: IOS Press. http://page.mi.fu-berlin.de/cbenzmueller/papers/C40.pdf.

Boolos, G.S., Burgess, J.P., Jeffrey, R.C. (2003). *Computability and Logic*, 4th edn. Cambridge: Cambridge University Press.

Bringsjord, S. (2015). A Vindication of Program Verification. *History and Philosophy of Logic*, *36*(3), 262–277. http://kryten.mm.rpi.edu/SB_progver_selfref_driver_final2_060215.pdf. This url goes to a preprint.

Bringsjord, S. (2018a). *Introducing Pure General Logic Programming (PGLP)*. Paris: Agence Nationale de la Recherche (ANR). http://kryten.mm.rpi.edu/PRES/SBringsjord_PGLP_Bertinoro.pdf. The URL provided here goes to the slide deck for the lecture in which PGLP was introduced. In Bertinoro, Italy, at the conference center there owned by the University of Bologna. The deck provides details on PGLP.

Bringsjord, S. (2018b). Logicist Remarks on Rapaport on Philosophy of Computer Science^{+}. *Newsletter on Philosophy and Computers*, *18*(1), 28–31. http://kryten.mm.rpi.edu/SBonBR.pdf. The URL here is to a preprint only.

Bringsjord, S., Govindarajulu, N., Banerjee, S., Hummel, J. (2018). Do Machine-Learning Machines Learn?. In Müller V (Ed.) *Philosophy and Theory of Artificial Intelligence 2017* (pp. 136–157). Berlin: Springer SAPERE. http://kryten.mm.rpi.edu/SB_NSG_SB_JH_DoMachine-LearningMachinesLearn_preprint.pdf. This book is Vol.44 in the book series. The paper answers the question that is its title with a resounding No. A preprint of the paper can be found via the URL given here.

De Mol, L., & Primiero, G. (2015). When Logic Meets Engineering: Introduction to Logical Issues in the History and Philosophy of Computer Science. *History and Philosophy of Logic*, *36*(3), 195–204.

Dowek, G. (2015). *Computation, Proof, Machine: Mathematics Enters a New Age*. Cambrige: Cambridge University Press.

Ebbinghaus, H.D., Flum, J., Thomas, W. (1994). *Mathematical Logic*, 2nd edn. New York: Springer.

Eden, A. (2007). Three Paradigms of Computer Science. *Minds and Machines*, *17*(2), 135–167.

Gardner, M. (1958). *Logic Machines and Diagrams*. New York: McGraw-Hill.

Govindarajulu, N.S. (2016). ShadowProver. https://naveensundarg.github.io/prover/.

Govindarajulu, N., & Bringsjord, S. (2017). On Automating the Doctrine of Double Effect. In Sierra, C. (Ed.) *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17), International Joint Conferences on Artificial Intelligence* (pp. 4722–4730). https://doi.org/10.24963/ijcai.2017/658.

Marquand, A. (1885). A New Logical Machine. *Proceedings of the American Academy of Arts and Sciences*, *21*, 303.

McKeon, R. (Ed.) (1941). *The Basic Works of Aristotle*. New York: Random House.

Pollock, J. (1995). *Cognitive Carpentry: A Blueprint for How to Build a Person*. Cambridge: MIT Press.

Primiero, G. (2016). Information in the Philosophy of Computer Science. In Floridi, L. (Ed.) *The Routledge Hhandbook of Philosophy of Information* (pp. 90–106). New York: Routledge.

Rapaport, W.J. (2019). Philosophy of Computer Science. http://www.cse.buffalo.edu/∼rapaport/Papers/phics.pdf , Current draft in progress at the URL given here.

Ross, J. (1992). Immaterial Aspects of Thought. *The Journal of Philosophy*, *89*(3), 136–150.

Smith, R. (2017). Aristotle's Logic. In Zalta, E. (Ed.) *The Stanford Encyclopedia of Philosophy*. https://plato.stanford.edu/entries/aristotle-logic.

Turner, R. (2009). *Computable Models*. London: Springer.

Turner, R. (2018). *Computational Artifacts: Towards a Philosophy of Computer Science*. Berlin: Springer. This volume is in the Series Theory and Applications of Computability.

Turner, R., & Angius, N. (2017). The Philosophy of Computer Science. In Zalta, E. (Ed.) The Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/entries/computer-science.

Wos, L., Overbeek, R., Lusk, E., Boyle, J. (1992). *Automated Reasoning: Introduction and Applications*. New York: McGraw Hill.