A Vindication of Program Verification

Selmer Bringsjord
Department of Cognitive Science
Department of Computer Science
Lally School of Management & Technology
Rensselaer AI & Reasoning Laboratory
Rensselaer Polytechnic Institute (RPI)
Troy NY 12180 USA
selmer@rpi.edu

Received 00 Month 200x; final version received 00 Month 200x

Fetzer famously claims that program verification isn't even a theoretical possibility, and offers a certain argument for this far-reaching claim. Unfortunately for Fetzer, and like-minded thinkers, this position-argument pair, while based on a seminal insight that program verification, despite its Platonic proof-theoretic airs, is plagued by the inevitable unreliability of messy, real-world causation, is demonstrably self-refuting. As I soon show, Fetzer (and indeed anyone else who provides an argument- or proof-based attack on program verification) is like the person who claims: "My sole claim is that every claim expressed by an English sentence and starting with the phrase 'My sole claim' is false." Or, more accurately, such thinkers are like the person who claims that *modus tollens* is invalid, and supports this claim by giving an argument that itself employs this rule of inference.

1. Introduction

Fetzer (1988) famously claims that program verification isn't even a theoretical possibility,[1] and seeks to convince his readers of this claim by providing what has now become a widely known argument for it. Unfortunately for Fetzer, and like-minded thinkers, this position-argument pair, while based on a seminal insight that program verification, despite its Platonic proof-theoretic airs, is plagued by the inevitable unreliability of messy, real-world causation, is demonstrably self-refuting. As I soon show, Fetzer (and indeed anyone else who provides an argument- or proof-based attack on program verification) is like the person who claims: "My sole claim is that every claim expressed by an English sentence and starting with the phrase 'My sole claim' is false." Or, more accurately, such thinkers are like the person who claims that *modus tollens* is invalid, and supports this claim ($\neg\mu$) by giving an argument (where r is any rule of inference from some proof or argument calculus) of the form shown in the following table.

	1	ϕ_1	r
	2	ϕ_2	r
$\vdots$	$\vdots$	$\vdots$	$\vdots$
	k	$\mu \rightarrow \psi$	r
	$k+1$	$\neg\psi$	r
$\therefore$	$k+2$	$\neg\mu$	*modus tollens* $k, k+1$

Table 1. Self-Refuting Argument-Schema Against *Modus Tollens*

[1] E.g., he writes: "The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility." (Fetzer 1988, 1048)

Please note immediately that, regardless of what content instantiates the variables ϕ_j, r, and ψ in this argument-schema, the result will be self-refuting, since whether one assumes that this argument is sound or unsound, one can instantly use *proof by cases* to show that it's unsound.[2]

Of course, deductive arguments against *modus tollens* that employ *modus tollens*, let's hope, are rather few and far between; I offer this example (and the equally shallow ones in note 2) for illustrative purposes. For these examples, a clever programmer could write a program that would decide whether the argument in question is self-defeating, since the mark of self-defeatingness in them is purely syntactic, and internal to the argument in question. When it comes to the evaluation of philosophical arguments, bald forms of self-defeatingness that reduce to syntactic circularity are exceedingly rare. In Fetzer's case, and in the structurally parallel case of any other serious argument for the proposition ($\bar{\varphi}$) that program verification is both theoretically and practically impossible, it is terminologically prudent to refer to what can be called *intensional* self-defeatingness. I say this because the relevant context is that those arguing for $\bar{\varphi}$ are urging rational acceptance of this proposition by the recipients of the argument they provide for $\bar{\varphi}$. This means that were we to model the situation using formal logics,[3] such logics would include intensional operators for belief, knowledge, intention, communication, etc. More specifically, if agent $\mathfrak{a}$ provides a serious argument $\mathcal{A}$ to agent $\mathfrak{b}$ in order to impel $\mathfrak{b}$ to accept $\bar{\varphi}$, then $\mathfrak{a}$ intends that $\mathfrak{b}$ believes that $\bar{\varphi}$ on the basis of $\mathcal{A}$. (Many other such propositions, containing intensional operators, are entailed; for instance, the straightforward one that $\mathfrak{a}$ believes that $\bar{\varphi}$.) As we shall see, the intensional self-defeatingness that overthrows Fetzer's case is essentially that Fetzer can have a rational belief that recipient $\mathfrak{b}$ believes $\bar{\varphi}$ on the strength of Fetzer's $\mathcal{A}$ only if it's possible that $\mathfrak{b}$ assimilate and verify that $\mathcal{A}$ indeed establishes $\bar{\varphi}$ — but if $\bar{\varphi}$ is established, then it's not possible that $\mathfrak{b}$ assimilate and verify this.

I'll also show that once the self-refutation in Fetzer's case is revealed, a corollary is that an impressive form of program verification turns out to be both theoretically and practically possible. My demonstration exploits some previously published structures and distinctions arising from careful consideration of proofs of the four-color theorem (*Arkoudas and Bringsjord* 2007).

The plan for the remainder is this: In the next section (2) I rapidly review program verification and the core issue before us (viz., whether such verification is possible); then explain that Fetzer's argument necessarily conforms to an argument-schema that must be physically checked if it's to convince any reader (§3); then uncover the aforementioned self-refutation (§4); and end by explaining that the future for the brand of program verification here presented is bright (§5). Along the way, numerous objections are considered, and rebutted.

2. Program Verification Encapsulated

As some readers may be unfamiliar with the topic of program verification, and its importance practically and philosophically speaking (let alone with the unprecedented firestorm that Fetzer's paper ignited), at least a brutal review is in order.[4]

[2]Some readers may be under the impression that if one wants to (rationally) dispute an argument, one needs to either accept the premises (at least temporarily, for the purpose of evaluating the argument) and show why the conclusion doesn't follow, or on the other hand dispute the premises. This impression is a mistaken one, in light of the fact that one can dispute/refute an argument by claiming/showing that it's self-refuting (e.g., that it's an instance of the argument-schema shown above). But there are more mundane counter-examples to the impression: The one-step argument of ψ from ψ, where ψ is known by all to be true, points the way to one. For this argument is fallacious (*petitio principii*), yet the conclusion follows with deductive perfection from the premise, and — by hypothesis — the premise is true. Another example is found in *refutation by parody*, which doesn't attack any premise in the targeted argument, and may well concede that the reasoning from premise(s) to conclusion is logically valid, but is at least agnostic on whether the reasoning in question is valid.

[3]This modeling is of course far beyond the scope of the present paper. For information regarding this kind of project, see e.g. (*Bringsjord* 2008); and for a multi-modal quantified computational logic ($\mathcal{DCEC}^*$) that constitutes the machinery needed for such a project, see e.g. (*Bringsjord et al.* 2014, *Bringsjord and Govindarajulu* 2013), which has its roots in the computational logic presented and used in (*Arkoudas and Bringsjord* 2009).

[4]A full review would require a book, and we don't have the luxury of that much space. For an efficient book-length introduction at the undergraduate level, the reader can consult (*Almeida et al.* 2011). A shorter, elegant introduction is provided in (*Klein* 2009). As the reader will soon see, I end by describing and advocating a *specific* form of program verification that exploits the Curry-Howard Correspondence. But my approach will ultimately be best understood by

What does it (customarily) mean to verify that a (computer) program *Prog* is correct? Certainly the uncontroversial kernel of any acceptable, standard answer will commit to the proposition that a verification of *Prog* consists in proving that it conforms to some *specification Spec*. Let's consider a painfully simple but nonetheless instructive example.

Suppose that our task is to verify that computer program `sum-up` conforms to the *Spec′* that when upon receiving as input a natural number n it returns $0 + 1 + 2 \ldots + n$. Let's assume that `sum-up` is this Common Lisp program:

```
(defun sum-up (n)
 (if (zerop n) 0
     (+ n (sum-up (- n 1)))))
```

How does this program work? Even those unfamiliar with Common Lisp should be able to quickly grasp the procedural algorithm (Alg_{sum-up}) here coded: viz.,

(1) Upon receiving as input NUM some natural number n, check to see if NUM is zero.
(2) If this identity holds, return `num` and halt; otherwise, add n to the result of proceeding again to line 1. with $n - 1$ as input.

What now does verification of the correctness of `sum-up` consist in? It consists at least in significant part in a correct proof P of the proposition that `sum-up` meets `Spec′`. Such a proof can be a trivial inductive one, easily supplied by most readers.[5] A formal version of such a proof, expressed as a rigid sequence of steps that can be mechanically checked, is also easily obtained.[6] Of course, `sum-up`, just like multi-million-line-long programs, have value because they can be physicalized and executed; we can thus refer to this resultant thing as $\texttt{sum-up}_p$. As we shall soon see, the plain distinction between `sum-up` and $\texttt{sum-up}_p$ is one Fetzer wisely makes much of, and this is a distinction I too affirm. But as we shall also see, he and others have apparently overlooked the parallel plain fact that physicalization and execution isn't restricted to silicon (etc.) hardware: after all, *humans* can execute (or "hand-simulate") `sum-up`, and when they do so, they certainly can err. To make this painfully concrete, one need only "run" `sum-up` on, say, 27. Assuming you do this, are you sure that your answer is correct? Isn't it possible that you're wrong, because of a failure of short-term memory, or a mini-stroke, or a tiny optical illusion? The same worries apply to the checking of arguments offered to you in print by philosophers (or of proofs offered to you in print by logicians), including then Fetzer's argument; this fact will be key.

And to also set the stage for the brand of program verification recommended on the heels of refuting Fetzer's argument, consider the proof-as-program $\texttt{sum-up}^L$ that consists in code that, upon receiving some input n, yields a proof P_{sum-up} of $\exists x SumUp(n, x)$ (from relevant arithmetic axioms $\psi_1, \psi_2, \ldots, \psi_k$ and a definition of $SumUp$) by proving in its penultimate line that $SumUp(n, a)$, and then outputs a along with the proof. Our manual verification here would merely consist in checking the following table line by line. Note that once we have the proof P_{sum-up} we can restrict our program-verification attention to checking it, since this proof, in and of itself, generates the needed output. In the proof, $\delta(x, y)$ is the definiens, a formula with free variables x and y.

readers who begin with standard coverage of program verification, which routinely leaves C-HC aside.

[5] An informal version of such a proof is provided in "Proving programs correct," section 16.5 of (*Barwise and Etchemendy* 1999).

[6] I omit a deeper and broader technical account of program verification as it is vibrantly pursued in the field of computer science, as my desire in the present paper is to wholly avoid the perception that my refutation hinges on the details of contemporary program verification. Even the brand of program verification I recommend below (a *single*, *tiny* classically verified program (viz., a proof checker), and — guided by the Curry-Howard Correspondence — proofs as programs with the associated theorems as output) is intended to be perfectly understandable to those without knowledge of the technical ins and outs of today's program-verification scene. In fact, that knowledge, I suspect, would only hinder understanding of the *sola logica* route I recommend.

	1	ψ_1	*supp*
	2	ψ_2	*supp*
⋮	⋮	⋮	⋮
	k	ψ_k	*supp*
	$k+1$	$\forall x \forall y(SumUp(x,y) \leftrightarrow \delta(x,y))$	Def
⋮	⋮	⋮	⋮
$\therefore$	$k+p$	$SumUp(n,a)$	r
$\therefore$	$k+p+1$	$\exists x SumUp(n,x)$	$\exists$ Intro $k+1$

Table 2. The Proof to Check in Order to Verify `sum-up`^{L}

3. The Logico-Historico-Philosophical Depth of Program Verification, and Fetzer

Some readers may be initially puzzled as to how from the perspective of the history and philosophy of logic such a pursuit as program verification, explicated as above, could be of serious, let alone profound, interest.[7] The sources of the philosophical depth of program verification are many and varied, and many of these sources have long been noted by thinkers working in the intersection of logic and computation.

I shall here first provide a brutally compressed and high-level look at the history of program verification, and move from that perspective to my focus on the aforementioned paper by Fetzer, a focus that drives the remainder of the present paper. Time-wise, I start from the current chapter in the history in question, which I refer to as 'Phase 4,' work backwards immediately all the way to the other temporal "bookend" of the history, Phase 1, and then explain why Fetzer's paper, the second part of what catalyzed Phase 3, is argubaly the pivot around which the entire historical narrative revolves.

The history of program verification, as alert (computationally inclined) logicians and philosophers of today know, has now expanded to be about nothing less than the nature of the relationship between abstract logical systems and the physical world. Phase 4, today's chapter in the narrative of program verification, features some very rich, nuanced proposals regarding the nature of this relationship. For example, in a recent paper, *Turner* (2012) proposes that the relationship between the abstract and the corporeal consists in part in the human desire that the former govern the latter, not merely in some sort of traditional alignment between the two. Simpler proposals regarding the nature of this relationship, motivated by the fear that any description of computation can apply to any physical process whatever, have been issued (e.g., *Chalmers* 1996). The present paper can also be viewed as taking place in Phase 4, since its fundamental claim is that *if* we get our logico-mathematics right, and correctly deploy it, we can forge a connection between the abstract and the practical that is so iron-strong as to be maximally comforting.

In the case of program verification, the relationship between the abstract and the corporeal is specifically between those logical systems that specify information-processing, and the causal processes that correspond to the execution of those specifications. In that light, program verification's dawn is without question that first shining moment when some rather clever human mind, in command of an abstract, logical description D of some information-processing, asked in earnest whether D does in fact describe the processing it's intended to describe, and then asked how an answer could be demonstrated. Given this, the genesis of program verification, as every educated reader must agree, coincides with cognition on the part of one or more of the

[7] I assume that *all* readers will either already know, or immediately see, that program verification is of enormous *practical* importance. The Internet now carries a vast number of profoundly disturbing real-life cases in which unverified programs were naïvely predicted to operate in desired ways, but didn't, in some cases at the cost of lives, and in countless cases at the cost of staggering amounts of money. As computing machines and robots continue to increase in autonomy and power, it seems undeniable that the practical importance of program verification will only grow.

"founders" of general-purpose[8] information-processing. In this regard, scholars can take their pick between for instance Church, Turing, and Post, whose specifications (later of course proved equivalent) were expressed, respectively, as and the λ-calculus (*Church* 1932, 1933), Turing machines (as they are now called) (*Turing* 1937), and "worker-box" models (*Post* 1936). But when it comes specifically to program verification, it's apparently Church, Turing, and von Neumann who offer us the first clear, *preserved textual confirmation* of their cognition in this direction, and the fruit borne by that cognition: (*Church* 1960); (*Turing* 1949), wonderfully analyzed and cleaned up in (*Morris and Jones* 1984); and in the case of von Neumann (and collaborators): (*Goldstine and von Neumann* 1947). Turing proves that a program (or, as he calls it, a "routine") for computing $n!$ without multiplication is correct. von Neumann, too, puts the focus on the processing of — as he puts it — "numerical problems." But von Neumann's analysis is remarkably extensive, and anticipates program verification in its modern guise as I described it above with shocking prescience and precision. I would go so far as to opine that all students of the history and logic/philosophy of program verification, and indeed of program verification itself, should begin by analyzing Turing's two-page proof, and von Neumann's remarkable essay — by thus reviewing, as I shall heretofore refer to it, the 'T-vN Start' (= Phase 1). One might be allowed to rather passionately make this recommendation in no small part because the logico-philosophical terrain of program verification, as conceived by Fetzer, can be fairly classified as "Phase 3," coming as it does *after* the seminal work of Turing and von Neumann.[9]

Phases 2 and 3 are masterfully summed up in one indispensable volume: (*Colburn et al.* 1993). It's fair, I think, to view Phase 2 as constituted by two things: one, a more mature, contemporary expression of the T-vN Start, and two, sustained optimism that this approach would eventually provide full-blown verification. A well-known paper, reproduced in (*Colburn et al.* 1993), serves perhaps better than any other to express this two-fold hallmark of Phase 2: Hoare's (1969) "An Axiomatic Basis for Computer Programing."[10]

But what then is Phase 3? That's easy. Phase 3 consists of two blows which together catalyzed a bit of a crisis for the optimists of Phases 1 and 2. Each blow was a paper.

The first paper in the pair is a highly influential one by *De Millo et al.* (1979), which appeared about a decade ahead of Fetzer's. In a nutshell, De Millo et al. argued that since intuition is the fundamental driver of progress in mathematics, and since the production of mathematical proofs is an irreducibly social process, a formal, proof-based approach to program verification (and by extension *a fortiori* the proofs-as-programs approach I promote below!) is fundamentally infeasible, and should not be allowed to drive the design of programming languages and programs. It is not unreasonable to hold, from the vantage point of the start of the new millennium, that (*De Millo et al.* 1979) has been to an appreciable degree *empirically* refuted, since success stories for formal program verification (based not on the informal proofs that logicians and mathematicians share with each other, but rather on formal, step-by-step proofs), and the energy and seriousness

[8]Full historical analysis would by the way need to consider *non*-general-purpose computation, and the relationship between a description of that more limited computation and what (at least) input-output behavior that computation does in fact enforce. Given this, my personal suspicion is that Leibniz will in the end be seen as the genuine originator of program verification (in a parallel to our learning that over one and a half centuries before Boole, Leibniz had more than the propositional calculus, and had standard modal logic; see (*Lenzen* 2004)), but a defense of this position reaches too far given present purposes.

[9]For note that Fetzer et al.'s compendium on program verification, (*Colburn et al.* 1993), while incuding a reprint of a number of classic papers on program verification from computer scientists themselves (including a classic one from *McCarthy* (1962) reminiscent of the approach to program verification defended in the present paper), starts after the aforementioned papers of Turing and von Neumann (i.e., after the T-vN Start).

[10]By the way, *Turner* (2013) observes that *Hoare* (1969) apparently — to use the verb Turner chooses — 'alludes' to the very issue that Fetzer presses against the "optimists" of Phases 1 and 2:

> When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics. (*Hoare* 1969, 579)

In Fetzer's defense, this is at best a vague reference to some of Fetzer's concerns.

with which it is currently pursued, are at odds with what these authors advanced.[11] Regardless, while there can be little doubt that intuition plays a major role in the discovery of proofs by humans, and that logic and mathematics are in part social, multi-agent processes, such truths don't entail any such Fetzerian claim as that program verification, even theoretically speaking, is impossible.

This brings us to the second and pivotal source of Phase 3, and clearly a source of the logico-philosophical depth of program verification: Fetzer's (1988) enormously influential "Program Verification: The Very Idea."[12] His reasoned argument for the claim that program verification — as we have adumbrated it above (§2), and as we have historically contexualized it immediately above — is a theoretical impossibility is clearly of profound interest to those interested in logic and its philosophy, if for no other reason than that this argument would seemingly show that what we saw above unfolding before our eyes (e.g., the line-by-line verification of P_{sum-up}), and what we have seen from the giants of computation like Church, Turing, and von Neumann, is some kind of illusion, or at minimum profoundly misleading.

Fetzer's argument is encapsulated by the following, taken directly from his abstract, verbatim:

> The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.

We can assume Fetzer to be in total, crisp control of the categories deployed in our above synopsis of program verification. Astute readers will recall that in that synopsis we made use of: *algorithms* ($Alg_{sump-up}$), *programs* (`sum-up` and $\texttt{sum-up}^L$), *proofs* (P_{sum-up}), and *physicalized programs* ($Prog_p$, $\texttt{sum-up}_p$, $\texttt{sum-up}_p^L$). If we suppose that in each case Fetzer can meet the burden of suitably defining these concepts, then his argument includes at least the following crisp variant of triadic *proof by cases* at its highest level:

Top-Level Proof-by-Cases Structure of Fetzer's Argument

P1 Program verification must consist in the formal, deductive verification of *physicalized* programs, not merely the formal, deductive verification of algorithms and purely linguistic programs.

Note: This corresponds to what I earlier labeled Fetzer's 'seminal insight.'

P2 Formal, deductive verification of an algorithm *Alg* doesn't entail the formal, deductive verification of any physicalized program (including those based on programs that regiment *Alg*).

P3 Formal, deductive verification of a (purely linguistic) program *Prog* doesn't entail the formal, deductive verification of any physicalized program (including therefore the corresponding physicalized program $Prog_p$).

P4 Formal, deductive verification of physicalized programs isn't theoretically possible.

$\bar{\varphi}$ Program verification isn't a theoretical possibility.

Two quick but important points:

One, this is my charitable, formally valid interpretation of the top level of Fetzer's argument, which I take to pivot on a core distinction between those things that he concedes are susceptible of formal analysis and proof, and those which, by their informal, "messy," and physico-causal nature, aren't.[13] But — and here point #1 — *any* inferential structure would be sufficient to anchor my analysis and reasoning in the next section, since Fetzer must hold that inferential structure, whatever it is, can be verified by human scrutiny — despite the brute fact that that

[11] See, e.g., the insightful (*Asperti et al.* 2009).

[12] The paper is reproduced in (*Colburn et al.* 1993).

[13] Fetzer speaks of algorithms versus programs, but since for us (confirmed by the analysis of §2) programs are — following standard professional and educational practice — linguistic objects intended to conform to formal grammars supplied in programming textbooks, we can understand his reference to programs to be a reference to what we have called *physicalized* programs. This creates no gulf between Fetzer and me; indeed it marks an *affirmation* of Fetzer's chief insight; the point, then, is merely terminological.

scrutiny is at the mercy of human perception, misperception, bias, and so on.[14] In the case of the triadic version of proof by cases given here, the brute empirical fact is that the verification of the reasoning to which I allude is indeed potentially compromised by misperception. For example, when the reader identifies the three relevant cases ($c_1 :=$ verification of algorithms, $c_2 :=$ verification of (linguistic) programs, $c_3 :=$ verification of physicalized programs), the reader will then need to rule out one c_i in order to arrive at a remaining pair, but then when it comes to ruling out one of the remaining pair, can the reader be 100% sure what integer i was set to?

And the second point: Certainly Fetzer's full argument includes sub-arguments for each of **P1**–**P4**. But those sub-arguments have inferential structure, and Fetzer must hold that that structure can be verified despite the vulnerabilities and vicissitudes of fallible human perception, memory, and judgment.

Is Fetzer's top-level argument a valid one? My question here is directed at the same sort of sedulous readers who either attempted to prove that `sum-up` satisfies the specification given above for it to meet, or better, at those who attempted to determine the result of `sum-up` on 27. How is such an energetic reader to proceed in order to ascertain whether Fetzer's top-level reasoning is right? Obviously, by unpacking in some deductive rule-set that $\bar{\varphi}$ can be derived from {**P1**, **P2**, **P3**, **P4**}, and by then checking that that progression of steps is correct. This activity is isomorphic to the activity of checking the proof P_{sum-up}, and is also activity that is highly fallible, since for example any number of glitches in the hardware of human perception can rear up at any time during checking.

We have now set the stage to make the self-refuting nature of Fetzer's argument transparent. (I suspect many readers already see this nature, even at the informal level we have reached.)

4. The Self-Refutation

Denote, as immediately above, the proposition that program verification isn't a theoretical possibility by $\bar{\varphi}$, and let $\mathcal{A} = (\Gamma, \vdash^*, \bar{\varphi})$ refer to Fetzer's full argument, which by standard background knowledge about the minimal structure of arguments we can be sure starts with finitely many premises Γ and proceeds by inferential steps $\vdash_1, \vdash_2, \ldots, \vdash_n$ to the conclusion, $\bar{\varphi}$. Obviously, the top-level proof-by-cases moves we saw above would be subsumed by this complete argument $\mathcal{A}$. ($\vdash^*$ is simply the sequence of all $\vdash_i$; each step can itself be considered a quadruple consisting of a label, a declarative statement, a rule of inference, and a citation composed of a label or labels indicating what prior individual statements and or sub-proofs are used in the inference. See the Tables 1 and 2 given above.)

Next, note that argument $\mathcal{A}$ is what we can harmlessly call an *abstract type*. The basis for this generic terminology is the same as what allows various thinkers to refer to Fetzer's argument in (for over three decades now, often animated) conversation, without any relevant physical object in play. This is no different than folks referring (successfully) to, say, Gödel's proof of the completeness theorem, in the absence of any particular inscriptions upon paper, or a computer display, etc. (It's the *tokens* of Gödel's proof that vary considerably across textbooks, classrooms, notebooks, etc.) I denote a physical token of this type as $\widehat{\mathcal{A}}$. Argument tokens are physical instantiations of the corresponding abstract arguments; they can be written down on paper and other media, read, inspected, erased, copied, transmitted, and so on. (In fact, resting on a table in front of me right now, courtesy of a printout of (*Fetzer* 1988), sits a particular $\widehat{\mathcal{A}}$.)

We can immediately deduce from Fetzer's position and standard background knowledge in such matters (such as that Fetzer published $\widehat{\mathcal{A}}$ in order to allow others to read his argument,

[14] This is as good a point as any to state for the record that I'm not in the least incredulous that Fetzer would hold the position that program verification is a theoretical impossibility. It's actually easy enough to see how he could advance this claim despite the fact that program verification has long been a healthy corner of computer science. After all, that healthy corner consists in the activity of human beings. Fetzer saw and indeed sees that activity, he just doesn't think that it *actually* results in the verification of (physicalized) programs. What he didn't see is that, first, in order for his reasoning to succeed, it itself must be verified, and, second, that program verification (of the proofs-as-programs kind I advocate herein) can consist in nothing more than checking steps of reasoning.

and thereafter adjust their beliefs accordingly) that there is an argument-checking process AC that can be applied by rational, objective humans to arguments in order to evaluate them. (It's of course not enough to know that Fetzer devised an argument for $\bar{\varphi}_t$; one must study $\widehat{\mathcal{A}}$ in order to be convinced of his pessimism: We need to check his argument.) Accordingly, we write $AC(\mathcal{A}') \rightsquigarrow \phi$ to mean that applying the process AC to argument $\mathcal{A}'$ produces the statement ϕ; and we write $AC(\mathcal{A}') \rightsquigarrow \mathit{error}$ to indicate that the application of AC revealed an error, signifying that $\mathcal{A}'$ is unsound. (These are the only two output possibilities.) We can view the argument-checking process as itself an abstract object, so we write

$$\widehat{AC}(\widehat{\mathcal{A}'})\widehat{\rightsquigarrow}\widehat{\phi}$$

to say that a token of the argument-checking process (say, one that is stored in someone's brain) is applied to a particular token of the argument and yields a particular token of formula ϕ. Finally, let $B_X[\phi]$ abbreviate that X believes ϕ.

With the above machinery in place, we deduce from Fetzer's position, combined with the fact that one's knowing ϕ entails one's justified true belief that ϕ,[15] that he affirms the progression shown in Figure 1.

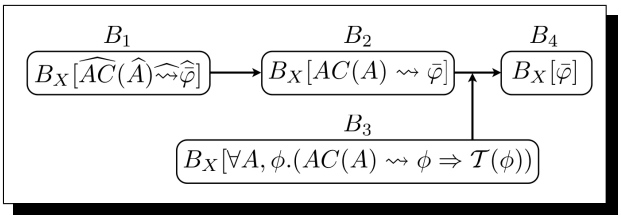

Figure 1. Justificational Structure Produced by Studying Fetzer's Paper

Figure 1 is straightforward. The use of arrows running from left to right, connecting the B_i, indicates a temporal progression. In the final stage, B_4, X simply believes $\bar{\varphi}$ (and if Fetzer is right, X knows this proposition). The first stage is self-explanatory at this point, as is the second. What, then, about B_3? The belief here simply consists in the general proposition that for any argument and purported-to-follow-from-it conclusion ϕ, if AC applied to this argument produces that conclusion, then the conclusion is true.

Summing up to this point, we have the following proposition:

(1) If Fetzer is right (i.e., if his position is correct), then rational, objective human persons can come to know $\bar{\varphi}$ via the progression shown in Figure 1.

Now, please note that the machinery set out above can be viewed as a schema, in which, specifically, reference to 'argument' is replaced by 'proof,' the argument-checking process AC is replaced by a *proof*-checking process PC, and inferential steps are entirely deductive.[16] For example, Bringsjord established the following when he taught today: First-order logic is complete ($= \chi$), and rational, objective[17] humans (students, in this case), by engaging in the proof-checking process $\widehat{PC}$ of his Gödel-style proof $\widehat{P}$ of this claim, will come to know that χ is true. (Of course,

[15] Note that I don't assume the not-uncontroversial converse: I don't assume that justified true belief entails knowledge.

[16] In the case of Fetzer's $\mathcal{A}$, much of the reasoning is in fact intended to be deductive (the triadic proof-by-cases reasoning discussed above was of course deductive), but my overall argument loses nothing if we allow that some $\vdash_i$ are non-deductive, e.g., inductive or abductive or analogical or what have you. Put in terms of the tabular argument-schema with which the present paper started (Table 1), that schema is self-refuting regardless of what the occurrences of r in it are instantiated to, and regardless of any distinctions, however ingenious and otherwise telling they may be, made between such concepts as deductive proof and inductive evidence. For an argument calculus that integrates these various modes of reasoning, and can be used to formalize and check Fetzer's argument as idealized in the second paragraph of the present paper, see (*Bringsjord et al.* 2008).

[17] And sufficiently smart — but in all systematic investigations of the matters before us, charity dictates that we assume those involved to be sufficiently intelligent to grasp what's going on.

what Bringsjord did, on the very same day, was being done across globe, not only with respect to this theorem, but thousands of others.) With obvious substitutions, the situation can be captured by Figure 2.

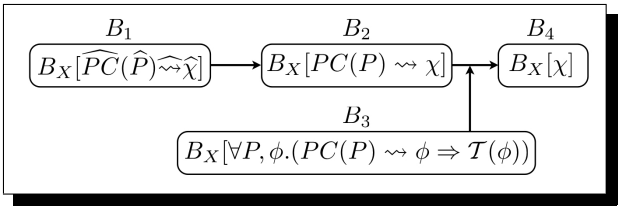

Figure 2. Justificational Structure Produced by studying Bringsjord's Gödel-style Proof

Next, note that we clearly must assent to the following proposition, in light of the foregoing (and in light of the undeniable fact that checking an informal natural-language argument like Fetzer's is much more challenging than checking a Gödel-style completeness proof).

(2) If rational, objective human persons can come to know $\bar{\varphi}$ via the progression shown in Figure 1, then rational, objective human persons can come to know χ via the progression shown in Figure 2.

Now for the next stage of my demonstration.

It's a logico-mathematical fact, rather easily proved, that computer programming can be carried out as the design and execution of proofs. This fact is known as the Curry-Howard Isomorphism (or Correspondence) (C-HI), sometimes, given our present dialectical context, tellingly called the 'Proofs-as-Programs Correspondence' (and as is well-known, there are numerous additional labels in the literature).[18] The particular instance of C-HI that is likely to be of most interest to philosophers is the direct correspondence between natural deduction (the type of proof theory usually taught in philosophy programs) and the λ-calculus, a specific correspondence established by Howard (1980) after Curry's seminal introduction of the general correspondence in question.[19] Please notice that I say programming can be *carried out* as proof design and execution. I'm not talking about designing proofs *about* programs. Nor am I talking about the capture of computations in deduction, which is standard fare in proofs of the undecidability of first-order logic [see e.g. the classic proof in (Boolos, Burgess & Jeffrey 2003)]. Rather, I'm talking about doing programming as doing proof construction: writing proofs *qua* programs, which, courtesy of the remarkable C-HI (not to mention a significant period of concrete practice), we now know to be entirely possible.[20] More specifically, the view is that for any specification-program pair $(Spec, Prog)$ in the proof-as-programs approach sanctioned by C-HI: there exists a standard proof P' (in a proof calculus of one's choosing), and a first-order formula ψ, such that $Prog$ is correct relative to $Spec$ if and only if (= iff) P' goes from the first-order formulas Φ_{Spec} representing $Spec$ to ψ; i.e., iff — in standard notation — $\Phi_{Spec} \vdash_C \psi$, where C is a variable for some particular proof calculus.[21] This means that under the C-HI approach, the challenge of program verification can be *identified* with the challenge of checking a proof. Notice what the previous sentence says: the challenge of *checking* a proof, *not* the challenge of *finding* a proof. Proof *search* is notoriously difficult. Proof *checking* is infinitely easier: it consists in being able to verify the correct, sequential application of painfully simple rules like *modus ponens*, reflexivity, and transitivity. Proof checking, in short, is easier than grade-school division. There are familiar theorems that undergird and illuminate the distinction just made: The decision problem for

[18] A nice overview, freely available online, is (*Girard* 1989).

[19] See also the excellent (*de Queiroz et al.* 2011).

[20] Personally, I have no idea why such activity isn't discussed more in philosophy programs, and indeed in the field of philosophy itself.

[21] E.g., the $\mathcal{F}$ of (*Barwise and Etchemendy* 1999). In an uncanny notational coincidence, *another* proof theory, also denoted by $\mathcal{F}$, is quite relevant to the proofs-as-programs paradigm (because of the frequent use of higher-order logic in this paradigm), viz. the type-theoretic $\mathcal{F}$ of (*Andrews* 2002).

theoremhood in first-order logic is Turing-undecidable, as is program verification in the general case; but a Turing machine exists which decides, for any proof expressed in a standard, finitary proof calculus, whether or not it's steps are all valid.

Not only that, but consider that *one* proof checker can handle *all* proofs (expressed in a fixed proof calculus).[22] There is, therefore, one fixed algorithm that can check any given proof; to ease exposition, we shall refer to it as $PC^{\star}$. Please note that whereas earlier I referred to generic *processes* for checking arguments and proofs (AC and PC, resp.), and left it quite open as to whether this process was literally an algorithm or computable function or anything of the sort, I'm now saying something much more determinate about $PC^{\star}$: I am saying that this is in fact an algorithm — indeed a dirt-simple one. I'm also saying that a person can apply this algorithm (i.e., apply $\widehat{PC^{\star}}$) to a proof in order to see if it's correct.[23]

How do we know this? We know this because it's infinitely easier to apply the fixed and fully specified $\widehat{PC^{\star}}$ to an arbitrary proof than it is to apply the process $\widehat{PC}$ to a proof like Bringsjord's Gödel-style $\widehat{P}$! — and we have already noted that using $\widehat{PC}$ is something the relevant class of persons routinely do. In point of fact, the entire, longstanding program verification problem can be solved by $\widehat{PC^{\star}}$, which can be but a page or two of code in an advanced programming language such as ML or Haskell, and has linear-time average-case complexity in the size of the input proof. $\widehat{PC^{\star}}$ is also completely domain-independent: It can be used to check proofs, and likewise programs-as-proofs, pertaining to anything under the sun.

In keeping with the pictorial scheme used above, the justificational structure that undergirds successful program verification can be summed up in Figure 3.

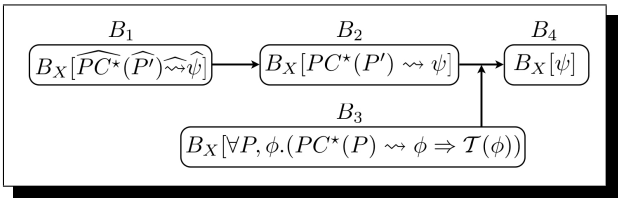

Figure 3. Justificational Structure Associated with Successful Program Verification.

We can express the situation we have arrived at explicitly, via this pair of propositions:

(3) If rational, objective human persons can come to know χ via the progression shown in Figure 2, then such persons can use $\widehat{PC^{\star}}$ to come to know whether or not some arbitrary proof $\widehat{P'}$ yields $\widehat{\psi}$ or not, via the progression shown in Figure 3.

(4) If rational, objective human persons can use $\widehat{PC^{\star}}$ to come to know whether or not some arbitrary proof $\widehat{P'}$ yields $\widehat{\phi}$ or not, then such persons can come to know whether or not some arbitrary program *Prog* (relative to a specification) is correct, i.e., program verification is a practical possibility (φ_p), and hence *a fortiori* also a theoretical one (φ).

Showing that Fetzer's position is self-refuting is now trivial:

Proof. From {**(1)**–**(4)**} by repeated applications of hypothetical syllogism it follows immediately that if Fetzer is right, program verification is both a practical and a theoretical possibility. Suppose Fetzer is right. Then by *modus ponens* program verification is both a practical and a theoretical

[22] For a fixed, finitary proof calculus, this can be easily proved. But a more concrete route is to simply note that there are many fixed proof checkers available for sale, and many of them are used to train students of beginning formal logic. The fixed, single proof checker Fitch, for instance, is available on CD along with (*Barwise and Etchemendy* 1999).

[23] This is perhaps the best place to point out that 'correct proof' is not redundant. Like 'correct computer programs,' proofs have rich, recursive syntactic structure. Both can be given inputs (viz., assumptions or premises) and evaluated, and the evaluation can produce an error, or it can generate an output, viz., the conclusion. Just as programs are correct relative to specifications, proofs are correct relative to input premises and output conclusions. Note that an argument or proof can be composed of only formally valid inferences, and produce a conclusion, but can nonetheless be incorrect, if the produced conclusion is not the desired one. For a robust treatment of these matters, and formal details regarding an elegant proofs-as-programs route, consult (*Arkoudas* 2000).

> possibility. But also, if Fetzer is right, then both $\bar{\varphi}_p$ and $\bar{\varphi}$ can be immediately deduced; i.e., both the practical and theoretical *im*possibility of program verification can be deduced. Our supposition has led to a contradiction, and hence by *reductio ad absurdum* it follows that Fetzer is wrong. **QED**

While I caution readers who want an intuitive encapsulation of this result that that is to demand something that is by its informal and less rigorous nature compromised relative to the more explicit reasoning just given, I nonetheless oblige via the following:

> The core of the argument given in the paper, which is new in the literature on program verification, is that the messy world of human perception, memory, etc., all of which is real-world physical rather than some idealized Platonic activity, makes the checking of Fetzer's argument (and any other argument offered by one thinker to another) fallible. The same must hold for program verification, which is (as Fetzer sagaciously points out) plagued by the messiness of real-world causality. Hence, if Fetzer is right and can be seen to be right, he is thus wrong, because if he is right and can be seen to be right, the real-world messiness that plagues the checking of physicalized argumentation by fallible human perception and memory can be surmounted, and the very real-world messiness that plagues program verification can *mutatis mutandis* also be surmounted. "**QED**"

Now four important points.

First, the fallibility of human perception in the case of visual symbolic information that is as simple and clear as possible is a matter now of empirical, scientific fact. Here I refer to symbolic information that is *much* simpler than any such argument expressed by a skeptic of program verification intent on justifying their skepticism to others. Much of the cognitive science to which I allude has been carried out by Landy and Goldstone (e.g. 2010, 2007). As an example, consider the request to subjects (college students in the U.S.) that they compute the following arithmetic expression.

$$2 + 4 \quad \times 7$$

A large number of neurobiologically normal college students believe that the result of this arithmetic is 42. Of course, the result is really 30. One can increase the likelihood of the erroneous result by simply tweaking the sense data just a bit. For instance, the percentage of those responding with '42' is higher when the following is the visual input.

$$2 + 4 \quad \times 7$$

Many, many other errors produced merely by slight variations in perceptual input can be found in the relevant corner of cognitive science, but we have no need to canvass the relevant terrain, for the moral of the story for us is clear: Judgments by humans about normative validity of symbolic physicalized expressions is to a significant degree at the mercy of the fallibility of human perception. Of course, we know that this fallibility can be made vanishingly small (we can insist in cases such as the above equations that parentheses be included throughout, and that computation work from the "inside out"), but it can't be brought to zero here, and nor can it be brought to zero in the case of multi-page arguments of the sort made by Fetzer (1988). Hence if Fetzer's argument is valid, and we can come to know that it is, which is his expectation, the messiness of real-world physico-causation can be surmounted.

Second point: It should be clear that any hair-splitting linguistic and logical distinctions at the heart of the top-level argument for $\bar{\varphi}$ from premises **P1**–**P4** provide no escape from my refutation, for the simple reason that the verification of programs via verification of proofs is maximally reliable despite the possibility of all manner of physical glitches in the neural system of humans who study and certify these proofs and their theorems. In a word, as I have shown, his argument $\mathcal{A}$ succeeds only if the verification of physicalized proofs is theoretically possible — but then what's sauce for the goose is sauce for the gander: verification of physicalized programs must be accepted as well, since such verification, as has been shown, can, on the shoulders of C-HI, *be* verification of physicalized proofs. In the case of physicalized programs, yes, the execution thereof is at the

mercy of physico-causal processes (gamma rays, e.g.) that may introduce stray, unanticipated, unpreventable, and undetectectable errors. But again, the physico-causal processes required for human verification of proofs and arguments, *mutatis mutandis*, put such processes at the mercy of stray, unanticipated, unpreventable, and undetectable errors (optical illusions, mini-strokes that deceive the eyes or cause false short-term memories, and so on *ad indefinitum*).[24]

Third point: This proof, and structures that enable it, have nothing in particular to do with Fetzer. Indeed, it would greatly reduce the point and reach of the present paper to plumb the depths of the Fetzerian sub-arguments underneath the top-level reasoning we isolated in section 3. *Any* even marginally rigorous case for the impossibility of program verification will, by exactly parallel reasoning, self-refute. (If the case is not based on rational argument, it's insulated from self-refutation, but what force would such a case have?) In light of the foregoing, no one can rationally attack program verification and succeed, because if they succeed, they will do so only by virtue of the fact that their chain of reasoning has been verified — but such verification, as we have seen, will need to be accepted despite the fact that it can in principle go awry because of glitches in the physical sensorimotor substrate that is active during the verification.

Fourth point: As astute readers will have long realized, both φ_p and φ are independently provable from the conjunction of **(3)** and **(4)**, given that rational, objective human persons can indeed come to know χ via the progression shown in Figure 2. This shorter proof stands as a vindication of program verification, independent of Fetzer's attack, and of an attack issued by anyone else.

5. Confidence Going Forward

Inevitably some readers will doggedly insist that while Fetzer's case may be neutralized, program verification, Figure 3's framework and the above proofs notwithstanding, remains either practically or theoretically impossible because despite $\widehat{PC^{\star}}$ and the ability of persons armed with it to verify or reject all programs (as proofs), the following six familiar things still need to be trusted.

(1) The various algorithms used by the underlying operating system (e.g., algorithms for implementing file operations on disk in terms of inode schemes).
(2) The implementation of the underlying operating system (even if the aforementioned algorithms are logico-mathematically correct, their implementation might have bugs).
(3) The semantics of the programming language L in which $PC^{\star}$ is implemented (e.g., does type soundness[25] hold in L?).
(4) The implementation of the compiler for L (e.g., bugs might creep into the compiler even if the semantics are theoretically sound).
(5) The integrity of the particular machine — hardware, operating system, and compiler for L — in which $\widehat{PC^{\star}}$ is performed (e.g., we need to know that no malicious agents have tampered with the system).

[24]The use of '*mutatis mutandis*' obviously blocks any literal *identification* of the vulnerabilities afflicting human proof-/argument-checking with those afflicting verification of physicalized programs; all that is needed are conditionals; that should be clear from the structure of my above proof, which exploits simple chaining over material conditionals (viz., **(1)**–**(4)**). So, I presuppose no such premise as either of the following two:

- All instances of human proof-/argument-checking are instances of (physicalized) program verification.
- Human proof-/argument-checking is identical to (physicalized) program verification.

Instead, for readers insisting on distillation, my reasoning uses the conditional that *if Fetzer is right, then human proof-/argument-checking is theoretically possible and thus surmounts the aforementioned processes*, and the conditional that *if human proof-/argument-checking is theoretically possible and thus surmounts the aforementioned processes, then program verification is theoretically possible and likewise surmounts the aforementioned processes.*

[25]This may be a concept unfamiliar to some readers. However, readers are likely to at least be familiar with the concept of types in programming languages; e.g., *boolean*, *integer*, etc. In light of this, we can very roughly say that type soundness ensures that if a program e is of type t, and the execution of e produces some result r, then r will be of type t.

(6) Exotic causal factors — random hardware malfunctions, influence of cosmic rays, and so on.

But these items are not in the least obstacles to my brand of program verification: The first item doesn't need to be trusted, since the algorithms in question, if to be taken seriously, can themselves be re-expressed as proofs, and subjected to $\widehat{PC^\star}$. Items 2 through 4 can be dispensed with by simply implementing $PC^\star$ in silicon. That is, we can build one special-purpose hardware device whose sole function is to check formal proofs expressed in a particular proof calculus; and the design of that hardware can then be thoroughly, and easily, verified logico-mathematically.[26] In addition — and this addresses item 5 — we need then only maintain security for this one device. We are left, then, having to "worry" only about stray cosmic rays, and the like.

But this is an infinitesimally small amount of worry. In fact, the possibility of error here is larger than the possibility of a still-undetected error infecting the collective human assimilation and apparent certification of received results like printed versions of Gödel's completeness theorem (χ, above), or like the equation $2 + 4 \times 7$ we visited above. In other words, given that human persons clearly know, with near *certainty*,[27] propositions like χ, despite the fact that their knowledge requires all sorts of guarantees regarding what has happened and will happen in the physical, causal sensorimotor world (just as program verification in the physical world requires non-deduction-based assurance that what has happened in the physical world can be trusted), they can, via the structure of Figure 3, clearly know, with near *certainty*, whether or not a program is correct.[28] Or to put the point in even more stark terms: The amount of worry that remains is about the same as the worry that between the reading of an axiom π of Peano Arithmetic at time t_1 and assenting to it *immediately* thereafter at t_2, the reality is that at t', where $t_1 < t' < t_2$, unbeknownst to you, because a stray cosmic ray hit your neocortex, you actually forgot what you read and are confusing it with some other declarative proposition (such as that by the *cogito* you undeniably exist), and only *think* you are assenting to π at t_2.[29] If you don't worry about cosmic rays hitting your neocortex when you assent to a simple axiom (or when you check Gödel's deduction, etc.), then don't worry about program verification as I have prescribed it, and don't worry about Fetzer's uncontroversial point that in the realm of the

[26] Note that we are not in any way here verifying *arbitrary* hardware, but rather only one specific, fixed hardware device, and so the traditional bugaboos of hardware verification (e.g., cache coherence and protocol correctness) evaporate.

[27] Following Chisholm (1966), we can reserve absolute certainty for such things as the direct, unmediated-by-any-sensory-perception grasping of, say, $1 \neq 0$ or 'I seem to be thinking,' and hence move down one notch to what he calls *evident* propositions (for us, "near-certain" propositions). In Chisholm's scheme, the verification of programs (whether pure or physicalized), would be at the level of the evident, as would the verification of chains of reasoning encoded in physical media and offered by one agent to another (because sensorimotor activity is involved). The next notch down is 'beyond reasonable doubt,' and then down to 'probable' or 'likely,' and then to 'counterbalanced,' at which point corresponding negative levels begin. The bottom line, in Chisholmian terms, is that both program verification and argument verification are equally trustworthy, at just one "notch" below beliefs that are utterly indubitable.

[28] Ultimately, it would be nice to have a formalization, informed by axioms and associated theorems, setting out the relationship between the uncertainty of argument verification and the uncertainty that Fetzer has drawn our attention to: that of physicalized (= actual) programs. Alas, I have no such body of logico-mathematical content to provide. But those arguing that program verification isn't a theoretical possibility can't provide a formalization of uncertainty in either case, and the burden of proof would clearly be on them. That said, it does seem clear that proof design and verification in the proofs-as-programs paradigm is exceedingly close to rigorous argument design and verification in first-rate philosophy of computer science; and this favors my case, not the arguments from Fetzer and other skeptics.

[29] Here's yet another way to put the point: Fetzer might say:

> Suppose that a program is flawless and deductively verified: would that guarantee that, when it is executed by a physical machine, nothing, such as a power outage or an unanticipated failure of one or more of its components, could go wrong? Obviously not. QED

But this can be effortlessly shot down by parody, showing that then we would need to be unsure about Gödel's completeness theorem and indeed every other such thing:

> Suppose that a proof is flawless and deductively verified by a human: would that guarantee that nothing, such as a subtle quantifier fallacy or a tiny but profound typo, could be wrong with the proof, but missed by the humans due to glitches in their brains? Obviously not. QED

physical, *perfect* reliability cannot be had. Confidently go forward into the future of a software-driven world knowing, despite arguments by Fetzer and company, that computer programs can in principle be verified, completely, utterly, and unshakably.

Acknowledgements

I'm *profoundly* indebted to three anonymous referees for remarkably insightful comments and objections, and to the special-issue editors for sagacious guidance and preternatural patience.

References

Almeida, J., Frade, M., Pinto, J., and de Sousa, S. 2011. *Rigorous Software Development: An Introduction to Program Verification*. New York, NY: Spinger.

Andrews, Peter. 2002. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. New York, NY: Springer.

Arkoudas, K. 2000. *Denotational Proof Languages*, Ph.D. thesis, MIT.

Arkoudas, K. and Bringsjord, S. 2009. 'Propositional Attitudes and Causation', *International Journal of Software and Informatics*, **3**, 47–65, URL `http://kryten.mm.rpi.edu/PRICAI_w_sequentcalc_041709.pdf`.

Arkoudas, Konstantine and Bringsjord, Selmer. 2007. 'Computers, Justification, and Mathematical Knowledge', *Minds and Machines*, **17**, 185–202, URL `http://kryten.mm.rpi.edu/ka_sb_proofs_offprint.pdf`.

Asperti, Andrea, Geuvers, Herman, and Natarajan, Raja. 2009. 'Social Processes, Program Verification and All That', *Mathematical Structures in Computer Science*, **19**, 877–896.

Barwise, J. and Etchemendy, J. 1999. *Language, Proof, and Logic*. New York, NY: Seven Bridges.

Boolos, G. S., Burgess, J. P., and Jeffrey, R. C. 2003. *Computability and Logic (Fourth Edition)*. Cambridge, UK: Cambridge University Press.

Bringsjord, S., Govindarajulu, N.S., Ellis, S., McCarty, E., and Licato, J. 2014. 'Nuclear Deterrence and the Logic of Deliberative Mindreading', *Cognitive Systems Research*, **28**, 20–43, URL `http://kryten.mm.rpi.edu/SB_NSG_SE_EM_JL_nuclear_mindreading_062313.pdf`.

Bringsjord, Selmer. 2008. 'Declarative/Logic-Based Cognitive Modeling'. in *The Handbook of Computational Psychology,* , edited by Ron SunCambridge, UK: Cambridge University Press, 127–169. URL `http://kryten.mm.rpi.edu/sb_lccm_ab-toc_031607.pdf`.

Bringsjord, Selmer and Govindarajulu, Naveen Sundar. 2013. 'Toward a Modern Geography of Minds, Machines, and Math'. in *Philosophy and Theory of Artificial Intelligence, Studies in Applied Philosophy, Epistemology and Rational Ethics*, vol. 5, edited by Vincent C. M¨llerNew York, NY: Springer, 151–165. URL `http://www.springerlink.com/content/hg712w4l23523xw5`.

Bringsjord, Selmer, Taylor, Joshua, Shilliday, Andrew, Clark, Micah, and Arkoudas, Konstantine. 2008. 'Slate: An Argument-Centered Intelligent Assistant to Human Reasoners'. in *Proceedings of the 8th International Workshop on Computational Models of Natural Argument (CMNA 8),* , edited by Floriana Grasso, Nancy Green, Rodger Kibble, and Chris Reed1–10. Patras, Greece: University of Patras. URL `http://kryten.mm.rpi.edu/Bringsjord_etal_Slate_cmna_crc_061708.pdf`.

Chalmers, David. 1996. 'Does a Rock Implement Every Finite-State Automaton?', *Synthese*, **108**, 309–333.

Chisholm, R. 1966. *Theory of Knowledge*. Englewood Cliffs, NJ: Prentice-Hall.

Church, Alonzo. 1932. 'A Set of Postulates for the Foundation of Logic (Part I)', *Annals of Mathematics*, **33**, 346–366.

Church, Alonzo. 1933. 'A Set of Postulates for the Foundation of Logic (Part II)', *Annals of Mathematics*, **34**, 839–864.

Church, Alonzo. 1960. 'Application of Recursive Arithmetic to the Problem of Circuit Synthesis'. in *Summaries of Talks Presented at the Summer Institute for Symbolic Logic (1957),* Princeton, NJ: Institute for Defense Analyses, 3–50.

Colburn, Tiimothy, Fetzer, James, and Rankin, Terry, eds. . 1993. *Program Verification: Fundamental Issues in Computer Science*. Dordrecht, The Netherlands: Kluwer Academic Publishers.

De Millo, Richard, Lipton, Richard, and Perlis, Alan. 1979. 'Social Processes and Proofs of Theorems and Programs', *Communications of the ACM*, **22**, 271–280.

de Queiroz, R., de Oliveira, A., and Gabbay, D., eds. . 2011. *The Functional Interpretation of Logical Deduction; Advances in Logic **5***. London, UK: Imperial College Press.

Fetzer, James. 1988. 'Program Verification: The Very Idea', *Communications of the ACM*, **37**, 1048–1063.

Girard, Jean-Yves. 1989. *Proofs and Types*. Cambridge, UK: Cambridge University Press. Translated and with appendices by Paul Taylor and Yves Lafont. Some helpful corrections were made for the 1990 version from CUP. A 2003 web version is freely available.

Goldstine, Herman and von Neumann, John. 1947. 'Planning and Coding of Problems for an Electronic Computing Instrument', *IAS Reports*, This remarkable work is available online from the Institute for Advanced Study, at the URL provided here. Please note that this paper is Part II of a three-volume set. The first volume was devoted to a preliminary discussion, and the first author on it was Arthur Burks, joining Goldstine and von Neumann., URL `https://library.ias.edu/files/pdfs/ecp/planningcodingof0103inst.pdf`.

Hoare, C. 1969. 'An Axiomatic Basis for Computer Programming', *Communications of the ACM*, 576–580.

Howard, William. 1980. 'The Formulae-as-Types Notion of Construction'. in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism,* , edited by Jonathan Seldin and J. Hindley-Boston, MA: Academic Press, 479–490. Manuscript is originally from 1969.

Klein, Gerwin. 2009. 'Operating System Verification—An Overview', *Sādhanā*, **34, Part 1**, 27–69.

Landy, D. and Goldstone, R. 2007. 'The Alignment of Ordering and Space in Arithmetic Computation'. in *Proceedings of the 29th Annual Meeting of the Cognitive Science Society,* , edited by D. S. McNamara and J. G. TraftonMahwah, NJ: Lawrence Erlbaum.

Landy, D. and Goldstone, R. 2010. 'Proximity and Precedence in Arithmetic', *Quarterly Journal of Experimental Psychology*, **63**, 1953–68.

Lenzen, Wolfgang. 2004. 'Leibniz's Logic'. in *Handbook of the History of Logic,* , edited by Dov Gabbay, John Woods, and Akihiro KanamoriAmsterdam, The Netherlands: Elsevier, 1–83.

McCarthy, John. 1962. 'Towards a Mathematical Science of Computation'. in *Proceedings of IFIP Congress 62,* , edited by C. Popplewell21–28. Amsterdam, The Netherlands: Noth-Holland, Amsterdam.

Morris, F. L. and Jones, C. B. 1984. 'An early program proof by alan turing', *Annals of the History of Computing*, **6**, 139–143.

Post, Emil. 1936. 'Finite Combinatory Processes – Formulation 1', *Journal of Symbolic Logic*, **1**, 103–105.

Turing, A. 1949. 'Checking a Large Routine'. in *Report of a Conference on High Speed Automatic Calculating Machines,* Cambridge, UK: University Math Lab, 67–69.

Turing, A. M. 1937. 'On Computable Numbers with Applications to the *Entscheidungsproblem*', *Proceedings of the London Mathematical Society*, **42**, 230–265.

Turner, Raymond. 2012. 'Machines'. in *A Computable Universe: Understanding and Exploring Nature as Computation,* London, UK: Imperial College Press, 63–76.

Turner, Raymond. 2013. 'The Philosophy of Computer Science'. in *The Stanford Encyclopedia of Philosophy,* , edited by Edward ZaltaURL `http://plato.stanford.edu/entries/computer-science`.