THE JOHN MARSHALL
REVIEW OF INTELLECTUAL PROPERTY LAW

ABSTRACTION IN SOFTWARE PATENTS (AND HOW TO FIX IT)

ATHUL K. ACHARYA

ABSTRACT

Software has long posed a quandary for patent law. As many have observed, software is an abstract technology—but abstract ideas are supposedly ineligible for patenting. This Article explores just what that means, what it doesn't mean, and what might fix the problem of abstraction in software patents.

This Article offers two related ways to understand the abstract nature of software. First, computer science defines itself as a "science of abstraction," and that self-definition finds real doctrinal purchase. Second, software code is designed to be what the doctrine calls "functional"—to describe abstract results that can be executed on heterogenous hardware without regard to how the results are achieved. Because software is functional, claims to software must necessarily also be functional. But functional claiming is exactly what the doctrine forbids.

This Article also examines and refute a third reason some have offered: the idea that software algorithms are "just math." Algorithms involve math and can be described by math, but they are not themselves math. In fact, this Article proposes that the way to fix software patents is to *require* patentees to claim algorithms—*concrete* algorithms, written in pseudocode, just as they would communicate their invention to other programmers.

Cite as Athul K. Acharya, *Abstraction in Software Patents (and How to Fix it)*,
18 J. MARSHALL REV. INTELL. PROP. L. 364 (2019).

ABSTRACTION IN SOFTWARE PATENTS (AND HOW TO FIX IT)

ATHUL K. ACHARYA

ABSTRACTION IN SOFTWARE PATENTS (AND HOW TO FIX IT)

ATHUL K. ACHARYA*

I. INTRODUCTION

The abstract-ideas doctrine is a mess. Courts and commentators, defenders and detractors, all agree. It is a "morass,"[1] an "incoherent body of doctrine,"[2] a "foggy standard cloaked as a rule."[3] It is "unprincipled and vague";[4] it is "indeterminate and often leads to arbitrary results."[5] One judicial critic recently declared that the law "renders it near impossible to know with any certainty whether [an] invention is or is not patent eligible."[6] In fact, the former head of the Patent and Trademark Office has publicly called to scrap the whole thing.[7]

Much of the *sturm und drang* in the abstract-ideas doctrine—nearly all of it, in fact—is over software patents. The Supreme Court case that established the current doctrinal landscape, *Alice Corp. Pty. Ltd. v. CLS Bank Intern.*,[8] was a software-

* © Athul K. Acharya 2019. Impact Litigation Associate, BraunHagey & Borden LLP. I am indebted to Mark Lemley and Brian L. Frye for their helpful comments and suggestions on a draft of this paper.

[1] John M. Golden, Flook *Says One Thing,* Diehr *Says Another: A Need for Housecleaning in the Law of Patentable Subject Matter*, 82 GEO. WASH. L. REV. 1765, 1766 (2014); *see* Tun-Jen Chiang, *The Rules and Standards of Patentable Subject Matter*, 2010 WIS. L. REV. 1353, 1394 (2010) (describing the state of affairs as "doctrinal chaos"); Mark A. Lemley *et al.*, *Life after* Bilski, 63 STAN. L. REV. 1315, 1316 (2011) (asserting that "no one understands" it).

[2] Andrei Iancu, *Remarks by Director Iancu at the Intellectual Property Owners Association 46th Annual Meeting* (2018), https://www.uspto.gov/about-us/news-updates/remarks-director-iancu-intellectual-property-owners-46th-annual-meeting (quoting Interval Licensing LLC v. AOL, Inc., 896 F.3d 1335, 1348 (Fed. Cir. 2018) (Plager, J., concurring in part and dissenting in part)).

[3] Michael Risch, *Nothing is Patentable*, 67 FLA. L. REV. FORUM. 45, 45 (2015); *see* Robert Merges, *Symposium: Go ask Alice—what can you patent after* Alice v. CLS Bank*?*, SCOTUSBLOG (June 20, 2014, 12:04 PM), http://www.scotusblog.com/2014/06/symposium-go-ask-alice-what-can-you-patent-after-alice-v-cls-bank/ (describing the patent-eligibility analysis in *Alice Corp. Pty. Ltd. v. CLS Bank Int'l*, 573 U.S. 208 (2014), as "brief, yet somehow baroquely obscure").

[4] Peter S. Menell, *Forty Years of Wondering in the Wilderness and No Closer to the Promised Land:* Bilski*'s Superficial Textualism and the Missed Opportunity to Return Patent Law to Its Technology Mooring*, 63 STAN. L. REV. 1289, 1305 (2011).

[5] Smart Sys. Innovations, LLC v. Chicago Transit Auth., 873 F.3d 1364, 1377 (Fed. Cir. 2017) (Linn, J., concurring in part and dissenting in part).

[6] *Interval Licensing*, 896 F.3d at 1348 (Plager, J., concurring in part and dissenting in part); *see also* Berkheimer v. HP Inc., 890 F.3d 1369, 1377 (Fed. Cir. 2018) (Reyna, J., dissenting from denial of rehearing en banc) (protesting that the court's recent decisions "offer[] no meaningful guidance to the bar, the government, or the public"); *see also* Kevin Emerson Collins, Bilski *and the Ambiguity of "An Unpatentable Abstract Idea,"* 15 LEWIS & CLARK L. REV. 37, 39 (2011) (arguing that patent-eligibility is an "'I know it when I see it' jurisprudence").

[7] Ryan Davis, *Kappos Calls for Abolition of Section 101 of Patent Act*, LAW360 (Apr. 12, 2016), http://www.law360.com/articles/783604/kappos-calls-for-abolition-of-section-101-of-patent-act (describing the doctrine as "a real mess, and you could actually use much stronger language than that").

[8] 573 U.S. 208 (2014).

patent case.[9] In one study, 1,274 out of 1,522 patents challenged under *Alice*, or 83.7 percent, were software patents.[10] As the Federal Circuit has noted, "[c]omputer software-related inventions—due to their intangible nature—can be particularly difficult to assess under the abstract idea exception."[11]

Some judges of the Federal Circuit would go much further. In one judge's view, "claims directed to software implemented on a generic computer are categorically not eligible for patent."[12] The court itself has rejected such a broad view of subject-matter exclusions to patentability.[13] But there is an extensive literature going back several decades debating whether software should be patentable at all.[14]

The question is more than just academic. The facts of software patenting are grim: More than half of patents issued are software patents;[15] they tend to stake extremely broad claims, creating "thickets" of overlapping rights that deter follow-on innovation;[16] they account for most of the increase in patent litigation in recent years, mostly by patent trolls;[17] and patent trolling had cost the economy around $500 billion by 2010, and surely much more since.[18]

I do not argue in this Article that software should be categorically ineligible for patenting. Whatever the merits of that position, the law as it stands permits patents on software and neither the Supreme Court nor the Federal Circuit has much

[9] *Id.* at 2352 (explaining that the patent at issue required "generic computer implementation" of an abstract idea).

[10] *Alice* Through the Looking Glass 8 (2018) (on file with author), DOCKET NAVIGATOR, http://brochure.docketnavigator.com/alice/.

[11] Interval Licensing LLC v. AOL, Inc., 896 F.3d 1335, 1343 (Fed. Cir. 2018).

[12] Intellectual Ventures I LLC v. Symantec Corp., 838 F.3d 1307, 1322 (Fed. Cir. 2016) (Mayer, J., concurring).

[13] *See Interval Licensing*, 896 F.3d at 1344 (reasoning that "[s]oftware can make non-abstract improvements to computer technology just as hardware improvements can" (quoting Enfish LLC v. Microsoft Corp., 822 F.3d 1327, 1335 (Fed. Cir. 2016))).

[14] For just a few examples, *see, e.g.,* Jonathan Stroud & Derek M. Kim, *Debugging Software Patents After Alice*, 69 S.C.L. REV. 177 (2017); Lemley *et al.*, *supra* note 1, at 1326–27; Michael Risch, *Everything Is Patentable*, 75 TENN. L. REV. 591, 622 (2008); Robert A. Kreiss, *Patent Protection for Computer Programs and Mathematical Algorithms: The Constitutional Limitations on Patentable Subject Matter*, 29 N.M.L. REV. 31 (1999); GREGORY A. STOBBS, SOFTWARE PATENTS (1995); Jur Strobos, *Stalking the Elusive Patentable Software: Are There Still* Diehr *or Was It Just a* Flook*?*, 6 HARV. J.L. & TECH. 363 (1993); Pamela Samuelson, Benson *Revisited: The Case Against Patent Protection for Algorithms and Other Computer Program-Related Inventions*, 39 EMORY L.J. 1025 (1990); Donald S. Chisum, *The Patentability of Algorithms*, 47 U. PITT. L. REV. 959 (1986).

[15] Raymond Millien, Alice *Who? Over Half the U.S. Utility Patents Issued Annually are Software Related!*, IPWATCHDOG (May 21, 2017), https://www.ipwatchdog.com/2017/05/21/alice-over-half-u-s-utility-patents-issued-annually-software/id=83367/.

[16] Mark A. Lemley, *Software Patents and the Return of Functional Claiming*, 2013 WIS. L. REV. 905, 906–07, 929.

[17] Stroud & Kim, *supra* note 14, at 180 (citing U.S. GOV'T ACCOUNTABILITY OFF., GAO-16-490, INTELLECTUAL PROPERTY: PATENT OFFICE SHOULD DEFINE QUALITY, REASSES INCENTIVES, AND IMPROVE CLARITY 14, 20 (2016)); John R. Allison et al., *Patent Quality and Settlement among Repeat Patent Litigants*, 99 GEO. L.J. 677, 708 (2011) (finding that patent trolls assert 64.3 percent of the most-litigated patents).

[18] Michael Meurer, James Bessen & Jennifer Ford, *The Private and Social Costs of Patent Trolls*, 34 REGULATION 26, 26 (2012).

appetite to change that.[19] My inquiry here is more descriptive: Just *why* does software pose such a challenge for patent-eligibility and the abstract-ideas doctrine? What makes software inventions different from, say, mechanical inventions or chemical inventions? And, perhaps most importantly, is there a way to fix software patents?

To answer these questions, it's necessary to start with a quick tour of the doctrine itself—what it means for a patent claim to be impermissibly abstract. That discussion occupies Part I of this Article. The doctrine is not entirely clear, to say the least, but at least one strong emergent theme is that functional claiming—which, in patent law, means claiming a bald *result* rather than how to achieve it—is impermissible.[20] More generally, the doctrine treats "abstract" as an antonym of "concrete" or "specific."[21]

Part II explores one reason software claims run afoul of the doctrine: computer science is, by its own self-description, "a science of abstraction."[22] Programmers model real-world concepts and entities in software by abstracting away irrelevant details and leaving only those aspects necessary to manipulate them. Many software programs, in other words, are just amalgamations of abstractions. A claim to such an invention will naturally be prone to abstractness itself. The worst offenders are mere "do it on a computer" claims, which describe some longstanding real-world practice, add a computer, and seek monopoly rents.

But many software innovations don't involve real-world concepts at all. Consider, for instance, a new programming language or a better compression algorithm—innovations that improve processes and products intrinsic to computers. Claims to such innovations, too, often turn out to be impermissibly abstract. That's because programming a computer is, by design, a functional exercise—at one level of abstraction or another, programmers tell the computer *what to do*, not *how to do it*.[23] And if the invention is functionally defined, it must necessarily be claimed in functional terms as well. I explore this second and perhaps more fundamental reason that software claims tend to be abstract in Part III.

In Part IV, I explore and reject a common objection to software patents: that software consists of algorithms, and algorithms are just math. That might mean one of two things—that algorithms *solve* mathematical problems, or that algorithms

---

[19] *See, e.g., Alice*, 573 U.S. 208, 223 (2014) (reaffirming that *Diamond v. Diehr*, 450 U.S. 175 (1981), which permitted software claimed in an industrial process, is good law); Finjan, Inc. v. Blue Coat Sys., Inc., 879 F.3d 1299, 1304 (Fed. Cir. 2018).

[20] *See, e.g.,* Elec. Power Grp., LLC v. Alstom S.A., 830 F.3d 1350, 1356 (Fed. Cir. 2016) (explaining that "the essentially result-focused, functional character of claim language has been a frequent feature of claims held ineligible under § 101"). "Functional" can mean different things in different IP contexts—from the perspective of copyright law, for instance, patents are *supposed* to claim functionality, which copyright excludes. *See* Christopher Buccafuso & Mark A. Lemley, *Functionality Screens*, 103 VA. L. REV. 1293, 1300, 1307 (2017). But from the internal perspective of patent law, patentees have to claim the *way* the function is performed, and not merely the idea of performing it. *See* Kevin Emerson Collins, *Patent Law's Authorship Screen*, 84 U. CHI. L. REV. 1603, 1614 & n.43 (2017) (explaining that copyright screens out even—or especially—what patent "opt[s] not to protect").

[21] Epic IP LLC v Backblaze, Inc., 2018 WL 6107029, at *3 (D. Del. Nov. 21, 2018) (Bryson, J.).

[22] ALFRED V. AHO, JEFFREY D. ULLMAN, FOUNDATIONS OF COMPUTER SCIENCE: C EDITION 1 (1994).

[23] *See infra* notes 73–79 and accompanying text.

involve mathematical insights. Neither is a meaningful reason to deny patent protection to algorithms. Even so, something like this objection underlies the Supreme Court's very first software-patent case, *Gottschalk v. Benson*.[24]

In fact, software patents that claimed actual algorithms, rather than vague results, would be an enormous improvement over most software patents today.[25] And they *can* be claimed concretely, using what programmers call "pseudocode"—an informal but precise language that programmers use to communicate algorithms with each other. I therefore close by proposing that *Benson* be overruled, and that software patents instead be *required* to claim an algorithm in pseudocode. That would go a long way towards fixing software patenting—and bringing order to the abstract-ideas doctrine.

II. THE ABSTRACT-IDEAS DOCTRINE: A QUICK TOUR

Section 101 of the Patent Act, which governs what sorts of inventions may be patented, affords a "wide and permissive scope" for patent-eligibility.[26] By its terms, an inventor may patent any new and useful "process, machine, manufacture, or composition of matter."[27] But from the earliest days of the patent system, the Supreme Court has held that certain types of subject matter are ineligible for patenting.[28] The modern formulation is "[l]aws of nature, natural phenomena, and abstract ideas"[29]—these things cannot be patented even if they fit within one of the statutory categories.[30]

Of course, all inventions harness natural phenomena, all inventions are governed by natural laws, and all inventions begin with an idea.[31] To separate out permissible invention from impermissible abstraction, the Supreme Court in *Alice* announced a two-part test: First, courts must examine whether the claim is "directed to a patent-ineligible concept"; and second, if so, they must determine whether it contains an "inventive concept" that "transform[s]" it into a "patent-eligible application."[32]

---

[24] 409 U.S. 63 (1972).

[25] Lemley, *supra* note 16, at 947.

[26] Smart Sys. Innovations, LLC v. Chicago Transit Auth., 873 F.3d 1364, 1377 (Fed. Cir. 2017) (Linn, J., concurring in part and dissenting in part); *see* Lemley, *supra* note 1, at 1328 ("[B]ecause patent claims almost never fall *outside* of the four fundamental categories of § 101, when they do it is noteworthy.").

[27] 35 U.S.C. § 101 (2012).

[28] *See, e.g.,* Le Roy v. Tatham, 55 U.S. (14 How.) 156, 175 (1852).

[29] Alice Corp. Pty. Ltd. v. CLS Bank Intern., 573 U.S. 208, 216 (2014) (quoting Ass'n for Molecular Pathology v. Myriad Genetics, Inc., 569 U.S. 576, 589 (2013)). The tripartite distinction is of relatively recent vintage and has changed even over that short time. It appeared first in *Gottschalk v. Benson* as "[p]henomena of nature, . . . mental processes, and abstract intellectual concepts." 409 U.S. 63, 67 (1972). As recently as *Bilski v. Kappos*, the Court used "physical phenomena" instead of "natural phenomena," 561 U.S. 593, 601 (2010) (quotation marks omitted), but the formulation appears to have stabilized as of *Myriad* and *Alice*.

[30] Lemley *et al.*, *supra* note 1, at 1328 (observing that these exclusions operate "irrespective of [a claim's] categorical status").

[31] Mayo Collaborative Servs. v. Prometheus Labs., Inc., 566 U.S. 66, 71 (2012).

[32] *Alice*, 573 U.S. at 217, 221 (quoting *Mayo*, 566 U.S. at 72, 79).

The first step acts as something of a high-pass filter: Claims that obviously describe something concrete, like "a chip architecture [or] an LED display," need not be analyzed further.[33] Software claims—even those ultimately held valid—often fail this first step.[34] The second step is harder to pin down. It is most often defined in negative terms: a claim directed to an abstract idea must contain something "more than 'well-understood, routine, conventional activity.'"[35] And somewhat orthogonal to both these steps is the "underlying concern" that a patent not inordinately "preempt"—that is, inhibit or prevent by laying claim to—too much follow-on innovation.[36]

Alice arose in the context of a common sort of software claim—"do it on a computer" claims.[37] The patents at issue in *Alice* took an ordinary business method—intermediated settlement—and simply claimed performing it on a computer.[38] The claims at issue in *Bilski* were similar.[39] Five years after *Alice*, there is little doubt that such claims are ineligible for patent. In *Alice*'s terms, they are directed to the idea of performing the ordinary business method, and they claim no inventive concept simply by performing it on an ordinary computer.[40] Courts have since expanded this strand of the doctrine to encompass not only business methods but any "method of organizing human activity."[41]

A related strand of the abstract-ideas doctrine is the bar on functional claiming.[42] Claims that "simply demand[] the production of a desired result . . . without any limitation on how to produce that result" are directed to the

[33] *See* Enfish LLC v. Microsoft Corp., 822 F.3d 1327, 1335 (Fed. Cir. 2016).

[34] *See, e.g.,* Berkheimer v. HP Inc., 881 F.3d 1360, 1369 (Fed. Cir. 2018); Bascom Global Internet Servs., Inc. v. AT&T Mobility LLC, 827 F.3d 1341, 1353 (Fed. Cir. 2016); DDR Holdings, LLC v. Hotels.com, L.P., 773 F.3d 1245, 1257 (Fed. Cir. 2014); *but see, e.g., Enfish*, 822 F.3d at 1337; McRO Inc. v. Bandai Namco Games Am., 837 F.3d 1299, 1313 (Fed. Cir. 2016).

[35] Affinity Labs of Texas, LLC v. DirecTV, LLC, 838 F.3d 1253, 1262 (Fed. Cir. 2016) (quoting *Mayo*, 566 U.S. at 79).

[36] *See Mayo*, 566 U.S. at 82; Lemley *et al.*, *supra* note 1, at 1329 (explaining that exclusions to patent-eligibility aim to "prevent[] patentees from claiming broad ownership over fields of exploration rather than specific applications of those fields").

[37] *Alice*, 134 S. Ct. at 2358 (holding that claiming an abstract idea "while adding the words 'apply it with a computer'" is not enough to be eligible for patenting); *see* Apple, Inc. v. Ameranth, Inc., 842 F.3d 1229, 1243 (Fed. Cir. 2016) ("It is not enough to point to conventional applications and say 'do it on a computer.'").

[38] *Alice*, 573 U.S. at 223.

[39] Bilski v. Kappos, 561 U.S. 593, 611 (2010).

[40] *See, e.g.,* Mortgage Grader, Inc. v. First Choice Loan Servs. Inc., 811 F.3d 1314, 1324–25 (Fed. Cir. 2016); Versata Development Group v. SAP America, Inc., 793 F.3d 1306, 1333–34 (Fed. Cir. 2015); Intellectual Ventures I LLC v. Capital One Bank (USA), 792 F.3d 1363, 1367–69 (Fed. Cir. 2015).

[41] *See, e.g., In re* Marco Guldenaar Holding B.V., No. 2017-2465, slip op. at 5 (Fed. Cir. Dec. 28, 2018); Bascom Global Internet Servs., Inc. v. AT&T Mobility LLC, 827 F.3d 1341, 1348 (Fed. Cir. 2016); Intellectual Ventures I LLC v. Capital One Bank (USA), 792 F.3d 1363, 1367 (Fed. Cir. 2015).

[42] In theory, Congress addressed functional claiming in the 1952 Patent Act with a compromise: "patentees could write their claim language in functional terms, but when they did so the patent would not cover the goal itself, but only the particular means of implementing that goal described by the patentee and equivalents thereof." Lemley, *supra* note 16, at 907 (citing 35 U.S.C. § 112(f)). In practice, however, § 112(f) rarely applies unless a claim is written in expressly functional terms—"means for" and "step for" performing a given function. Williamson v. Citrix Online, LLC, 792 F.3d 1339, 1347–49 (Fed. Cir. 2015). In some sense, the abstract-ideas doctrine has stepped in to fill the gap.

abstract idea of performing that result.[43] And, as with do-it-on-a-computer claims, simply claiming that result performed on an ordinary computer or an ordinary network is not inventive.[44]

Instead, what a patent must claim is "*how* the desired result is achieved."[45] This sounds in the principle against disproportionate preemption: Claims that focus on a bald result "capture[] ownership not of what [the patentee] built, but of anything that achieves the same goal, no matter how different it is."[46] That is, they preempt all follow-on and design-around invention. The bar on functional claiming is especially relevant to software patents, because "100% of troll software patents and 50% of non-troll software patents use functional claiming."[47]

There are other strands of the doctrine, though they are less important for our purposes. Claims to simply manipulating and reporting information are also often held abstract.[48] This principle has obvious ramifications for many software patents, but is also not infrequently honored in the breach.[49] Along similar lines, mental steps or steps that can be performed with pencil and paper are abstract, as are mathematical algorithms.[50] I will have more to say about algorithms later in Part V, but by and large, these other strands of reasoning largely buttress results that can be reached with the bar against functional claiming alone.

III. Computer Science: The Science of Abstraction

Why are software claims prone to abstractness? The Federal Circuit's explanation is that software, by its very nature, is intangible.[51] But so are all process claims, which are clearly permitted by the statute.[52] Another explanation is that "software is an abstract technology."[53] This type of blanket assertion, with little explanation, is common.[54] But what does it mean?

[43] *See, e.g.,* Interval Licensing LLC v. AOL, Inc., 896 F.3d 1335, 1345 (Fed. Cir. 2018); Two-Way Media Ltd. v. Comcast Cable Commc'ns, LLC, 874 F.3d 1329, 1337 (Fed. Cir. 2017).

[44] *Interval Licensing*, 896 F.3d at 1346; Apple, Inc. v. Ameranth, Inc., 842 F.3d 1229, 1244 (Fed. Cir. 2016).

[45] Elec. Power Grp., LLC v. Alstom S.A., 830 F.3d 1350, 1355 (Fed. Cir. 2016); *see Interval Licensing*, 896 F.3d at 1338.

[46] Lemley, *supra* note 16, at 908.

[47] *Id.* at 920 n.65.

[48] *See Interval Licensing*, 896 F.3d at 1344–45 (collecting cases); SAP Am., Inc. v. Investpic, LLC, 898 F.3d 1161, 1167 (Fed. Cir. 2018).

[49] *See, e.g.,* SRI Int'l, Inc. v. Cisco Sys., Inc., No. 2017-2223 (Fed. Cir. Mar. 20, 2019), slip op. at 9–10; *id.* at 3 (Lourie, J., dissenting) ("This case is hardly distinguishable from *Electric Power Group*."); Core Wireless Licensing S.A.R.L. v. LG Elecs., Inc., 880 F.3d 1356, 1362–63 (Fed. Cir. 2018).

[50] Synopsys, Inc. v. Mentor Graphics Corp., 839 F.3d 1138, 1145 (Fed. Cir. 2016); Intellectual Ventures I LLC v. Capital One Bank (USA), 792 F.3d 1363, 1368 (Fed. Cir. 2015); DDR Holdings, LLC v. Hotels.com, L.P., 773 F.3d 1245, 1256 (Fed. Cir. 2014).

[51] Interval Licensing LLC v. AOL, Inc., 896 F.3d 1335, 1343 (Fed. Cir. 2018).

[52] 35 U.S.C. § 101 (2012).

[53] James Bessen & Michael J. Meurer, Patent Failure: How Judges, Bureaucrats, and Lawyers Put Innovators at Risk 187 (2008) (emphasis removed).

[54] *See, e.g.,* Stroud & Kim, *supra* note 14, at 205 (observing that "software patents, by their very nature, generally have broad, ambiguous claims"); Note, *Everlasting Software*, 125 Harv. L. Rev.

An important clue is found in Alfred Aho and Jeffrey Ullman's classic introductory computer-science textbook, *Foundations of Computer Science: C Edition*.[55] The very first chapter is titled "Computer Science: The Mechanization of Abstraction."[56] Its explanation of what that means is worth quoting in full:

> [F]undamentally, computer science is a science of abstraction—creating the right model for thinking about a problem and devising the appropriate mechanizable techniques to solve it. Every other science deals with the universe as it is. The physicist's job, for example, is to understand how the world works, not to invent a world in which physical laws would be simpler or more pleasant to follow. Computer scientists, on the other hand, must create abstractions of real-world problems that can be understood by computer users and, at the same time, that can be represented and manipulated inside a computer.[57]

In other words, software by its very nature deals in representations several times removed from physical reality.[58] For one example, Aho and Ullman describe an algorithm for scheduling a university's final exams.[59] The only detail relevant to whether two exams can be scheduled at the same time is whether they have a student in common.[60] Thus, that's all the algorithm uses to model each class—the binary relationships, student in common or not, between it and every other class.[61] Importantly, the algorithm *abstracts away* all other details, like how *many* students each pair of classes has in common, who the students are, how many credits the class is worth, or when the class itself is scheduled.[62] Those are real attributes of the class in the real world—what programmers call "meatspace"—but they don't matter for solving the exam-scheduling problem.

Or consider an example more immediately relevant to your author: word-processing programs. Microsoft Word represents a document as a window into a page or so of text, albeit with none of the physicality of paper and ink (let alone the rest of the document). That's because none of those things matter to solving the

1454, 1459–60 (2012). Bessen and Meurer do explain that "many of the standard terms of art are themselves abstract ideas that are meant to apply to a wide variety of possible applications," BESSEN & MEURER, *supra* note 53, at 195, but that seems circular and is not even necessarily true. *See* Lemley, *supra* note 16, at 930 (explaining that software doesn't really have standard terms of art the way chemistry or biotechnology do).

[55] AHO & ULLMAN, *supra* note 22.

[56] *Id.* at 1.

[57] *Id.*

[58] Aho and Ullman are far from alone in observing that abstraction is fundamental to computer science. *See also, e.g.,* Jacob C. Perrenet, *Levels of Thinking in Computer Science: Development in Bachelor Students' Conceptualization of Algorithm*, 15 EDUC. & INFO. TECHS. 87, 88–89 (2010) (summarizing many theories of abstraction in computer science from different computer scientists and mathematicians); Peter J. Denning *et al.*, *Computing as a Discipline*, 32 COMM. OF THE ACM 9, 10 (1989) ("[C]omputer science focuses on analysis and abstraction; computer engineering on abstraction and design.").

[59] AHO & ULLMAN, *supra* note 22, at 1.

[60] *See id.* at 1–3.

[61] *See id.*

[62] *See id.*

problem at hand—showing me what I'm writing so I can edit it. So, as with the exam-scheduling software, the software abstracts them away.

Now consider what a claim on either of these examples would look like. It would recite a "document" or a "class," but those words would refer not to real-world documents or classes but to software models of them—abstractions. The same goes for real-world software patents, with words like "media product" and "sponsor message,"[63] "credit card" and "debit card,"[64] or "mail object" and mail "sender."[65] Those words in those claims refer to software abstractions, not entities in the real world. On top of that, patent claims are *meant* to abstract out irrelevant details of specific embodiments and retain only the key aspects that make the claim novel and nonobvious.[66] If you invent an improved car seat, and the improvement is in how the headrest responds to a rear-end collision, you don't need to specify in the claims whether the seat is covered with cloth or leather.[67] You can abstract away that irrelevant detail. But if you invent an abstract model of the real world, and then you abstract away even more of it in the claims, you practically have a recipe for irredeemable abstractness.

That largely explains why do-it-on-a-computer software claims are abstract (and, after *Alice*, categorically ineligible for patenting). But that's far from the only type of software innovation (if indeed manipulating models of real-world objects is innovative at all). There are also innovations like webpages and memory systems, which are wholly intrinsic to computers and model nothing in the real world; innovations like faster algorithms for sorting and searching data, whose value lies not in manipulating particular abstractions but in *how* they are manipulated; and innovations like a more compact way to store video data, which improve the *way* a real-world entity is modeled. Doctrinally, *Alice* marks this distinction as the line between do-it-on-a-computer claims and inventions that "improve[] an existing technological process" or "improve the functioning of the computer itself."[68]

But claims in that latter class, too, are often irredeemably abstract.

---

[63] U.S. Patent 7,346,545, held ineligible in Ultramercial, Inc. v. Hulu, LLC, 772 F.3d 709 (Fed. Cir. 2014).

[64] U.S. Patents Nos. 7,566,003, 7,568,617, 8,505,816, and 8,662,390, held ineligible in Smart Sys. Innovations, LLC v. Chicago Transit Auth., 873 F.3d 1364 (Fed. Cir. 2017).

[65] U.S. Patents Nos. 7,814,032, 7,818,268, 8,073,787, 8,260,629, 8,429,093, 8,910,860, and 9,105,002, held ineligible in Secured Mail Sols. LLC v. Universal Wilde, Inc., 873 F.3d 905 (Fed. Cir. 2017).

[66] Golden, *supra* note 1, at 1766–70; *see also* Risch, *supra* note 3, at 53 ("Practically speaking, . . . [e]very invention will look like an abstract idea or natural phenomenon at some level.").

[67] *See, e.g.,* U.S. Patent No. 6,017,086.

[68] Alice Corp. Pty. Ltd. v. CLS Bank Intern., 134 S. Ct. 2347, 2358, 2359 (2014); *see, e.g.,* Enfish LLC v. Microsoft Corp., 822 F.3d 1327, 1338 (Fed. Cir. 2016) (reasoning that even though the claimed invention "r[an] on a general-purpose computer," it was patent-eligible because it was "directed to an improvement in the functioning of a computer"); DDR Holdings, LLC v. Hotels.com, L.P., 773 F.3d 1245, 1257 (Fed. Cir. 2014) (holding claims patent-eligible because they were "necessarily rooted in computer technology in order to overcome a problem specifically arising in the realm of computer networks").

IV. SOFTWARE: IT'S FUNCTIONAL ALL THE WAY DOWN

The abstract nature of computer science doesn't end at modeling real-world things like documents and bank accounts. To the programmer, *the computer itself* is abstracted away. The programmer doesn't direct the flow of the electrons through the silicon; she doesn't manipulate transistors, gates, circuits, or registers; by and large, she doesn't know anything about the machine on which her code will run.[69] No—she merely directs the computer to accomplish a certain result.

Consider this simple snippet of code, the classic first program one writes when learning the C programming language[70]:

```
int main(void) {
   printf("Hello, world!");
}
```

This program displays "Hello, world!" on the screen. Don't sweat the details—the first and last lines merely designate where to start and end execution. All the magic is in the second line. But that line doesn't *do* any displaying, exactly. Before the program can be run, it must be translated into executable form by another program, known as a *compiler*.[71] So all that second line does is tell the compiler the *result*—displaying "Hello, world!"—that the programmer would like to produce.[72]

Importantly, the programmer has no idea *how* the desired result is produced. That ignorance is not happenstance or a contingent fact. *By design*, the programmer describes results and the compiler translates her description into machine-language commands that do what she wants done.[73] This design allows the same code to be compiled and run on dozens of architectures, generating wildly different machine code for each,[74] yet ultimately displaying "Hello, world!" on them all. It's a step toward accomplishing one of the most atavistic aspirations of computer science—"write once, run anywhere."[75]

[69] There are exceptions, of course. It is possible, if rare, to write code directly in assembly language—ostensibly the language of the processor itself. But modern processors rewrite that too, so that even machine code becomes nothing more than commands to the system to effect a certain result.

[70] *See "Hello, World!" program*, WIKIPEDIA, https://en.wikipedia.org/w/index.php?title=%22Hello,_World!%22_program&oldid=879818585 (last visited Jan. 23, 2019).

[71] ALFRED V. AHO, MONICA S. LAM, RAVI SETHI, & JEFFREY D. ULLMAN, COMPILERS: PRINCIPLES, TECHNIQUES, & TOOLS 1 (2d ed. 2007) ("[B]efore a program can be run, it must first be translated into a form in which it can be executed by a computer . . . . The software systems that do this translation are called *compilers*.").

[72] *See id.* ("Programming languages are notations for *describing* computations to people and to machines." (emphasis added)).

[73] *See id.* at 1–3.

[74] You can see this in action at the excellent *Compiler Explorer* resource. The x86-64 compiler output of the above code is available at https://godbolt.org/z/CvUtAu, and the arm64 version is available at https://godbolt.org/z/PvdwxY.

[75]*See Write once, run anywhere*, WIKIPEDIA, https://en.wikipedia.org/wiki/Write_once,_run_anywhere (last visited Aug. 1, 2018); *JavaSoft Ships Java 1.0*, TECH INSIDER, https://tech-insider.org/java/research/1996/0123.html (last visited July 25, 2019). The phrase originated in the mid-1990s with the release of the Java programming language, but the idea dates back at least to the 1970s. In computer science, the phrase typically refers to a

What's more, code has this property at every level of abstraction.[76] Drill down to the very bottom layer of software—the machine-language loads, moves, compares, and other operations[77]—and all you'll see are commands to the hardware to accomplish a result (albeit a small one) by whatever means are available. To be sure, these instructions cause a "series of gates in a computer chip to open and close in a particular sequence," ultimately creating the desired effect.[78] But there's no physical connection between the instruction and the result, as evinced by the fact that the same instructions can accomplish the same result on a 1985 Intel 80386 and a 2018 Intel Core i7-8086K, even though they have radically different architectures. Code is thus intrinsically functional. At every level of abstraction, it describes results, and by design, it doesn't much care "by what process or machinery the result is accomplished."[79] It's functional all the way down.[80]

Compare this to patent law's "archetypal subject matter," the mechanical and chemical arts.[81] There, claim language typically refers to physical structure—machinery and chemicals.[82] That makes it relatively easy to tell when claim language is referring to functionality in the abstract—just look for a distinct absence of physical referents.[83] Software programs, however, are not physical machines but "virtual" ones; they're machines constructed "in the medium of text"—*i.e.*, descriptive code.[84] So no matter at what level of abstraction a software patentee claims her invention, the referents of her claim language will still be functional, not physical.[85] Little wonder, then, that software claims often have an "essentially result-focused,

specific type of platform independence involving a virtual machine that abstracts away the hardware entirely, *see* AHO, *supra* note 71, at 2–3, 507–08, but it can describe compiling to different hardware targets as well.

[76] Kip Werking, *The Illogic of the Algorithm Requirement for Software Patent Claims*, IPWATCHDOG (Oct. 12, 2012, 7:20 AM), http://www.ipwatchdog.com/2012/10/12/the-illogic-of-the-algorithm-requirement-for-software-patent-claims/id=28635/. Werking's solution is to submit to unbridled functionalism in software claims, which I resist for the reasons described in Part V.B.

[77] AHO, *supra* note 71, at 512–13 (describing some basic machine-language instructions present on most platforms).

[78] *See* Lemley, *supra* note 16, at 969.

[79] *Cf.* O'Reilly v. Morse, 56 U.S. (15 How.) 62, 113 (1854). Of course, sometimes—as in device drivers—code does care about the machinery involved. But if a patent claim involves a device driver, typically, it also involves the device, thus claiming hardware as well as function.

[80] *See* Kevin Emerson Collins, *Patent Law's Functionality Malfunction and the Problem of Overbroad, Functional Software Patents*, 90 WASH. U.L. REV. 1399, 1402 (2013).

[81] *See* Mark P. McKenna & Christopher Jon Sprigman, *What's In, and What's Out: How IP's Boundary Rules Shape Innovation*, 30 HARV. J.L. & TECH. 491, 502 (2017).

[82] *See id.*

[83] *Id.* at 509. Even process claims "prototypically refer[] to industrial processes, like methods for manufacturing a drug." *Id.*

[84] *See* Pamela Samuelson, *Functionality and Expression in Computer Programs: Refining the Tests for Software Copyright Infringement*, 31 BERKELEY TECH. L.J. 1215, 1273 (2017) (quoting Pamela Samuelson, et al., *A Manifesto Concerning the Legal Protection of Computer Programs*, 94 COLUM. L. REV. 2308, 2316 (1994)).

[85] Lemley, *supra* note 16, at 960. Professor Lemley argues that even physical objects are often characterized by their functions—consider the "seat," the "jackhammer," or even the screwdriver. *Id.* But even though we *label* these physical objects using functional terms, we all understand that the word "screwdriver" refers to a particular class of physical tools that drive a screw by fitting a particularly shaped driver bit into a matching screw head, and not in any other way.

functional character."[86] Just like software *code*, software *claims* are intrinsically functional.

The upshot is that abstractness in software claims is not a question of kind. Software claims cannot be categorized as "functional" or "not functional." Software claims are always functional. But they *can* be made concrete enough for patenting by adding in enough implementation detail.[87] How much is enough? That's the 500-billion-dollar question.[88]

V. ALGORITHMS: NOT THE PROBLEM, BUT THE SOLUTION

One common objection to software patents is that software is "just math."[89] This is an almost *metaphysical* objection—that allowing software to be patented is some sort of category mistake. And it has intuitive appeal. Software is made up of algorithms, and students are first introduced to that word in their mathematics classes. What's more, the word "algorithm" comes from the Latinized name of a 9th century mathematician, Muḥammad ibn Musa *al-Khwarizmi*.[90] And computer science courses on algorithms are also generally math-intensive.[91] If nothing else, software and mathematics are deeply related.

But look past the intuition, and the claim that algorithms are "just math" doesn't hold water. It might conceivably mean one of two things: that algorithms solve mathematical problems, or that algorithms involve math. Both are true to some extent, but neither is a good reason to deny patents to algorithms. In its very first software-patent case, however, the Supreme Court embraced that notion, and software patentees have been scared off claiming algorithms ever since.[92] That was a mistake. In fact, claiming algorithms might just be the *solution* to the problem with software patents.

[86] *See* Electric Power Group, LLC v. Alstom S.A., 830 F.3d 1350, 1356 (Fed. Cir. 2016).

[87] *See, e.g.,* Data Engine Technologies LLC v. Google LLC, 906 F.3d 999, 1009 (Fed. Cir. 2018) (contrasting claims that covered a "specific implementation" and thus were patent-eligible with one claim that did not and thus was not); McRO Inc. v. Bandai Namco Games Am., 837 F.3d 1299, 1315 (Fed. Cir. 2016).

[88] *See supra* note 18 and accompanying text; *see also* BESSEN & Meurer, *supra* note 54, at 200 (arguing that "software patents might be particularly prone to strategic use of vague language by applicants to gain undeserved scope").

[89] *See, e.g.,* Timothy B. Lee, *Why a 40-year-old SCOTUS ruling against software patents still matters today*, ARS TECHNICA (July 21, 2018, 1:15 PM), https://arstechnica.com/features/2018/06/why-the-supreme-courts-software-patent-ban-didnt-last/ (quoting Mark Lemley); Katherine Noyes, *It's Clear Why Software Patents Need to Disappear*, PC WORLD, https://www.pcworld.com/article/238173/its_clear_why_software_patents_need_to_disappear.html (last visited Aug. 16, 2011).

[90] JOHN L. ESPOSITO, THE OXFORD HISTORY OF ISLAM 188 (2000). The word "algebra," too, can be attributed to al-Khwarizmi—it comes from the name of his seminal treatise on mathematics. *See id.* at 186–88.

[91] *See, e.g.,* THOMAS H. CORMEN ET AL., INTRODUCTION TO ALGORITHMS xv (3d ed. 2009).

[92] *See infra* notes 114–116 and accompanying text.

A. Just math? Not quite.

One of the oldest algorithms in common use is the Sieve of Eratosthenes, which can be used to find the prime numbers between 1 and any natural number n.[93] Students often learn it in elementary or middle school. Here are the steps:

1. Write down all the integers from 1 to n.
2. Cross out all integers divisible by 2.
3. Find the smallest remaining integer greater than 2.
4. Cross out all integers divisible by it.
5. Continue until you reach $\sqrt{n}$.
6. All remaining integers are prime numbers.[94]

Another well-known example is Euclid's algorithm for finding the greatest common divisor of two natural numbers.[95] These algorithms plainly solve mathematical problems. So do many algorithms in the classic computer science tome, *Introduction to Algorithms*.[96]

But many do not. The "most fundamental problem in the study of algorithms" is *sorting*—reordering the elements in a set of records from lowest to highest according to some measurement or value (age, account number, and so on).[97] The only mathematical operation involved, if it can be called that, is comparison—less than and greater than.[98] In fact, algorithms are used to do all sorts of things other than solve math problems. They're used to store data,[99] to search efficiently,[100] to figure out the best chess move,[101] and to multitask several programs running at the same time[102]—for just a few examples. In general, an algorithm is simply "a collection of simple instructions for carrying out some task."[103] So recipes, driving directions, IKEA instructions, knitting patterns—all these things are algorithms.[104]

To be sure, an algorithm might involve math, and many of the algorithms mentioned above do. Consider Google's PageRank, the algorithm Google first used to "measure the relative importance of web pages."[105] The gist of the algorithm is that pages receive a higher ranking if other pages with a high ranking link to them.[106] A

[93] *See* THEONI PAPPAS, THE JOY OF MATHEMATICS: DISCOVERING MATHEMATICS ALL AROUND YOU 100–01 (1989).

[94] *Id.*

[95] *See* C. STANLEY OGILVY & JOHN T. ANDERSON, EXCURSIONS IN NUMBER THEORY 27–29 (Dover ed. 1988).

[96] *See generally* CORMEN, *supra* note 91 (explaining hundreds of algorithms to solve a variety of problems in mathematics).

[97] *See id.* at 147–48.

[98] *See, e.g., id.* at 17 (describing "insertion sort," the method many people intuitively use to sort a hand of playing cards).

[99] *Id.* at 253 (describing "hash tables," an efficient data structure for storing and retrieving records).

[100] *Id.* at 39 (describing "binary search").

[101] *See* STUART RUSSELL & PETER NORVIG, ARTIFICIAL INTELLIGENCE: A MODERN APPROACH 163–67 (3d ed. 2010) (describing the "minimax" algorithm).

[102] *See* ABRAHAM SILBERSCHATZ *ET AL.*, OPERATING SYSTEM CONCEPTS 266–77 (9th ed. 2013).

[103] MICHAEL SIPSER, INTRODUCTION TO THE THEORY OF COMPUTATION 182 (3d ed. 2013).

[104] *See id.*

[105] Lawrence Page *et al.*, *The PageRank Citation Ranking: Bringing Order to the Web* 2 (1998), http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf.

[106] *Id.* at 2–3.

formal description of that process certainly involves some math, even if it's mostly arithmetic.[107] But there's also math involved in baking—in measuring the flour, scaling the recipe for more or fewer people, calculating cooking time based on weight—and no one says baking is "just math." What matters are the search results and the bread, respectively.

More to the point, math is *involved* in all sorts of inventions that are plainly eligible for patent. For example, rock climbers use camming devices to attach rope to the wall for protection.[108] These "cams" work because their surface has a particular, mathematically defined curvature that exerts maximum force into the rock.[109] And yet even the most ardent proponent of the "software is just math" position must admit that despite the involvement of mathematics, cams can be patented.[110]

Or, to think about it another way, patenting math is disfavored for a specific reason—because granting private monopolies on the "building blocks of human ingenuity" might "tend to impede innovation more than it would tend to promote it."[111] That makes sense for foundational mathematical relationships like $e^{\pi i} + 1 = 0$,[112] or basic algorithms like the Sieve of Eratosthenes, or even sorting algorithms, which are often used as "key subroutines" in all sorts of applications.[113] But that reasoning has much less purchase when math is merely *involved*, be it in a rock-climbing cam device or a webpage search algorithm.

B. Pseudocode claims.

The conflation of algorithms with math has had a significant and harmful effect on the development of software patent law and the quality of software patents. When the Supreme Court first considered software patents in *Gottschalk v. Benson*, it assumed that algorithms were only for solving mathematical problems and therefore that patents on algorithms were categorically impermissible.[114] A decade later in *Diamond v. Diehr*, the Court used the words "mathematical formula" and

[107] *See generally id.* at 2–6. Analyzing algorithms for speed and space efficiency can involve much more complicated math. *See* CORMEN, *supra* note 91, at 150.

[108] *See, e.g.,* U.S. Patent No. 4,184,657, at 1:7–11.

[109] *See id.* at 3:45–60 (teaching that the cam surface must be shaped so that "the line of action through the contact point [of the cam surface with the rock wall] and the axis of the spindle should always be less than 78° to the longitudinal axis of the support bar").

[110] *Cf.* Timothy B. Lee, *Software Is Just Math. Really.*, FORBES (Aug. 11, 2011, 2:29 PM), https://www.forbes.com/sites/timothylee/2011/08/11/software-is-just-math-really/ ("A computer program is a sequence of symbols represented by strings of 1s and 0s. A physical object is a collection of atoms. The distinction couldn't be clearer or more fundamental.").

[111] *See* Alice Corp. Pty. Ltd. v. CLS Bank Intern., 134 S. Ct. 2347, 2354 (2014); Mayo Collaborative Servs. v. Prometheus Labs., Inc., 132 S. Ct. 1289, 1293 (2012) (quoting Gottschalk v. Benson, 409 U.S. 63, 67 (1972)).

[112] This is known as Euler's identity and is considered one of the most beautiful relationships in mathematics. *See* James Gallagher, *Mathematics: Why the brain sees maths as beauty*, BBC NEWS (Feb. 13, 2014), https://www.bbc.com/news/science-environment-26151062.

[113] *See* CORMEN, *supra* note 91, at 148.

[114] Gottschalk v. Benson, 409 U.S. 63, 65, 72 (1972); *see* Julie E. Cohen & Mark A. Lemley, *Patent Scope and Innovation in the Software Industry*, 89 CAL. L. REV. 1, 8 (2001) ("Mathematical algorithms (not just formulae) were declared non-patentable subject matter in an early Supreme Court case, *Gottschalk v. Benson*.").

"algorithm" interchangeably, and reaffirmed that algorithms, like laws of nature, could not be the subject of a patent.[115]

Even though *Diehr* actually weakened that prohibition in practice, the die was cast: Software innovators understood that "being branded an algorithm was the kiss of death."[116] To avoid "being labeled an algorithm or looking too much like math," they began to claim their innovations in plain English—by the result they accomplished, rather than the technical algorithm used to accomplish it.[117] In other words, the bar on claiming algorithms *created* the problem of excessive functional claiming in software patents.[118]

Professor Lemley has argued persuasively that most of the problems with software patents stem from "broad functional claiming of software inventions."[119] In my own experience as a computer scientist and a patent litigator, I have come to agree. And the opposite of claiming functions, for software, is claiming algorithms. The way to fix software patents, then, is to require that they disclose the algorithms used to achieve the results they claim, and to limit their reach to those algorithms and their equivalents.[120] Ironically, that means that *Benson*—which ultimately held the software patent at issue ineligible[121]—should be overruled, at least in part.

But, as I argued above, software claims are *necessarily* functional. There is no clear boundary between claiming software ends and software means. Claim language at the level of machine-language commands would be no use to anyone and would *still* describe results.[122] More realistically, even actual source code is probably too granular a level of detail for patent claims. So what is the right level?

There are no easy answers,[123] but a good first step would be to require that software claims be written in "pseudocode"—the loosely defined notation programmers use to communicate algorithms to others in the field.[124] The description of pseudocode in *Introduction to Algorithms* is useful:

> What separates pseudocode from "real" code is that in pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes, the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section of "real" code. Another difference between pseudocode and real code is that pseudocode is not typically concerned with issues of software engineering. Issues of data abstraction, modularity, and error

[115] Diamond v. Diehr, 450 U.S. 175, 186 (1981).

[116] ROBIN FELDMAN, RETHINKING PATENT LAW 108 (2012).

[117] *Id.* at 108–09.

[118] Lemley, *supra* note 16, at 924–25 n.85.

[119] *Id.* at 908.

[120] *See id.* at 927; *see also* Randall M. Whitmeyer, Comment, *A Plea for Due Processes: Defining the Proper Scope of Patent Protection for Computer Software*, 85 NW. U.L. REV. 1103, 1106 (1991) ("[I]n the computer software context only narrow algorithms, as the term is understood by computer scientists, should be patentable.").

[121] *Gottschalk*, 409 U.S. at 72–73.

[122] *See supra* notes 76–79 and accompanying text.

[123] *See* Lemley, *supra* note 16, at 961.

[124] *See* CORMEN, *supra* note 91, at 3.

> handling are often ignored in order to convey the essence of the algorithm more concisely.[125]

*Pseudo*code, in other words, conveys just enough detail to enable a programmer to implement the algorithm in *real* code.[126]

An example might help make my proposal more concrete (so to speak). This is the pseudocode for "insertion sort," one way to sort a list of numbers:

```
INSERTION-SORT($A$)
for $j = 2$ to $A.length$
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i + 1] = A[i]$
        $i = i - 1$
    $A[i + 1] = key$[127]
```

This code cannot be compiled or executed on any machine. It just describes to programmers the way—the specific, concrete way—that the insertion-sort algorithm works. In plain English, this is what it does: first, it compares the first number in the list with the second. If the second is smaller, it swaps them. Either way, now the first two numbers in the list are sorted relative to each other. It then compares each number in the rest of the list with those to its left and moves the ones that are bigger one space to the right, which opens up a space into which it drops the number being sorted—in between the biggest element smaller than it and the smallest element bigger than it.[128] Plainly, there's math involved in executing this algorithm, but the algorithm itself isn't math—it's just moving things around. There's no metaphysical reason it can't be patented.

Now compare that pseudocode with the way a patent might claim it, especially under pre-*Alice* standards:

> A method of sorting numbers in a list, comprising:
>
> moving each number to a position within such list, wherein:
>
> said same number is greater than all numbers in positions prior to such position, and
>
> said same number is lesser than all numbers in positions subsequent to such position.

That "claim" correctly describes the desired result, but it doesn't actually disclose anything about how to move each number to the desired position. In fact, it could cover any of the myriad ways of sorting a list of numbers.[129] It's a classic abstract idea—it describes a result and "it matters not by what process or machinery

[125] *Id.* at 17.
[126] *Id.* at 3.
[127] *Id.* at 18.
[128] *Id.* at 18–20.
[129] *See* CORMEN, *supra* note 91, at 147–212 (describing several other sorting algorithms).

the result is accomplished."[130] Still, it's wordy enough that a beleaguered patent examiner might just let it slide.[131]

On the other hand, the pseudocode above doesn't include every last detail necessary to execute the algorithm either. Both it and the mock claim are in some sense functional. But pseudocode is how programmers communicate their algorithms to *each other*. Its *purpose* is to convey the "structure" of an algorithm in just enough detail to enable other programmers to "implement it in the language of [their] choice."[132] So if a programmer wants a patent monopoly on her algorithm, that's also how she should disclose the scope of that monopoly to the world.[133]

A coda to this proposal: requiring that software claims be written in pseudocode might make for a right with very "thin" scope.[134] So under the dominant paradigm in which claims stake out the very edge of the patent monopoly, known as "peripheral claiming," it would incentivize patentees to claim every trivial variation on their algorithm, which would be good for no one but the patent lawyers.[135] Instead, the tradeoff should be this: you claim your algorithms in "thin" pseudocode, but you get a "thick" doctrine of equivalents in return.[136] For algorithmic inventions, which can be implemented in an infinite variety of ways, this return to a sort of "central claiming" is a natural fit.[137] And in addition to the carrot of a thick doctrine of equivalents, an appropriate stick can be found in Professor Lemley's proposal for patents written under the current rules: limit functional claim language—not just explicit "means for" and "step for" elements, but all language that, fairly evaluated, describes ends rather than means—to algorithms actually disclosed in the specification, if any.[138] In

[130] *Cf.* O'Reilly v. Morse, 56 U.S. (15 How.) 62, 113 (1854).

[131] For some similarly egregious examples, see U.S. Patent No. 8,688,085 cl. 14, invalidated in *Affinity Labs of Tex., LLC v. Amazon.com, Inc.*, 838 F.3d 1266 (Fed. Cir. 2016); U.S. Patent No. 6,384,850 cl. 1, invalidated in *Apple, Inc. v. Ameranth, Inc.*, 842 F.3d 1229 (Fed. Cir. 2016); or U.S. Patent No. 6,038,295 cl. 17, invalidated in *In re TLI Commc'ns LLC Patent Litig.*, 823 F.3d 607 (Fed. Cir. 2016).

[132] CORMEN, *supra* note 91, at 3.

[133] *See* PSC Computer Prods., Inc. v. Foxconn Int'l, Inc., 355 F.3d 1353, 1358 (Fed. Cir. 2004) ("[T]he claim provides notice as to the scope of the invention."); Mark A. Lemley & Mark P. McKenna, *Scope*, 57 WM. & MARY L. REV. 2197, 2202 (2016) (explaining that "scope" is "the range of things the IP right lawfully protects against competition").

[134] *Cf., e.g.,* Apple Computer, Inc. v. Microsoft Corp., 35 F.3d 1435, 1443 (9th Cir. 1994) (discussing "broad" versus "thin" copyrights on software).

[135] *See* Dan L. Burk & Mark A. Lemley, *Fence Posts or Sign Posts? Rethinking Patent Claim Construction*, 157 U. PENN. L. REV. 1743, 1745–46, 1769–70 (2009) (explaining peripheral claiming). For just one example of this type of trivial variation, a patentee could replace all iterative loops with recursions, or vice versa. *See generally* Olin Shivers, *The Anatomy of a Loop: A story of scope and control*, PROCEEDINGS OF THE TENTH ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING 2 (2005) (explaining that any iterative algorithm can be rewritten as a recursive algorithm).

[136] The doctrine of equivalents permits a patentee to expand the scope of her claim to cover a defendant's product that differs from the patented invention in only a minor respect." Burk & Lemley, *supra* note 135, at 1763. The doctrine has been largely vitiated as peripheral claiming and claim construction have come to dominate the reach of the patent monopoly. *Id.* at 1763–66.

[137] On the relationship between central claiming and a thick doctrine of equivalents, see *id.* at 1772 ("[M]odern courts engaged in doctrine of equivalents analysis follow a form of central claiming while denying that they do so.").

[138] *See* Lemley, *supra* note 16, at 943–49.

other words, either you claim pseudocode and get its equivalents, or you claim functions and get only what you disclose in the specification.

VI. Conclusion

Software is hard for patent law. From tip to toe, it's an abstract, functional enterprise. But at the same time, there plainly *is* innovation in software, the sort of innovation we usually encourage with patent protection. And as I have shown, software innovation *can* be specified concretely enough to grant it protection, so there's no metaphysical reason not to. There might be other reasons—perhaps the costs just outweigh the benefits.[139] But much of those costs come from the presently abstract nature of software claims.[140] And my suggestion in this Article—to require that software claims be written in pseudocode—would go a long way toward solving the problem of abstraction in software patents.

[139] IP rights incur substantial social costs, which can only be justified if outweighed by the incremental innovation encouraged. Christopher Buccafuso, Mark A. Lemley, & Jonathan S. Masur, *Intelligent Design*, 68 DUKE L.J. 75, 87, 89 (2018); John M. Golden, *Patentable Subject Matter and Institutional Choice*, 89 TEX. L. REV. 1041, 1070–73 (2011). Those costs might be especially high, and especially unjustified, for software innovation. Cohen & Lemley, *supra* note 114, at 5–6 & n.5, 41, 46; Lemley, *supra* note 16, at 935; Lemley *et al.*, *supra* note 1, at 1340. As the Supreme Court observed in *Benson*, software innovation saw "substantial and satisfactory growth in the absence of patent protection." Gottschalk v. Benson, 409 U. S. 63, 72 (1972) (quoting PRESIDENT'S COMMISSION ON THE PATENT SYSTEM, TO PROMOTE THE PROGRESS OF USEFUL ARTS 13 (1966)).

[140] Lemley, *supra* note 16, at 908 ("[T]he most important problem a product-making software company faces today is not suits over claims with unclear boundaries but suits over claims that purport to cover any possible way of achieving a goal."); Cohen & Lemley, *supra* note 114, at 15 & n.49.