1+1 (pat. pending) — Mathematics, Software and Free Speech

Why Software and Patents Need To Get a Divorce

~ by PolR

This article provides a detailed factual explanation of why software is mathematics, complete with the references in mathematical and computer science literature. It also includes a detailed factual explanation of why mathematics is speech, complete once again with references. My hope is that it will help patent lawyers and judges handling patent litigation understand these fundamental truths, so they can apply that technical knowledge to their field of skill.

Case law on software patents is built on a number of beliefs about how computers and software work. But as you will see, when you compare the technical facts presented in this article and in the authoritative works referenced, with expressions in case law on how computers and software work, you will find they are often in complete opposition. I believe this is a foundational problem that has contributed to invalid patents issuing.

If you are a computer professional, I hope you pay attention to another aspect of the article, on how the lawyers and judges understand software. This is critical to understanding their point of view. After reading case after case on the topic, I have concluded that the legal view of software relies on beliefs that are in contradiction with known principles of computing. Computer professionals explain their profession based on an understanding that is, on its face, the opposite of a few things the legal profession believes to be established and well understood facts. Moreover, the law is complex and subtle. Computer professionals don't understand it any better oftentimes than patent lawyers understand software, and so they can make statements that make no legal sense.

I believe that coming to a clear and fact-based definition of what an algorithm is can help both sides to communicate more effectively. So let's do that as well.

Criteria to Evaluate the Costs And Benefits of Software Patents to Society

Why do people believe patents are a good idea? The usual explanation is that they do more good than harm. Patents are exclusive rights granted to inventors in exchange for public disclosure of their invention. The table below summarizes a frequent understanding of the costs and benefits to society of patents.

Benefits to Society	**Costs to Society**
Promote progress of useful arts by rewarding inventors	Limited time exclusive patent rights to the invention
Support the economy by encouraging innovation	Administrative costs (we need a patent office)
Disclosure of what would otherwise be trade secrets	Legal costs

	Liability risks for potential infringement and costs of patent defense strategies

The generally held belief is that the benefits from the left column outweigh the costs from the right column. This is from the perspective of society. The costs and benefits for the inventor are a different calculation, but for the purposes of this article they are irrelevant. The topic is why we have patents in the first place, and the answer is not that the inventors have a right to them. It is that the policy makers have decided that it is beneficial to society to grant them to inventors.

But the above table is completely wrong for software. The error in the table is that it doesn't include copyright and it doesn't include an analysis of Free and Open Source Software, or FOSS. It also doesn't include the facts that mathematics is speech and software is mathematics. Here is what a corrected table looks like.

Benefits to Society	**Costs to Society**
Promote progress of software by rewarding inventors above and beyond the rewards already provided by copyrights and community contribution to FOSS projects	Limited time exclusive patent rights to the invention
Support the economy by encouraging innovation above and beyond the rewards already provided by copyrights and community contribution to FOSS projects	Administrative costs (we need a patent office)
Disclosure of what would otherwise be trade secrets above and beyond disclosure inherent to the release of source code by FOSS projects	Legal costs
	Liability risks for potential infringement and costs of patent defense strategies
	Harm to FOSS development limiting its positive contribution to progress, the economy and to disclosure of source code
	First Amendment issues resulting from exclusive rights granted to the exercise of mathematical speech

As you see the effect of copyrights and FOSS on the analysis is two-fold. First, software will progress under the incentives of copyrights and community support in the absence of patents exactly as it did before *State Street*. This is a far cry from comparing the benefits of patents with total lack of benefits due to the absence of alternatives as might be appropriate in other disciplines. The benefits of software patents are incremental at best, assuming there are any benefits at all. Second, the damage of patents to FOSS is a negative that must be subtracted from the benefits. The goals of progress, innovation, economic incentive and disclosure are all met by FOSS using means other than exclusive rights. FOSS developers and users must bear the social costs of patents but FOSS has no use for patents and won't derive the corresponding benefits. The resulting harm is a reduction in the benefits of FOSS to society that offset at least part of the presumed benefits of software patents.

This is important to explain because, I think, courts and policy makers believe the same costs/benefits calculations that apply to patents in general also apply to software patents in particular. They will be reluctant to rule software unpatentable as long as they believe these patents are beneficial. They won't want to do anything disruptive without a good reason. For example here is an extract from the recent Supreme Court ruling in *Bilski*:

> It is true that patents for inventions that did not satisfy the machine-or-transformation test were rarely granted in earlier eras, especially in the Industrial Age, as explained by Judge Dyk's thoughtful historical review. *See* 545 F. 3d, at 966—976 (concurring opinion). But times change. Technology and other innovations progress in unexpected ways. For example, it was once forcefully argued that until recent times, "well-established principles of patent law probably would have prevented the issuance of a valid patent on almost any conceivable computer program." *Diehr*, 450 U. S., at 195 (STEVENS, J., dissenting). But this fact does not mean that unforeseen innovations such as computer programs are always unpatentable. *See id*., at 192—193 (majority opinion) (holding a procedure for molding rubber that included a computer program is within patentable subject matter). Section 101 is a "dynamic provision designed to encompass new and unforeseen inventions." *J. E. M. Ag Supply, Inc. v. Pioneer Hi-Bred Int'l, Inc*., 534 U. S. 124, 135 (2001). A categorical rule denying patent protection for "inventions in areas not contemplated by Congress … would frustrate the purposes of the patent law." *Chakrabarty*, 447 U. S., at 315.
>
> The machine-or-transformation test may well provide a sufficient basis for evaluating processes similar to those in the Industrial Age—for example, inventions grounded in a physical or other tangible form. But there are reasons to doubt whether the test should be the sole criterion for determining the patentability of inventions in the Information Age. As numerous amicus briefs argue, the machine-or-transformation test would create uncertainty as to the patentability of software, advanced diagnostic medicine techniques, and inventions based on linear programming, data compression, and the manipulation of digital signals. *See, e.g*., Brief for Business Software Alliance 24— 25; Brief for Biotechnology Industry Organization et al. 14—27; Brief for Boston Patent Law Association 8—15; Brief for Houston Intellectual Property Law Association 17—22; Brief for Dolby Labs., Inc., et al. 9—10.

This part of the decision is in a part of the opinion of Justice Kennedy which is endorsed by 4 out of the 9 justices. It nevertheless shows their concerns. They see the Information Age as a new ballgame which requires a flexible interpretation of the law. If we convince them that the costs benefits calculation for software patents is different from the costs benefits of patents in general it may influence their assessment of what is good for society. They may become more receptive to the pleas that harm is being done. [1] See the appendix of the Joint OSI and FSF Position Statement on CPTN Transaction , which develops this theme.

I will concentrate on the themes of disclosure, innovation, mathematics and speech. At the beginning, this discussion will discuss the costs and benefits of patents, but as I develop the points of mathematics and then speech I will raise questions on the more fundamental issue of whether software should be patentable at all.

Disclosure

Both the Free Software Definition and the Open Source Definition require disclosure in the form of source code. The Free Software Definition requires among other things:

> The freedom to study how the program works, and change it to make it do what you wish (freedom 1). Access to the source code is a precondition for this.

The Open Source Definition requires among other thing:

> The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.

This release of source code is disclosure. In the event the software represents an innovation, its release under a FOSS license is disclosure of the innovation. In other words, patents are not the only form of disclosure available.

How does this disclosure compare to what is found in patents? Well-known patent attorney Gene Quinn explains what kind of disclosure he recommends when filing a patent application:

> So what information is required in order to demonstrate that there really is an invention that deserves to receive a patent? When examining computer implemented inventions the patent examiner will determine whether the specification discloses the computer and the algorithm (e.g., the necessary steps and/or flowcharts) that perform the claimed function in sufficient detail such that one of ordinary skill in the art can reasonably conclude that the inventor invented the claimed subject matter. An algorithm is defined by the Patent Offices as a finite sequence of steps for solving a logical or mathematical problem or performing a task. The patent application may express the algorithm in any understandable terms including as a mathematical formula, in prose, in a flow chart, or in any other manner that provides sufficient structure. In my experience, flow charts that are described in text are the holy grail for these types of applications. In fact, I just prepared a provisional patent application for an inventor and we kept trading flow charts until we had everything we needed. Iterative flow charting creates a lot of detail and the results provide a tremendous disclosure.
>
> …
>
> The short of it is that you need to articulate the invention so that others know you have more than just an abstract idea and you need to articulate how a programmer would set out to create the code to bring about the desired functionality. This doesn't mean that you need to know how to write the code, but it does mean that you need to take a systems approach to the project. Treating the software project like an engineering problem and setting out to create a design document that can and will be used by those writing the code is the goal. Essentially you want to answer as many questions that the programmer will have as possible, with the goal being to answer each and everything they could ever want to know so that the programming is a ministerial task akin to translating your design document (i.e., patent application) into code.

And here Quinn further clarifies that source code is not required:

> Software and computer implemented process don't protect the code, anyone can write code, some of which works and much of which has the added feature of unintended bugs. The code is not the magic, the code is the translation of the innovation into terms capable of being processed by a machine. The innovation is the overall system from a computer

> engineering perspective that takes into consideration anything that can and will go wrong and addresses those possible occurrences, whether likely or not. That is why you don't need code to receive a patent on software or a computer implemented process. What you need is a design document that would direct those who would be coding so that they are not interjecting any creativity. You want them to just code and not get creative, the vision is the inventors and the coders are the means to the end.

Here we have two very different kinds of disclosure. FOSS discloses complete and fully operational software in the form of source code. Patents disclose an “engineering document” where the bulk of the work remains to be done.

If providing an engineering document to reduce the coding to a ministerial task were the key to writing bug-free software then bug-free software would be commonplace. This is not the case. It may help to get a sense of of the basics of the software development process. Several methodologies are used by professional developers. The simplest, conceptually, that matches Quinn's vision of preparing an engineering document before coding is the waterfall model. This is developing software through the successive execution of several consecutive phases in this order.

- Requirements Analysis
- Design
- Implementation
- Verification (Testing)
- Maintenance

There are many variants of the waterfall, and the exact list of phases is not always exactly as shown. But for purposes of this discussion this summary will do. What Quinn suggests is that the work product of the design phase is sufficient disclosure for a patent. The implementation — which is when we write the code — and the verification — which is when we test that the software works as intended — are not required to produce a design document. However these two phases represent over 60% of the actual work. Moreover, the earlier the phase where an error occurs the greater the cost of fixing the problem. Errors in the design are more damaging and more costly to fix than those occurring during implementation. This is basic software engineering 101 which is explained in any good introductory textbook.

Where is the requirement that the disclosure in a patent specification has been tested? Where is the requirement that the invention has been implemented? I don't find this in Quinn's explanation. As far as I can tell, one may stop at the design phase, never go through the other phases, and still get a patent, obtaining exclusive rights for 40% of the cost of someone who works through a complete implementation. Further cost reductions are achieved when one limits his design activity to a narrow aspect of the complete software which will meet the requirements of novelty and obviousness and ignores the other aspects of real-life software development.

I have to ask, is this kind of disclosure sufficient to meet the societal bargain of a grant of temporary monopoly rights in exchange for disclosure? I don't see any requirements that the invention is implemented let alone tested. How could this disclosure be deemed sufficient? We are not even sure there is a working invention.

The costs/benefits analysis of software patents with respect to disclosure works like this. Software patents will prevent the disclosure to society of fully working FOSS implementations because they can't be written without a patent license. This is part of the harm done to FOSS. But in exchange, society gets the disclosure of engineering documents that cover a narrow aspect of the software without any guarantee that the invention has ever been implemented, let alone been tested.

Innovation

Patents proponents like to pose as the enablers of innovation. They argue that without patents innovators will stop working and creativity and inventiveness will wither. FOSS advocates shouldn't leave such claims unopposed. They should make a strong case of their own innovative capabilities.

There was innovation in software before *State Street* issued. Those who have knowledge of the history of computing or personal knowledge of important events can speak out and post out this information in places where the writers of amicus briefs may find it and quote it. Perhaps it would be important to discuss the role of sharing source code in the development of early UNIX, the Internet, programming tools and open standards in organizations like the IETF and W3C. Please don't forget to list programming languages such as Perl and PHP which are widely used to implement web sites. They are innovative. The Supreme Court showed an interest in the role of patents in the Information Age. They should know which role the sharing of source code played in the development of the key technologies that brought this Information Age into existence. If this knowledge isn't clearly communicated we run the risk the justices may assume that FOSS is an insignificant industry outlier which may safely be ignored or that innovation will cease without patent protection.

FOSS innovation is not all technology. The work methods of FOSS are very innovative in themselves. The "*release early and release often*" development model is one of Linus Torvald's greatest innovations. The relationship between Red Hat and the Fedora community is a huge business innovation. The Debian community and its relationship with an array of businesses building Linux distributions atop of their work is also innovative. Then there is the productive use of FOSS in many enterprises leading to new business models. Look at Google, Yahoo and others.

The harm done to FOSS by patents isn't limited to technology, either. It curtails the creativity of businessmen and developers in their search for new business and software development models and their ability reap the rewards of these innovations.

I want to point out two specific FOSS projects which display some very interesting innovations. They are Coq and Standard ML. They illustrate that FOSS is a source of innovation and that harming FOSS will harm this source of innovation.

Coq

Coq is Free Software distributed under the LGPL v. 2.1 license. Coq is described as follows (emphasis in the original):

> Coq implements a program specification and mathematical higher-level language called *Gallina* that is based on an expressive formal language called the *Calculus of Inductive Constructions* that itself combines both a higher-order logic and a richly-typed functional programming language. Through a *vernacular* language of commands, Coq allows:
>
> - to define functions or predicates, that can be evaluated efficiently;
> - to state mathematical theorems and software specifications;
> - to interactively develop formal proofs of these theorems;
> - to machine-check these proofs by a relatively small certification "kernel";
> - to extract certified programs to languages like Objective Caml, Haskell or Scheme
>
> As a proof development system, Coq provides interactive proof methods, decision and

> semi-decision algorithms, and a *tactic* language for letting the user define its own proof methods. Connection with external computer algebra system or theorem provers is available.
>
> As a platform for the formalization of mathematics or the development of programs, Coq provides support for high-level notations, implicit contents and various other useful kinds of macros.

Coq is a system for writing software specification, for developing proofs and extracting the corresponding algorithms. This concept of extraction of certified program is based on a mathematical principle called the Curry-Howard correspondence. When mathematical proofs are written in a certain manner, an algorithm is implicit in the proof. When we extract this algorithm we get working software whose correctness is guaranteed by the mathematical proof it comes from. In this context 'correctness' means the software conforms to the mathematical specification and is bug-free to the extent there are no errors in the specification. In this sense the software is 'certified' to be correct. Coq may be used in some cases as an alternative to writing code in the more usual way.

This is innovative. It is being disclosed as source code under a FOSS license, the LGPL. Moreover this innovation has already found practical industrial uses [2]:

> Coq was then an effectively usable system, thus making it possible to start fruitful industrial collaborations, most notably with CNET and Dassault-Aviation.
>
> …
>
> After a three-year effort, Trusted Logic succeeded in the formal modeling of the whole execution environment for the JavaCard language. This work on security was awarded the EAL7 certification level (the highest level in the so-called common criteria). This formal development required 121000 lines of Coq development in 278 modules.

Standard ML

The Standard ML programming language has a semantics that is defined mathematically. The language designers have published the definition.[3] They have deliberately chosen not to define it in terms of machine activity. Instead the semantics of the language is a series of mathematical formulas specifying the behavior of a conforming implementation. Here is how the language designers explain their approach to the language semantics[4] (emphasis in the original):

> The job of a language-definer is twofold. First—as we have already suggested—he must create a world of meanings appropriate for the language, and must find a way of saying what these meanings precisely are. Here, he meets a problem; notation of *some* kind must be used to denote and describe these meanings—but not a *programming language* notation, unless he is passing the buck and defining one programming language in terms of another. Given a concern for rigour, mathematical notation is an obvious choice. Moreover, it is not enough just to write down mathematical definitions. The world of meanings only become meaningful if the objects possess nice properties, which make them tractable. So the language-definer really has to develop a small *theory* of his meanings, in the same way that a mathematician develops a theory. Typically, after initially defining some objects, the mathematician goes on to verify properties which indicate that they are worth studying. It is this part, a kind of scene setting, which the language definer shares with the mathematician.

> Of course he can take many objects and their theories directly from mathematics, such as functions, relations, trees, sequences, … But he must also give some special theory for the objects which make his language particular, as we do for types, structures and signatures in this book; otherwise his language definition may be formal but will give no insight.
>
> The second part of the definer's job is to define *evaluation* precisely. This means that he must define at least *what* meaning, *M*, results from any phrase *P* of his language (though he need not explain exactly *how* the meaning results; that is he need not give the full detail of every computation). This part of his job must be formal to some extent, if only because the phrases *P* of his language are indeed formal objects. But there is another reason for formality. The task is complex and error-prone, and therefore demands a high level of explicit organisation (which is, largely, the meaning of 'formality'); moreover, it will be used to specify an equally complex, error-prone and formal construction: an implementation.

The effect of such a approach is that every program written in Standard ML has a known meaning expressed by a series of mathematical formulas found in the language definition manual. These formulas are tied to a mathematical theory which serves as a foundation and a motivation for the language definition. This approach to defining a language is innovative.

There are several implementations of Standard ML. According to the Wikipedia they are all open source. For example the Standard ML of New Jersey implementation is released under a BSD-like license. Note that links to Wikipedia are to let you verify the meanings of some technical terms as they are introduced in the article, but these links are for quick informational purposes only, and are based on the definitions at the time of this article's preparation. The authoritative sources are the published literature.

Mathematics

I will now substantiate the idea that software is mathematics.

Let's cast aside the effect of the real world semantics on the patentability of mathematics for the moment. I will return to this question when I explain why mathematics is speech. Then I will explain that patenting a mathematical computation on the basis of its semantics is granting exclusive rights on speech. For now the focus is on showing that the patented software method is always a mathematical algorithm.

Let me be clear. I am not saying the work of the programmer is mathematics. I am saying the work of the computer is mathematics. Microsoft, Symantec and Phillips in their amicus brief to the Supreme Court explain their theory of what it means for software to be patentable:

> As with any patentable process, it is the real-world implementation—the actual acting out, or physical execution—of the process that makes it new and useful. In a computer-implemented process, the acting out consists primarily of the rapid activation and deactivation of millions of transistors to perform some useful function, such as displaying images and solving problems. Such functions, implemented and made real, physical, and useful by the activity of transistors, are the inventor's actual process.

I am saying this transistor activity is the implementation of a mathematical algorithm.

Let's assume that we have some design document for software which is some patent specification written in accordance with Quinn's recommendations. The question is whether this patent reads on a mathematical algorithm. A possible answer may be to write the software using a development tool such as Coq or a language such as Standard ML. If we succeed, we have evidence that the implementation of a mathematical formula corresponds to the functionality of the method recited in the patent.

Note that this argument is not based on any definition of mathematical algorithm. Deciding whether a method is legally a mathematical algorithm based on the definition of algorithm has turned out to be a very difficult task from a legal perspective. The courts, from my research, seem to have given up on that approach and now prefer to tackle the issue of patentability of software from the abstract idea angle. I am saying that there is another way. We may use a trail of documentation. The first document is the definition manual for the language. It shows that the language may only express mathematical algorithms corresponding to a series of formulas recited in the language definition. The second document is the source code. It shows that the program is written in this language. Together these documents prove that the program is factually a mathematical algorithm. Therefore the corresponding patent reads on a mathematical algorithm which, depending on how the patent is written, may possibly be limited to a specific real-world semantics layered on top of the mathematical semantics.

Another key idea is the notion of model of computation. Oftentimes mathematicians and computer scientists find it convenient to give a mathematically rigorous definition to a class of algorithms. Then the proof that a particular procedure is an algorithm may be broken down into a two steps. First we identify where in published literature the model of computation is defined. This establishes that all procedures belonging to the model are accepted as mathematical algorithms in the fields of mathematics and computer science. Then we show that the procedure under study conforms to the model. This is a mathematically rigorous test because the model has been defined with mathematical rigor. The semantics of programming languages such as Coq and Standard ML are examples of models of computation. The trail of documentation is the two-step process I have just identified, with one document corresponding to each of the two steps. Such a procedure is factual and avoids the legal difficulties with the definition of algorithm the courts have encountered so far.

People may object, what if the patent method is written in a language such as C which doesn't have a mathematical definition? The answer is that such languages may implement mathematical algorithms even though the language doesn't have a mathematical definition. We may draw no conclusion on whether or not a C program is a mathematical algorithm from the C language definition alone. Besides the issue is whether the patent claim reads on a mathematical algorithm and not whether the programming language is implemented according to a mathematical specification. If we find one implementation of the patent claim which is provably a mathematical algorithm, then the claim reads on it.

A Warning about the Libraries

In the above argument there is a limitation concerning the libraries. The documentation trail will work only if all the software is mathematically defined. If the program links with libraries written in languages other than Standard ML then this part is not proven to be mathematical. This may require the programmer to write his libraries from scratch in Standard ML to establish a solid documentation trail. Note that writing code in language other than Standard ML doesn't prove it is not mathematics. It just breaks the documentation trail.

This issue is aggravated by the fact the authors of the Standard ML basis library which contains all the standard functions have not bothered to define them mathematically.[5] Only the Core language is so defined. This problem is fixable. Part of the library is the mathematical functions. Their mathematical definitions are implicit even though they have not been explicitly spelled out as formulas in the documentation. Another part of the library is about the character strings. This too is definable mathematically by those who are knowledgeable in abstract language theory.

The parts that are tricky are the input and output routines and the interface with the operating systems. There is a workaround. The Concurrent ML library is mathematically defined.[6] It includes a reimplementation of the most commonly used input and output functions as well as several of the services that are normally provided by the operating system. These versions are mathematically defined. There is also eXene, a X-Windows toolkit that is entirely written in Concurrent ML. I didn't check the code but this is suggestive that the user interface functions should inherit a mathematical definition from the underlying Concurrent ML library. Therefore it should be possible to program in Standard ML while using only library functions that are mathematically defined.[7] However for many applications this could be crippling if the necessary functions don't have an already written mathematically defined version.

It may be a good idea if someone packages a legalese dialect of Standard ML that contains only mathematically defined library functions together with the documentation of their mathematical definitions. In an ideal world this dialect would be validated by a lawyer to ensure it will resist adverse examination in litigation. Then the mere fact that a program compiles in this dialect would guarantee that it is a mathematical algorithm.

A Virtual Machine Project

It would be helpful if we can extend the trail of documentation approach to languages other than Standard ML and remove the limitations on the libraries. I suggest a virtual machine project.[8] The idea is to give a mathematical semantics to the instructions of a virtual machine in such manner that the virtual machine itself is a known mathematical algorithm. Think of something like Bochs except that the target is a synthetic instruction set which has a mathematically defined semantics like Standard ML. Then any program that runs in this virtual machine is provably a mathematical algorithm computing a known series of mathematical formulas. The next step would be to port the GNU compilers to this architecture and run entire software stacks from operating systems to applications in the virtual machine. At this point the entire software stack is proven to be the execution of a mathematical algorithm.

Abstract Machines

An algorithm such as this proposed virtual machine belongs to the general category of abstract machines. Here is how Marvin Minsky explains how an abstract machine is abstract.[9] This notion originated in theoretical studies of what are the fundamental limitations of computations made by machines. The intent is to look beyond the physical constraints of implementations and examine the essence of what is a machine-implemented computation.

> [I]t is important to understand from the start that our concern is with questions about the ultimate theoretical capacities and limitations of machines rather than with the practical engineering analysis of existing mechanical devices.
>
> To make such a theoretical study, it is necessary to abstract away many realistic details and features of mechanical systems. For the most part, our abstraction is so ruthless that it leaves only a skeleton representation of the structure of sequences of events inside a machine—a sort of “symbolic” or “informational” structure. We ignore, in our abstraction, the geometric or physical composition of mechanical parts. We ignore questions about

> energy. We even shred time into a sequence of separate disconnected moments, and we totally ignore space itself! Can such a theory be a theory be a theory of any "thing" at all? Incredibly, it can indeed. By abstracting out only what amounts to questions about the logical consequences of certain cause-effect relations, we can concentrate our attention sharply and clearly on a few really fundamental matters. Once we have grasped these, we can bring back to the practical world the understanding, which we could never obtain while immersed in inessential detail and distraction.

If you assume in these two paragraphs that the meaning of the word "mechanical" includes the term "electronic" you will understand what it means for the virtual machine algorithm to be an abstract machine. Here the fundamental matter is whether there is an abstract idea called a mathematical algorithm in a process claimed in a software patent. By abstracting away the physical elements and reducing everything to information we can see the mathematics of computing clearly without being immersed in inessential detail and distraction.

A common argument used in favor of software patents is that everything is reducible to hardware. When the programmer writes code it will get translated into machine instructions that eventually will result into something hardware and patentable. The Microsoft-Symatec-Phillips brief used this logic, arguing the patentable process is transistor activity. This argument is faulty because not everything in a computer is reducible to hardware. If we proceed with Minsky's ruthless abstraction we are left with something which is not hardware.

I like to use the book as an analogy. We may imagine an argument where a book is ultimately marks of ink on paper. The writer may write a novel but once he is done writing, what is left is a stack of paper with marks of ink and a cover bound around it. If you try to patent your next novel you will fail. The patent system and the courts understand very well the notions of alphabet, text, language and semantics. They know that when one subjects a book to something like Minsky's ruthless abstraction, one gets a series of symbols in the alphabet that tells a story.

On the other hand the similar reduction to hardware argument is used on computers with success. The bits are symbols in the binary alphabet. They carry a semantics in the language of mathematics. And because mathematics may carry a semantics in the real world, the bits will carry a real-world semantics too. But still the computation is viewed as a hardware process, and a patent will issue provided the other requirements of patent law are met.

The challenge is to show people the abstract mathematical part. Arguing with words is difficult. The opposing side is equally skilled with words and the decision rests with laymen, judge and juries, who may not be inclined to see things as computer professionals do. I suggest to show them visually. The formulas for the abstract machine are produced in a document. They can see these formulas. The source code of the virtual machine implements the formulas. Then they can see in a demo the computer running the virtual machine loading various Linux and BSD distributions and running the program.

Technology such as the printing press produces static information. The book once printed will retain the same content forever. But there are uses of the language which are not static. We may hold a conversation interacting with others, we may maintain written business records that must be updated, or we may carry out a pencil and paper calculation. While the printing press is technology appropriate for static uses of the language, the computer is a device that implements the dynamic uses. The trouble is that patent proponents see these dynamic uses as processes and want to patent them. They do so by reducing the dynamic features of mathematical languages to hardware activity. This is the argument the virtual machine demonstration should overcome.

Another common objection is that a mathematical description doesn't make the described device nonpatentable. The court in *In re Bernhart* summarizes this principle of law (emphasis in the original):

> [A]ll machines function according to laws of physics which can be mathematically set forth if known. We cannot deny patents on machines merely because their novelty may be explained in terms of such laws if we are to obey the mandate of Congress that a machine is subject matter for a patent. We should not penalize the inventor who makes his invention by discovering new and unobvious mathematical relationships which he then utilizes in a machine, as against the inventor who makes the *same machine* by trial and error and does not disclose the laws by which it operates.

I understand this to mean that if we describe a rocket mathematically the rocket is still patentable. The *Bernhart* court said nothing about patenting the mathematical formulas for the rocket. However there will be people who say that the formulas for the virtual machine project describe the transistor activity in the computer and the principle stated in *Bernhart* means this activity remains patentable. The answer is to read Minsky's ruthless abstraction once again. There are no transistors in these formulas. They have all been thoroughly abstracted away.

The same is true of the formulas used in Coq and Standard ML. They don't describe any physical computer part either.

Register Machines and their Variations

Which kind of abstract machine will this virtual machine project be? It will belong to the family of register machines. This is a series of similar abstract machines where a finite-state machine is used to control and access numeric information located in a finite number of locations called registers. Each of these locations is addressed by a number. The finite state machine more or less corresponds to the CPU of a physical computer while the registers correspond to the main memory (and not the CPU registers).

Multiple flavors of these machines have been studied and different authors use different names: abacus machines, program machines, unlimited register machines, random-access machines (RAM). The different names often correspond to differences in the exact features which have been incorporated in the mathematical definition.

Here is how R. Gregory Taylor describes the generic register machines.[10]:

> Whereas most of the models that we consider in this text—for example, Turing machines, Post systems—predate the advent of the modern digital computer, the register machine model, which we introduce next, is of later vintage. It will come as no surprise, then, that this model reflects modern computer design to a degree. As a consequence, the reader will probably find register machines natural to work with.
>
> Any register machine *M* is assumed to have some nonempty collection of registers R_1, R_2, R_3, …. The contents of any register R_i will always be a natural number. Consequently, register incrementation will always make sense. Usually it will be sufficient to assume that the collection of *M*'s registers is finite, although occasionally it will be convenient to assume that the number of registers is unbounded. In other words, we shall permit a register machine to have either a finite or an infinite collection of registers. To this extent, register machines represent an idealization of modern computers. On the other hand, even in the case where *M* has access to infinitely many registers, it will be true that, up to any point in M's computation for a given input, only finitely many registers will have been used.

We see how the register machine corresponds in the abstract universe of mathematics to the memory architecture of real-life computers. Each abstract register corresponds to a memory location, indexed by a natural number. The relationship with infinity is documented. When presented with Turing machines software patent proponents sometimes object that this model is inapplicable because there is no infinite tape in computers. The register machine model clarifies this point. R. Gregory Taylor explains that in the abstract world of mathematics we may have as much memory as we want but we will only use a finite quantity of memory locations. In real world terms this means we don't need an actually infinitely big computer to run our programs. The program will run if we have enough hardware.

Gregory continues his explanation[11] (emphasis in the original):

> Any register machine M will be associated with a finite, labeled sequence of instructions, each of which is one of the five types listed below.
>
> [Ed: list of instruction types omitted]
>
> For the time being, this completes our informal presentation of the syntax and semantics of register machine programs. Later on, we shall be adding to our instruction set, but we prefer to consider a couple more examples before doing so. In any case, we shall keep our instruction set small; this will be an advantage later in presenting proofs about register machines. What we shall mean by a *register machine computation* is implicit in the flowcharts and pseudocode used to define particular machines.

Here we see the correspondence between register machines and the concept of instruction set architecture we find in real-life computers. The mathematician's version is limited to a very small set of instructions to make the mathematical proofs easy. But the door is open to expanding the instruction set. John Hopcroft and Jeffrey Ullman elaborate on this point using a variant of register machines called random access machines, or RAM[12] (emphasis in the original):

> Logicians have presented many other formalisms such as λ-calculus, Post systems, and general recursive functions. All have been shown to define the same class of functions, i.e. the partial recursive functions. In addition, abstract computer models, such as the *random access machine* (RAM), also give rise to the partial recursive functions.
>
> The RAM consists of an infinite number of memory words, numbered 0, 1, … , each of which holds any integer, and a finite number of arithmetic registers capable of holding any integer. Integers may be decoded into the usual form of computer instructions. We shall not define the RAM model more formally, but it should be clear that if we choose a suitable set of instructions, the RAM may simulate any existing computer. The proof that the Turing machine formalism is as powerful as the RAM is given below.

This ability to choose a suitable set of instructions is being incorporated in this theorem[13]:

> **Theorem 7.6**: A Turing machine can simulate a RAM, provided that the elementary RAM instructions can themselves be simulated by a TM.

This makes it clear that alterations to the instruction set don't bring the random access machine outside the bounds of mathematics. The resulting machine is still an abstract mathematical machine amenable to mathematical methods such as this theorem 7.6.

Alfred Aho, John Hopcroft and Jeffrey Ullman analyze a different variation of the RAM in chapter 1 of

[Aho 1974]. This particular flavor is augmented with input and output capabilities using Turing machine-like tapes. On the other hand this flavor of the RAM doesn't store the program instruction in the registers. They are built into the finite-state control of the abstract machine. This same chapter from this same book documents a related model, the Random Access Stored Program or RASP. The RASP instructions for the machine are stored in the registers and the machine finite-state control executes them. This RASP is otherwise identical to the RAM from [Aho 1974] including the input/output capabilities. This replicates mathematically the stored program architecture of real-life computers where software is loaded in memory for execution by the CPU. This is the flavor of register machines that is most suited for the virtual machine project.

The take home point of this discussion of register machines is that you will find in mathematical literature the information necessary to show software is mathematics if you look for it. The questions of what is a computation and what is an algorithm are of considerable importance to mathematicians. If engineers develop new ways to carry out computations, mathematicians will want to know whether these discoveries have consequences in their discipline. Their analyses are found in the literature.

Universality and Random Access Stored Program

The reader may have noticed a peculiarity of the proposed virtual machine project. I am not directly arguing that every program is a mathematical algorithm. I am arguing the virtual machine is a mathematical algorithm and therefore every program it executes is a mathematical computation. Can we show the individual program is a mathematical algorithm on its own? To prove this point we need to invoke the individual program semantics. If the program is written in Standard ML and we managed to work around the library limitation we are in business. But if we use C we have a steeper hill to climb. With the virtual machine argument we have one single algorithm that catches all the programs. This peculiar mathematical property is called universality.

The grandfather of universal algorithms is the universal Turing machine . R. Gegory Taylor explains [14] (emphasis in the original):

> In this section we have seen how Turing machines may be encoded as natural numbers in two distinct ways. Our stated motivation for doing this is our intention that Turing machines be capable of taking other Turing machines—albeit in encoded form—as input. The reader is no doubt wondering why one would be interested in doing such a thing in the first place. Why should it be desirable that one Turing machine operate on another in this sense? We will answer the question in the next section, where the important concept of *universal Turing machine* is introduced.
>
> [section heading omitted]
>
> All Turing machines considered so far have been machines in a peculiar sense: They run only under a single program or set of instructions. Change the program and you have changed the machine. This use of a “machine” is at odds with current usage, of course, and no doubts reflects the fact that the beginnings of automata theory predate the advent of modern digital computers. The more usual sense in which computer scientists use the term “machine” allows that a machine (hardware) be capable of running under a variety of programs (software). To put this another way, the modern digital computer might be described as a *universal computing device* in the sense that, suitably programmed and ignoring resources limits, it is capable of computing any number-theoretic functions that is, in principle, computable. The Turing machines that we have considered up to this point lack

> this property of universality, as noted above. Our interest now is in describing a notion of universality for Turing machines.

There is quite a lot in these two paragraphs. Let me start by clarifying into layman's terms some of the mathematical speak, so the fine points are not missed. Turing machines are a model of computation that captures in mathematically precise terms the informal notion of algorithm, more exactly the flavor of algorithms called effective methods. This is called the Church-Turing thesis. In layman terms this means that if something can be computed at all, then a Turing machine can compute it. When one wants a mathematically precise definition of what is computable as opposed to what is not computable, the notion of Turing machine does the trick.

Another point to clarify: Taylor's chosen language limits his discussion to number theoretic functions. Make no mistake. There is a mathematical device called Gödel numbers which means that when we have the capability to make any possible computation with numbers then we can also make any form of computation, whether or not it relates to numbers. Therefore this apparent limitation about numbers is no limitation at all. This is why mathematicians focus intently on number-theoretic functions when they study computability theory. They know that by doing so they implicitly capture all forms of computations.

A last point to clarify: This description of universality is specific to Turing machines. Other such algorithms are known. For the family of register machines, the universal algorithm is the Random Access Stored Program or RASP. Our virtual machine project is to define mathematically and implement a universal algorithm similar to a RASP.

This description of universality from Taylor is in total opposition to current patent law. While *In re Alappat* says programing a computer makes a new machine, Taylor says expressly that computer scientists understand the term “machine” to mean a single universal device that runs a variety of programs. While patent law sees the programming of a computer as a configuration of the circuitry occurring at the hardware level, Taylor says a program is the input given to an algorithm with the universal property.

The significant point I want to raise is that universal algorithms exist. They are documented in the literature. They have been implemented. They cannot be denied or ignored in law because the facts are that they exist and have been implemented.

A legal view of computer programming is explained in *In re Prater*:

> No reason is now apparent to us why, based on the Constitution, statute, or case law, apparatus *and* process claims broad enough to encompass the operation of a programmed general-purpose digital computer are necessarily unpatentable. In one sense, a general-purpose digital computer may be regarded as but a storeroom of parts and/or electrical components. But once a program has been introduced, the general-purpose digital computer becomes a special-purpose digital computer (i. e., a specific electrical circuit with or without electro-mechanical components) which, along with the process by which it operates, may be patented subject, of course, to the requirements of novelty, utility, and non-obviousness. Based on the present law, we see no other reasonable conclusion.

They say a “storeroom of parts”. This gives the vision of heaps of nonfunctional disconnected hardware. But once the parts have been connected by programming, the specific circuit brought into existence can start working. Such a computer can be built. The older version of the ENIAC was programmed in this manner with a plug board that had to be manually wired.

Let's imagine we have such a storeroom of computer parts. We program it with a universal algorithm.

Now we no longer have a storeroom of parts. We have a working computer which has been programmed to execute a mathematical algorithm. Because this algorithm has the universal property, we still have a generic machine able to execute any program of our choosing. However we have changed the programming method. We no longer need to configure the circuitry. We achieve the same result by mathematical means.

This is the significant point. There are many ways to program a computer. Some of them require physical configuration of circuitry. Others use mathematical means. The ENIAC used the configuration of circuitry. Modern computers use the mathematical way.

Here I expect the objection: how about those programs which are not mathematical algorithms? According to this objection they are not covered by this mathematical theory. The answer is there is no such thing. All programs are mathematical algorithms. The point of the virtual machine project is to prove this experimentally.

The unprogrammed virtual machine is software running on a powered up and fully functioning computer. It displays a user interface which reacts to user input. If we show this in court, all notions that the unprogrammed computer is a nonfunctional storeroom of parts is untenable. The computer is already in a working state before the virtual machine has been programmed. The next step is when the demonstrator opens up a file open dialog that displays a choice of Linux and BSD distributions. This is input. This dialog is the standard way we inform a program of which input it must use. Everyone will clearly see this is not hardware configuration but the usual ordinary operation of supplying input to a program. The final step is to show the working distribution. All programs run. The demonstration shows everyone the universal algorithm works exactly like the mathematical theory says it works.

This is unsettling to legal doctrines. For example there is a question of novelty. Imagine a patent drafted as per Quinn's recommendation. It contains a design document saying how to write the software but there is no information on how it is compiled into executable form. What exactly is patented? This patent will read on a new input on an old machine running an old universal mathematical algorithm. Or consider the Microsoft-Symatec-Phillips theory that the activity of transistors is the patentable process. When we implement a universal algorithm, the activity of transistors is this universal algorithm. All programs are input to this algorithm. When executing a specific program the transistor activity is always the universal algorithm.

Implication on Hardware Architecture

Attorney Thomas Gallagher makes this point in his newsletter:

> The mistake many people make is reading the idea of software absent the hardware that makes it work. Most software patent claims are written in "means plus function"** style USC §112 ¶6). Software cannot function without hardware, but as one gifted inventor once told me "the only difference between hardware and software is bandwidth." He was referring to implementing a function in dedicated hardware as compared to implementing it on a general purpose processor programmed with software to mimic the dedicated hardware.
>
> When you consider software as a functional description of a process carried out by a machine, rather than prose, it makes perfect sense. To think of software as merely prose is superficial and delusional. Everything written in software could be accomplished by a hardwired system of gates. Thus, "means plus function". Why should a dedicated processing system be granted greater protection than a programmed general processing

> system?

This point applies to a universal algorithm. If we take the virtual machine of our project and implement it in circuitry we get a CPU connected to memory and IO ports on a motherboard. This is the reverse of Minsky's ruthless abstraction. Instead of abstracting away everything that isn't information to get an abstract machine, we look at how the abstract machine ends up being physically implemented.

The two key concepts are the instruction cycle and the instruction set architecture.

The instruction set architecture is the definition of all the instructions the computer can execute. The program can only be written as a series of the instructions from the instruction set of this computer. Patterson and Hennessy describe this instruction set architecture like this[15] (emphasis in the original):

> Both hardware and software consist of hierarchical layers, with each lower layer hiding details from the level above. This principle of *abstraction* is the way both hardware designers and software designers cope with the complexity of computer systems. One key interface between the levels of abstraction is the *instruction set architecture*—the interface between the hardware and low-level software. This abstract interface enables many *implementations* of varying cost and performance to run identical software.

The instructions are executed one after another in a loop. This is called the instruction cycle. Here is how Hamacher, Vranesic and Zaky describe the instruction cycle[16] (emphasis in the original):

> Let us consider how this program is executed. The processor contains a register called the *program counter* (PC) which holds the address of the instruction to be executed next. To begin executing a program, the address of its first instruction (*i* in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing*. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Move instruction at location *i* + 8 is executed the PC contains the value *i* + 12 which is the address of the first instruction of the next program segment.
>
> Executing a given instruction is a two-phase procedure. In the first phase, called *instruction fetch*, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register* (IR) of the processor. At the start of the second phase, called *instruction execute*, the instruction in IR is examined to determine which operation to be performed. The specified operation is then performed by the processor. This often involve fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point at the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

This is an outline of the virtual machine algorithm translated in hardware terms as per Thomas Gallagher's suggestion.

If we take our virtual machine software and etch it in circuitry we will find that the RASP-like algorithm defines the instruction set architecture and the instruction cycle of a computer. The converse is also true. If we run a physical computer through Minsky's ruthless abstraction what will be left is an abstract version of the instruction set architecture and the instruction cycle and this is a RASP-like universal algorithm.

This correspondence between hardware implementation and mathematical definition is part of why all software is a mathematical algorithm, possibly limited in a patent claim to a particular real-world semantics.

Speech

Now let's return to the question, What if the algorithm is limited to a specific real-world semantics? The question is relevant for two reasons. The first one is the legal definition of algorithm. In *Benson* the Supreme Court defined an algorithm to be a “procedure for solving a given type of mathematical problem”. This is the court's own words. What if the problem the patent attempts to solve is defined in terms that are not mathematical? Does it mean the procedure is not a mathematical algorithm? Note the holding of the Supreme Court in *Diehr*: ”a claim drawn to subject matter otherwise statutory does not become nonstatutory simply because it uses a mathematical formula, computer program, or digital computer.” Therefore even though all software is a mathematical algorithm, there is still a question of where the claim is drawn to subject matter that is otherwise statutory. In such case the patent may issue. But should it?

Let's consider the notion of a mathematical description of the real world. If we describe a rocket with mathematical equations, the rocket is still patentable. The problem being solved, flying a rocket in the sky, is not a mathematical problem. How about the equations and the resulting calculations? Are they patentable because they describe a rocket?

Mathematics is a language with an alphabet, a syntax and a semantics.[17] Mathematical language is used to state facts. Mathematical logic is used to derive more truths from already known truths. This is called proving a theorem. Mathematical language is used for reasoning and increasing knowledge by processes of thought.

Mathematics may be given an additional layer of semantics when abstract mathematical ideas are used to describe facts about the real world. Mathematics is speech, in other words. Keith Devlin gives us an example[18] (emphasis in the original):

> It is mathematics that allows us to 'see' electromagnetic waves, to create them, to control them, and to make use of them. Mathematics provides the only description we have of electromagnetic waves. Indeed, as far as we know, they are *waves* only in the sense that the mathematics treats them as waves. In other words, the mathematics we use to handle the phenomenon we call electromagnetic radiation is a theory of wave motion.

The universal algorithm etched into circuitry doesn't describe the electronic activity of transistors in a computer any more than a mathematical formula describes the marks of inks representing it in a textbook of physics. Minsky's ruthless abstraction sees to this point. But this algorithm has the power to make the calculations that will let us know the geographic coverage of a radio station broadcast signal. The computer is useful in solving real-world problems because it permits a more efficient use of mathematical speech than pencil and paper calculations.

When we use a computer to control an industrial process to cure rubber, the industrial process is currently patentable. When we use mathematical equations to describe how to build an antenna, the antenna is patentable. But the equations involved in these two situations are mathematical speech. Even though some real-world semantics is added to the mathematics, the resulting speech is speech. The semantics of speech is not something patent law is intended to patent.

The Effect of Real World Semantics on Abstract Mathematical Problems

Real-world semantics doesn't change the underlying mathematics. The written symbols are the same and the rules of the mathematical language and mathematical logic are the same. In terms of computation, the mathematical operations are the same regardless of whether we compute for the sake of doing mathematics or whether there is a practical application.

One may ask how exactly real-world semantics can turn an abstract mathematical idea into some concrete use of mathematics. If we want to test an industrial process for curing rubber, we need to cure actual rubber. If we want to test an antenna built according to mathematical specification, we need the actual antenna. But when we test a computer program, we often use fictitious test data. Fiction is abstract. A mathematical calculation works in a fictional situation just the same as in a real-world situation. So what exactly is concrete in a program? Imagine a patent on software. It will read on a computer running a mathematical algorithm on fictional data. Where is the concrete part in this scenario? It takes more than semantics to make a concrete use of mathematics. We need the actual concrete thing, not just a reference to it.

Mathematics has a long tradition of describing problems using real-world terms. You will find many examples of such problems in recreational mathematics.[19] However this tradition is not limited to games and puzzles. It is also the source of many fundamental mathematical discoveries. Keith Devlin's book *The Language of Mathematics, Making the Invisible Visible* describes several actual occurrences of mathematical discoveries made in this manner. Here is one example[20] (emphasis in the original, figures omitted):

> As is so often the case in mathematics, the broad-ranging subject known as topology has its origins in a seemingly simple recreational puzzle, the Königsberg bridges problem.
>
> The city of Königsberg, which was located on the river Pregel in East Prussia, included two islands, joined together by a bridge. As show in Figure 6.1, one island was also connected to each bank by a single bridge, and the other island had two bridges to each bank. It was the habit of the more energetic citizens of Königsberg to go for a long family walk each Sunday, and, naturally enough, their paths would often take them over several of the bridges. An obvious question was whether there was a route that traversed each bridge exactly once.
>
> Euler solved the problem in 1735. He realized that the exact layout of the islands and bridges is irrelevant. What is important is the way in which the bridges connect—that is to say, the *network* formed by bridges, illustrated in figure 6.2. The actual layout of the river, islands, and bridges—that is to say, the geometry of the problem—is irrelevant.

You will find a version of the figures in the Wikipedia article on the Königsberg bridges problem together with Euler's analysis of the problem. This particular problem, and others like it, shows that one cannot tell whether a problem is abstract or concrete from a statement of the problem alone. The language used may be full of irrelevant details that must be abstracted away during the solution-finding process until we are left with the core abstract mathematical ideas. This abstraction is part of what mathematics is about. A key part of the solution of the Königsberg bridges problem is to abstract away the bridges, the river, the island and people making a journey in the city.

This is typical of all problems having a mathematical solution. The real world references must be abstracted away and the problem must be solved by mathematical means. Only then, if the problem is not a fictional one, we reintroduce the real world semantics to translate the mathematical solution back into real-world terms. This is what is done when a computer-implemented mathematical algorithm is used to solve a real-world problem.

The Effect of Real World Semantics on Reduction of Software to Hardware

If we define the patented process in terms of transistor activity and the patented machine in terms of hardware configuration, the transistor activity and the machine configuration will be the same whether or not the mathematical algorithm carries a real-world semantics. We may ask the question of what exactly is patented when a layer of semantics is added to an otherwise mathematical computation. This is not patenting anything definable in hardware terms. This is applying a field-of-use limitation which is defined in terms of semantics.

The Notion of Formal System

When a computer-implemented computation is viewed as something entirely reducible to hardware, the issue of speech doesn't arise. The patent just reads on either a machine or machine-implemented process. But we have shown this view is incorrect. Software is not reducible to hardware in the same way as a book is not reducible to ink and paper. If we submit a computer to Minsky's ruthless abstraction procedure, we are left with a RASP-like universal algorithm. The question is how exactly is this algorithm speech? The answer requires a deeper understanding of how exactly mathematics is speech. Let's look into this.

A first clue is given by Marvin Minsky, citing Emil Post[21] (emphasis in the original):

> Even the most powerful mathematical system or logical system is ultimately, in effect, nothing but a set of rules that tell how some *strings of symbols* may be transformed into other *strings of symbols*.

This is a reference to the written nature of mathematics. The abstract mathematical ideas may be defined in any way we please; they are inaccessible unless we use text to describe them. But it goes further than this. It alludes to the peculiar fact that mathematical logic may be defined using syntax alone. The symbols may — and usually have — a semantics but the semantics is not needed to define the rules that logically transform a mathematical statement into another mathematical statement. Here I am referring to the concept of mathematical proofs.

Mathematicians and logicians have analyzed the laws of mathematics itself using mathematical means. They call this metamathematics, or proof theory. They have found that the criteria for validity of mathematical proofs are definable without making reference to the semantics of the symbols. Stephen Kleene explains the procedure[22] (emphasis in the original):

> To discuss a formal system, which includes both defining it (i.e. specifying its formation and transformation rules) and investigating the result, we operate in another theory language, which we call the *metatheory* or *metalanguage* or *syntax language*. In contrast, the formal system is the *object theory* or *object language*. The study of a formal system, carried out in the metalanguage as part of informal mathematics, we call *metamathematics* or *proof theory*.
>
> For the metalanguage we use ordinary English and operate informally, i.e. on the basis of

> meanings rather than formal rules (which would require a metalanguage for their statement and use). Since in the metamathematics English is being applied to the discussion only of the symbols, sequences of symbols, etc. of the object language, which constitutes a relatively tangible subject matter, it should be free in this context from the lack of clarity that was one of the reasons for formalizing.
>
> Since a formal system (usually) results by formalizing portions of existing informal or semiformal mathematics, its symbols, formulas etc. will have meanings or interpretations in terms of that informal or semiformal mathematics. These meanings together we call the (*intended* or *usual* or *standard*) *interpretation* or *interpretations* of the formal system. If we were not aware of this interpretation, the formal system would be devoid of interest for us. But the metamathematics, to accomplish its purpose, must study the formal system as just itself, i.e. as a system of meaningless symbols, and may not take into account its interpretation. When we speak of the interpretation, we are not doing metamathematics.

Howard Delong fills in more details on what it means to view a mathematical proof as a system of meaningless symbols[23] (emphasis in the original):

> When we apply our transformation rules to the initial formulas the result is a theorem. The exhibition of the application of the rules is a *proof*. More explicitly, a *proof* is a finite sequence of formulas, such that each formula is an initial formula or follows from an earlier formula by the application of a transformation rule. The last line of the proof is a *theorem.* We require that the transformation rules be such that there is a merely a mechanical procedure to determine whether or not a given sequence of formulas is a proof. Note that this requirement is important; it ultimately derives from the idea of the Pre-Socratics that there is no royal road to knowledge. If knowledge is claimed, and a proof is given as evidence, this proof must be open to inspection by all. This requirement distinguishes logical proofs from some theological "proofs" (of God's existence) where "faith" or "grace" is needed to "see" the so-called proofs. What is needed is that no ingenuity or special insight be needed; in other words, that it is mechanical.

This requirement that the test of validity of the proof must be mechanical implies that there must be an algorithm applicable to text. Delong himself makes this point on page 132 (emphasis in the original):

> Any effective method by which it can be determined whether or not an arbitrary formula of a formal system is a theorem is called a *decision procedure*. By *effective finite method* is meant the same thing that used to be called an *algorithm*.

One of the points of using an algorithm is to achieve mathematical rigor. There is no place for human judgment in an algorithm. This makes the validity of a mathematical proof a matter of objective truth, something that can be verified without relying on the opinion of a human. This removes the possibility that different humans may hold different opinions on what is the mathematically proven truth. This is very different from the law where human judgment is required at every turn.

Most of the time mathematicians write their proofs informally. They don't bother to work out the tedious details required by the formal proof. But mathematicians know what the proof should look like if they work out such details. In case of a dispute on the validity of a proof they will fill in the formal details until they can ascertain whether or not the proof holds. The formal system serves as a reference, an objective test, as to which formulas are mathematically proven.

How it is possible to do away with human judgment and replace it with an algorithm? Kleene explains what is known as the axiomatic method[24]:

> The system of these propositions must be made entirely explicit. Not all of the propositions can be written down, but rather the disciple and student of the theory should be told all the conditions which determine what propositions hold in the theory.
>
> As the first step, the propositions of the theory should be arranged deductively, some of them, from which the others are logically deducible, being specified as the axioms (or postulates).
>
> This step will not be finished until all the properties of the undefined or technical terms of the theory which matter for the deduction of the theorems have been expressed by axioms. Then it should be possible to perform the deductions treating the technical terms as words in themselves without meaning. For to say that they have meanings necessary for the deduction of the theorems, other than what they derive from the axioms which govern them, amounts to saying that not all of their properties which matter for the deductions have been expressed by axioms. When the meanings of the technical terms are thus left out of account, we have arrived at the standpoint of formal axiomatics.

So leaving meanings out of account is a question of being exhaustive in making explicit what is normally implicit in informal theories. The above explanation is about axioms, the initial formulas of the theory from which all proofs must derive. We must also take care of the transformation rules which are the rules of logic for making inferences. Kleene resumes his explanation:[25] (emphasis in the original):

> At any rate, we are still short of our goal of making explicit all the conditions which determine what propositions hold in the theory. For we have not specified the logical principles to be used in the deductions. These principles are not the same for all theories, as we are now well aware.
>
> In order to make these explicit, a second step is required, which completes the step previously carried out for the so-called technical terms in respect to the non-grammatical part of their meanings. All the meanings of all the words are left out of account, and all the conditions which govern their use in the theory are stated explicitly. The logical principles which formerly entered implicitly through the meanings of the ordinary terms will now be given effect in part perhaps by new axioms, and in some part at least by rules permitting the inference of one sentence from another or others. Since we have abstracted entirely from the content or matter, leaving only the form, we say that the original theory has been *formalized*.

At this point, we can test based only on syntactic form whether a sequence of symbols is a formula in the theory, whether a given formula is an axiom, and whether a sequence of formulas is a theorem. All these determinations are made by an algorithm without resorting to human judgment. Once a theory has been formalized, logic has been reduced to syntax. This is why the study of formal systems is often called proof theory.

This is a point where the gap between mathematical logic and legal logic is wide. It is easy for a lawyer unaware of the mathematical viewpoint to make mistakes. Imagine a lawyer, thinking that mathematical logic works like legal logic, arguing that computer algorithms are not speech because they are the automated work of electronics unable to understand semantics. To an audience of lawyers this argument will probably sound plausible. In front of an audience of mathematicians and computer scientists the reaction will probably be bursts of laughter or heads shaken in disbelief. I don't say this to be dismissive of the legal profession. I am warning how easy it is to make basic mistakes when one assumes his understanding of legal logic applies to mathematics. The error is that this argument ignores the effect of formalizing a theory on the relationship between syntax and semantics. Meaning is not ignored or suppressed. It is made exhaustively explicit in the syntax to the point that there is no need to refer to semantics when writing mathematical proofs or carrying out a computation. Therefore issues of meaning can be analyzed mathematically by algorithms looking at syntax alone.

All of this doesn't mean the semantics is ignored. Mathematicians also use model theory that must complement and agree with proof theory. Then the pure syntactical manipulations of symbols have a semantics in the model. This relationship is explained by Kleene as follows[26] (emphasis in the original):

> The formal systems which are studied in metamathematics are (usually) so chosen that they serve as models for parts of informal mathematics and logic with which we are already more or less familiar, and from which they arose by formalization. The meanings which are intended to be attached to the symbols, formulas, etc. of a given formal system, in considering the system as a formalization of an informal theory, we call the (*intended*) *interpretation* of the system (or of its symbols, formulas, etc.). In other words, the interpretations of the symbols, formulas, etc. are the objects, propositions, etc. of the informal theory which are correlated under the method by which the system constitutes a model for the formal theory.

In other words we have some mathematical theory which is used informally. Proof theory turns it into a rigid system of syntactic transformations called a formal system. But mathematicians want the formal system to retain the original semantics of the informal version. Therefore they correlate the formal system with the informal semantics with model theory. When this correlation is done successfully the syntactic transformation correctly preserves the semantics even though the semantics is not used in the formal manipulations of the language. This is how syntax, rules of logic, semantics and algorithms work together in mathematical languages.

How Algorithms Solve Problems

Here I have only half-answered the original question of how a computation is speech by pointing to the connection between algorithms and mathematical logic. The other half is how we use algorithms to solve problems. This point is important because it connects to the legal understanding of algorithms as procedures to solve mathematical problems.

Computers don't have the cognitive capabilities of humans. They can only take bits as input and produce bits as output. The input may come to the algorithm from an external source like a keyboard, or the algorithm may find its input already loaded in memory. Similarly the output may be left in memory or it may be sent to an external party, like a display. In any case, the algorithm is a transformation from a string of symbols, the bits, into another string of symbols. The only way the semantics can be accounted for is by formalizing the problem to such details that everything that matters is represented syntactically as bits. Only then will the mathematical algorithm be able to produce a meaningful answer.

Note that this doesn't mean the semantics is absent. On the contrary the bits have a semantics. Without a semantics the problem would never get solved. But the computer doesn't need the semantics and doesn't use it. The computer is physically incapable of using anything that isn't explicitly represented syntactically as bits.

Raymond Greenlaw and James Hoover have dedicated a chapter of their book on the relationship between language, problems and algorithms solving them.[27] Here is an extract of the introduction[28] (emphasis in the original):

> Even equipped with a fancy graphical user interface, a computer remains fundamentally a symbol manipulator. Unlike the natural languages of humans, each symbol is precise and unambiguous. A computer takes sequences of precisely defined symbols as inputs, manipulates them according to its program, and outputs sequences of similarly precise symbols. If a problem cannot be expressed symbolically in some languages, then it cannot be studied using the tools of computation theory.
>
> Thus the first step of understanding a problem is to design a language of communicating that problem to a machine. In this sense, a *language* is the fundamental object of computability.

Then the authors proceed in this chapter to give a mathematically precise description of what it means to define a language suitable to represent a problem and how an algorithm can solve problems by processing the strings of symbols in the language.

The Connection of Computation with Language and Logic

The relationship between computation, language and logic is deep. When we dig in this direction, we find the fundamental mathematical principles that make possible the development of tools like Coq. I give here an outline of some of the fundamentals. This only scratches the surface of the issues but for the purposes of this article this is sufficient.[29]

Algorithms are related to the notion of mathematical functions. Minsky explains[30] (emphasis in the original):

> *What is a function*? Mathematicians have several more or less equivalent ways of defining this. Perhaps the more usual definition is something like this:
>
> > A *function* is a rule whereby, given a number (called the argument), one is told how to compute another number (called the *value* of the function for that argument).
>
> For example, suppose the rule that defines a function F is "the reminder when the argument is divided by three." Then (if we consider only non-negative integers for arguments) we find that
>
> > $F(0) = 0, F(1) = 1, F(2) = 2, F(3) = 0, F(4) = 1,$ etc.
>
> Another way mathematicians may define a function is:
>
> > A function is a set of ordered pairs $\langle x, y\rangle$ such that there are no two pairs with the same first number, but for each x, there is always a pair with that x as its first number.

> If we think of a function in this way, then the function *F* above is the set of pairs
>
> ⟨0,0⟩ , ⟨1,1⟩ , ⟨2,2⟩ , ⟨3,0⟩ , ⟨4,1⟩ , ⟨5,2⟩ , ⟨6,0⟩ …
>
> Is there any difference between these definitions? Not really, but there are several fine points. The second definition is terribly neat; it avoids many tricky logical points—to compute the value of a function for any argument *x*, one just finds the pair that starts with *x* and the value is the second half of the pair. No mention is made of what the *rule* really is. (There might not even be one, though that leaves the uncomfortable question of what one could use the function for, or in what sense it really exists.) The first definition ties the function down to some stated rule for computing its values, but that leaves us the question of what to do if we can think of different rules for computing that same thing!

I like Minsky's description of a function because it is very accessible to a layman. However be aware that Minsky is oversimplifying things for the sake of clarity.

Minsky has limited himself to number-theoretic functions but mathematicians have a notion of function that uses any type of arguments and produces any type of values. They also have a notion of functions that takes multiple arguments or returns multiple values bundled in a pair, a triple etc.

Algorithms are rules that are used in defining functions. Therefore there is a correspondence between the notion of algorithm and the notion of function in that every algorithm defines a function. But again we must beware of oversimplifications. Some flavors of algorithms allow infinite loops when some arguments are provided. Then the function is only partially defined in that there is no value for the argument that causes the infinite loop. Other flavors of algorithms are non-deterministic or probabilistic. In this case the function produces a set of values instead of a specific value.

In the language of mathematics function symbols are used to express terms. These terms are the equivalent to nouns and noun phrases. [31] For example suppose I say “I have four apples on the left side of my desk and six oranges on the right side for a total of ten fruits.” The phrase “total of ten fruits” is a noun phrase. It refers to the result of a computation which is making the addition of the numbers 4 and 6. Why an addition? This is because the definition of “total of” requires an addition. “Total” is a name given to a function. The well-known procedure for computing an addition we have learned in school is the algorithm defining this function. The statement linking four apples and six oranges with the total of ten fruits is a logical inference based on this definition. This is a simple example but it illustrates accurately the link between computation, logic, and the parts of speech.

In mathematical language, definitions of functions often take the form of an equation. The function being defined is on the left-hand side of the equal sign and the defining calculation the right-hand side. For example this may be a (humorous) definition of the tax being owed.

> Tax(income, expenses) = income - expenses

An equation such as this one may be true or false. In this case it is false. The calculation is not the one specified in the tax code. A definition which states a correct calculation will be true. This example shows that an equation may state a truth without stating a mathematical truth. A tax code definition states a truth about the tax code and not a truth of mathematics.

Abstract Ideas

In an amicus brief in the *Bilski* case when it was before the US Supreme Court, the Software Freedom

Law Center explained why the First Amendment precludes the patenting of abstract ideas. The SFLC also made a strong plea that software in its source code form shouldn't infringe on the patent.

The ideas explained in this article go well beyond the SFLC argument in that case. First, mathematical speech is not limited to abstract ideas. The semantics of the software may be concrete and still be speech. If for any reason the courts decide that semantics is a valid reason to decide a patented method is not abstract, and I understand the courts have taken this view in the past, this would not excuse the patent from being a patent on speech.

Next, I argue that the computation as executed by the transistors in the computer is speech. This is because software does not reduce strictly to a hardware phenomenon. If we perform Minsky's ruthless abstraction procedure on a computer we are left with a universal mathematical algorithm in the same manner that a similar abstraction procedure performed on a book leaves us with an intangible sequence of letters in an alphabet constituting a text in some language such as English. This algorithm is implementing one of the dynamic features of mathematical language and such features don't stop being speech just because they are dynamic.

A possible objection to the speech nature of computing may be that the machine activity is not meant to be watched by a human mind. The argument might be some variation on the theme that there can be no speech if there is no human to receive the message. The answer to this objection is that the semantics of software and computers exist. This semantics is used for speech purposes even though humans are not watching the bits as the computation progresses. Here is an example provided by Keith Devlin[32] (emphasis in the original):

> One of the earliest topological questions to be investigated concerned the coloring of maps. The four color problem, formulated in 1852, asked how many colors are needed to draw a map, subject to the requirement that no two regions having a stretch of common border be the same color.
>
> Many simple maps cannot be colored using just three colors. On the other hand, for most maps, such as the county map of Great Britain shown on plate 15, four colors suffice. The four color conjecture proposed that four colors would suffice to color *any* map in the plane. Over the years, a number of professional mathematicians attempted to prove this conjecture, as did many amateurs. Because the problem asked about all possible maps, not just some particular maps, there was no hope of proving that four colors would suffice by looking at any particular map.
>
> …
>
> In 1976, Kenneth Appel and Wolfgang Haken solved the problem, and the four color conjecture became the four color theorem. A revolutionary aspect of their proof was that it made essential use of a computer. The four color theorem was the first theorem for which no one could read the complete proof. Parts of the argument required the analysis of so many cases that no human could follow them all. Instead, mathematicians had to content themselves with checking the computer program that examined all those cases.

Mathematical logic is a means to deduce more truths from known truths using pure logic. But, as the four color theorem shows, there is no requirement in mathematics that the use of logic fit within the limitations of live humans.

Computers are tools used to access truths that would otherwise be inaccessible to humans. This is true for automated theorem proving. This is also true for other information processing applications.

Further Readings

This completes my promised tour of the four topics: disclosure, innovation, mathematics and speech. There is much more to say on the themes of software being mathematics and mathematics being speech. There is ample factual evidence of these two propositions available to those who set out to investigate the relevant literature. I promise this research will reveal additional arguments and fine points. If you are interested, here is my list of favorite sources and what you may find useful in such a quest.

Explaining the Language of Mathematics to Laymen

I highly recommend [Devlin 2000]. If you can afford to read only one of my suggested references this one should be your choice. This is a wonderful book explaining the language of mathematics to an audience of laymen able to understand high school mathematics. The prologue explains what is mathematics in accessible language. This book discusses the relationship between mathematics, the physical world, the written language and the cognitive capabilities of the human mind. It is a treasure trove of historical details, practical examples and clear and lucid explanations of the fundamental issues. The second chapter is entirely dedicated to mathematical logic. If you need to make a case that requires explaining some fundamentals of mathematics to an audience of laymen you will find this book very useful.

Philosophy of Mathematics

I also recommend learning the basics of the philosophy of mathematics. Questions such as what is a mathematical proof, what is an abstract mathematical object, whether they exist in the human mind, and what is a mathematical truth belong to philosophy of mathematics. This is clearly relevant to questions such as what is an abstract idea, what is a fundamental truth in computer science and what is mathematical speech. Mathematicians concerned with the foundations of mathematics have spent considerable efforts thinking about the philosophy of mathematics. The very definition of when a proof is acceptable in mathematics depends on it.

Philosophy of mathematics is a controversial topic. Oftentimes laymen will bring up their conceptions of mathematics they think are obvious and stand without saying. Oftentimes this amounts to taking a position on a very controversial issue of philosophy of mathematics. Oftentimes a layman will hold to some belief that every expert thinks is indefensible. Knowledge of these controversies will help a lawyer steer a court clear from many quagmires and refute many apparently plausible but fundamentally faulty arguments. Some good sources are:

- The article on philosophy of mathematics on Wikipedia, as of April 26, 2011
- The article on philosophy of mathematics in the Stanford Encyclopedia of Philosophy
- The article on abstract objects in the Stanford Encyclopedia of Philosophy
- Chapter III of [Kleene 1952]
- Chapter IV of [Kleene 1967]
- I have not yet read [Benacerraf 1984]. I still want to share this reference because the table of contents is very appealing. This is an anthology of essays written by the foremost mathematicians of the twentieth century including all the main proponents of the major philosophies.

Mathematical Logic

The easiest introduction to mathematical logic for laymen is the second chapter of [Devlin 2000].

If you want to dig more on the topic, [Delong 1970] is a university level textbook that explains the historical roots, the nature and the philosophical implications of mathematical logic. Without this background this is a dry and very technical topic. Delong eases the learning curve greatly and requires only the knowledge of high school mathematics.

[Kleene 1967] is a classic. It is highly technical but it includes a detailed explanation of the relationship between the formal mathematical logic and the informal logic of everyday language. This is the only source I know which supplies this information in this degree of details. You will find it on pages 58-73, 134-147 and 164-169.

Another classical text, [Kleene 1952], is a graduate level textbook that provides an alternative exposition for large parts of [Kleene 1967] but not the connection with everyday language and logic. Section 15 is by far the most lucid and clearly written description of the process of formalizing a mathematical language I know of. It is vastly superior to its equivalent from [Kleene 1967]. Extracts from this explanation have been quoted in this article.

[Ben-Ari 2001] takes a more algorithmic approach to mathematical logic compared to the other texts. The author moves quickly over the fundamentals to reach topics such as how to write algorithms for automated theorem proving and how to write and verify the specifications of programs. An important feature is its coverage of the resolution procedure which forms the basis of logic programming.

The Definition of Mathematical Algorithm and Computation Theory

Pretty much every book on the theory of computation has a section explaining what is a mathematical algorithm. Few write a definition. You can find one in [Kleene 1967] on page 223, however, with a continuation on page 226. [Knuth 1973] section 1.1 is a must read. I also recommend [Minsky 1967] chapter 5 and [Rogers 1987] section 1.1.

[Greenlaw 1998] chapter 2 is a description of how mathematical algorithms solve problems by means of language. Chapter 9 is dedicated to the "Boolean circuit" mathematical model of computation. This is the model we get when we run the notion of digital circuits made of logic gates through Minsky's ruthless abstraction.

[Minsky 1967] is a must read. This book is my the second most recommended reference after [Devlin 2000]. This author makes a deliberate and conscious effort to explain the theory of computation in simple and plain English to the maximum extent possible. He shows that a very large part of it can be explained in this manner. He resorts to mathematical formulas only when there is no other way. He also thoroughly covers all important aspects known at the time. Unfortunately this book is out of print.

Usually books on computation theory will provide a technical reference to a selection of models of computation. This sort of text is not for laymen. I like the selections of models of computation found in [Minsky 1967] and from [Taylor 1998]. Together these two sources cover many of the more important models. Unfortunately both books are out of print.

History of Computing

An historical account of the development of mathematical logic, theory of computation and invention of the digital computer is found in [Davis 2000].

The Turing Archives for the History of Computing is also a great source of material. See in particular their Brief History of Computing.

Appendix A—The Definition of Mathematical Algorithm

The term "mathematical algorithm" is a term of art in mathematics. It has a definition which is found in textbooks of mathematics. I understand that the courts have treated this term as a legal term with a legal definition. There is no reason to give this term any meaning other than the one used in mathematics. When the courts have tried to determine the legal meaning of "mathematical algorithm" they failed. In particular the Federal Circuit has given up understanding what is an algorithm, from all I can tell. In *In re Warderman* they ruled:

> The difficulty is that there is no clear agreement as to what is a "mathematical algorithm", which makes rather dicey the determination of whether the claim as a whole is no more than that. *See* Schrader, 22 F.3d at 292 n. 5, 30 USPQ2d at 1457 n. 5, and the dissent thereto. An alternative to creating these arbitrary definitional terms which deviate from those used in the statute may lie simply in returning to the language of the statute and the Supreme Court's basic principles as enunciated in *Diehr,* and eschewing efforts to describe nonstatutory subject matter in other terms.

Reliance on the knowledge of mathematicians will solve this difficulty. The method of identifying a model of computation and then verifying that the method may be expressed in this model is unambiguous and based on factual mathematical knowledge. There is no need and no justification for referring to the statute or some ambiguous and arbitrary legal definition to understand a term of art in mathematics. This is certainly an approach which has a much better factual foundation than trying to figure out what is an abstract idea for purposes of patent law.

In my opinion, the so-called definition from *Benson* reads best as a statement of one of the facts of the case in front of the court, located as it is in the middle of the summary of the facts of the case. A mathematically correct definition should be responsive to this fact of the *Benson* case while a mathematically incorrect definition would not. I believe, and this is my personal theory based on the cases I have read so far, that much of the difficulties the courts have encountered in implementing *Benson* come from the inability of the parties to bring to the courts a correct definition in the discipline of mathematics. The courts have properly rejected the mathematically invalid definitions that were brought in front of them but without the knowledge of a mathematically valid definition they were left with the words of the Supreme Court as their sole guidance. This proved insufficient to make good law to match realities of mathematics.

Where do we find a good definition of 'algorithm'? I propose a definition from Stephen Kleene. This author is highly competent to write on these matters. He is a doctoral student of Alonzo Church of the Church-Turing thesis fame. He is qualified to write a competent definition of algorithm. According to the biography (found at the above link):

> At a lecture in the University of Chicago in 1995, Robert Soare described Kleene's work in these terms:
>
>> Kleene's formulation of computable function via six schemata is one of the most succinct and useful, and his previous work on lambda functions played a major role in supporting Church's Thesis that these classes coincide with the

> intuitively calculable functions.From 1930's on Kleene more than any other mathematician developed the notions of computability and effective process in all their forms both abstract and concrete, both mathematical and philosophical. He tended to lay foundations for an area and then move to the next, as each successive one blossomed into a major research area in his wake.Kleene developed a diverse array of topics in computability: the arithmetical hierarchy, degrees of computability, computable ordinals and hyperarithmetic theory, finite automata and regular sets with enormous consequences for computer science, computability on higher types, recursive realizability for intuitionistic arithmetic with consequences for philosphy and for program correctness in computer science.

Kleene's definition is below. [33] It defines procedures for solving mathematical problems as legally required by *Benson*.

> Consider a given countably infinite class of mathematical or logical questions, each of them which calls for a “yes” or “no” answer.
>
> Is there a method or procedure by which we can answer any question in the class in a finite number of steps?
>
> In more detail, we inquire whether for the given class of questions a procedure can be described, or a set of rules or instructions listed, once and for all to serve as follows. If (*after* the procedure has been described) we select *any* question of the class, the procedure will then tell us how to perform successive steps, after a finite number of which we will have the answer to the question we selected. In performing the steps we have only to follow the instructions mechanically, like robots; no insight or ingenuity or intervention is required of us. After any step, if we don't have the answer yet, the instruction together with the existing situation will tell us what to do next[34]. The instructions will enable us to recognize when the steps come to an end, and to read off from the resulting situation the answer to the question, “yes” or “no”.
>
> In particular, since no human performer can utilize more that [sic] a finite amount of information, the description of the procedure, by a list of rules or instructions, must be finite.
>
> If such a procedure exists, it is called a *decision procedure* or *algorithm* for the given class of questions. The problem of discovering a decision procedure is called the *decision problem* for this class.

The core idea is that the instructions must be self-contained. They must be executed mechanically using only the rules and the current situation for guidance on the next step without having to make additional decisions based on insight or ingenuity. But still the rules must be such that the correct answer will be reached. A key feature of algorithms is that all information required for the accuracy of the answer must be contained in the rules and the information being processed.

This definition appears to limit the definition of algorithm to problems, here dubbed “questions”, that demand a “yes” or “no” answer. This is not the case. Kleene later follows on[35]: (emphasis in the original)

> We begin by observing that, just as we may have a decision procedure or algorithm for a countably infinite class of questions each calling for a “yes” or “no” answer, we may have a *computation procedure* or *algorithm* for a countably infinite class of questions which require as answer the exhibiting of some object.
>
> For example, there is a computation procedure for the class of questions “What is the sum of two natural numbers *a* and *b*?”. We learned this procedure in elementary school when we learned to add. The long division process constitutes an algorithm for the class of questions “For given positive integers *a* and *b*, what are the natural numbers *q* (the quotient) and *r* (the remainder) such that $a = bq + r$ and *r* b?”.

From this we learn that the procedures for carrying out ordinary pencil and paper arithmetic that we have learned in school are examples of algorithms.

Kleene's concept of a class of questions correspond to the set of possible inputs to the algorithm. This is Kleene's chosen terminology for circumscribing the type of problems the algorithm will solve.

Flavors of Algorithms

Mathematicians also consider various variants of the above definition where one or another of the provisions are altered.

Kleene himself has mentioned some flavors by considering alternatives to his constraint that the class of questions must be “countably infinite”.[36] This is a rather technical requirement whose practical consequence is to require that there must be a way to write down all the questions in the class with finitely many symbols in a finite alphabet.

If the questions are infinite in numbers but not “countably infinite” we allow computation models where we use infinite precision arithmetic, that is all the real numbers are elaborated with the full expansion of their infinitely long decimals. This is akin to analog computers, except that the law of physics and the errors of the instruments are such that analog computations don't really have infinite precision.

If there is a finite number of questions then, according to Kleene,[37] the algorithm is reducible to a table where corresponding inputs and outputs are stored, assuming we have enough storage capacity. Then, the computation of the algorithm becomes a simple table lookup operation. Kleene himself admits that although such a table may be constructed in principle, in practice it may not be available. He himself brings algorithms for the game of chess as an example of this circumstance.

Other flavors arise when one removes the requirements of termination or determinacy. Werner Kluge explains why this is sometimes done.[38] (emphasis in the original)

> However, *termination* and *determinacy* of results may not necessarily be desirable algorithmic properties. On the one hand there are algorithms that (hopefully) never terminate but nevertheless do something useful. Well-known examples are the very basic cycle of issuing a prompter, reading a command line, and splitting off (and eventually synchronizing with) a shell process for its interpretation, as it is repeatedly executed by a UNIX shell, or, on a larger scale of several interacting algorithms, the operating system kernel as a whole. which must never terminate unless the system is shut down.
>
> On the other hand, there are *term rewrite* and *logic-based systems* where the transformation

> rules are integral part of the algorithm themselves. Given the freedom of specifying two of more alternative rules for some of the constructs, these algorithms may produce different problem solutions for the same input parameters, depending on the order of rule applications.

Sometimes one encounters the objection that computer implemented processes are not mathematical because real-life computations as performed in actual computers have non deterministic elements. Such objections have no foundations in mathematics. Nondeterministic algorithms and probabilistic algorithms are part of mathematics.

Still more flavors may arise when one considers the possibility of parallel computing and distributed computing where several agents cooperate in the execution of the algorithm.

The potential difficulty for the courts of keeping track of all these flavors and determining where are the boundaries of mathematics is part of why I propose the model of computation approach. This procedure eschews the need of defining exactly the extent of the notion of mathematical algorithm. A specific model of computation that is known from mathematical literature can be tested on the instant patented method with mathematical rigor. Besides this is how mathematicians themselves eschew the difficulties of defining the term algorithm. They use models of computation to circumscribe their studies to classes of algorithms that support rigorous mathematical definitions.

The Correlation with Pencil and Paper Calculations

The particular flavor of algorithm captured by Kleene's definition is an important one. It corresponds to the notion of a computation that could be performed, in principle, by a human working with pencil and paper using only the information that is explicit in the written symbols and the rules governing the computation. Use of of human insight or ingenuity beyond what is required to a strict application of the rule is forbidden. As Kleene puts it “In performing the steps we have only to follow the instructions mechanically, like robots; no insight or ingenuity or intervention is required of us.”

This particular flavor of algorithm is often called effective methods.[39] The goal is to capture the notion of providing an actual and exact answer to the problem by strict mathematical and logical means. This notion of mechanical application of the rules without using additional insight and ingenuity is a close relative to the notion of formal systems that has been described previously.

When we relax the definition to allow non termination, then it may happen that the algorithm solves the problem partially. It may solve the problem only for the inputs that lead to termination. However in the course of the execution of an infinite loop an algorithm may still produce useful results as Werner Kluge has pointed out. When we relax the definition to allow a non deterministic element such as a probabilistic choice the algorithm will produce a set of possible answers as opposed to a uniquely determined answer.[40] In both cases we have a variation of the same fundamental idea of producing a solution to a problem by mathematical means.

Please note the use of the phrase “in principle”. It means that the definition of algorithm ignores the real life limitations of the agent doing the computation. An algorithm doesn't stop being an algorithm because the human gives up and stops computing, dies or runs out of stationery before being done. The limitations of the algorithm must be inherent to the procedure and not to the agent carrying it out. However one should not conclude that mathematicians won't care about whether or not it is practical to compute the algorithm. They do care but this analysis is not part of the definition. It is done separately. This is called computational complexity theory. The algorithm is analyzed to determine how many steps and how much storage are required to produce a solution. Then a determination of whether the algorithm is feasible is made on this basis.

This notion of effective method is an informal one. As usual in mathematics, this informal notion has been formalized for purpose of achieving mathematical rigor and accuracy. This is done by means of a model of computation called Turing machines. The statement that Turing machines are equivalent to human computers carrying out pencil and paper calculations is known as the Church-Turing thesis. This thesis states, in effect, that the problems which are solvable, in principle, by pencil and paper calculations are the exact same problems as those which are solvable by Turing machines. There are many reasons why mathematicians believe this is the case.[41] The one they found most convincing is an analysis done By Alan Turing himself and published in the original paper where he first introduced the Turing machines. Here are two selected paragraph of this analysis[42]:

> Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think it will be agreed that the two-dimensional of paper is no essential of computation. I assume then then that the computation is carried out on one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrary small extent. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or 999999999999999 is normally treated as a single symbol. Similarly in European languages words are treated as single symbols (Chinese, however attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.
>
> The behaviour of the computer at any moment is determined by the symbol he is observing, and his “state of mind” at the moment. We may suppose that there is a bound *B* to the number of symbols or squares which the computer may observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of minds which must be taken into consideration is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be “arbitrarily closed” and will be confused. Again, the restriction is not one which seriously affect computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.

Here we see stated explicitly the relationship between Turing machines, pencil and paper calculations, and the state of mind of the human computer. Turing's argument is a form of reducibility. The activity of the human is transformed into a corresponding activity of the Turing machine in such manner that they both solve the same problems. The existence of such transformation is considered evidence that the Church-Turing thesis is correct.

This is typical of the methods used in computation theory.[43] Several theorems showing one or another model of computation is equivalent to a Turing machine, and by way of consequence of pencil and paper calculations, use such transformations to show both models solve the same set of problems. In mathematical speak we say they both compute the same class of computable functions. This is called Turing-reducibility or Turing-reduction.

The register machines and random access stored program models of computation have been shown equivalent to Turing machines in this manner. The mathematical analysis of these models is the topic of chapter 1 of [Aho 1974].[44] The consequence is that the stored program architecture of modern computers is mathematically equivalent to Turing machines and human computers carrying out pencil and paper calculations in the sense that they all solve the same set of problems.

Perhaps this may be useful in showing that digital computers always solve mathematical problems. If the problem is mathematical when solved by a Turing machine then it must be mathematical when solved by a digital computer.

Historical Connections Between Mathematical Logic and the Invention of Modern Computers

There is a correspondence between the notion of formal system in mathematical logic where all the information has been made explicit in the syntax and the notion of effective procedures which use only the information which is explicit in the computation. These two ideas fit together like the neighboring pieces of the same puzzle. This discovery predates the invention of computers and guided Alan Turing in his research. R. Gregory Taylor explains[45] (emphasis in the original):

> We have described Turing and Markov as intending *analyses* or *models* of some intuitive concept of computability. Talk of computability is sure to conjure up images of motherboards, disk drives, and the like, in the mind of the contemporary reader. In the interest of historical accuracy, however, it must be said that, especially in the case of Turing, the original purpose of the analysis had little to do with computing devices themselves. Rather, Turing was motivated by a desire to provide a secure foundation for mathematics—that is, some way of establishing, once and for all, that provable mathematical propositions are indubitable. (Philosophers and logicians would themselves more likely describe this project as an attempt to show that the theorems of mathematics are *logically necessary* or *analytic*.)
>
> Most likely, the reader will be puzzles by the suggestion that Turing machines could be used to justify mathematics. After all, how could anyone think that an analysis of mere computability—recall the computation involved in determining whether *n* is prime—would provide an epistemic foundation for the entire edifice of mathematics, including both number theory and analysis? Implausible as this may seem initially, the reader should remember that epistemic foundations are not concerned with the act of *mathematical discovery*, whose study properly belongs to cognitive science and psychology. Rather, epistemic foundations focus on the process of *verifying mathematical proofs*—verifying that what purports to be a proof truly is a proof. Logicians and philosophers of mathematics working in the 1930s had come to think of this verification process as being essentially computational—they themselves would have described it as *formal*—a matter of mere symbolic manipulation involving no consideration of the meaning of the symbols.

We have seen that story earlier in this article. The new point here is that historically the beginnings of computation theory have been a quest to provide a foundation to mathematics. But an important unintended discovery came out of these investigations. Turing and others discovered that computation can be automated. Howard Delong explains[46] (emphasis in the original):

> Both Post and Turing decided to approach the problem of the meaning of *effectively calculable* by first considering a paradigm case of computing, then, second, by ignoring

> inessential features of that case, and third, by analyzing the essential features into combination of very simple operations. The following account is in the spirit of Turing and Post with some details changed.
>
> We begin by imagining some human who is faced with a specific computational problem. It might be some such problem as computing the sum of 101 and 102 and 103 and … and 198 and 199, where the three dots indicate one occurrence each of all the natural numbers between 103 and 198. We assume that he is working according to a finite set of rules which have been fixed before the problem was given and that he is using pencil and paper. We also assume that after a finite amount of time he stops with the correct answer.
>
> If we examine this paradigm case of computing with a view toward eliminating inessential features, a number of such features come to mind, such as the use of pencil and paper or the particular computational problem chosen. However, the most striking one appears to be the *human*: the computer might as well be a machine.

This point has been noticed by Turing and others. They proceeded to the next logical steps which was to build such a machine. In particular the discovery of the universal Turing machine suggested the construction of a general purpose computer able to carry out every possible computation, within the limits of available resources, as opposed to the construction of a specific device for each computation. We may read some of this history on the Turing Archives for the History of Computing.[47] In England, Alan Turing himself contributed to the development of early computers. In the US, John von Neumann played a major role in the development of the computers. The Turing Archives explains:

> In 1944, John von Neumann joined the ENIAC group. He had become 'intrigued' (Goldstine's word) with Turing's universal machine while Turing was at Princeton University during 1936-1938. At the Moore School, von Neumann emphasised the importance of the stored-program concept for electronic computing, including the possibility of allowing the machine to modify its own program in useful ways while running (for example, in order to control loops and branching). Turing's paper of 1936 ('On Computable Numbers, with an Application to the Entscheidungsproblem') was required reading for members of von Neumann's post-war computer project at the Institute for Advanced Study, Princeton University (Julian Bigelow in personal communication with William Aspray, reported in the latter's *John von Neumann and the Origins of Modern Computing* Cambridge, Mass.: MIT Press (1990), pp. 178, 313).

If historical circumstances are of any use in showing that software is mathematics and mathematical speech, there you have them.

The Definition of "Algorithm" in Computer Science

Physicists and engineers don't redefine the mathematical concepts that serve as a foundation of their disciplines. Neither do computer scientists. Computation theory is one of the mathematical foundations of computer science and the notion of mathematical algorithm is part of it. Computer science uses the same notion of algorithm as mathematicians. This idea is straightforward and obvious to computer scientists but it happens that the courts have ruled otherwise. For example *Paine Webber v. Merrill Lynch* says (emphasis and links in the original):

> Although a computer program is recognized to be patentable, it must nevertheless meet the

> same requirements as other inventions in order to qualify for patent protection. For example, the Pythagorean theorem (a geometric theorem which states that the square of the length of the hypotenuse of a right triangle equals the sum of the squares of the lengths of the two sides—also expressed A2 + B2 = C2) [sic] is not patentable because it defines a mathematical formula. Likewise a computer program which does no more than apply the theorem to a set of numbers is not patentable. The Supreme Court and the CCPA has clearly stated that a mathematical algorithmic formula is merely an idea and not patentable unless there is a new application of the idea to a new and useful end. *See* *Gottschalk v. Benson*,[48] 409 U.S. 63, 93 S.Ct. 253, 34 L.Ed.2d 273 (1972); *In re Pardo* , 684 F.2d 912 (Cust. & Pat.App.1982) .
>
> Unfortunately, the term "algorithm" has been a source of confusion which stems from different uses of the term in the related, but distinct fields of mathematics and computer science. In mathematics, the word algorithm has attained the meaning of recursive computational procedure and appears in notational language, defining a computational course of events which is self contained, for example, A2 + B2 = C2. [sic] In contrast, the computer algorithm is a procedure consisting of operation to combine data, mathematical principles and equipment for the purpose of interpreting and/or acting upon a certain data input. In comparison to the mathematical algorithm, which is self-contained, the computer algorithm must be applied to the solution of a specific problem. *See* J. Goodman, *An Economic Analysis of the Policy Implications of Granting Patent Protection for Computer Programs* (scheduled for publication Vand. L.Rev. (Nov.1983)). Although one may devise a computer algorithm for the Pythagorean theorem, it is the step-by-step process which instructs the computer to solve the theorem which is the algorithm, rather than the theorem itself. Sometimes you see proposals that software be patentable as a process, but that does not provide escape from algorithms, which are processes but are not patentable.
>
> The confusion that has resulted by the dual definition of the term "algorithm" has been exemplified by the different findings by the PTO and CCPA. The PTO, in the past, has had the tendency to hold that a computer program, which is expressed in numerical expression, is not statutory subject matter and thus unpatentable because the computer program is inherently an algorithm. *See* *In Application of Toma,* 575 F.2d 872 (Cust. & Pat.App.1978) ; *In Application of Phillips,* 608 F.2d 879 (Cust. & Pat.App.1979) ; *In re Pardo* , 684 F.2d 912 (Cust. & Pat.App.1982) . The CCPA, however, has reversed the findings of the PTO and held that a computer algorithm, as opposed to a mathematical algorithm, is patentable subject matter.

This case is from 1983. This view has permeated patent law for decades.

In deciding that computer algorithms and mathematical algorithms are different, the *Paine Webber* court relied on a number of precedents and on the authority of J. Goodman, author of an article titled *An Economic Analysis of the Policy Implications of Granting Patent Protection for Computer Programs*. Here is the personal home page of J. Goodman. He appears to be a lawyer whose practice is primarily in the areas of family and commercial litigation. Note the table of contents of his article.

Whether or not computer science and mathematics use the same definition is a question of fact in the relevant disciplines. Either both disciplines use the same definition or they don't. This question should be answered by experts in these fields. Law can't helpfully redefine algorithms any more than Congress can usefully decide to pass a law redefining π or declaring that 1 + 1 = 3 by law now.

Donald Knuth doesn't need to be presented to computer professionals. He is a celebrity. He is both a theoretician and a practitioner of computer science. His achievements have given him his place among the mathematicians whose biographies are in the MacTutor History of Mathematics Archive .

In the above biography, his achievements in software development are described as follows:

> Starting in 1976 Knuth took ten years off his other projects to work on the development of TeX and METAFONT, a computer software system for alphabet design.
>
> TeX has changed the technology of mathematics and science publishing since it enables mathematicians and scientists to produce the highest quality of printing of mathematical articles yet this can be achieved simply using a home computer. However, it has not only changed the way that mathematical and scientific articles are published but also in the way that they are communicated. In the 17th century a mathematician would have written a letter to another mathematician and they would discuss their everyday lives in English, French or German, say, but whenever they came to explain a piece of mathematics they would use Latin. Now mathematicians communicate by e-mail and whenever they want to explain a piece of mathematics they require mathematical symbols which almost always they communicate using TeX.

His mathematical background is described as follows:

> It is a real achievement to publish a mathematics paper while still a doctoral student, but Knuth managed to publish two papers in the year he completed his undergraduate degree. These were *An imaginary number system* and *On methods of constructing sets of mutually orthogonal Latin squares using a computer I* the latter paper being written jointly with R C Bose and I M Chakravarti. In the first Knuth describes an imaginary number system using the imaginary number 2*i* as its base, giving methods for the addition, subtraction and multiplication of the numbers. In the second paper Knuth and his co-authors give two sets of five mutually orthogonal Latin squares of order 12.
>
> In the autumn of 1960 Knuth entered the California Institute of Technology and, in June 1963, he was awarded a Ph.D. in mathematics for his thesis *Finite semifields and projective planes.*

According to this biography his theoretical achievements include work on the semantics of programming languages; the Knuth-Bendix algorithm, attribution grammar ; the development of *LR* , (, *k*,) parsing , the Knuth-Morris-Pratt algorithm which searches for a string of characters; and structured documentation and literate programming.

He has earned many awards in his prestigious career. The same biography summarizes:

> For his quite remarkable contributions Knuth has received many honours - far too many to be mentioned in an article of this length. Let us just list a small selection. He was the first recipient Grace Murray Hopper Award from the Association for Computing Machinery in 1971; he was elected a Fellow of the American Academy of Arts and Science in 1973; in 1974, he won the Alan M Turing Award from the Association for Computing Machinery; he was elected to the National Academy of Sciences in 1975; in the same year he won the Lester R. Ford Award from the Mathematical Association of America; he was awarded the National Science Medal in 1979 (presented to him by President Carter); he was elected to the National Academy of Engineering in 1981; he was elected an honorary member of the

> IEEE in 1982 and awarded their Computer Pioneer Award in the same year; he was awarded the Steele Prize for Expository Writing from the American Mathematical Society in 1986; he was awarded the Franklin Medal in 1988; he was elected to the Académie des Sciences in 1992; he was awarded the Adelskold Medal from the Swedish Academy of Sciences in 1994; he was awarded the John von Neumann Medal from the IEEE in 1995; and the Kyoto Prize from the Inamori Foundation in 1996.
>
> Let us mention a few of the honours that Knuth has received since 2000. He has received honorary degrees from a large number of universities world-wide: Waterloo University, Canada (2000), Tübingen University (2001), the University of Oslo (2002), Antwerp University (2003), Harvard University (2003), the University of Macedonia (2003), Montreal University (2004), ETH Zurich (2005), Concordia University (2006), Wisconsin University (2006), the University of Bordeau (2007). In 2003 he was elected to the Royal Society of London, and in 2008 to the Russian Academy of Sciences . He was awarded the Gold Medal from the State Engineering University of Armenia in 2006, and in the same year the gold medal from Yerevan State University. In 2001 the minor planet "(21656) Knuth" was named after him.

His series of books *The Art of Computer Programming* has been used by computer professionals as a reference on algorithms for decades.

At this point there should be no doubt that Donald Knuth is qualified to know whether or not computer science and mathematics use the same definition of algorithms being himself an expert in both disciplines.

He sent a pair of letters to the US Commissioner of patents and Trademark and to the President of the European Patent Office where he said unambiguously that all computer algorithms are mathematical.

> I am told that the courts are trying to make a distinction between mathematical algorithms and nonmathematical algorithms. To a computer scientist, this makes no sense, because every algorithm is as mathematical as anything could be. An algorithm is an abstract concept unrelated to physical laws of the universe.
>
> Nor is it possible to distinguish between "numerical" and "nonnumerical" algorithms, as if numbers were somehow different from other kinds of precise information. All data are numbers, and all numbers are data. Mathematicians work much more with symbolic entities than with numbers.

There is more. Donald Knuth wrote a definition of 'algorithm' for use in computer science in *The Art of Computer Programming*, *Volume 1*[49].

We know that Knuth really means the same definition as the one from computation theory because he too states his definition is equivalent to Turing machines. On page 7-9 he further explains (emphasis in the original):

> So far our discussion of algorithms has been rather imprecise, and a mathematically oriented reader is justified in thinking that the proceeding commentary makes a very shaky foundation on which to erect any theory about algorithms. We therefore close this section with a brief description of one method by which the concept of algorithm can be firmly grounded in terms of mathematical set theory. Let us formally define a *computational method* to be a …

> [a lengthy mathematical discussion is omitted]
>
> Such a computational method is clearly “effective ,“ and experience shows that it is also powerful enough to do anything we can do by hand. There are many other essentially equivalent ways to formulate the concept of an effective computational method (for example, using Turing machines). The above formulation is virtually the same as that given by A. A. Markov in 1951, in his book *The Theory of Algorithms* (tr. from Russian by J. J. Schorr-Kon, U.S. Dept. of Commerce, Office of Technical Services, number OTS 60-51085).

The Markov algorithms are covered in some textbooks on the mathematical theory of computation. Â For example they are the topic of chapter 4 of [Taylor 1998]. The opening paragraph of this chapter is[50] (emphasis in the original):

> Historically, the first attempts at giving a precise account of the notion of algorithm took our function computation paradigm as basic. In contrast, the *string-rewriting systems* that were first described in the 1950s by Russian mathematician A. A. Markov (1903—1979) are an attempt to give an analysis of sequential computation in its fullest generality. In this sense, Markov took our transduction paradigm as his starting point and stressed symbol manipulation. We shall see how each of the three computational paradigms introduced in § 1.1 can be implemented using Markov's model. Moreover, that model will turn out to be formally equivalent to Turing's.

There can be no reasonable doubt that computer science uses the same definition of “algorithm” as mathematics. The legal distinction between these two uses of the term is factually incorrect.

Appendix B—The Relationship of Data and Functionality

When confronted with the argument that software is data, software patent proponents sometimes raise objections based on the concept of functionality. It appears that patent law makes a distinction between data which brings no functionality to the computer and software which instructs the computer of some functionality. This is a distinction without a difference because all data affect the computer functionality.

The reason is that all data may be tested during computation. For example a program may test a numeric value at some point of its execution and do one thing if it is greater than or equal to zero and another thing when it is less than zero.[51] Think of a banking system that does one thing when there are sufficient funds for a withdrawal and another thing when funds are lacking. In such case the numeric value has as much influence on the execution of the program as the instruction. When the numeric value is the result of some calculation, then the calculation has an indirect influence on the rest of the program execution. The programmer may use this phenomenon to choose whether he will control the execution either by providing explicit instructions or with data that will be tested. This gives him the power to write an algorithm that is either very specific to the intended functionality, or one that is very generic and specify the details of the functionality within the data to be processed. The ability of a computer to modify the data increases the programmer's power because this gives him the ability to alter the functionality as the program execution proceeds.

The functionality of the code is what the programmer decides to be in the code. If he doesn't like the legal consequences of putting functionality in the code, he can put it in the non executable data. And if he wants the legal consequences of using code, he can turn non executable data into executable code.

The choice is his.

The Programmer Has the Choice to Put Functionality in Data

It is usually considered that documents are data and are not patentable. Did you know that there are document formats in widespread use that are actually byte codes executed by a virtual machine? They are PostScript and PDF. Every time you load a PostScript or a PDF file in your reader you actually load byte code that must be executed in order to display the document.

You can read on PDF and PostScript directly from the Adobe site . Here is how they describe PostScript.

> So, we've established that PostScript is a language, like BASIC, Fortran, or C++. But unlike these other languages, PostScript is a programming language designed to do one thing: describe extremely accurately what a page looks like.

Kas Thomas wrote a good comparison on MacTech which includes this salient point:

> To the untrained eye, much of PDF may look like PostScript. But there are significant differences, the main one being that whereas PostScript is a true language, PDF is not: PDF lacks the procedures, variables, and control-flow constructs that would otherwise be needed to give it the syntactical power of a bonafide language. In that sense, PDF is really a page-description protocol.

These two examples demonstrate the wide range of possibilities a programmer has to his disposition when organizing data. He can develop a full powered programming language that is specific to his data like PostScript. Or he can use a dumb format like plain text. Or he can use some byte code whose functionality is somewhere in between these two extremes like PDF. Data may be anything in this range. If we define functionality as executable instructions then byte code is functionality because it is executable instructions even though it is not the native machine instructions of the CPU. Functionality can be displaced from the program that handles the data to the data itself to an extent that depends on the expressive power of the data format.

I must ask here. How would the safeguards in patent law that keep patents compatible with the First Amendment work in such case?

Code and Algorithms may be Machine Generated

Data is not only read. It is also written. This means that code and algorithms are not necessarily designed and/or written by human beings. They may be generated by other programs and executed on the fly. An example of such a technique is metaprogramming . A consequence is that programs can be written where even the authors of the programs will have a hard time knowing which algorithm is actually running without inspecting the live execution of the program with the help of debugging tools.

I must ask here. How does one ensure non-infringement when no one knows which instructions are actually executed? How does one perform a patent search on machine generated code? What happens when code is generated on-the-fly and discarded after use? How would the patent trade off between disclosure and exclusive rights work to benefit society in this context?

Functionality May Be Specified from Data that Is Modified as the Computation Progresses

The universal Turing machines and the RASP are two examples of universal algorithms. They are not the only ones. Robert Sedgewick and Kevin Wayne list several universal models of computability in their page on universality. The understanding that functionality equates to instructions is, assuming it may be defended at all, attuned to a specific model, the RASP, or stored program architecture. It is possible to pull the rug from under this understanding by changing the model of computation.

Let me give a specific example, but please keep in mind that this example is not the only one possible. The concept of beta-reduction in lambda-calculus allows to define several universal algorithms where functionality is not provided by a stable program comprised of instructions. Werner Kluge explains the underlying principle:[52] (emphasis in the original)

> This language must be complete in the sense that all intuitively computable problems can be specified by finite means. It must feature a *syntax* that defines a set of precise rules for the systematic construction of complex algorithms from simpler parts and a *semantics* that defines the meaning of algorithms, i.e. *what* exactly that are supposed to compute, and at least to some extent also *how* these computations need to be carried out conceptually.
>
> We will call this language *expression-oriented* since the algorithms are composed of expressions and are *expressions* themselves, and the objective of executing algorithms is to compute the *values* of expressions. These computations are realized using a fixed set of *transformation rules*. The systematic application of these rules is intended to transform expressions step by step into others until no more rules are applicable. The expressions thus obtained are the values we are looking for.

Lambda-calculus is a mathematical language operating under this principle. Beta-reduction is an example of such transformation rule. This form of computing is called a rewrite system. If you provide a strategy, I mean an algorithm, which specifies in which order to apply the rewrite rules you may get, as happens in the case of lambda-calculus, a universal algorithm. But this algorithm doesn't work by providing instructions to some computing agent. The specific computation arises from the initial text to be rewritten. When the algorithm applies the rewrite rules to this text until no more rules are applicable it reaches the solution of the problem. When we compile source code for a language based on such an algorithm we don't get machine executable code. We get the initial data for the universal algorithm.

The important point is that in this model there is no fixed executable code that stays in memory for the duration of the program outside of the universal algorithm. There is only data being constantly modified. How does this fit with the view that ultimately software is reduced to hardware? How does this fit with the view that functionality dictates the executable code? It doesn't fit. The specific functionality entirely lies in modifiable data.

Real-life implementations of lambda-calculus for purposes of implementing programming language exist. Languages such as Lisp and Haskell are based on it.

To be fair, not all implementations use rewrite rules. Some of these implementations use abstract machines such as the SECD machine. Kluge's book documents several of them.[53] But these abstract machines do not belong to the family of register machines like the RASP do. The instructions are stored in dynamic data structures that are constantly rewritten as the computation progresses. These machines literally and constantly reprogram themselves as they carry out the computation.

In such a scenario code is necessarily data. It is volatile data that must be modified for the computation to proceed.

References

[Aho 1974] Aho, Alfred V., Hopcroft, John E, and Ullman, Jeffrey D.. *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company 1974

[Benacerraf 1984] Benacerraf, Paul, Putnam, Hilary, *Philosophy of Mathematics: Selected Readings*, Cambridge University Press, 1984

[Ben-Ari 2001] Ben-Ari, Mordechai, *Mathematical Logic for Computer Science, Second Edition*, Springer-Verlag, 2001

[Bertot 2004] Bertot, Yves, Castéran, Pierre, *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*, Springer, 2004

[Davis 1965] Davis, Martin, *The Undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Raven Press Books, 1965, Corrected republication by Dover Publications 2004.

[Davis 2000] Davis, Martin, *Engines of Logic, Mathematicians and the Origin of the Computer*, W.W. Norton and Company, 2000. This book was originally published under the title *The Universal Computer: The Road from Leibnitz to Turing.*

Here is Martin Davis' Curriculum Vitae [PDF].

[Delong 1970] Delong, Howard. *A Profile of Mathematical Logic*. Addison-Wesley Publishing Company. 1970. I use the September 1971 second printing. Reprints of this book are available from Dover Publications.

[Devlin 2000] Devlin, Keith, *The Language of Mathematics, Making the Invisible Visible*, W.H. Freeman, Henry Holt and Company, 2000

[Gansner 2004] Gansner, Emden R., Reppy, John H., *The Standard ML Basis Library*, Cambridge University Press, 2004

[Greenlaw 1998] Greenlaw, Raymond, Hoover, H. James, *Fundamentals of the Theory of Computation, Principles and Practice*, Morgan Kaufmann Publishers, 1998.

[Hamacher 2002] Hamacher, V. Carl, Vranesic, Zvonko G., Zaky. Safwat G., *Computer organization, Fifth Edition*, McGraw-Hill Inc. 2002

[Hopcroft 1979] Hopcroft, John E . and Ullman, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation* , Addison-Wesley Publishing Company, Inc. 1979

[Kleene 1952] Kleene, Stephen Cole, Â *Introduction to Metamathematics*, D. Van Nostrand Company, 1952. I use the 2009 reprint by Ishi Press International 2009.

[Kleene 1967] Kleene, Stephen Cole, *Mathematical Logic*, John Wiley & Sons, Inc. New York, 1967. I use the 2002 reprint from Dover Publications.

Here is a biography of Stephen Kleene, from the MacTutor History of Mathematics Archive.

Programmers will enjoy knowing Kleene is the inventor of regular expressions and has contributed greatly to the theory of finite automata.

[Kluge 2005] Kluge, Werner, *Abstract Computing Machines, A Lambda Calculus Perspective*, Springer-Verlag Berlin Heidelberg 2005

[Knuth 1973] Knuth, Donald E. , *The Art of Computer Programming* , *Volume 1* , *Fundamental Algorithms* , Second Edition, Addison-Wesley Publishing Company, Inc. 1973

Here is a biography of Donald Knuth. from the MacTutor History of Mathematics Archive

[Milner 1991] Milner, Robin , Tofte, Mads , *Commentary on Standard ML* , The MIT Press, 1991.

[Milner 1997] Milner, Robin , Tofte, Mads , Harper, Robert , MacQueen, David , *The Definition of Standard ML (Revised)* , The MIT Press, 1997

[Minsky 1967] Minsky, Marvin L., *Computation, Finite and Infinite Machines*, Prentice-Hall, 1967

[Patterson 2009] Patterson, David A. , Hennessy, John L. , *Computer Organization and Design* , Fourth Edition, Morgan Kaufmann Publishers, 2009.

See also Wikipedia: David Patterson , John Henessy

[Pierce 2006] Pierce, Benjamin C., Ed., *Advanced Topics in Types and Programming Languages*, The MIT Press 2005.

[Reppy 2007] Reppy, John H. , Concurrent Programming in ML, Cambridge University Press, First published 1999, Digitally printed version (with corrections) 2007.

[Rogers 1987] Rogers, Hartley Jr, *Theory of Recursive Functions and Effective Computability*, The MIT Press, 1987

[Taylor 1998] Taylor, R. Gregory, *Models of Computation and Formal Languages*, Oxford University Press, 1998

[Turing 1936] Turing, Alan, *On Computable Number with and Application to the Entscheidungsproblem*, Proceeding of the London Mathematical Society, ser. 2, vol 42 (1936), pp. 230-67. Correction: vol 43 (1937) pp. 544-546.

This paper could be ordered from the publisher here [Link]

This paper is available on-line here [link]

Here is a biography of Alan Turing, from the MacTutor History of Mathematics Archive .

[Winkler 2004] Winkler, Peter, *Mathematical Puzzles, A Connoisseur's Collection*, AK Peters Ltd, 2004.

[Winkler 2007] Winkler, Peter, *Mathematical Mind-Benders*, AK Peters Ltd, 2007.

Footnotes

1 Groklaw user rebentisch gives us this hint: we should read this economic review of the patent system. This is a study of professor Fritz Machlup, professor of political economy, John Hopkins University, for the subcommittee of Patents, Trademarks and Copyrights as part of its study of the United States patent system pursuant to resolutions 55 and 236 of the 85th Congress. (1958) He also argues that (lack of) scarcity would be a strong economic argument in the eyes of economists. He says there is not logical reason in the science of economy to implement an incentive system such as patents when there is no scarcity of a commodity such as software.

2 See [Bertot 2004] p. XIII and p. XV. This book is the Coq user manual. It is available for purchase from bookstores.

3 See [Milner 1997] for the definition manual. This is a tough reading for those who don't know how the formulas are supposed to work. There is an accompanying commentary [Milner 1991] that explains the formulas in the definition but it is written to an earlier version of the language. According to the authors there isn't that much difference between the two versions so the commentary is applicable but

still, making the correlation between the two texts is a bit complicated because the reference numbers don't match.

4 See [Milner 1997] p. xi. This manual is the official definition of the language.

5 See [Gansner 2004] for the Basis library documentation and reference manual.

6 See [Reppy 2007] Appendix B for the mathematical definition of Concurrent ML.

7 There is an alternative that may work for those patents where the legal doctrines of insignificant data gathering steps and insignificant post solution activities could be applicable. The patented method and the IO may be segregated in a pair of separately compiled modules. Then the module file that contains the patented code has a mathematical semantics and the IO module is handled by the legal doctrines. I am not giving a legal advice. I am pointing to a technical possibility that may or may not work out legally. If you have a use for this possibility I recommend that you obtain the advice of a lawyer before proceeding.

8 I don't intend to develop it myself.

9 See [Minsky 1967] p. 2

10 See [Taylor 1998] pp. 293-294

11 See [Taylor 1998] pp. 294-295

12 See [Hopcroft 1979] p. 166

13 See [Hopcroft 1979] p. 167

14 See [Taylor 1998] p. 140

15 See [Patterson 2009] p. 21

16 See [Hamacher 2002] p. 43.

17 The formal definition of mathematical language together with the rules of the logic are found in textbook of a discipline aptly named “mathematical logic”. A number of references will be found in the “Further Readings” section of this article.

18 See [Devlin 2000] p. 306

19 For instance see Peter Winkler's books [Winkler 2004] and [Winkler 2007]. They contain may such problems.

20 See [Devlin 2000] pp. 222-223

21 See [Minsky 1967] p.219

22 See [Kleene 1967] pp. 199-200

23 See [Delong 1970] p. 92

24 See [Kleene 1952] pp. 59-60

25 See [Kleene 1952] p. 60

26 See [Kleene 1952] pp. 63-64

27 The is chapter 2 of [Greenlaw 1998]

28 See [Greenlaw 1998] p. 19

29 If you are interested in such topics you may investigate the following themes. There is logic

programming which uses models of computation based on algorithms arising from mathematical logic, more precisely model theory. Then there is type theory which studies the correspondences between computation and mathematical logic. The Curry-Howard correspondence and the Coq programming environment are applications of type theory. Other applications exist. For example [Pierce 2005] chapters 4 and 5 cover the concept of "proof carrying code" where low level language, as in assembly languages or virtual machines byte codes, could be constructed in such manner that they have a dual semantics: one semantics is the usual machine execution semantics and the other semantics is a mathematical proof that the code has some desirable property, such as it will never try to use memory that wasn't properly allocated. A possible application is that you could load code from untrusted source and automatically validate it by verifying that its proof semantics indeed proves that it is well behaved.

The point is that the notions of computation and mathematical logic are deeply connected and this connection is revealed when one studies the relevant parts of mathematics.

30 See [Minsky 1967] pp. 132-133

31 See [Kleene 1952] p. 72. Stephen Kleene explains how the symbols are assembled into terms and formulas in a sample formal system of mathematical logic dedicated to number theoretic functions: (*emphasis* in the original, **bold** from me)

> First we define 'term', which is **analogous to noun in grammar**. The terms of this system all represent natural numbers, fixed or variable. The definition is formulated with the aid of metamathematical variables "s" and "t", and the operation of juxtaposition as explained above. It has the form of an inductive definition, which enables us to proceed from known examples of terms to further ones.
>
> 1. 0 is a *term*. 2. A variable is a *term*. 3—5. If *s* and *t* are *terms*, then (*s*)+(*t*), (*s*)·(*t*) and (*s*)' are *terms*. 6. The only *terms* are those given by 1—5.
>
> Example 1. By 1 and 2, 0, *a*, *b* are terms. Then by 5, (0)' and (*c*)' are terms. Applying 5 again ((0)')' is a term; and applying 3, ((*c*)')+(*a*) is a term.
>
> We now give a definition of 'formula', **analogous to (declarative) sentence in grammar**.
>
> 1. If *s* and *t* are terms, then (*s*)=(*t*) is a *formula*. 2—5. If A and B are *formulas*, the (A) ⊃ (B), (A) & (B), (A) ∨ (B) and ¬(A) are *formulas*. 6—7. If x is a variable and A a formula, then ∀x(A) and ∃x(A) are formulas. 8. The only formulas are given by 1—7.

This explanation is technical, but the point I want to make is not technical. The reason I show this quote is that Kleene expressly indicates the correspondence between the linguistic features of his formal system and those of normal English. A term, which express a computation with function symbols such a +, plays the role of a noun or, perhaps more accurately, of a noun phrase. The complete formula plays the role of a declarative sentence. This gives us the connection between computation and speech in terms that are familiar to persons knowledgeable of English grammar.

32 See [Devlin 2000] p. 240

33 See [Kleene 1967] p. 223.

34 Here Kleene inserts this footnote: "In practice such procedures are often described incompletely, so that some inessential choices may be left to us. For example, if several numbers are to be multiplied together, it may be left to us in what order we multiply them."

35 See [Kleene 1967] p. 226.

36 See [Kleene 1967] pp. 226-227

37 See [Kleene 1967] pp. 226-227

38 See [Kluge 2005] p. 12

39 See [Kleene 1967] p. 231.

40 This is unless, of course, all possible nondeterministic or probabilistic choices eventually result in the same answer being produced.

41 You will find much more details on Jack Copeland's page on the Church Turing thesis. Jack Copeland is the maintainer of the Alan Turing archives. Another version of the same page is found in the Stanford Encyclopedia of Philosophy.

42 See [Turing 1936] section 9. It is also available from [Davis 1965] anthology pp. 135-140. The most relevant part has been quoted in extenso in [Minsky 1967] pp. 108-111.

43 It is usual that arguments of this kind compare two mathematically defined models of computation and the reduction is proven by mathematical means. Turing's argument is unique in computation theory in that the notion of effective method is informal. This notion is not amenable to mathematical proofs until it is formalized and the purpose of the argument is to justify the formalization. This is why Turing's argument is informal and written in plain English.

44 The mathematical analysis of these models is the topic of chapter 1 of [Aho 1974]. This book more specifically focuses on the variant of register machines called random access machines (RAM). It also discusses the random access stored program (RASP). The equivalence of RAM and RASP with Turing machines is on pages 31-33.

45 See [Taylor 1998] p. 285

46 See [Delong 1970] p. 197

47 Another account of this history is in chapters Seven and Eight of [Davis 2000]

48 Here the court inserts this footnote:

> In *Benson,* the patent claimed a method of programming a general purpose digital computer to convert signals from binary-coded decimal form into pure binary form. The procedure set forth in the claims provided a generalized formulation to solve the mathematical problem of converting one form of numerical representations to another. The Court stated that the mathematical formula was an algorithm and that an algorithm is merely an idea, if there is no application of the idea to a new and useful end. Thus *Benson* established the principle that an algorithm, in the mathematical sense of the word, cannot without a specific application to a new and useful end be patentable.

49 See [Knuth 1973] pp. 4-6.

50 See [Taylor 1998] p. 245.

51 In hardware terms, this is called a conditional JUMP instruction. This latter reference is from the Art of Assembly Programming.

52 See [Kluge 2005] p. 37.

53 This book is [Kluge 2005]. This is the main topic of the book.

149 comments

---

http://www.groklaw.net/article.php?story=20110426051819346