**A Simpler Explanation of Why Software is Mathematics**

*Exposing Errors of Fact About Software in Patent Case Law*

By PolR

This article is a follow up on my previous article 1+1 (pat. pending) — Mathematics, Software and Free Speech. My original intent was to write a shorter and simpler explanation of the software is mathematics argument which would be accessible to laymen. This proved to be a too ambitious goal. This is about mathematics, software, computers and patents. This topic is inherently technical. I settled on the next best thing. I tried to assume as little knowledge of mathematics, computers and software as possible and still explain things properly. There are professions like journalists in the trade press and lawyers practicing in software related fields of the law whose members know more about computing than the ordinary folks but don't have the programming skills of Linus Torvalds. These people are the target audience.

The argument is based on the explanation of how software, hardware and data combine to implement functionality. When we have a complete picture of how functionality happens we can see clearly the parts played by the mathematical concepts of computation and algorithm. Once we have this knowledge it becomes clear in which sense and why software is mathematics.

But when we look at software from this angle of the making of functionality we find that there are two competing explanations; one which is factually correct because it complies with the principles of computer science and mathematics, and one which is provided by case law about software patents. These two explanations are different and contradictory. Some principles of mathematics are not as case law says they are and computers don't work like case law says they do. These are errors of fact in the explanation of how software functionality is implemented. They are not errors of law although we may doubt that the courts would have reached the same legal conclusions if the correct facts were considered.

The software is mathematics argument is as much about exposing the errors of fact in case law as it is about proving software is mathematics. When we compare the two explanations of functionality it can be seen where are the errors (several of them) and why they are errors. The evidence that these errors are indeed errors is overwhelming. This evidence is also abundant and highly technical. It accounts for most of the length and obscurity of the argument.

This article is divided in two parts. Part I does not elaborate on the evidence. It concentrates on presenting the argument as concisely, simply and clearly as possible in order to give the reader the big picture. Part II is a much more detailed explanation of the argument complete with evidence.

## Part I – A Summary of Why Software Is Mathematics

### A Summary of the Factually Correct Explanation of Software Functionality

This is how software, hardware and data combine to implement functionality.

- Symbols are abstract entities which are represented physically, like letters in an alphabet. They may convey meaning. The relationship between the physical representation of symbols and their meanings is an intangible relationship of semantic.
- Mathematics is a language. This language has a syntax of written symbols. It has a semantics in the abstract world of mathematical ideas. Mathematical language is also used to describe the real world, which means the mathematical statements may have a dual semantics both in abstract mathematics and in concrete reality.
- Mathematical language may be used to solve problems by means of manipulations of symbols called computations. The procedures for carrying out the computations are called algorithms. Algorithms are used to reason about problems and deduce new truths from known truths by applying principles of mathematics and logic.
- When mathematical language is used to describe a real world problem an algorithm solving the underlying mathematical problem will solve the corresponding real world problem by

producing as an answer the symbols whose meanings are the solution. This involves the intangible relation of semantics between the language and its meaning.

- The requirements of mathematics for a procedure to be an algorithm are such that these procedures are amenable to automation. When the symbols are represented by electronic means, for example bits, an electronic circuit for manipulating the symbols according to the rules of the algorithms can be constructed.
- Mathematicians have discovered that some algorithms have the capability to mimic the behavior of other algorithms when they are given appropriately crafted data as input. This gives the performer of the algorithm a choice. He may execute the algorithm directly, or he may execute the algorithm indirectly by mimicking its behavior with some other algorithm.
- Mathematicians have discovered that some of the mimicking algorithms are universal. They can reproduce the behavior of every possible algorithm when given suitably crafted data as input.
- When a universal algorithm is implemented in electronic circuitry the resulting device is a general purpose computer. The specially crafted data used by the universal algorithm is the computer program.
- Most modern digital computers are built according to the principles of the stored program computer architecture. Computers built in this manner execute a universal algorithm called the instruction cycle. We know this algorithm is mathematical because it meets the requirements of mathematics for a procedure to be an algorithm and it is possible to exhibit the mathematical formulas.
- The data given to this algorithm is called instructions. The instruction cycle executes the instructions one after another, reproducing the behavior of the algorithm for which it has been programed.
- A computer programmer must carry out at least three tasks in order to program a computer.
- The first task is to define a symbolic representation of his data, including both the syntax and the semantics of the symbols. This is a prerequisite to programming the computer because algorithms are procedures to manipulate symbols. They can't solve problems unless the data is represented with symbols.
- The other task is to define an algorithm which will carry out a computation which will produce symbols representing the answer to the problem.
- The third task is to implement the algorithm. Here the implementer has choices.
    1. He may build a special purpose circuit for the algorithm. This is not programming a general purpose computer but I mention this option because it is available.
    2. He may generate instructions for the instruction cycle of some general purpose computer.
    3. He may implement some other algorithm which will mimic the behavior of his target algorithm according to the mathematical principles described above. The implementation of the mimicking algorithm is itself subject to the same three choices from this list.
- The combination of the data representation in symbols, the definition of the algorithm and its implementation on an actual computer produces the functionality of software.

This explanation makes apparent why software is mathematics. The digital computer is an electronic machine to manipulate symbols. The computation is a mathematical computation pursuant to an algorithm in the sense mathematicians give to this term. We can be sure of that because ultimately all software will be executed by the universal algorithm which is the computer instruction cycle and we know this algorithm is mathematical. It meets the requirements of mathematics for a procedure to be an algorithm and it is possible to exhibit the corresponding mathematical formulas.

The computation may have some significance in the real world when the symbols have real world meanings on top of their mathematical meanings. In such case the software is a use of the mathematical language to describe the real world because algorithms and computations are features of the mathematical language.

We also have the answer to a frequently asked question. What is the difference between hardware

and software? Software is a description of a computation, a procedure to manipulate symbols, which is executed by a universal algorithm. Hardware is the physical substrate representing the symbols as well as the agent carrying out the computation. This is akin to the difference between a physical book and the printed text or the difference between the pocket calculator and the calculations. This answer accounts for the difference between software and analog computers because an analog computation does not use symbols. It accounts for the difference between a general purpose computer and a special purpose circuit because the general purpose computer uses a universal algorithm and the special purpose circuit does not.

This was a short and simple explanation of why software is mathematics.

**A Summary of the Legal Explanation of Software Functionality**

In contrast, here is the legal explanation of how software, hardware and data combine to implement functionality. This explanation is fundamental in patent law because it is used to justify why software is patentable subject matter. It is also used in legal analysis for determining whether a patent claim is valid and whether an implementation infringes on a claim.

This explanation takes the form of a series of extracts from case law. The first series of extracts is concerned with the legal view of “algorithm”.

From *Gottschalk v. Benson*

> A procedure for solving a given type of mathematical problem is known as an "algorithm."

From *in re Freeman* (emphasis in the original)

> As a bare minimum, application of *Benson* in a particular case requires a careful analysis of the claims, to determine whether, as in *Benson,* they recite a "procedure for solving a given type of *mathematical* problem."

The same case also refers to a dictionary definition: (emphasis in the original)

> The broader definition of algorithm is "a step-by-step procedure for solving a problem or accomplishing some end." *Webster's New Collegiate Dictionary* (1976)

This leads the *Freeman* court to observe: (emphasis in the original)

> Because every process may be characterized as "a step-by-step procedure * * * for accomplishing some end," a refusal to recognize that *Benson* was concerned only with *mathematical* algorithms leads to the absurd view that the Court was reading the word "process" out of the statute.

From *in re Toma* (emphasis and links in the original)

> In applying the *Freeman* rationale to the case before us, we begin by rejecting the board's definition of algorithm recited in note 4, supra. While we agree with the board that the form in which an "algorithm" is recited, whether algebraic or prose, is of no moment, it is clear to us that the *Benson* Court used the term "algorithm" in a specific sense, namely "a procedure for solving a given type of *mathematical* problem." 409 U.S. at 65, 93 S.Ct. at 254, 175 USPQ at 674 (emphasis added). Using this definition, we have carefully examined the claims in this case and are unable to find any direct or indirect recitation of a procedure for solving a *mathematical* problem. Translating between natural languages is not a mathematical problem as we understand the term to have been used in *Benson.* Nor are any of the recited steps in the claims mere procedures for solving mathematical problems.

This is the contents of note 4 referenced in *Toma*:

> The board took the following definition from C. Sippl and C. Sippl, Computer Dictionary and Handbook 23 (2d ed. 1972):
>
> algorithm—1. A fixed step-by-step procedure for accomplishing a given result; usually a simplified procedure for solving a complex problem, also a full statement of a finite number of steps. 2. A defined process or set of rules that leads and assures development of a desired output from a given input. A sequence of formulas and/or algebraic/logical steps to calculate or determine a given task; processing rules.

From *in re Bradley* (emphasis in the original)

> The board said that the claims do not *directly* recite a mathematical formula, algorithm, or method of calculation, but, nevertheless, held the claims to be mathematical in nature. As appears from the quoted portion of the board opinion, the board regarded the fact that digital computers operate in some number radix as conclusive on the issue of whether the appealed claims recite a mathematical algorithm in the *Benson* and *Flook* sense. The board did not, however, direct attention to any specific formula it thought is utilized, or to what, if anything, the mathematical calculations alleged to be present in the claims are directed. We do not agree with the board.
>
> We are constrained to reject its reasoning. Such reasoning leads to the conclusion that any computer-related invention must be regarded as mathematical in nature, a conclusion which is not compelled by either *Benson* or *Flook.*
>
> The board's analysis confuses *what* the computer does with *how* it is done. It is of course true that a modern digital computer manipulates data, usually in binary form, by performing mathematical operations, such as addition, subtraction, multiplication, division, or bit shifting, on the data. But this is only *how* the computer does what it does. Of importance is the significance of the data and their manipulation in the real world, i. e., *what* the computer is doing. It may represent the solution of the Pythagorean theorem, or a complex vector equation describing the behavior of a rocket in flight, in which case the computer is performing a mathematical algorithm and solving an equation. This is what was involved in *Benson* and *Flook.* On the other hand, it may be that the data and the manipulations performed thereon by the computer, when viewed on the human level, represent the contents of a page of the Milwaukee telephone directory, or the text of a court opinion retrieved by a computerized law service. Such information is utterly devoid of mathematical significance. Thus, the board's analysis does nothing but provide a quick and automatic negative answer to the § 101 question simply because a computer program is involved.

From *Paine Webber v. Merrill Lynch* (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) 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*, 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. 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.

From *in re Warderman*: (links in the original)

> 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.

Now let's see how the courts understand the relationships between computer program, algorithms and hardware.

From *in re Prater*

> 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.

From *in re Bernhart*

> [I]f a machine is programmed in a certain new and unobvious way, it is physically different from the machine without that program; its memory elements are differently arranged.

From *in re Alappat*

> [S]uch programming creates a new machine, because a general purpose computer in effect becomes a special purpose computer once it is programmed to perform particular functions pursuant to instructions from program software.

From *WMS Gaming, Inc. v. International Game Technology*

> The instructions of the software program that carry out the algorithm electrically change the general purpose computer by creating electrical paths within the device. These electrical paths create a special purpose machine for carrying out the particular algorithm.

At this point *WMS Gaming* refers to this footnote: (emphasis in the original)

> A microprocessor contains a myriad of interconnected transistors that operate as electronic switches. *See* Neil Randall, *Dissecting the Heart of Your Computer,* PC Magazine, June 9, 1998, at 254-55. The instructions of the software program cause the switches to either

> open or close. *See id.* The opening and closing of the interconnected switches creates electrical paths in the microprocessor that cause it to perform the desired function of the instructions that carry out the algorithm. *See id.*

As you may see, the legal story and the factually correct story are very different. The software is mathematics argument does not merely ask for a different interpretation of the law. It shows that errors of fact about computers and software are elevated by the courts to the status of case law.

Failure to notice this point will lead to miscommunication. For example when we say software is mathematics we mean the procedures for manipulating the symbols, the computations, always meet the requirements of mathematics for a procedure to be an algorithm. But people who adhere to the legal explanation often understand “software is mathematics” as saying that the physical process implemented on the computer is mathematics. Then they object: when a physical process is described mathematically it doesn't mean that this process is mathematics. This is answering a point which is not the one being made with an argument which presumes that the legal understanding of software as a physical process is correct.

**A Summary of the Errors of Fact in the Legal Explanation of Software Functionality**

The implementation of software functionality requires the conjunction of three components. There must be a definition of the data representation, complete with syntax and semantics. There must be a definition of the algorithm and there must be an actual implementation on a computer. In the legal story the data representation is completely absent while the other two components are presented in factually erroneous terms. Let's review the errors I managed to identify starting with data representation.

The role of symbols is essential. They are absent from the legal explanation which reduces all software functionality to a physical process executed by the computer. This reduction misses the the abstract nature of symbols and the role of semantics. It is oblivious of a very fundamental difference between software and hardware because symbols and their semantics are central to the functionality of software and they are not physical phenomenons.

In printed text the letters are abstractions separate from marks of ink on paper. The meaning of the text is not a physical property of ink and paper. The same observations apply to bits. They are symbols. They are abstractions separate from the electronic signals which represent them because a wide variety of signals may be used to represent the same bits. Their meanings is not a physical property of digital electronics. The meanings of symbols are not part of a physical process for manipulating electronic signals because meaning is not a physical phenomenon. Assigning meanings to symbols is an operation of semantics.

Computer programming requires that the programmer represents the real world with bits in such manner that algorithms could be used. When this is done properly the solution to the problem is read in the bits which result from the calculations. This is called defining the data. Symbols, the bits, are arranged according to some syntactic rules chosen by the programmer and their semantics is defined. This aspect of the computer program is never translated into machine instructions because no machine instruction exist for giving meanings to bits. The instructions only indicate the operations to be performed on the symbols, the algorithm.

Mathematics is a language. When a real world problem is stated in mathematical terms, there is an underlying mathematical problem. When we solve this mathematical problem we obtain a statement in mathematical language of the answer to the real world problem. The procedures to solve the mathematical and real world problems are identical. They are both the same algorithm in the mathematician's sense of the word. If a machine is built to compute the algorithm then the same machine will solve both problems. The difference between pure mathematics and applied mathematics is in the semantics given to the mathematical language and not in the procedures or machines used to solve problems. However some people who adhere to the legal explanation of software contend that applied mathematics is a process different from a pure mathematical calculation and is patentable. This view amounts to say that there is a difference in the process of

computation when the difference is in the meanings of symbols, everything else being the same.

Once we have proof that the manipulation of symbols pursuant to computer programs are mathematical algorithms it follows that the semantics of software is semantics of mathematical language. This is because an algorithm is a procedure for manipulating symbols in mathematical language.

This relationship between mathematics and the real world is not sufficiently taken into consideration when the courts apply *Benson*. The law defines the metes and bounds of the process of software by the words written in the patent. We can see this from the above quotes in *Freeman*

> As a bare minimum, application of *Benson* in a particular case requires a careful analysis of the claims, to determine whether, as in *Benson,* they recite a "procedure for solving a given type of *mathematical* problem."

If the problem described in the patent is not a mathematical problem according to the court's understand of mathematics, often the courts will not care whether or not the physical process actually implemented is the act of carrying out a mathematical computation. They will say that according to the words of the claims a mathematical algorithm is not recited. For example see the reasonings from *Toma* and *Bradley*.

This kind of legal analysis ignores the difference between the expressive power of legal language used in patents and machine languages used in computers.[1] Legal language is able to add a limitation on the real world meaning of data to the description of a process. Machine language can't do this. Even when one incorrectly accepts the description of a computer algorithm found in *Paine Webber* and its effect on machine structure found in *WMS Gaming* the fact remains that no machine instruction exists for giving real world meaning to bits. This result is achieved by an operation of semantics. Meaning is not part of the machine structure and it is not part of the process by which the machine operates.

In the legal view of software the notion that in mathematics algorithms are procedures for manipulating symbols is absent. The court in *Bradley* incorrectly thought that they are procedures for manipulating numbers, excluding ordinary text. They erroneously decided that textual contents is “utterly devoid of mathematical significance”. This is contradicted by the existence of a branch of pure mathematics dedicated to the mathematical properties of symbols in an alphabet and their arrangement in text and languages. Computer programs using what is known as “regular expressions” routinely process text according to mathematical algorithms.

The courts don't appear to use often enough expert mathematicians to resolve factual issues of what is within the scope of mathematics. I have seen no reference to such experts in the cases I have read so far. It doesn't help that the parties don't seem to correctly explain the scope of mathematics in their briefs, at least not in the cases I have read so far. The result is that when given the description of a problem in real world terms the courts don't always see the underlying mathematical problem and they don't always see when the computer program does nothing more than solving the underlying mathematical problem with a real world semantics attached to it.

The understanding of mathematics is inconsistent from judge to judge leading to a diversity of opinions on what is a mathematical algorithm and what is a mathematical problem. These difficulties have led the Federal Circuit to give up trying to understand what an algorithm is. See *Warmerdam*. Now they try to sidestep this issue by determining whether a claim is drawn to an abstract idea under the assumption that mathematical algorithms fall under the categories of unpatentable abstract ideas. While it is correct that algorithms are abstract ideas, this view is incomplete because the role of symbols and their meanings is still absent from case law. The semantical relationship between mathematics as a language and the real world cannot be understood by looking only at the abstract character of algorithms. First Amendment issues cannot be completely understood without a complete understanding of how mathematical language relate to software.

Because of the gap between the courts' understanding and the reality of mathematics there is a difference between the legal understanding of the term “algorithm” which result from the application of *Benson* and *Freeman* and the actual meaning of this term in mathematics and computer science. There is a branch of mathematics called computation theory which, as the name indicates, is concerned with the mathematical studies of computations. This is one of the mathematical foundations of computer science, much like calculus is a foundation of most branches of engineering. The term “algorithm” has a specific meaning in computation theory and it is this meaning which is used in computer science. The court in *Paine Webber* is unclear on these concepts. This case incorrectly conflates the mathematical meaning of the term with the legal meaning and incorrectly concludes the computer science meaning is different from the mathematical meaning.

For purposes of explaining how functionality is implemented in software one must use the term “algorithm” as it is used in computer science. The legal conceptions are unsuitable for this purpose because the implementation of functionality is a factual issue. Computers work according to the principles of computer science, including its mathematical foundations.

The definition of “algorithm” from computer science and mathematics is limited to procedures for manipulating symbols. Using this definition does not read the word “process” out of the statute because processes which don't manipulate symbols are not algorithms. The concern expressed by the court in *Freeman* does not apply to this definition. Besides the definition from mathematics and computer science is *mathematical* because it is within the boundaries of mathematics.

We know that all computer programs are mathematical algorithms because they all meet the requirements of mathematics for a procedure to be an algorithm. This same conclusion also follows from the fact that a modern computer is built according to the stored program architecture. All such computers have a built-in algorithm which is the instruction cycle. It has been verified that the instruction cycle meets the requirements of mathematics for being an algorithm and the corresponding mathematical formulas may be exhibited. All computer programs are ultimately executed by the instruction cycle.

The instruction cycle is a universal algorithm. This means it has the ability to mimic the behavior of every other algorithm. It is a fact of mathematics that such universal algorithms exist. When we implement a universal algorithm in hardware we get a general purpose computer. This is exactly what a stored program computer is, a hardware implementation of a universal algorithm. When we define the process of software as the step-by-step execution of instructions like *Paine Webber* and *WMS Gaming* do, then this process can only be a mathematical computation carried out pursuant to the instruction cycle because this is how the computer has been built to operate.

The requirements of mathematics for an algorithm to be an algorithm, the concept of universal algorithm and the instruction cycle are absent from the legal explanation of how functionality is implemented in software. See *Bradley*, *Prater* and *WMS Gaming*. In particular this series of cases errs on how algorithms are implemented on a computer. The correct explanation is that the implementer has three options.

1. He may build a special purpose circuit for the algorithm. This is not programming a general purpose computer but I mention this option because it is available.
2. He may generate instructions for the instruction cycle of some general purpose computer.
3. He may implement some other algorithm which will mimic the behavior of the target algorithm according to the mathematical principles described above. The implementation of the mimicking algorithm is itself subject to the same three choices from this list.

The courts believe implementers always make special purpose circuits and that loading machine instructions in memory is how this specific circuit is made. See *Prater*, *Bernhart*, *Alappat* and *WMS Gaming*. This is ignoring the existence of the instruction cycle mentioned in option 2. This is also entirely ignoring option 3. When this option 3 is used the instructions stored in the computer do not correspond to the words of the patent but the functionality is still implemented.

The instruction cycle defines the process by which the moving parts of the computer perform their functions. The courts erroneously conflate the changes inherent to the activity of the moving parts with structural changes which define a new machine. In particular the pathways between transistors mentioned in *WMS Gaming* are transient.[2] They last only the time required to execute a single instruction and are torn down when comes the time to execute the next instruction. This is not making the structure of a machine for executing an entire algorithm. This is carrying out the computation pursuant to the instruction cycle. Besides these pathways are not established when the instructions are loaded in memory. They are established when the instructions are executed. *WMS Gaming* is erroneously conflating changes which result from the execution of the program with changes which result from programming a computer. This is inconsistent with the *Alappat* notion that programming the computer makes a special purpose machine because executing the program is not the same thing as programming the computer.

The legal explanation of how functionality is implemented assumes implicitly that there is a difference between instructions and non instruction data. This distinction cannot be justified by the physical effect of instructions on hardware. There is no difference between the physical changes to a computer which result from storing instructions in memory and storing non instruction data in memory. In both cases bits are stored in memory in the exact same manner with the exact same technology. Also both instructions and non instructions data result in pathways between transistors to be setup and torn down in the processor when the computation is carried out because the computation depends on the data as much as it depends on the instructions.[3] The difference is that the instructions are a special kind of data recognized by the instruction cycle. However software implementation of instruction cycles exist. Therefore the difference between instructions and non instruction data must be defined in algorithmic terms and not in hardware terms.

Sometimes people who adhere to the legal explanation of software define the distinction between instructions and non instructions data in terms of functionality instead of physical effect on the computer. This distinction is not justified by the facts because there is no difference between instructions and non instruction data in terms of ability to represent software functionality. There are programming techniques which implement software functionality by means of non instruction data and there are programming techniques which represent non functional data by means of instructions.

This completes our guided tour of the errors of fact in the legal explanation of how software functionality is implemented that I have been able to locate. This was a lengthy tour because there are many of them. Remember that this explanation is used to justify why software is patentable in the first place. It is also used in the analysis of whether a given patent claim is valid and in the analysis of whether a given implementation infringes on a claim.

We may wrap up the errors of fact as follows:

- The role of symbols and their meanings is absent.
- The distinction between software and hardware is not made because we find nowhere the notion that software describe a computation, that is a procedure for manipulating symbols, as opposed to hardware which is the agent performing the computation.
- The following principles established in the discipline of mathematics are absent: the definition of algorithm, the notion of universal algorithm and the body of mathematics applicable to text made of symbols.
- When the courts hold that programming a computer makes a special purpose machine, they do so on the basis of a factually incorrect understanding of computers where the instruction cycle and the notion of universal algorithm are absent and where applying voltage to an existing pathway is assimilated to the making of a new electronic path.
- When a software is interpreted as a process, the courts are often conflating three processes which are actually different: 1) the words of the patent, and, 2) the machine instructions as they are loaded in computer memory, and, 3) the activity of transistors executing the instruction cycle.
- The notion that a computer program may be meant to be executed by a universal algorithm implemented in software is absent because it is presumed that all software will

be hardware instructions.
- The legal distinction between instructions and non-instruction data is factually erroneous because it cannot be made on the basis of the effects of the bits on the hardware and it cannot be made on the basis of functionality. The correct distinction must be made in algorithmic terms and this cannot be done because the notion of instruction cycle is absent.
- A mathematical calculation with a real world semantics is treated as a process and not as a semantics given to the mathematical language. The computer doing the calculations is treated as a different machine when the calculations have a real world semantics.

Please think of this XIX$^{th}$ century proposal for a law setting the value of pi to 3. The Indiana legislature refused to vote this bill for obvious reasons. But imagine what would have happened if they adopted it. This is the kind of factually fallacious law that is imposed on computer programmers, the software industry and anyone who uses a computer.

**A Summary of the Evidence for the Software Is Mathematics Argument**

We have an abundance of evidence that the software is mathematics argument is factually correct and that the legal view of software functionality is factually wrong. Here is a summary of the sources of evidence.

- **Textbooks of mathematics and computer science**. Specialists in both fields have documented what is a mathematical algorithm and how computer works. The story found in the writings of experts doesn't match the one found in case law. This article will provide some references.
- **Mathematical formulas**. The formulas for specific computer programs as well as formulas for the instruction cycle of computers can be exhibited. This article and an accompanying exhibit explain how. These formulas describe a mathematical procedure for manipulating symbols, an algorithm. They don't describe a physical phenomenon such as the activity of a circuit. The given formulas are describing the manipulation of symbols performed by a comprehensive range of hardware features such as DMA, interrupts, networking, multiprocessing, etc which are found in real world implementations.
- **Programing techniques**. There are programming techniques which implement functionality without generating machine instructions corresponding to the words of a patent claim. Even if we expand the legal concept of machine structure to arbitrary data there are programming techniques which will implement the functionality without producing structure corresponding to the words of the claim according to this expanded definition. Conversely there are programming techniques which represent non functional data with machine instructions. This article will explain some of these techniques and provide some references.
- **Experiments and demonstrations**. This article presents a series of experiments which are designed to produce one result if the legal explanation is correct and another result if the principles of mathematics computer science are correct. Needless to say the result of such experiments show that the legal explanation is wrong. These experiments may be used to prove who is right to laymen without having to teach them more computer science and mathematics than they can learn. Laymen may not understand the technical details but they are able to see which result occurs when the experiment is run. These experiments are such that laymen should be able to draw the correct conclusions from information they can understand.
- **Nonsensical patent claims**. There are a few patent claims which on their face are perfectly valid when we apply the principles of case law but common sense shows they attribute to a computing machine some elements of functionality which cannot reasonably be the result of the execution of machine instructions. This article presents a few claims of this nature.

This completes the first part of this article. This was, I hope, a short enough and intelligible enough presentation of the software is mathematics argument. It would have been much shorter if it were not for the need to explain why case law is factually erroneous. The explanation of why software is

mathematics can be told simply. The difficulty is to convince the courts to accept the factually correct understanding of software instead of flatly rejecting it as contrary to case law. Someone will have to tell them that case law is built on factually fallacious foundations and this requires very convincing explanations and strong evidence. Navigating the fallacies and showing why they are erroneous requires a discussion comparing the law, technology and mathematics. This kind of topic is inherently complex and technical.

**Part II – Detailed Explanation and Presentation of the Evidence**

I now go over the argument in more detail including the specifics of the evidence which I have not yet provided. This part is much longer and more technical. I have tried to make it as easy as possible for the layman; but, this is about computers, software and patents. There is only so much which can be done to make this highly technical topic accessible to non experts. When laymen are overwhelmed they should defer to experts.

Much of the evidence has already been presented in my previous article mentioned in the opening paragraph: 1+1 (pat. pending) — Mathematics, Software and Free Speech. When this evidence is relevant to what is said here I usually give a reference to this previous article instead of repeating the material. You need both articles for the complete story.

**The Significance of the Mathematical Notion of Algorithm in Computer Science**

The notion of algorithm is closely related to the notion of computation. We all have learned algorithms in school when we were taught the basic procedures for carrying out arithmetical calculations. But algorithms are not limited to arithmetic. All branches of mathematics may have algorithms for solving problems whether or not the problem involves numbers. R. Gregory Taylor explains:[4] (emphasis in the original, references to exercises omitted)

> For the contemporary reader, the term *computation* most likely conjures up images of computing *machines*, namely, computers and related devices such as calculators. But of course the concept of computation predates modern digital processing. Recall that, in earlier times, calculation of the positions of heavenly bodies, of navigational directions, and of actuarial statistics was performed without the aid of electronic calculating devices. Up until the 1930s, it was human beings alone who computed. As such, the concept of computation belongs to virtually all human cultures past and present. Although computation is no longer the exclusive domain of human beings, it remains an essentially human activity to this day.
>
> Talk of computation suggests numbers, of course. Clear examples of *numerical computations* are binary arithmetic operations such as addition and subtraction as well as unary operations such as finding the square root of a number–without looking it up in some table of course. However, computation need not involve numbers at all. Operations on strings of symbols provide many examples of *non-numeric computation*–for example, searching for an occurrence of one string within another longer one or sorting a collection of strings lexicographically.
>
> The concept of computation is related to that of an algorithm. For the moment, let us say only that by *algorithm* we mean a certain sort of general method for solving a family of related questions. Examples of familiar algorithms are the truth-table algorithm for argument validity in the propositional calculus, the column method for conversion of decimal to binary numerals, and the Euclidean algorithm for determining the greatest common divisor of two integers. Such algorithms are the end result of the mathematician's quest for a general method or procedure for answering any one of some infinite family of questions.

Please note that the examples of non-numerical computations involve text. This should answer this concern of the Court which was expressed in *in re Bradley* because mathematicians recognize that

text processing has mathematical significance.

> [I]t may be that the data and the manipulations performed thereon by the computer, when viewed on the human level, represent the contents of a page of the Milwaukee telephone directory, or the text of a court opinion retrieved by a computerized law service. Such information is utterly devoid of mathematical significance.

S. Barry Cooper explains how old the concept of algorithm is:[5] (emphasis in the original)

> [A]lgorithms have played an explicit role in human affairs since very early times. In mathematics, the first recorded algorithm is that of Euclid, for finding the greatest common factor of two integers, from around 300 BC.

Cooper also explains how the modern understanding of computation has developed during the twentieth century.[6] (emphasis in the original)

> We can see now that the world changed in 1936, in a way quite unrelated to the newspaper headlines of that year concerned with such things as the civil war in Spain, economic recession, and the Berlin Olympics. The end of that year saw the publication of a thirty-six pages paper by a young mathematician, Alan Turing, claiming to solve a long-standing problem of the distinguished German mathematician David Hilbert. A by-product of that solution was the first machine-based model of what it means for a number-theoretic function to be computable, and the description of what we now call a *Universal Turing Machine*. At a practical level, as Martin Davis describes in his 2001 book *Engines of Logic: Mathematicians and the origin of the Computer*, the logic underlying such work became closely connected with the later development of real-life computers. The stored-program computer on one's desk is a descendant of that first universal machine.

The work of Alan Turing, along with contemporary work of Alonzo Church, Stephen Kleene and others are at the origin of the discipline known as Computation Theory. This discipline is one of the mathematical foundations of computer science, much like calculus is one of the foundations of physics and various fields of engineering.[7] This is why the mathematical concepts of computation and algorithms are fundamental to a factually correct understanding of modern day digital computers.

**How the Legal Definition of Algorithm Relates to Mathematics and Computer Science**

The term “algorithm” is a term of art in mathematics and computer science. I normally use this meaning of the term. In patent context there is a separate legal definition of the term “algorithm”. This may introduce some confusion. People with legal background often doubt that non-legal definitions are relevant because they say the legal definition is the one which is needed to invoke case law about algorithms.

This article is about explaining some facts. The legal definition belongs to the law. For purposes of understanding the facts it is necessary to understand what is the actual entity that is implemented by programmers. The definition which provides this insight is the one used in mathematics and computer science. If we want to know how the law relates to the facts we need to consider and compare both the legal and non-legal definitions.

Let's start with the legal definition of “algorithm”. This definition is this sentence from the *Benson* Supreme Court case.

> A procedure for solving a given type of mathematical problem is known as an "algorithm."

This definition is interpreted by the courts to be a “field of use” definition, that a mathematical algorithm must be directed to solving a problem within the discipline of mathematics. See the extracts from *Freeman*, *Toma* and *Bradley* in the legal explanation of software functionality above. If the field of use is not mathematics then the algorithm is not mathematical in the legal sense of

the word. In particular *Freeman* explains:

> The broader definition of algorithm is "a step-by-step procedure for solving a problem or accomplishing some end." *Webster's New Collegiate Dictionary* (1976)

This leads the *Freeman* court to observe:

> Because every process may be characterized as "a step-by-step procedure * * * for accomplishing some end," a refusal to recognize that *Benson* was concerned only with *mathematical* algorithms leads to the absurd view that the Court was reading the word "process" out of the statute.

In computer science and mathematics an algorithm is not defined so broadly as to mean any procedure for accomplishing some end. It is not true that algorithms in the legal sense are a special kind of what is called algorithms in mathematics and computer science. The relationship between these definitions cannot be characterized by saying one is broad and the other is narrow. This observation of the *Freeman* court applies to a dictionary definition of “algorithm”. It doesn't apply to a mathematically correct definition.

For the sake of the example let's consider the procedure of ordinary addition. I mean the paper and pencil procedure everyone who has learned arithmetic in school uses to find out that 697537+9768=707305. This is an algorithm in the mathematician's sense of the word. If this procedure is programed into a computer, it is a computer algorithm. I like to use addition as an example because it is familiar to the layman reader. Familiarity helps make the point clear.

When addition is used to add abstract numbers the field of use is mathematics. When it is used to add quantities of goods in inventory in a grocery the field of use is the management of a grocer's business. When this procedure is used to add the loads on a building structure the field of use is civil engineering. In each case the procedure of addition is the same. This is why mathematicians and computer scientists don't consider the field of use when defining the procedure of addition. The real world semantics of the mathematical abstractions has no relevance to the definition and execution of the steps in the procedure of addition.

This observation applies to the computing machine as well. If the addition is done with a mechanical device, then the same device may be used in all fields of use. If an electronic circuit is used, then the same circuit may be used for all fields of use. The difference between a (legal) mathematical algorithm and an algorithm which is not (legally) mathematical is neither in the steps of the procedure nor in the device implementing them. It is in the real world semantics given to the abstract mathematical entities.

This observation carries to all algorithms in computer science and mathematics. They are all procedures which are defined, with either mathematical or programming languages, in terms which are independent from the field of use.

The above relates to the legal notion of utility. It also explains the difference between Pure and Applied mathematics which is sometimes raised by patents attorneys for the purpose of patenting applied mathematics. We see that the steps of the procedure are not useful in themselves. Utility results from the real world interpretation given to the mathematical entities. The same procedure is or is not useful in the concrete world depending on whether or not it is used in connection with a real world interpretation.

### The Mathematically Correct Understanding of Algorithm

The original researchers who discovered much of the foundations of computation theory, Alan Turing, Alonzo Church and Stephen Kleene, worked to capture a specific flavor of computation which is also called an effective method. This is the type of computation an idealized human being using pencil and paper is able to carry out. In this context “idealized” means no error is ever done

in carrying out the steps of the computation, the idealized human being will always carry the task until completion no matter how long it takes (he has infinite tenacity and doesn't die before he is done) and he will never run out of pencils and paper. The intent is that the limitations of the notion of effective method must come from the computation itself and not from circumstances unrelated to mathematics.

See appendix A of the previous article for a detailed discussion. In this article I concentrate on the two most important aspects of this notion.

The first aspect is that every algorithm is a procedure to manipulate symbols. This is implicit in the notion that the computation may be carried out by pencil and paper. The assumption is that the problem we want to solve must be represented by means of a written encoding. There are two elements in the encoding. First there are some rules on the arrangement of symbols, the syntax. The second element is the rules for determining the meanings given to the symbols, the semantics. Together the syntax and the semantics defines the written representation of the entities which belongs to the problem we want to solve. In the case of arithmetical problems such as addition, this representation may be the ordinary decimal system we have learned in school when we learned how to write numbers. Problems in branches of mathematics others than arithmetic will be represented using appropriate written encodings.

The importance of symbols in computing has been beautifully explained by Raymond Greenlaw and James Hoover:[8] (emphasis in the original)

> The purpose of computation is to solve problems. However, before we can attempt to solve a problem, we must communicate it to another person, a computer, or just ourselves. We do this with a *language*. In very general terms, a language is a system of signs used to communicate information between one or more parties. In this book the language we use to talk *about* computation is a combination of English and mathematics. But what about the language *of* computation? That is what language do we use to communicate problems to, and get answers from, our computing machines? Answering these questions is the goal of this chapter.
>
> Even equipped with a fancy graphical 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 language, then it cannot be studied using the tools of computation theory.
>
> Thus, the first step of understanding a problem is to design a language for communicating that problem to a machine. In this sense, a *language* is the fundamental object of computability.

The other important aspect of the definition of algorithm is that all the information necessary to carry out the algorithm must be explicitly found in either the symbols being processed or in the rules which define the procedure. Nothing must be left implicit. Nothing must be left to the imagination or the judgment of the human carrying out the procedure. This part has been explained by Stephen Kleene as follow:[9]

> In performing the steps we have only to follow the instructions mechanically, like robots; no insight or ingenuity or intervention is required of us.

It is this feature of algorithms which makes it possible to automate them. Machines don't have the ability to fill in the gaps in an incompletely specified procedure. They can't make judgment calls or bring in information they can't explicitly access. The only procedures machines may possibly carry out are those where everything is sufficiently explicit to allow this kind of mechanical execution.

We can now see why all computer algorithms and all computer programs are mathematical

algorithms. Being an algorithm is a requirement to make machine execution possible.

But the reverse inference is also possible. Computations carried out by computers will meet these requirements for being mathematical algorithms. Computers are symbol processors. The bits are electronic representations of the symbols 0 and 1.[10] Patterson and Hennessy explains how symbols are implemented as electronic signals:[11] (emphasis and bold in the original)

> The electronics inside a modern computer are *digital*. Digital electronics operate with only two voltage levels of interest: a high voltage and a low voltage, All other voltage values are temporary and occur while transitioning between the values. (. . . ) The fact that computers are digital is also a key reason they use binary numbers, since a binary system matches the underlying abstraction inherent in the electronics. In various logic families, the values and relationships between the two voltage values differ. Thus, rather than refer to the voltage levels, we talk about signals that are (logically) true, or 1 or are **asserted**; or signals that are (logically) false, or 0, or are **deasserted**.

Here we see why the bits are symbols. They are abstractions different from the signals representing them because they are not defined by the specific voltages. They are defined by the convention used on which voltage is 1 and which is 0 and this convention varies from circuit to circuit.

Computers execute procedures for manipulating bits stored in their memory. These procedures are executed in a mechanical manner, like robots. Computers are machines. They add no insight, ingenuity or intervention beyond what is explicitly required by the rules of the procedure. By this standard computer programs meet the requirements of mathematics for a procedure to be an algorithm. Later in this article I will further support this conclusion by showing how computer programs meet a mathematically precise definition of algorithm, by showing how to exhibit the mathematical formulas for the computer algorithms and also by showing how the computer circuits implement the concept of a universal algorithm.

It is important to distinguish the algorithm from the mathematical function it computes. Hartley Rogers Jr explains:[12] (emphasis in the original)

> It is, of course, important to distinguish between the notion of *algorithm*, i.e., procedure, and the notion of *function computable by algorithm*, i.e., mapping yielded by procedure. The same function may have several different algorithm.

The function is a mathematical relationship between the inputs and the result. The algorithm is the procedure to follow in order to obtain the result. While the two notions are different, they are related. A function is computable when there is an algorithm for it. A function without a corresponding algorithm is uncomputable. In the latter case the relationship between input and output cannot be determined by means of a computer no matter how programed because there is no procedure we may give to the computer which will yield the desired result.

For example the number 707305 is the sum of 697537 and 9768. This relationship is the function. The pencil and paper procedure which when carried out yields 707305 when applied to 697537 and 9768 is an algorithm for computing this function. As you can see the function is a relationship between numbers but the algorithm is a procedure to manipulate the written digits, the symbols, which represents the numbers. There is more than one algorithm to compute an addition. For example if the numbers are stored in binary format in a computer memory the ordinary procedure for the decimal format is not applicable. The computer must use another algorithm suitable to the binary format.

In this example a function on numbers is presented but as was noted above, not all computations are about numbers and similarly not all mathematical functions are about numbers.[13] There are several branches of mathematics which are not about numbers and they use algorithms to solve non numerical problems. For example I may mention geometry, boolean algebra, set theory, graph theory, mathematical logic and lattice theory.

The term “function” may be seen as mathematical speak for the problem we want to solve. There is some data which is given as inputs to the problem and we seek an answer which is the solution of the problem. The association of the inputs with the resulting answer is the function.

Another way to put it is the function is *what* is being computed while the algorithm is *how* the computation proceeds. For this reason reciting a function in a patent instead an algorithm doesn't change the fact that the implementation of the computation is an algorithm.

Manipulation of symbols and mechanical execution are the two main requirements of mathematics for an algorithm to be an algorithm. How about the others? Some of them are the sort of things most people will take for granted, like the computation will always produce the correct result when carried out without errors, or the numbers of rules for the computation are in a finite number, or that the problem may be expressed with finitely many symbols.

There are also optional requirements in the sense that there are several flavors of algorithms and these additional requirements may be changed depending on which flavor is considered. One frequently mentioned example of such a requirement is termination. Most people expect the computation to produce an answer and this implies that it will reach some point where is is done computing. This is why the vast majority of people understand the term algorithm to include a requirement for termination. But there are meaningful computations which never terminate, like finding out all the decimals of pi. There is an infinite number of them. This computation will never terminate on its own. Either the user of the program stops it at an arbitrary point or the computer breaks or runs out of resources before being done. This is why some people relax the definition of algorithm to allow nonterminating procedures. This is the case in this article because there are examples of useful software which won't terminate on their own without outside intervention.

Another optional requirement is determinism, that is each step in the computation is completely determined by the rules of the algorithm. If we relax this requirement we get the notions of probabilistic algorithms and nondeterministic algorithms where some form of randomness is allowed. Another requirements which may be subject to variations is the number of agents which carry out the computation (distributed algorithm). These various flavors of algorithms are all part of mathematical foundations of computer science. Whether or not the requirements of a particular flavor are met doesn't change the mathematical nature of the algorithm as long as the requirements of *some* mathematically recognized flavor are met.

**How Mathematicians Achieve Mathematical Rigor When Discussing Algorithms**

The notions of computation and algorithm are intuitive concepts. There is no generally accepted definition of algorithm. This is why R. Gregory Taylor writes:[14]

> We must grant that the algorithm concept is somewhat vague to the extent that notions like “next step,” “carrying out,” and “output,” although clear enough for most purposes, are not themselves characterized in any way and are hence subject to varying interpretations. As such, the algorithm concept belongs to the philosophy of mathematics.

Then how could mathematicians reason about algorithms and computations with the rigor and exactness which is the hallmark of mathematics? This connects to the question of how mathematicians determine whether a procedure is an algorithm according to the requirements of mathematics. If a court is interested in answering this question they may want to know how mathematicians themselves answer it. This may save them the difficulty of designing their own test in absence of a universally accepted definition.

Mathematicians rely on models of computations. These models are groups of algorithms which are defined with mathematical precision.[15] Only algorithms constructed in a manner permitted by the model out of elementary operations permitted in the model are allowed. An example of a model is the random access stored program machine, or RASP. This is the mathematical model which corresponds to the stored program architecture of computers.

When reasoning about a model of computation the mathematician does not contemplate every possible algorithm. He considers a specific group of algorithms which must be defined in a specific manner out of specific basic operations. Then rigorous mathematical logic may be used. Of course, such mathematical analysis will be limited to the algorithms which conform to the model. This is the price mathematicians must pay to achieve mathematical rigor.

How big a price is this? It turns out that it is not so big. Some models are powerful enough to include at least one algorithm for every possible computable function. If we take the issue from the function angle instead of the algorithm angle, these privileged models are exhaustive. They can compute all functions which can be computed. But when used in this sense, "can be computed" must be interpreted as "can be computed in principle by an idealized human being" which never makes mistake, never gives up or dies before he is done computing and never runs out of pencil and paper.[16] This is the concept of effective methods which has been previously mentioned. This concept forms the basis of the intuitive notion of "algorithm".[17]

This result is known as the Church-Turing thesis. It may be paraphrased as this: if a problem may be solved by some algorithm no matter how defined, then each of the privileged models of computation contains at least one algorithm which will solve this same problem. The models identified in the Church-Turing thesis are the Turing machines, the recursive functions and lambda-calculus. All three models are equivalent in the sense that every algorithm in any of the three models has corresponding algorithms for computing the same function in the other two models.

In my experience this point has been met with skepticism in legal circles. A common objection is that computers are not Turing machines, therefore the Church-Turing thesis is legally irrelevant because the patents are drafted to what runs on actual computers. This is failing to notice that the Church-Turing thesis is about *computations*. It is not about computers. This objection does not answer the point which is being made because this point brings no requirement for the computer to be a Turing machine. The requirement is for the computation to be Turing-computable.

The difference between software and hardware is that the hardware is the agent carrying out the computation but the software is the description of the computation, which is a procedure for manipulating symbols. The question is whether this procedure meets the requirement of mathematics for being a mathematical algorithm. The difficulty is that the notion of algorithm mathematicians use is intuitive. An algorithm is a certain type of procedures a human being using pencil and paper is able to follow mechanically, like robots. This is not a mathematically exact formulation. This makes a mathematically exact determination of whether a procedure is an algorithm hard to achieve. The solution mathematicians have found to this problem is the Church-Turing thesis.[18]

Those who deny that Turing machines and the Church-Turing thesis are legally relevant are in effect saying that the mathematically exact test recognized in mathematics for telling when a procedure is a mathematical algorithm is not legally relevant. My point is that if the courts care about what a mathematical algorithm is they should pay attention to what mathematicians have to say on the question.

Here is a description of the Church-Turing thesis by Stephen Kleene. He is one of the researchers who helped in the discovery of the thesis.[19] (emphasis in the original, some mathematical notation adapted to make it friendly to web browsers)

> The situation in 1935 was that a certain exactly defined class of computable number-theoretic functions considered by Church and Kleene during 1932-35, called the "λ-definable functions", had been found to have properties strongly suggesting that it might embrace all functions which can be regarded as computable under our vague intuitive notion. This result was somewhat unexpected, since it was not clear the class contained even the particular function *pred*(*a*) mentioned above, and a proof in 1932 (publ. 1935) that it did was the present author's first piece of mathematical research. Another class of computable functions, called the "general recursive functions", defined by Gödel in 1934

> building on a suggestion of Herbrand, had similar properties. It was proved by Church 1936 and Kleene 1936a that the two classes are the same, i.e. each λ-definable function is general recursive and vice-versa.
>
> Under these circumstances Church proposed the thesis (published in 1936) that all functions which intuitively we can regard as computable, or in his words “effectively calculable”, are λ-definable, or equivalently general recursive. This is a thesis rather than a theorem, in as much as it proposes to identify a somewhat vague intuitive concept with a concept phrased in exact mathematical terms, and thus is not susceptible of proof. But very strong evidence was adduced by Church, and subsequently others, in support of the thesis.
>
> A little later but independently, Turing's paper 1936-1937 appeared in which another exactly defined class of intuitively computable functions, which we shall call the “Turing computable functions”, was introduced, and the same claim was made for this class; this claim we call *Turing's thesis*. It was shortly shown by Turing 1937 that his computable functions are the same as the λ-definable functions, and hence the same as the general recursive functions. So Turing's and Church's theses are equivalent.

Kleene is hinting as what the evidence is for the Church-Turing thesis. The notion of algorithm is intuitive. Mathematicians can't prove the thesis with a mathematical theorem because algorithm is not a concept which is defined with mathematical rigor. The inability to write theorems about algorithms is the very issue the thesis is meant to solve.

A large part of the evidence for the thesis is that every attempt to define a model of computation which would compute functions which are not Turing-computable have failed. Marvin Minsky gives us more details.[20]

> Perhaps the strongest argument in favor of Turing's thesis is the fact that, over the years, all other noteworthy attempts to give precise yet intuitively satisfactory definitions of “effective procedure” have turned out to be equivalent – to define essentially the same class of processes. In the 1936 paper Turing proves that his “computability” is equivalent to the “effective calculability” of A. Church. A very different formulation of effectiveness, described at about the same time by Emil Post, also turned out to have the same effect: we show the equivalence of Post's “canonical systems” and Turing machines in chapter 14. Another quite different formulation, that of “general recursive functions” due to S. C. Kleene and others is also equivalent, as we show in chapters 10 and 11. More recently a number of other formulations have appeared, with the same results (e.g., Smullyan's “Elementary Formal Systems” [1962]). Whenever a system has been proposed which is not equivalent to these, its deficiencies or excesses have always been intuitively evident.
>
> Why is this an argument in favor of Turing's thesis? It reassures us that different workers with different approaches probably did really have the same intuitive concept in mind – and hence leads us to suppose that there is really here an “objective” or “absolute” notion.

The Church-Turing thesis is under permanent on-going validation. New models of computations are constantly being discovered with the progress of research. Each of these models is analyzed mathematically to see how it fits with the thesis. Some models were found which are more powerful than Turing machines, but the corresponding computations are mathematical abstractions which may not be actually carried out. Mathematicians say these models are not effective. Other models were found whose computations may be carried out. Either they were found weaker than Turing machines, i.e. they perform some Turing computable functions but not all of them, or they ended up being added to the list of Turing-equivalent models of computation which have been discovered. This list keeps growing as research progresses. Robert Sedgewick and Kevin Wayne list dozens of universal models of computability in their page on universality. You will find on this page that models of computations implementing multiple flavors of algorithms, like non-deterministic and probabilistic Turing machines, have been researched and found to be Turing

computable. This is how we know that changing the requirements of the intuitive notion to bring up a new flavor of algorithm doesn't invalidate the Church-Turing thesis.

How do we know that two models of computation are equivalent? Mathematicians use a proof technique called reducibility. This technique takes the computations in a model, say lambda-calculus, and shows mathematically how the same manipulations of symbols could be done with a Turing machine. Then mathematicians do the analysis in the reverse direction showing the manipulations of symbols of Turing machines can also be done in lambda-calculus.[21]

The method of proof is important. It shows that the computations are the same regardless of the differences in the details of the models. The language of the thesis says the computable *functions* are the same. But when one looks at the details of the proof one finds that what is actually proven is that the manipulation of *symbols* is the same. If one wants to be finicky he will find differences. The elementary operations of lambda-calculus are not done as single operations by the Turing machines. Several elementary Turing machine operations are necessary to replicate a single lambda-calculus elementary operation and vice-versa. But when we look at *what* is being done instead of *how* it is done, we see that the computation is the same because the symbols end up being manipulated in the same manner. This is what mathematical reducibility is about, showing that despite the apparent differences the same procedure is being carried out.

This is emphasizing a point already made, that this mathematical analysis is about computations and not computers. It makes no sense to require that the computation be done by an actual Turing machine. Such requirement defeats the point of Turing reducibility which is to show that the computations are the same despite of apparent differences.

Mathematicians use the thesis to write proofs of theorems which are applicable to all computable functions. They take one of the models which is recognized as Turing equivalent and prove that their theorem applies to all functions in this model. Then by the thesis they have covered all computable functions no matter how computed. I am suggesting a different use of the thesis. When confronted with a computation, courts may verify that it is the kind of computation which results from a mathematical algorithm by verifying that it belongs to one of the models which has been proven Turing computable. Then the Church-Turing thesis will guarantee that this computation results from a mathematical algorithm. This is a mathematically rigorous test. It eschews the definitional difficulties resulting from the lack of a universally recognized definition of algorithm while resulting into an answer which is mathematically accurate.

Mathematicians are constantly testing, using reducibility, whether their mathematically precise definition of “algorithm” is correct. They verify that the newly discovered models of computations are just new ways of doing the same old computations. As long as no new model is found which permits new computations the Church-Turing thesis remains valid. Finding such a model would result in enlarging the range of procedures which qualify as algorithms. Then the thesis would need to be revised to include the newly discovered computations. Procedures which are proven to be algorithms by means of the current version of the thesis will remain algorithms under the new version. A court which uses the thesis to verify whether a procedure is an algorithm runs no risk of having a finding that it is indeed an algorithm being contradicted by future mathematical discoveries.

For those who want to see what a reducibility argument looks like, there is an example written in plain English. This is Alan Turing's own analysis of how a Turing machine is equivalent to a human carrying out a mathematical algorithm with pencil and paper. This analysis is part of the evidence that Turing machines are equivalent to the intuitive concept of effective methods. He shows that the limitations inherent to pencil and paper calculations are such that all algorithms can be fitted on the Turing machine. His analysis is conducted in English instead of mathematical language because pencil and paper calculations is not a mathematically defined concept. But for this limitation Turing's analysis is exactly the sort of thing reducibility entails.

Imagine that a human is required to carry out all calculation on paper of a special format. Two-dimensional sheets are not used. The calculations must fit on a long ribbon divided into squares,

with one symbol fitting into the squares. This ribbon could be inserted into a typewriter. The human carries out the calculations with the operations permitted by the typewriter: write a symbol, possibly erasing what what previously written in the square, or moving the tape one square to the left or to the right. Then is the procedure such that we can automate the typewriter to do the calculation all of its own? If so then the computation is carried out mechanically, like robots. It meets the requirements for being a mathematical algorithm. This automated typewriter is the Turing machine.

Turing doesn't use the word “typewriter” when describing a Turing machine, but I believe this metaphor makes the concept more intelligible. Turing also use the word “computer” to refer to the human being carrying out the calculation. This was written in 1936. This word had a different meaning back then.

Here are the first two paragraph of Turing's analysis.[22]

> 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 mind 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 affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.

You may read the rest of the analysis in the original paper. It is available on-line here. The analysis begins at page 01u, lasts for three pages (including the first) and ends on the top of the fourth page.

**Mathematically Correct Answers to Useful Legal Questions from the Church-Turing Thesis**

Please observe that Turing's analysis includes a proof that the number of states of mind of the human carrying out the calculation must be finite. If you go read the complete analysis you will find that this result is used to show that it is OK for a Turing machine to have a finite number of states because the machine states are corresponding to the states of mind of the human.

We now have a second legally useful question which may be answered with mathematical precision. The first one is whether a procedure for manipulating symbols meet the requirements of mathematics for being an algorithm. The other one arises in the context of *Cybersource*

*Corporation v. Retail Decisions, Inc.*

> [W]e find that claim 3 of the ’154 patent fails to recite patent-eligible subject matter because it is drawn to an unpatentable mental process—a subcategory of unpatentable abstract ideas.

and also from the same case

> It is clear that unpatentable mental processes are the subject matter of claim 3. All of claim 3’s method steps can be performed in the human mind, or by a human using a pen and paper.

We may ask, what is the relationship between abstract mathematical algorithms and mental processes which can be performed in the human mind? Alan Turing's analysis answers this question. Algorithms and the computations of Turing computable functions are mental processes except for two differences.

First a mental process may involve a human judgment call. This goes against Kleene's prohibition: “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 is not something which could be automated into a Turing machine either. This means that mental processes which involve judgment calls are not mathematical algorithms. They may still be abstract from the point of view of the law but this would be for reasons unrelated to being mathematical algorithms.

The other difference between algorithms and mental processes is that mathematical algorithms are not subject to the limitations of live human beings. For example an addition doesn't stop being a mathematical algorithm because the numbers have billions of digits or there are too many numbers to add for a human to handle in his lifetime. Mathematicians define “algorithm” on the basis of an idealized human being who is not subject to down to earth limitations.

If the Federal Circuit accepts that these differences are immaterial the principles of *Cybersource* may be applied to procedures which are reducible to Turing machines on the basis that they are abstract ideas.

Here is another legally useful question the Church-Turing thesis answers in a mathematically correct manner. Is a mathematical algorithm a procedure to manipulate symbols or is it a procedure to compute numbers? Where should the courts draw the line? The mathematically correct answer is that the difference is immaterial and the courts should not try making such distinctions.

Turing machines and lambda-calculus are models of computation which manipulate symbols and they are not limited to symbols representing numbers. They don't directly do arithmetic but they may carry out arithmetic calculations indirectly when the symbols are chosen to represent numbers. On the other hand recursive functions are arithmetic calculations on the natural numbers. It is possible to represent symbols numerically with Gödel numbers. Then every manipulation of symbols from Turing machines or lambda-calculus may also be performed arithmetically on the corresponding Gödel numbers with recursive functions.

These ideas were analyzed by mathematicians using reducibility. We have mathematical proof that whatever we can compute with any one of these three models can also be computed with the other two. This is one of the discoveries which led to the formulation of the Church-Turing thesis. The consequence is that the difference between manipulation of symbols and arithmetic is not a difference in substance. It is a difference in how we represent the information.

The form of mathematical analysis compares entire models of computation at once. Mathematicians write theorems which provide a recipe to reduce all algorithms belonging to some model into a Turing machine, recursive functions or whatever model is the target of the reduction.

I suppose this phenomenon could be convenient to a court. Instead of having to analyze algorithms one by one to find out whether they are mathematical algorithms they may verify whether they belong to one of the models mathematicians have already analyzed. If so they have a ready-made determination that they are mathematical algorithms using a test which is accepted in mathematics.

Models of computations matching the behavior of computers have been analyzed in this manner and it has been found that these models are equivalent to Turing machines.[23] The manipulations of symbols done by a digital computer may be done by Turing machine. The proof works on the model of computation and is applicable to all computations matching the model. This means that all the computations carried out by software meet the requirement of mathematics for being an algorithm by virtue of this theorem.

John Hopcroft and Jeffrey Ullman elaborate on this point using a model of computation called random access machines, or RAM.[24] (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.[25]

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

The RAM and the previously mentioned RASP model are related. The difference is that in the RASP the machine instructions are stored in the machine memory and in the RAM they are stored separately. They are otherwise identical. It has been proven that these two models are equivalent because they may be reduced to each other.[26] Then the above theorem shows that the computation carried out by a RAM can also be done by a Turing machine. This theorem shows that software passes the test of reducibility to a Turing machine. By this standard software meets the requirements of mathematics for being a mathematical algorithm.

It bears repeating that the argument is about computations and not computers. For example I have received the objection that computers don't have an infinite tape or infinite memory. According to the objection the computer cannot implement all the algorithms a Turing machine can compute because of its limited memory and this somehow invalidates the argument of equivalence with Turing machines. But the test goes in the other direction. We don't have to show that all Turing computable functions run on an actual computer. It is good enough to show that if a computation fits in the limited memory of the computer this same computation may be carried out by a Turing machine according to the theorem. This means this computation meet the requirements of mathematics for being a mathematical algorithm. It doesn't matter that some other computation will fail to run by lack of memory because this other computation is not the one being tested.

However it is easier to think of this issue in terms of the facet of the definition of mathematical algorithms which makes abstraction of real life limitations. For most practical purposes the limited quantity of memory doesn't matter because the computation fits in the available resources. When the computation doesn't fit in the resources available the solution may be to get a bigger computer. In this sense, for purpose of determining whether software is a mathematical algorithm,

the limitation on memory size should be disregarded.

This sort of things happens in computer arithmetic too. The machine instructions for addition work on a limited number of bits, say 64 bits. One may argue this is not the true operation of arithmetic because it will fail when the numbers require more than 64 bits. This is akin to arguing addition is not a mental step a live human being may perform because the numbers could require trillions of digits and this is too unwieldy for an actual human to compute. But reasonably we have to disregard the effect of real life limitations on the execution of the procedure when determining whether an abstract mathematical computation is carried out, otherwise nothing we compute will be mathematics.

**We Can Show the Formulas**

There is another way to show that all software describes a computation pursuant to a mathematical algorithm. We can exhibit the mathematical formula. This may be done in at least two different manners

1. *The language semantics approach*: there is a considerable amount of scientific literature on how to define mathematically which computation is the semantics of a programming language. [Stoy 1981] is a well regarded reference. The computations carried out by a program in the language are described mathematically by means of a model provided by the language semantics. All programs have a mathematical semantics which may be stated by means of these formal methods.

   Some languages have their official definition stated mathematically. In such case the mathematical computation corresponding to a given program is given by the language definition. Any correct implementation of the language must carry out this mathematical computation and any patented method implemented in the language is implemented by means of a mathematical algorithm whose formula is known. The previous article indicated two languages which are mathematically defined in this manner: Standard ML and Coq.

2. *The circuit approach*: in modern computers the circuitry is designed in such manner that the device is incapable of doing anything but carry out a mathematical algorithm. It is possible to find out which algorithm by inspecting the circuitry. The algorithm which is implemented in a modern stored program architecture computer is a variation on the kind of algorithms mathematicians call a Random Access Stored Program or RASP. We may exhibit the formula for the particular algorithm which correspond to a specific computer.

When a computation is described in mathematically precise terms using formulas, this is evidence that the algorithm is mathematical. The knowledge of the formulas eschews the definitional difficulties due to the lack of a universally accepted definition of algorithm. It also avoids the objection that a computer is not a Turing machine because this argument doesn't rely on them.

I have received in response to my previous article a number of comments which stated in various forms the same idea: that it is not because something may be described mathematically that it is unpatentable mathematics. The point being that everything can be described mathematically somehow. This is no excuse to say everything is mathematics and nothing is patentable. My understanding of these comments is that they are variations on this theme, expressed by the Court or Customs and Patent Appeals in *in re Bernhart*. (emphasis in the original)

> Moreover, all 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.

This argument would be carrying a lot of weight had the mathematics of software been a description of the computer circuitry, or even a description of the process by which the circuit operates. But the mathematics of software describes a computation which is an abstract entity separate from the circuit. This distinction is of the same nature as the distinction between the marks of ink on paper and the printed text because a computation is a manipulation of symbols.

This objection and the objection that computers are not Turing machines share the same fundamental flaw. They assume software defines some physical process or machine. The error is to ignore that the discussion is about symbols. They are abstractions different from their physical representations and their semantics is not a physical phenomenon. Once symbols are taken into consideration it is clear that the objections do not answer the point being made.

The question is whether the symbol manipulations carried out by a computer qualify as a mathematical algorithm. It is perfectly appropriate to describe computations with formulas (or Turing machines) and conclude they are indeed part of mathematics on this basis.

Another objection I often received is that Applied Mathematics, as opposed to Pure Mathematics, is patentable. This sends us to the difference between the legal definition of algorithm and the one mathematicians use. Pure Mathematics is not different from Applied Mathematics except for the real world semantics given to the symbols. The procedures for the computations and the machines carrying them out are identical.

Let's suppose the task is to show that a specific patent claim is reading on a mathematical algorithm, possibly with a limitation on the real world meanings of the mathematical entities. This is a narrow goal, targeted to a particular patent claim. We want to exhibit the mathematical formula corresponding to an algorithm reading on this claim. Most probably the simplest way to proceed would be the language semantics approach. We program the patented method in a language such as Standard ML which has a mathematically defined semantics. Then the patented method may be proven to read on a mathematical formula as follows:

- We show the language definition. Everyone may see that the definition is a series of mathematical formula. Not everyone will understand the formulas but at least their eyes will tell them that the formulas are mathematical because they have this inimitable mathematical gobbledygook appearance and an expert witness will confirm that they can trust their eyes.

- The method is written in the language which means the computation is defined mathematically in accordance with the language semantics.

- When we run the program we can verify that its behavior is the one mentioned in the claim.

These three bullets together show that the claim reads on a mathematical algorithm. Notice how this approach doesn't require an audience of laymen to understand mathematics and computer science. It requires the audience to trust their eyes and believe the experts' explanations. The objection we sometimes hear that the software is mathematics argument is unintelligible to laymen doesn't apply to this procedure.

Notice also that this procedure is targeted at a specific patent. No effort is made to prove that all software patents are mathematical algorithm but practically speaking the effect may be almost the same. It is an unusual software patent that can't be programmed in the language of the programmer's choice, therefore Standard ML is almost always an option. Once the courts have accepted this kind of demonstration it can be repeated for other software claims.[27]

Let's suppose that we broaden the task to show that all software no matter how written is specifying a mathematical computation according to a mathematical algorithm, possibly with a limitation on the real world meanings of the mathematical entities. We want a recipe to exhibit the mathematical formulas for all software. Then the simplest way to proceed would probably be the circuit approach. If we can show that the computer can only compute a computation pursuant to a

mathematical algorithm corresponding to a known series of mathematical formulas then we are done. All software is eventually executed by the hardware pursuant to these formulas.

I have not found an actual computer whose mathematical definition is known in the same manner as the definition of Standard ML is known. There is some work to be done here. Someone would have to write such a definition and show it is applicable.

Computer professionals will often refer to boolean algebra. This is a branch of mathematics which is the foundation of digital electronics. Modern digital circuits are often built from logic gates which implement logical operators which are mathematically defined with boolean algebra. Therefore boolean algebra forms a mathematical basis suitable to define mathematically the algorithm corresponding to a circuit.

Personally I prefer to use lambda-calculus, in part because it includes boolean algebra as one of its subsystems and in part because it is applicable in circumstances where no logic gates are involved. The advantage is we can define algorithms corresponding to a broader range of physical implementations.

I have an exhibit to submit to the readers. It shows how to define the mathematical semantics of a circuit using lambda-calculus. I mean the exhibit contains a method that if applied to a specific circuit it will result into the definition of the mathematical formulas corresponding to the computations done by this circuit. This is a document full of mathematical language. Mathematical skills are required to understand this material. If you don't have such skills then the main interest of this exhibit is to let you know that people with the right skills can indeed write down the mathematical definition of the computation performed by a computer. The Exhibit: Hybrid Denotational/Operational Semantics for Physically Implemented Computations

Here is a list of hardware features. Each of these features has an effect on the symbols. This effect is the contribution of the feature to the computation. I have provided a formula describing the effect of the symbols of each feature. Some people will object that the listed hardware features are patentable. Remember that the computation is not the circuit. The formulas describe the computations. They relate to the hardware only to the extent the hardware carries out the computation.

- Main memory
- CPU flags and registers
- Selected instructions: JUMP, MOV, ADD, CMPXCHG and HLT
- Addressing modes
- Input and output instructions
- Hardware random number generators
- Interrupts and return from interrupts
- Direct Memory Access (DMA)
- Page tables and page fault exceptions
- Symmetric Multi-Processing (SMP)
- Resilient computations that resume after powering the computer off and on again
- Point-to-point networking
- Networking over noisy lines that may mangle the data during transit
- Networking over shared communication channels like coaxial cables and wireless
- Networks comprised of devices that may power off and on while the network is running
- Networks where devices and connections are added and removed as the computation is running

The point of this list is to show the breadth of coverage permitted by the lambda-calculus approach.[28] Every now and then I have encountered someone asking how about such and such feature, thinking this particular feature is outside the reach of mathematics. A list such as this one answers this kind of argument, partly because all the features which have been brought to my attention as contentious are in the list, and partly because the breadth of this list shows that the method I use is very flexible. With a little work more features may be added as needed.

**Mathematics Is Speech**

This issue in this section is to explain why mathematics is speech.

Mathematical formulas may have several meanings depending on context. For example:

1. It can be some real world semantics within the field of use, like some accounting rule or the shape of an antenna for electromagnetic waves; or,
2. it can be a description of the computer and/or its operating principles; or,
3. it can be a description of the abstract mathematical calculations.

Option 3 applies to all formulas. The calculations are always mathematical. The question is what else is described. Formulas may describe both a computation and something physical in the real world. The real world semantics or the computer are not mathematics but the calculations are mathematics.

Judging from the quote from *Bernhart* in the preceding section, it appears that case law admits that mathematics is a language which is used to state facts and reason about the real world. This is a point which is known to everyone who has studied physics or chemistry in school. It is also known to everyone who knows the basics of accounting, business management or economics among other disciplines.

This is also revealed by the history of mathematics. Keith Devlin explains the birth of arithmetic during antiquity as this:[29]

> Up to 500 B.C. or thereabouts, mathematics was indeed the study of numbers. This was the period of Egyptian and Babylonian mathematics. In those civilizations, mathematics consisted almost solely of arithmetic. It was largely utilitarian, and very much of a 'cookbook' nature (“Do such and such to a number and you will get the answer”).

Then, after discussing the ancient Greeks mathematics, Devlin continues with how calculus was discovered.[30] (emphasis in the original)

> There were no major changes in the overall nature of mathematics, and hardly any significant advances within the subject, until the middle of the seventeenth century, when Newton (in England) and Leibniz (in Germany) independently invented the calculus. In essence, the calculus is the study of motion and change. Previous mathematics had been largely restricted to static issues of counting, measuring and describing shape. With the introduction of techniques to handle motion and change, mathematicians were able to study the motion of planets and of falling bodies on earth, the workings of machinery, the flow of liquids, the expansion of gases, physical forces such as magnetism and electricity, flight, the growth of plants and animals, the spread of epidemics, the fluctuation of profits, and so on. After Newton and Leibniz, mathematics became the study of number, shape, *motion*, *change* and *space*.

We see that throughout history the development of mathematics has been motivated in a large part by the desire to describe the world and reveal some of its properties which would have otherwise remained unknown to mankind. Sure much of the research in mathematics has been done for the sake of mathematics. But the practical motivation has always been present. There is much more in Devlin's book if more evidence is required.[31]

How about algorithms? Their discovery is part of this development of mathematics for practical purposes. Algorithms are procedures to solve problems. S. Barry Cooper reports:[32] (emphasis in the original)

> The word *algorithm* is derived from the name of the mathematician al-Khwarizmi, who worked at the court of Mamun in Baghdad around the early part of the 9$^{th}$ century AD. The

sort of algorithms he was interested in had a very practical basis. This we can see from his description of the content of his most famous book *Hisab al-jabr w'al-muqabala* ("The calculation of reduction and restoration"), said to be the first book on algebra, from which the subject gets its name.

> . . . what is easiest and most useful in arithmetic, such as men constantly require in case of inheritance, legacies,partition, lawsuits, and trade, and in their dealings with one another, or where the measuring of lands, the digging of canals, geometrical computations, and other objects of various sorts and kinds are required.

This should settle the "description of the real world" aspect of mathematics. How about the language aspect?

Mathematics is written with symbols. This written language has a syntax and a semantics. It has rules of logic for carrying out deductions and find out new truths based on known truths. More exactly there are many mathematical languages depending on which branch of mathematics is being discussed and which encoding is used to represent the specifics of the mathematical problem, each with its own alphabet, syntax, semantics and rules of logic. The meanings of the languages belong to an abstract universe of mathematical entities. Also these entities may be used to describe the real world, giving the mathematical language a real world utility.

Mathematicians have explicitly recognized the role of symbols.[33] They define the foundations of mathematics by means of formal systems of symbols. This is called mathematical logic and involve the study of the syntax and semantics of formal systems based on formal languages. Marvin Minsky, citing Emil Post explains:[34] (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*.

Stephen Kleene fills in more details:[35] (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.

Algorithms are procedures for manipulating the symbols and the computations are the manipulations of the symbols. This makes them features of the mathematical language. These symbolic manipulations are related to the operations of mathematical logic which are used to

deduce new truths from known truths.[36] In particular algorithms is the solution mathematicians have found to the issue of rigor. No intuition, insight or ingenuity is required to find objectively what is the answer to a problem. Algorithms also allow to determine whether a mathematical proof is valid when it has been written in a formal language. It is possible to verify all the inferences with a mechanical process, an algorithm. This ensures that there is an objective test as to whether or not something is proven in the mathematical language. There is no room for human subjectivity once the axioms and the rules of the formal system have been defined. Marvin Minsky explains[37] (emphasis in the original)

> We have to be sure, in accepting a logical system, that it is really “all there” – that the methods for deriving theorems have no dubious “intuitive” steps. Accordingly, we need to have an *effective procedure* to test whether an alleged proof of a statement is really entirely composed of deductive steps permitted by the logical system in question. Now “theorems” are obtained by applying rules of inference to previously deduced theorems, and axioms. Our requirement must then be that there is an *effective* way to verify that an alleged chain of inferences is entirely supported by correct applications of the rules.

Algorithms and the corresponding formulas are also used to define concepts and reason about them. For example consider the notion of force in physics. It can be truly understood only with the knowledge of the mathematical equations which formulates the relevant law of physics. It is possible to predict the movement of bodies with the calculations dictated by these equations. Mathematical calculations may be used to define concepts and reason about them in a variety of fields such as accounting, tax law, economics, biology, land surveying etc.

Keith Devlin describes this linguistic aspect of mathematics as follow:[38] (emphasis in the original)

> Without its algebraic symbols, large parts of mathematics simply would not exist. Indeed, the issue is a deep one, having to do with human cognitive capabilities. The recognition of abstract concepts and the development of an appropriate language to represent them are really two sides of the same coin.
>
> The use of a symbol such as a letter, a word, or a picture to denote an abstract entity goes hand in hand with the recognition of that entity *as an entity*. The use of the numeral '7' to denote the number 7 requires that the number 7 be recognized as an entity; the use of the letter *m* to denote an arbitrary whole number requires that the *concept* of whole numbers be recognized. Having the symbol makes it possible to think about and manipulate the concept.

This should make clear that mathematics is a language with a syntax, a semantics and rules of logic for conducting inferences. They should make clear that algorithms are features of this language and are an essential tool to achieve mathematical rigor.

**How Programmers Make Useful Functionality and the Nature of Software Processes**

In this section I will start looking at the issue I have summarized in the first part of this article: how hardware, software and data work together to make functionality. The focus is on the programmer's activities point of view. The physical implementation point of view will come later.

One of the programmer's tasks is to find out a representation of the elements of the problem in terms of machine symbols. The bits, the 0s and the 1s stored in the computer, are these symbols. The programmer must decide on an arrangement of bits, this is a syntax, and the meanings he will give to the bits, this is semantics. Together the syntax and the semantics are the representation of the elements of the problem. This is one key requirement for an algorithm to be an algorithm.

Another task of the programmer is to define procedures that are entirely explicit. Every step must be completely defined by the symbols being processed, the bits, and by the rules for manipulating them, the program. This must include the rules on how to decide when the procedure is completed and the rules on which step must come next when the current step is over. Only then is the

procedure machine executable. This is another key requirement for an algorithm to be an algorithm.

This is a short explanation of why every computer program is an algorithm in the sense mathematicians give to this word. Of course a complete proof should use the more elaborate evidence described earlier in this article and the previous article.

This is also a short explanation of how software functionality is implemented. It is easy for a layman to believe all functionality is produced by a machine. The layman sees the computer working and he sees the functionality happens. This is a superficial walk like a duck and quack like a duck kind of analysis. But half of the work is in the definition of the data representation. This has nothing to do with instructions executed by a computing machine. Think of a printing press. If we claim “a printing device configured to print a paper form comprising boxes for writing such and such data”, should we say the definition of the paper form is a functionality of the printing device? This sort of things happens frequently in computer programming because the work of a programmer is as much about the data as it is about the computer.

Let's see an example of functionality resulting from data definition which occurs in an actual patent claim. This is claim 1 from patent US7,222,078. It is being asserted against iOS developers by Lodsys. This claim has been previously discussed on Groklaw, also here, here and here.

1. A system comprising:

    - units of a commodity that can be used by respective users in different locations,

    - a user interface, which is part of each of the units of the commodity, configured to provide a medium for two-way local interaction between one of the users and the corresponding unit of the commodity, and further configured to elicit, from a user, information about the user's perception of the commodity,

    - a memory within each of the units of the commodity capable of storing results of the two-way local interaction, the results including elicited information about user perception of the commodity,

    - a communication element associated with each of the units of the commodity capable of carrying results of the two-way local interaction from each of the units of the commodity to a central location, and

    - a component capable of managing the interactions of the users in different locations and collecting the results of the interactions at the central location.

Please take a look at the element of functionality of being “further configured to elicit, from a user, information about the user's perception of the commodity”. How does one do this? Perhaps the user interface displays this line of text and then accepts input from the user as an answer.

> What do you think of this software?

Assuming that the software is considered a unit of commodity then this display of text would meet this element of functionality. What if instead we change the text for this one?

> What do you think of Barrack Obama?

By changing the two words “this software” to “Barrack Obama” we have changed the functionality from customer feed-back to political opinion polling. There is no need to change the program. The instructions in memory for execution by the computer will be exactly the same. The only change is two words shown to the user. It also happens that changing these two words means the software

no longer infringe on the patent because it no longer elicit information about the user's perception of the commodity.

Often the text of the user interface is kept in a separate data file for internationalization purposes. It is much easier to translate software into foreign language when we only have to change the data in a file without having to rewrite any software. When we use such a programming technique to implement this claim, the change of the two words is an update to this file with no change to any file containing software.

This is showing one of the ways where functionality results from defining the data instead of writing executable code. Many other methods are known and some of them will be discussed later in this article. As was said, data definition is an integral part of every program because every program uses data. We can see that the notion that all functionality is resulting from the software procedure, the instructions, without considering the data is erroneous. We can also see that the notion that software is a physical process defined by the instructions is also erroneous.

What is the mathematical calculation in a computer program? It is the pure manipulation of symbols, with their semantics abstracted away. There is a branch of mathematics called formal language theory. These formal languages consists of strings of characters arranged according to the rules of syntax absent of any semantics. These strings are the mathematical entities which are manipulated by algorithms.[39] Of course in many cases the symbols have an abstract mathematical semantics. But this semantics is not the calculation. Remember we need to distinguish the function from the algorithm. The function is about the relationship between the mathematical abstractions represented by the symbols. The algorithm is the procedure manipulating the symbols.

How does an algorithm differs from a procedure which produces a useful result? This is a key question to patent law because such procedures are processes and are patentable. This question amounts to asking for the difference between an algorithm and a patentable process.

One difference is that algorithms require the use of symbols. Processes which don't involve symbols are not algorithms. But the main difference is that algorithms don't have a real world utility by themselves. The utility depends on whether or not the symbols are given an interpretation in the real world. Without such an interpretation the algorithm is purely abstract mathematics.

Another aspect is that all information to execute the algorithm must be in the symbols or in the rules one must follow to manipulate them. This means the execution of the algorithm doesn't depend on the meanings of the symbols. The required abilities are to recognize the symbols and manipulate them according to the rules of the procedure. This explains how algorithms may be carried out in absence of a real world semantics for the symbols and how they may be carried out in absence of actual real world utility.

The symbols semantics is abstracted away when defining the syntax of the representation and the rules for the manipulations of symbols but it is not done away. The semantics is used to interpret the symbols when reading the answer.

This gives us some considerable insight on the nature of the processes which are claimed in software patents.

> *The software processes described in software patents are (collections of) mathematical algorithms where mathematical entities are used to describe some real world objects. The real world usefulness of these processes is a consequence of the use of the mathematical language to describe the real world.*

I have included in parentheses the phrase “collections of” to refer to claims which are broadly written, often in functional terms. It is possible that more than one mathematical algorithm will read on the same claim.

This allows us to explain with more precision the relationship between mathematical algorithm and field of use. This is a clarification of the observation that was previously made.

When the procedure is carried out for the sake of manipulating the symbols without regards to their semantics, say we want to see in action what happens when we do this particular manipulation, then the field of use is mathematics. In this case I suppose the algorithm would be mathematical in the legal sense of the word. Similarly if the symbols are manipulated for the sake of their mathematical meanings but without assigning a real world interpretation to the mathematical meanings, the field of use is also mathematics. But when the algorithm is given a practical utility because the symbols are given a real world interpretation, then the field of use is not mathematics and the algorithm is no longer mathematical in the legal sense of the word.

In all three cases the procedure as defined in terms of symbol manipulation is the same. If it is machine-implemented, the machine will be the same in all three cases. The difference lies in the semantics given to the symbols and not in the manipulation of the symbols. This is why legally mathematical algorithms are not a narrow group of procedures within a broader category of algorithms. All algorithms may be either mathematical or non mathematical according to the legal definition depending on how they are used.

I have been told that the law makes a distinction between pure and applied mathematics. I am told pure mathematics is unpatentable and applied mathematics is. Such a legal distinction doesn't take into consideration that mathematics is a language. Whether mathematics is pure or applied it is the same mathematics and the same computations. This has always been the case throughout the history of mathematics. This is a consequence of the previously mentioned phenomenon that algorithms are not changed when real world meanings are attached to the symbols or mathematical entities. The difference between pure and applied mathematics is whether or not the mathematical language is used with a real world semantics attached to it. If we take the view that the pure mathematical calculation is not a process in the sense of patent law then applied mathematics is not a new use of a process. It is at most a new mathematical description of the real world. I understand that the reality described by a mathematical description may be patentable but I don't see why the description itself should be patentable. This is granting exclusive rights to the use of the mathematical language. This should raise serious First Amendment concerns.

**Why Algorithms Are Abstract**

Here I am discussing algorithms according to the mathematician's definition.

Please see for reference Mitchelmore, Michael and White, Paul — *Abstraction in Mathematics and Mathematical Learning* (where you may download the PDF of the article)

These authors are university researchers specializing in the psychology of learning mathematics with a focus on the role of abstraction. They are authorities on the issue of what is an abstract idea when mathematics is concerned. Given that it may be difficult to define precisely the concept of “abstract idea” in the sense of *Bilski* and *Diehr* it may be worthwhile to seek some basis from authorities which have studied the question. This is especially important because the definition of algorithms is not the same in case law as it it is mathematics. The question arises of whether algorithms according to mathematics and computer science are abstract ideas. If the courts are unwilling to hold that *Benson* is applicable to this definition they will have to give it a fresh look.

These authors identify two main forms of abstractions: what they call *abstract-apart* and what they call *abstract-general*. The abstract-apart form of abstraction is about definitions of pure mathematics. The mathematical entities are considered for themselves, within the system where they are defined. The abstract-general concept is about how humans learn the mathematical abstractions empirically by observing them occurring in the real world. For example they may learn about angles by observing angular objects and they may learn about circles by observing round objects. Abstract ideas learned in this manner are abstract-general.

Here is how they describe the concept of abstract-apart:

> We claim that the essence of abstraction in mathematics is that mathematics is self-contained: An abstract mathematical object takes its meaning only from the system within which it is defined. Certainly abstraction in mathematics at all levels includes ignoring certain features and highlighting others, as Sierpinska emphasises. But it is crucial that the new objects be related to each other in a consistent system which can be operated on without reference to their previous meaning. Thus, self-containment is paramount.
>
> . . .
>
> To emphasise the special meaning of abstraction in mathematics, we shall say that mathematical objects are abstract-apart. Their meanings are defined within the world of mathematics, and they exist quite apart from any external reference.

Mathematical algorithms exhibit this form of abstraction. The algorithm is an abstract manipulation of symbols where their semantics has been abstracted away. The rules for carrying out the algorithm are defined in the world of abstract symbols. In this sense they stand in their own world, apart from the external references provided by their semantics.

How is this notion of abstraction compatible with the practical utility of algorithms? The same authors explain:

> Mathematics is used in predicting and controlling real objects and events, from calculating a shopping bill to sending rockets to Mars. How can an abstract-apart science be so practically useful?
>
> One aspect of the usefulness of mathematics is the facility with which calculations can be made: You do not need to exchange coins to calculate your shopping bill, and you can simulate a rocket journey without ever firing one. Increasingly powerful mathematical theories (not to mention the computer) have led to steady gains in efficiency and reliability.
>
> But calculational facility would be useless if the results did not predict reality. Predictions are successful to the extent that mathematics models appropriate aspects of reality, and whether they are appropriate can be validated by experience. In fact, one can go further and claim that the mathematics we know today has been developed (in preference to any other that might be imaginable) because it does model significant aspects of reality faithfully. As Devlin (1994) puts it:
>
> > How is it that the axiomatic method has been so successful in this way? The answer is, in large part, because the axioms do indeed capture meaningful and correct patterns. . . . There is nothing to prevent anyone from writing down some arbitrary list of postulates and proceeding to prove theorems from them. But the chance of those theorems having any practical application [is] slim indeed. (pp. 54-55)
>
> Many fundamental mathematical objects (especially the more elementary ones, such as numbers and their operations) clearly model reality. Later developments (such as combinatorics and differential equations) are built on these fundamental ideas and so also reflect reality even if indirectly. Hence all mathematics has some link back to reality.

This explanation corresponds to the notion that mathematics is a language used to describe reality. The practical utility of algorithms is of semantical nature and not of procedural nature.

I have been able to verify that the quote from (Devlin 1994) is from an earlier edition of the book identified as [Devlin 2000] in the references section of this article. I have located this quote on pages 74-75 of the newer edition.

These authors bring up the notion that abstract mathematical ideas are found in real life. The notion of circle maybe abstracted from our experience with round objects. The notion of angle is similarly abstracted from our experience with angular objects. This form of abstraction is what these authors call “abstract-general”. It doesn't describe the abstract idea per se. It describes the human experience by which the abstract-apart idea is learned. This is building the tie between the abstract mathematics and it usefulness in modeling reality. The abstract property of the idea per se is characterized by the concept of abstract-apart.

We may verify the notion that the procedure of an algorithm is separate from its semantics by looking at the various ways to use the procedure.

One possibility is is to use the algorithm as manipulations of meaningless symbols. All algorithms may be used in this manner. This is sometimes done by programmers for purposes of testing. For example the programmer may write something random like akhsf9aw where some text is expected to supply test data to its program. The point is to verify whether this text is processed as the algorithm is supposed to. A possible intended result may be that this information is rejected with an error message. It may also be that the information is stored in a database or forwarded to another program by means of a communication protocol. Other outcomes may also be intended depending on the program. In each case the algorithm is used in this test as an abstract manipulation of meaningless symbols to verify that the expected manipulation occurs. This use of an algorithm is useful in practical terms. It allows to verify that the program is error free and will behave correctly. This is part of the testing arsenal programmers may use to control the quality of their work and determine whether some corrective action is appropriate.

Another possibility is to apply the algorithm to fictional data corresponding to a made up situation which didn't arise from real life. Again this may be a test procedure for programmers, but it may also happen when users use the software to analyze hypothetical scenarios. Then the algorithm doesn't have a concrete real world utility because the data doesn't correspond to anything concrete in real life. Fictional scenarios are abstract because they too are abstract-apart from reality.

This makes three usage scenarios for the same algorithm. It may be a procedure operating on meaningless symbols. It may be a procedure operating on fictional data. It may be a procedure operating on data which have a concrete real world meaning. In all three scenarios the procedure is the same. In all three scenarios the procedure is a computation which is a different entity from the circuit used to implement it. In all three scenarios the same range of very different circuits may be used to implement the same algorithm. The difference between the scenarios is purely a matter of the semantics given to the symbols. This semantics depends on the intent and purposes of the user and not on the physical layout of the circuit or on the steps of the procedure. This is why an algorithm in the mathematician's and computer scientist's meaning of the word is an abstract idea.

**Why Do We Have a Legal Definition of Algorithm?**

This is a question for the legally skilled readers. Why wouldn't the term of art in mathematics be used for purposes of interpreting patent case law such as *Benson*, *Flook* and *Diehr*?

I ask because it happens that the Federal Circuit has given up trying to understand the legal meaning of “algorithm”. In *in re Warderman* they ruled: (links in the original)

> 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.

The legal definition of algorithm is ambiguous, arbitrary and unnecessary legally speaking according to the Federal Circuit.

Another reason I ask is the context where *Benson*'s original definition appears. Here the relevant extract of the *Benson* case. For your convenience, I have highlighted the sentence of the definition in bold characters.

> A digital computer, as distinguished from an analog computer, operates on data expressed in digits, solving a problem by doing arithmetic as a person would do it by head and hand. Some of the digits are stored as components of the computer. Others are introduced into the computer in a form which it is designed to recognize. The computer operates then upon both new and previously stored data. The general-purpose computer is designed to perform operations under many different programs.
>
> The representation of numbers may be in the form of a time series of electrical impulses, magnetized spots on the surface of tapes, drums, or discs, charged spots on cathode-ray tube screens, the presence or absence of punched holes on paper cards, or other devices. The method or program is a sequence of coded instructions for a digital computer.
>
> The patent sought is on a method of programming a general-purpose digital computer to convert signals from binary-coded decimal form into pure binary form. **A procedure for solving a given type of mathematical problem is known as an "algorithm."** The procedures set forth in the present claims are of that kind; that is to say, they are a generalized formulation for programs to solve mathematical problems of converting one form of numerical representation to another. From the generic formulation, programs may be developed as specific applications.
>
> The decimal system uses as digits the 10 symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The value represented by any digit depends, as it does in any positional system of notation, both on its individual value and on its relative position in the numeral. Decimal numerals are written by placing digits in the appropriate positions or columns of the numerical sequence, *i. e.,* "unit" (10[0]), "tens" (10[1]), "hundreds" (10[2]), "thousands" (10[3]), etc. Accordingly, the numeral 1492 signifies (1X10[3]) + (4X10[2]) + (9X10[1]) + (2X10[0]).
>
> The pure binary system of positional notation uses two symbols as digits—0 and 1, placed in a numerical sequence with values based on consecutively ascending powers of 2. In pure binary notation, what would be the tens position is the twos position; what would be hundreds position is the fours position; what would be the thousands position is the eights. Any decimal number from 0 to 10 can be represented in the binary system with four digits or positions as indicated in the following table.
>
> [Table is omitted by PolR]
>
> The BCD system using decimal numerals replaces the character for each component decimal digit in the decimal numeral with the corresponding four-digit binary numeral, shown in the righthand column of the table. Thus decimal 53 is represented as 0101 0011 in BCD, because decimal 5 is equal to binary 0101 and decimal 3 is equivalent to binary 0011. In pure binary notation, however, decimal 53 equals binary 110101. The conversion of BCD numerals to pure binary numerals can be done mentally through use of the foregoing table. The method sought to be patented varies the ordinary arithmetic steps a human would use by changing the order of the steps, changing the symbolism for writing the multiplier used in some steps, and by taking subtotals after each successive operation. The mathematical procedures can be carried out in existing computers long in use, no new machinery being necessary. And, as noted, they can also be performed without a computer.

This series of six paragraphs is the part of *Benson* where the Court gives a summary of the facts of the case. As you can see the “definition” is the second sentence of the third paragraph. Why would the Supreme Court introduce without justification the definition of a new, unnecessary, arbitrary and ambiguous legal term in the middle of an enumeration of the facts of the *Benson* case? Why wouldn't the Supreme Court simply refer to a fact involving the term of art in mathematics? When

read in this manner the sentence of the Supreme Court is correctly stating a fact of mathematics.[40]

Let's suppose that the legal definition is indeed the appropriate one when applying *Benson*. This doesn't change the fact that there is a term of art in mathematics. This doesn't change the fact that algorithms according to the mathematicians' definition are mathematical abstract ideas. We should be able to make a case that these algorithms are unpatentable based on the principles enunciated by the Supreme Court in *Diehr* and *Bilski* and by the Federal Circuit in *Warderman* without having to refer to *Benson.* In such a scenario we may get a precedent which refers to the mathematical term of art but is otherwise identical in its effects to *Benson.* Then what is the point of keeping the legal definition around?

I strongly believe that the courts should understand and use the definition of the term of art in mathematics because the legal definition of “algorithm” is unsuitable to understand how software functionality is implemented. The correct definition for this purpose is the term of art in mathematics and computer science. This legal definition is a legal concept unrelated to the principles of mathematics and computer science.

Without an accurate understanding of the principles of computer science how do we determine when functionality results from the action of a machine? We have seen with the Lodsys patent claim how easily functionality which does not result from the action of a machine may be incorrectly attributed to a machine. This may lead to errors such as incorrectly finding defendants liable for infringing on machine patents when they didn't implement the functionality by means of a machine. More examples of how this may happen will be found later in this article.

**The Mathematical Concept of Universal Algorithm**

The reader may have noticed that the procedures used to exhibit the formulas for computer algorithms have an interesting feature. One series of formulas, always the same, is used for all computations. If we use the definition of Standard ML, the formulas found in the language definition are used for all Standard ML programs no matter how written. If we use some definition for the computation carried out by a computer circuit, the formulas from this definition will apply to all programs this computer may run no matter how they are written.

These two circumstances are examples of universal algorithms. Some algorithms have a *universal* property. In this context universal means that the algorithm is very broadly generic. It can reproduce the behavior of every possible algorithm when given input to this effect. The consequence is that a universal algorithm may be the only algorithm you will ever need. It can carry out all possible computations. This is a very important concept because it plays a key role in the design of computer hardware. A device implementing a universal algorithm is a universal computing device, a general purpose computer. But like symbols and their meanings, the notion of universal algorithm is absent from case law on how the functionality of software is produced. This is a major error.

Universal algorithms are known in the mathematical literature since the mid-1930s. In theory there are infinitely many of them. The previous article contains a few sections on universal algorithms with references to the mathematical and computer science literature for evidence.[41]

Perhaps non mathematicians will better appreciate this concept with the help of an analogy with music devices. A musical box is a mechanical device which typically can play only one tune which is built into the hardware. A player piano would typically be able to play any tune provided a piano roll is provided for the tune. A specific algorithm is like the musical box. It can carry out only one computation like the musical box can play only one tune. A universal algorithm is like the player piano. It can carry out any computation provided it is given appropriate directions in its input like a player piano can play any tune provided it is given the corresponding piano roll.

Another explanatory metaphor might be this one. You are told by your boss to await further instructions. Then, later, you receive some mail with the instructions and you do as you are told. Several (but not all) universal algorithms are like this. Mathematicians have found ways to define

in mathematical language procedures which receive instructions and do as the instructions say. Many of these procedures are universal algorithms.

Let's flesh out how this works.

- We need some set of predefined instructions called the “instruction set”. These instructions are the basic building blocks to write certain algorithms but with a pair of requirements: a) all the permitted algorithms are sequences of instructions and b) the only allowable instructions are those in the instruction set.

- Each instruction in the instruction set is an operation on symbols in the sense that its execution reads symbols and/or writes symbols.

- Each instruction has a written representation in symbols according to some convention. This is the encoding which is required for our universal procedure to be an algorithm.

- Each instruction must be of the kind which may be executed mechanically, like robots, otherwise they are unsuitable to describe how to compute an algorithm.

- When presented with a written list of such instructions the procedure is to execute them one by one. When you are done with one instruction you move to the next one in the list. If the instruction you are executing says that some specific instruction in the list is the next one, you go to this instruction next. If the instruction is that you must stop computing because you are done, you stop.

The procedure I have just described meets the requirements for being an algorithm. Among other things it operates on symbols and must be executed mechanically, like robots. It can execute any computation provided it is described with a list of instructions chosen from the instruction set. This list is called a *program*. The key for this procedure to be a universal algorithm is to have a comprehensive instruction set that will let the programmer express every procedure he may ever need by means of some combination of the permissible instructions. Whether or not the instruction set is comprehensive enough to permit the computation of all computable functions is a mathematical feature of the instruction set.[42]

An engineer wishing to build a circuit to carry out a computation pursuant to an algorithm has at least three main alternatives. There may be more options which escape me.

The first option is to build a circuit which is specific to his algorithm. For example if the algorithm is addition the engineer may want to implement an adder circuit which can do only addition. Nowadays these circuits are often called Application Specific Integrated Circuit or ASIC. In the music device analogy, these circuits are comparable to musical boxes which may play only one tune.

The second option is to build a flexible circuit which may be configured in a diversity of ways to enable the computation of several algorithms. Then the engineer configures the circuit according to his chosen algorithm. For example the early ENIAC was a computer built in this manner.[43] One needed to manually configure circuits on a patch panel to program this computer. In the music device analogy, these circuits are comparable to musical boxes one may reconfigure to change the tune. Each reconfiguration in effect makes a new musical box.

The third option is to build a circuit for some universal algorithm and then use the universal property of the algorithm by supplying the program as input. This is how modern digital computers work. Programmers write their programs in a human readable programming language and then they use tools such as compilers which translate the programming language into the machine language used to represent the instructions from the computer instruction set. It is this machine language which is the input for the computer universal algorithm. In the music device analogy, these circuits are comparable to player pianos and the machine language programs are comparable to piano rolls.

Nowadays early ENIAC-like configurable circuits are considered obsolete. Circuits for universal algorithms are the preferred solution.

It is important to understand that this concept of “universal algorithm” is a fact of mathematics, like the Pythagoras theorem, which is documented in mathematical literature.[44] It is not a legal conclusion.

**How Software Functionality Is Implemented – Hardware Perspective**

You must have noticed how this notion of universal algorithm provides further details on how programmers implement functionality by means of software.

1. They define a data representation with a syntax and a semantics and,
2. they define a procedure, an algorithm, for manipulating the data and,
3. they translate this algorithm into data to be given to the universal algorithm built into their computer and,
4. they supply these instructions as input to the computer.

This is an incomplete explanation. There is further complexity in that a universal algorithm may be implemented in software. A programmer has the choice between compiling his program into instructions recognized by hardware or producing data as input to a software universal algorithm. If he goes the software route he may use any universal algorithm known to mankind. This will be explained in more details in the article.

This story is very different from what is used in patent law circles. The usual legal explanation is that the computer is a useless device absent its program. The instructions “configure” the computer and this makes it useful. Software is patentable because the configuration of a machine to enable a new use of a machine is patentable.

A legal view of computer programming is explained in footnote 29 of *in re Prater*: (emphasis in the original)

> 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.

According to this explanation the computer is supposed to have changed electrically into a specific circuit once it has been programed. This is contrary to the explanation from computer science which states that a computer is already a circuit which computes a universal algorithm and no further making of a new specific circuit is necessary to carry out the computation. What is required to carry out a specific computation is input of data.

This is a disagreement on the facts. How can we tell which explanation is correct? One way may be to rely on expertise. Textbooks of computer science are written by specialists in the field. With all due respect to the courts, experts in computer science should know better on these factual matters.

Another way to tell the two explanations apart may be to setup experiments which should give one result which should give one result if the universal circuit is correct and another result when the specific circuit explanation is correct. This method has the advantage of being the most intelligible to laymen. They may not understand all the technical details but they can see what

happens when the experiments are done. Later in the article I will explain how to conduct such experiments.

For now let's consider still another method of telling the two explanations apart. We look into how computers work. If we find an electrical change which makes a specific circuit then *Prater* is right. If we find a universal algorithm being computed absent the creation of a specific circuit then the universal algorithm explanation is correct.

When we will be through this analysis we will have achieved something more than finding out who is right from *Prater* and computer science. We will have a still more complete description of how functionality is implemented from the hardware perspective. This knowledge is valuable for its own sake. It may be used in every context where it is relevant to understand how functionality is implemented.

**The Operating Principles of a Modern Digital Computer**

The description of computer programming in *Prater* is certainly applicable to the early ENIAC computer which was programmed by manually configuring wires in a plug board. This computer had several hardware modules which could be seen as a collection (a storeroom in *Prater's* description) of parts which becomes a specific electrical circuit once the program has been introduced. Manually plugging wires in a plug board makes a new electrical circuit.

Modern computers are of a different design called the stored program architecture also known as von Neumann architecture. This design implements a universal algorithm akin to what mathematicians call a random access stored program machine, or RASP.

This type of computers have two main components: the memory and the processor.[45] The memory is the place where bits are written. This component plays a role that is somewhat analogous to the pad of paper in a pencil and paper computation. The computer constantly reads and write bits in memory as it computes like a human reads and writes symbols on paper. The processor, also called the CPU, is somewhat analogous to the human moving the pencil. The processor is the component which reads and writes the bits in memory in order to carry out the computation.

Each make and brand of processor has the capacity to recognize and execute a limited number of instructions called the instruction set of the processor. The programmer programs the computer by selecting an appropriate sequence of instructions. He is not free to issue to the computer any instruction he pleases. The instructions must all be chosen from the instruction set because they are the only instructions the processor has the ability to execute. The instructions are represented by sequences of bits which must be stored in the memory. The task of specifying and storing the appropriate bits in memory is called programing the computer and the bits are the program.

In order to execute the program the computer must be instructed of where in memory is located the first instruction of the program. Then the processor reads and executes the instructions one by one in a sequential manner. This sequential execution of instructions is called the instruction cycle. Which instructions are in the instruction set and how they are represented with bits is defined in the instruction set architecture for this particular processor. Together the instruction cycle and the instruction set architecture are the universal algorithm which is built into the computer. For simplicity of nomenclature, hereafter I refer to this universal algorithm as the instruction cycle even though, properly speaking, the instruction set architecture should also be mentioned.

The instructions as well as the data they manipulate are represented by symbols, the bits. This is one of the requirements for an algorithm to be an algorithm. The instructions as well as the instruction cycle are carried out in a mechanical manner, like robots, by virtue of being implemented in circuitry. This is another requirement for an algorithm to be an algorithm.[46] The fact that this is an algorithm is verified by the theorem which proves that the corresponding model of computation, the RASP, is equivalent to a Turing machine. The instruction cycle may carry all the possible computations provided it is given the bits for the corresponding instructions as input. This makes it a universal algorithm.

We have just seen that the examination of a stored program computer reveals the universal algorithm which carries out the computations. This is further verified by the procedure used to define the mathematical formula for the circuit. The PDF exhibit which has been mentioned previously in this article contains a method for defining in the language of lambda calculus the instruction set and instruction cycle of a specific computer. This method also includes how to spell out in the formulas the interaction with the input and output peripherals as well as various form of multiprocessing and networked communications. When this method is followed to completion we have written in mathematical language the formulas for the universal algorithm of this specific computer. The knowledge of the formulas guarantees that the algorithm is indeed a mathematical universal algorithm in the sense of computation theory.

At this point we are also able to verify that the real world semantics of the data is not an electrical property of the circuit. The instruction set of computers is documented in its totality in reference manuals.[47] An inspection of the manuals reveals that the instructions are all defined in terms of operations of boolean algebra and arithmetic. They contain no reference to the real world semantics of the bits. This is the evidence that, like a pocket calculator which does arithmetic without knowledge of the real world interpretation of the numbers, the computer executes instructions as abstract operations on bits without knowledge of the real world meanings of the bits. This shows that software is abstract in exactly the same manner as mathematical algorithms are abstract.[48] This is expected. A computer program is a mathematical algorithm.

Let's continue this tour of the operating principles of computers with an examination of how they execute instructions. This information will be useful later.

The Art of Assembly Programming has an explanation of how a stored program architecture computer relate to an early ENIAC.

> One big problem with the patch panel programming method is that the number of program steps (machine instructions) is limited by the number of rows of sockets available on the machine. John Von Neumann and others recognized a relationship between the sockets on the patch panel and bits in memory; they figured they could store the binary equivalents of a machine program in main memory and fetch each program from memory, load it into a special decoding register that connected directly to the instruction decoding circuitry of the CPU.

This makes clear that programs are bits. They are data. Where the early ENIAC has a plug board with manually connected wires the stored program architecture computer has bits in memory. The bits will be eventually decoded and acted upon by the CPU circuitry. How does this work? This is done by the instruction cycle. The instructions are executed one after another in a loop.

The processor reads from memory the bits for one instruction. Then the processor decodes these bits to find out which operation must be executed. Then the processor does as the instruction says. When the processor is done with the instruction it reads the next one and the cycle repeats. Instructions are executed in this manner one after another in a sequential fashion.

When executing instructions, there is a task for the processor to configure itself in such manner as to be able to do as the instruction says. This occurs right after the processor has decoded the bits and has acquired the knowledge of what the instruction entails. This reconfiguration works in a manner similar to an early ENIAC. It sets up its internal circuitry, connecting the components as needed for executing the instruction. This lasts only for the time required to execute this particular instruction. Once the instruction is done the configuration has outlived its usefulness.[49]

While there is a similarity, there is also a fundamental difference between this activity of the processor and the programming of an early ENIAC. The manual configuration of an ENIAC covered the entire program because it is the programming of the computer. In a stored program computer the internal configuration of the processor is for a single instruction because it is a single step in the execution of the instruction cycle. Programming a computer is not the same thing as executing

a program, especially when the configuration is limited to a single instruction.

There is also a fundamental difference in technology. In the ENIAC programming physical wires must be connected in a plug board for every program. This is a physical change to the machine structure. In a digital computer all the physical connections between transistors in a processor are etched in the factory and never changed afterwards. The processor “configures” itself by applying different voltages to these pathways between transistors. No structural change to the layout of the circuit physically occurs. The only changes are new voltages being applied to existing portions of the same physical layout.

This shows that the internal configuration of the processor cannot be a specific circuit implementing the computer program. The same processor constantly configures and reconfigures itself instruction per instruction. No configuration stays stable for the whole duration of the program. Besides what is called “configuration” in this context is not making a new circuit in the usual sense of the word “circuit”. It is applying new voltages to an existing circuit.

If there is a specific circuit corresponding to the program its configuration must be elsewhere. Where could that be? This leaves us with the bits in memory. But memory too doesn't stay still for the duration of a program. Instructions constantly write data in memory at a rate ranging from thousands to billions of times per second depending on the speed of the computer.

All of this shows that the unprogramed computer is a device in working order which has already a process by which it operates. This process is the instruction cycle. An unprogramed computer is not an accumulation of disjointed parts which needs to be configured in order to become a device in working order. In such circumstances one may wonder why going out of our way to declare the changes in a computer to be changes in machine structure. These changes are the execution of the instruction cycle. They are the moving parts of the computer doing their work. No new structure is created when a machine does nothing more than execute the process by which it operates.

**The Structure of a New Machine According to the Federal Circuit**

Let's take a look at the flip side of the argument. What would be the machine structure according to case law? How does this match with actual programing practice?

The Federal Circuit has explained what is the structure of a machine resulting from inserting a program in a programmable computer in *WMS Gaming. Inc. v. International Game Technology*. (links and emphasis in the original)

> The structure of a microprocessor programmed to carry out an algorithm is limited by the disclosed algorithm. A general purpose computer, or microprocessor, programmed to carry out an algorithm creates "a new machine, because a general purpose computer in effect becomes a special purpose computer once it is programmed to perform particular functions pursuant to instructions from program software." *In re Alappat,* 33 F.3d 1526, 1545, 31 USPQ2d 1545, 1558 (Fed.Cir. 1994) (en banc); *see In re Bernhart,* 57 C.C.P.A. 737, 417 F.2d 1395, 1399-1400, 163 USPQ 611, 615-16 (CCPA 1969) ("[I]f a machine is programmed in a certain new and unobvious way, it is physically different from the machine without that program; its memory elements are differently arranged."). The instructions of the software program that carry out the algorithm electrically change the general purpose computer by creating electrical paths within the device. These electrical paths create a special purpose machine for carrying out the particular algorithm.[3]
>
> [PolR: This refers to footnote 3 which is quoted below]
>
> [3] A microprocessor contains a myriad of interconnected transistors that operate as electronic switches. *See* Neil Randall, *Dissecting the Heart of Your Computer,* PC Magazine, June 9, 1998, at 254-55. The instructions of the software program cause the switches to either open or close. *See id.* The opening and closing of the interconnected switches

> creates electrical paths in the microprocessor that cause it to perform the desired function of the instructions that carry out the algorithm. *See id.*

This case is from 1999. It has been reaffirmed in 2008 in *Aristocrat Technologies Australia Pty Ltd v. International Game Technologies* and in 2011 in *in re Katz*. *WMS Gaming* is a fascinating case for the insights it provides on how the Federal Circuit is viewing software. It is clear that they see the programming of the computer as literally making a new machine by creating a new electrical circuit.

Let me open a parentheses. Sometimes some patent attorneys bring up a theory that functionality legally defines the machine. How does that fit with these cases? In *Aristocrat* the Federal Circuit bring more details. They explicitly say they make a difference between the machine functionality and the machine structure. While it is OK in some circumstances to describe the machine in functional terms there are other circumstances where the knowledge of the structure is required.

> Whether the disclosure would enable one of ordinary skill in the art to make and use the invention is not at issue here. Instead, the pertinent question in this case is whether Aristocrat's patent discloses structure that is used to perform the claimed function. Enablement of a device requires only the disclosure of sufficient information so that a person of ordinary skill in the art could make and use the device. A section 112 paragraph 6 disclosure, however, serves the very different purpose of limiting the scope of the claim to the particular structure disclosed, together with equivalents. The difference between the two is made dear by an exchange at oral argument. In response to a question from the court, Aristocrat's counsel contended that, in light of the breadth of the disclosure in the specification, any microprocessor, regardless of how it was programmed, would infringe claim 1 if it performed the claimed functions recited in the means-plus-function limitations of that claim. That response reveals that Aristocrat is in essence arguing for pure functional claiming as long as the function is performed by a general purpose computer. This court's cases flatly reject that position.

This closes this parentheses.

Please consider what the Federal Circuit says in *WMS Gaming* about the myriad of interconnected transistors. An algorithm is part of the machine structure because, they say, “The instructions of the software program that carry out the algorithm electrically change the general purpose computer by creating electrical paths within the device.” This is an incorrect understanding of how instructions are processed. Such a change is not making a new machine structure for the reasons we have just discussed.

These electrical paths are transient. They last the time to execute one instruction and are immediately torn down and replaced by the paths for executing the next instruction. This is assuming we incorrectly assimilate changes in the voltages applied to existing paths to the making of new electrical paths. These changes are the execution of the instruction cycle. They are the process by which an existing machine operates to carry out a computation pursuant to a universal algorithm. The general purpose character of the computer doesn't result from electrical changes induced by instructions. It results from the mathematical properties of the universal algorithm.

Instructions in memory too may be transient. They usually are not transient because it is good programming practice to leave the instructions unchanged for the entire execution of the program. This is a choice of the programmers. This is not an obligation of technology. If word is out that one may reduce the risk of software patent infringement when the instructions are transient because the structure of a new machine cannot be made in such circumstances, programmers will be able to use this in actual programming. It is well within the capabilities of computers to run such programs.[50]

There is another reason *WMS Gaming* is incorrect and it is they misunderstand the role of transistors in implementing symbols. All bits stored in memory have the capability to make transistors turned on and off in the processor when the computation proceeds. This is not the

exclusivity of instructions. But whether it results from instructions or non instruction data this transistor activity is always carrying out the computation according to the instruction cycle.

When a transistor, acting as a switch, is turned on it means the symbol 1. When it is turned off it means 0. The pathways created in this manner are the manipulation of symbols.[51] For example an adder circuit[52] will produce the bits for the number 146 when given as inputs the bits for the numbers 42 and 102. But internally to the adder transistors will turn on and off creating and destroying pathways until the answer is produced. This kind of activity is cannot be the making of machine structure which correspond to functionality of addition because it is very transient. This is actually carrying out the computation on specific numbers and the electronics is such that different numbers will create different pathways.

When building a processor using the methods described in [Patterson 2009] chapter 4 we find that the pathways set by transistors are not configuring a circuit for the entire algorithm or even the entire instructions. They are the kind of transient pathways which do the actual calculation.

We find an idea similar to *WMS Gaming* in the amicus brief of Microsoft, Phillips and Symantec submitted to the Supreme Court in *Bilski*:[53]

> Purporting to analyze the patent-eligibility of software, as opposed to that of hardware, relies on an illusory distinction. The functionality of any digital device is the product of the same transistor activity, and it is the configuration of the pathways between those transistors that dictates their functionality. Like all patent-eligible processes, computer-implemented processes combine physical activity with human-directed logic. Irrespective of whether a particular configuration of transistors is accomplished using a soldering iron or by means of software, the processes conducted by these transistors are ultimately physical processes.

A complete reading will reveal that the notions of universal algorithm and stored program computers are absent from this brief. These authors explicitly assimilate the programming of modern computers with the configuration of physical pathways between transistors similar to the configuration of an early ENIAC patch panel.[54] As we have seen, this is incorrect. Modern computers are not programmed in this manner. But it is useful to know that this theory has been presented to the Supreme Court of the United States without a rebuttal.

I wonder what the Federal Circuit would say if someone brought the Bochs software to their attention. From the Bochs user manual:

> Bochs is a program that simulates a complete Intel x86 computer. It can be configured to act like a 386, 486, Pentium, Pentium II, Pentium III, Pentium 4 or even like x86-64 CPU, including optional MMX, SSEx and 3DNow! instructions. Bochs interprets every instruction from power-up to reboot, and has device models for all of the standard PC peripherals: keyboard, mouse, VGA card/monitor, disks, timer chips, network card, etc. Because Bochs simulates the whole PC environment, the software running in the simulation "believes" it is running on a real machine. This approach allows Bochs to run a wide variety of software with no modification, include most popular x86 operating systems: Windows 95/98/NT/2000/XP and Vista, all Linux flavors, all BSD flavors, and more.
>
> Bochs is written in the C++ programming language, and is designed to run on many different host platforms, including x86, PPC, Alpha, Sun, and MIPS. No matter what the host platform is, Bochs still simulates x86 hardware. In other words, it does not depend on the native instructions of the host machine at all. This is both a strength and a weakness, and it's the major difference between Bochs and many other x86 emulation software such as plex86, VMware, etc. Because Bochs uses software simulation for every single x86 instruction, it can simulate a Windows application on an Alpha or Sun workstation.

The difference between machine instructions and other forms of data which may be stored in memory is that the instructions are part of the instruction set of the computer. This is something

specific to the make and brand of the computer. In particular the instruction set of an x86 processor is different from the instruction set of a SPARC processor. If we assume that computers work according to the principles set forth in *WMS Gaming* and we store instructions for the x86 on a SPARC-based machine, say a Sun workstation, they won't modify the pathways between the transistors in the SPARC processor because they are not SPARC instructions. The x86 instructions are ordinary data on a Sun workstation.

According to the principles set forth in *WMS Gaming* loading the Bochs program on a Sun workstation changes the machine structure to implement the Bochs algorithm. This device can't execute x86 instructions natively because it has a SPARC processor. But once Bochs is stored in memory the x86 instruction cycle and instruction set become part of the Sun workstation structure.

If the instruction cycle of an x86 processor could be implemented as the structure of a Sun workstation this same algorithm must part of the structure of a native x86 processor. In both cases it is the same instruction set and the same instruction cycle. How does this work with the legal notion that an unprogramed computer is useless circuitry which needs hardware configuration by means of programing to become useful? It already has the structure for running a universal algorithm which can carry out all possible computations provided it is given appropriate data as input.

The notion that the process by which the machine execute is different from the instruction cycle is not tenable because programing a stored program architecture computer does not stop this computer from executing the instruction cycle. All computer algorithms are implemented indirectly through the execution of the instruction cycle. They are never implemented directly through the reconfiguration of pathways between electrical components.

**How Programmers Implement Functionality Without Making Machine Structure**

There is another way to refute the notion that programming a computer makes the computer structurally different.[55] There are a number of programming techniques which do not result into the machine structure described by the Federal Circuit in *WMS Gaming*. When these techniques are used software functionality is implemented without storing in memory corresponding machine instructions. In such case the algorithm is implemented but the structure described in *WMS Gaming* is not there. The interest of this refutation is that it does not attempt to invalidate any case law. This could be handy to a lawyer who can't or won't try to get existing law overruled. This refutation takes the Federal Circuit at their word and argue that in the case of this particular implementation no new machine has been made because of the programming techniques which have been used.

The reader already knows two programming techniques which give this result. The first one is defining the data. As was previously explained there is no instruction to assign a real world meaning to the data. There is no machine structure corresponding to the real world meanings of data. The other technique is Bochs. A program for the x86 is not translated into SPARC instructions. It is given as input to Bochs. Theres is no machine structure corresponding to a x86 program executed in this manner.

Think of the Lodsys patent claim discussed previously. The words “What do you think of this program?” or “What do you think of Barack Obama?” are not part of the machine structure as set forth in *WMS Gaming* even though they are displayed to the user because they are not instructions. They are data. But the choice of words makes the difference between a program eliciting customer feed-back about a product and a program for political opinion polling. This element of functionality has no corresponding structure in the machine.

Another frequent situation arises from programming languages which are not compiled into machine instructions. This is possible because universal algorithms may be implemented in software. Any universal algorithm could be the target of a programming language. Language designers do not need to target the instruction cycle of a physical computer.

For example there are several languages which are implemented with bytecode interpreters. An example is Python. These programming languages are translated into instructions from a special instruction set called a bytecode. This instruction set is not implemented in the processor hardware. It is implemented by a computer program called a bytecode interpreter. Even if we assume that the principles set forth in *WMS Gaming* on the effect of instructions on a processor are correct, these bytecode instructions will not result into the configuration of pathways in the processor because they are not machine instructions.

If we apply *WMS Gaming* to a Python program, or a similar language, we find the machine structure must be the bytecode interpreter because this is the only part which is machine executable instructions. By this standard the structure is the same for all Python programs regardless of functionality. This is normal because the bytecode interpreter is a universal algorithm in software, like Bochs. The bytecode which provides the specific functionality is not structure because it is not machine executable instructions.

This sort of situations is not restricted to universal algorithms. There are many algorithms which replicate the computations of other algorithms. Some of these algorithms are not universal because they replicate a narrow range of algorithms instead of replicating them all. But they still implement functionality without making the structure described in *WMS Gaming*.

For example consider a text search. The programmer wants to search all occurrences of words longer than four characters ending with the lowercase letter “s” and not including the letter “a”. One possibility is to write some code, some instructions, which will do exactly this. Another way is to use regular expressions.[56] This is a generic algorithm for doing a broad range of text searches based on a controlling character string called a regular expression. This algorithm is not universal because it is restricted to text searches. But it has the ability to implement the functionality of more specific algorithms for text searches.

The programmer uses the generic procedure and the exact search criteria is determined by some data input to this procedure. In this example the input may be “[^a][^a][^a][^a]+s”. This gobbledygook-looking string of characters is not machine instructions. It is the data which, when given as input to the regular expression procedure, will result into the text search mentioned above. In term of machine structure, what do we have here? If the programmer writes the software which is specific to his search, then this software is the machine structure as set forth in *WMS Gaming.* But if he uses the regular expression, then the machine structure is the regular expression procedure because this is what is implemented with instructions. This structure doesn't have the functionality of doing the required search by itself. The combination of the procedure with the mentioned input string does. But the input string is not machine structure because it is not machine instructions as required by *WMS Gaming*.

To see this point more clearly please consider another text search. We are looking for all series of characters which start with a capital letter and end with a period. Again the programmer may write some dedicated software to do this. Then the machine structure will be different from the other text search because the instructions are different. But if the programmer uses regular expressions, the same generic procedure will be used but with a different input. Now it is “[A-Z].*\.” which will do the trick. In this scenario we use the exact same machine structure as for the other search but with a different data input.

What if someone convinces the court to expand the legal definition of machine structure to include arbitrary data instead of just instructions. I don't know that would be the justification of such argument but let's contemplate this scenario for the sake of analysis. Then even with this expanded definition it is still possible to implement software functionality without making new machine structure. There are programming techniques which will achieve this result.

Data definition is assigning meanings to bits. This is never machine structure for the same reason the meaning of text is not part of the structure of a book. Semantics is not a physical property of objects.

Other ways to implement functionality with data without making some corresponding structure occur when the data is transient. Data cannot be machine structure for implementing functionality when it is constantly changed during the course of the computation. Whatever machine may have been made is too short-lived to implement all the elements of the claimed functionality.

For example, it is easier to write self-modifying code with bytecode interpreters than it is with hardware instructions. The programming technique of metaprogramming consists of having a computer program write software. Elements of functionality may be generated on the fly by a program which writes the corresponding code at the moment it is required and then this code is discarded when it is no longer used. In such circumstances the bytecode is transient.

Other examples are the family of of functional programming languages. These languages are based on the model of computation called lambda-calculus. This model is very different from the RASP. It does not rely on an instruction cycle. Its principle of operation is different. This is why there is no obligation to compile a functional programming language into instructions as we do with the more familiar imperative programming languages.

Think of high school algebra. We have an equation, or a series of equations, which must be solved. Then we rewrote the equations a different manner, applying some transformation permitted by the rules of algebra at every step. We kept rewriting the equations like this until we reached an answer. This sort of procedure is a term rewrite system. A lot of different problems may be solved by means of rewriting terms with a set of rules like algebra. The difference between a problem and another is the initial text which will be rewritten. The rules for rewriting the text remain the same.

Lambda-calculus is a term rewrite system which is so generic as to qualify as a universal algorithm. It is mentioned by name in the Church-Turing thesis as one of the equivalents of Turing machines. It is part of the foundations of computation theory. Because lambda-calculus is a term rewrite system a computation in lambda-calculus progresses by constantly rewriting the data. In a system like this a program is not instructions. It is the initial data which will be rewritten.

It is possible to implement a functional programming language with a generic term rewrite procedure which constantly rewrites information in memory according to the rules of lambda-calculus.[57] If we apply *WMS Gaming* on this system we will find that the structure is the generic procedure for rewriting the data. This is the same procedure for all programs therefore the structure will be the same for all programs. But the data being rewritten cannot be structure because it is transient. It is constantly modified during the execution of the computation.

**Functionality, Data and Instructions**

*WMS Gaming* and the precedents it invokes are standing for the notion that software functionality is implemented by instructions, and that inserting instructions in a computer change its structure. We have seen this notion is false in multiple ways.

More generally, the distinction between instructions and non instruction data doesn't have that much significance. Instructions is the data the instruction cycle will accept to particularize its behavior to match a specific algorithm. It has some hardware significance only to the extent the instruction cycle is implemented in hardware. But as programs such as Bochs show, instruction cycles may be implemented in software. Then instructions no longer have any particular hardware significance. Purporting to distinguish between data types on the basis of their relationship to hardware is relying on an illusory distinction.

There is no physical difference between the effect of instruction and non instruction data on the processor. Both type of information is used during the execution of the computation. The pathways between transistors will be configured and torn down on the basis of both types of information.[58] In both cases this transistor activity is the execution of the program. It is not the programming of the computer.

There is no physical difference between loading instruction and non instruction data in computer memory. In both cases this is storing bits in an electronic component meant to record bits. In both cases storing the bits in memory doesn't have any effect on the processor until the data is used for executing an instruction.

I have encountered people arguing that instructions and non instruction data are different in terms of functionality. I wonder how functionality is defined in this context. Is it machine functionality in the sense of the machine structure according to *WMS Gaming*? Or is it some subjective distinction where the use of some data is called functionality and other data does not have such designation?

Even if we take a subjective view of functionality there is no difference between instruction and non instruction data in terms of ability to implement functionality. We have seen that functionality may be implemented by means of non instruction data. Examples are the regular expressions and some implementations of functional programming languages. We are now going to see the converse is equally true. Data which does not describe functionality in a subjective sense may be represented by instructions. The consequence is that both types of data may implement functionality and both types of data may represent information which doesn't determine functionality. Purporting to distinguish between instructions and non instruction data on the basis of their relationship with functionality is relying on an illusory distinction.

This conclusion is logical when one considers that much of the functionality of software depends on the semantics of symbols. The notion that software is ultimately a physical process is erroneous because it ignores both the role of symbols and the mathematical properties of the instruction cycle.

Consider a claim like “A computer system comprising a computer and a printer configured to print William Shakespeare's *Hamlet*”. What is the functionality of printing *Hamlet*? Is this a function of the instructions executed by the computer? Or is this a function of defining the data? Most people would only consider the scenario where the printing program just copy the text to the printer. A copy is a copy and the instructions don't depend on contents. In this scenario the functionality results from defining the data. Or alternatively, one may say the functionality is printing and *Hamlet* is just non functional data this functionality is acting on. This would be the normal subjective interpretation of functionality.

How about this method: (see the text of Hamlet)

1. Print the text “Act 1, Scene 1”
2. On the next line, print the text “SCENE I. Elsinore. A platform before the castle. ”
3. On the next line, print the text “FRANCISCO at his post. Enter to him BERNARDO ”
4. etc for all the lines of text in the play

A computer programed like this will print the play. This program is a series of print statements as they are commonly found in most programming languages. Each print statement is a series of computer instructions dedicated to do one task: print a specific line of the text of play. Can we say that printing *Hamlet* in this case is a function of the instructions?[59] This program is limited to printing *Hamlet*. It will never print *Romeo and Juliet*. What is the function of this program if it is not to print *Hamlet*?

In terms of implementation one may store each character as a constant in an instruction using the immediate addressing mode. The print procedure must first execute this instruction to recover the character and then print it. Translated into simple English, this means we are choosing our instructions in such manner that each printable character of *Hamlet* is part of some machine instruction. No character is stored separately as data. If we apply the principles set forth in *WMS Gaming* to this programming technique the text of *Hamlet* is part of the machine structure because because all information is built into the instructions.

You think this example is far fetched? How about PostScript? This is a file format where the contents is made of instructions on how to print the file. This format is commercialized by Adobe

Computer Systems. See for yourself how they describe PostScript: (link in the original)

> In the earliest days of PostScript, drawings could be created only by manually typing in the PostScript language. Programmers would read the PostScript Language Reference Manual, type PostScript "code" into a text file, and then send it to the printer to be "processed" (more on that in a moment). Illustrator was the first "graphical PostScript interface," much in the same way that MicrosoftÆ Windows 1.0 put a graphical user interface on top of MS/DOS. Illustrator allows the designer to draw with graphic tools while it automatically writes a PostScript program in the background.
>
> 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.
>
> Every programming language needs a processor to run or execute the code. In the case of PostScript, this processor is a combination of software and hardware which typically lives in a printer, and we call it a RIP - a Raster Image Processor. A RIP takes in PostScript code and renders it into dots on a page. So a PostScript printer is a device that reads and interprets PostScript programs, producing graphical information that gets imaged to paper, film, or plate.

PostScript is an example of a commercially supported file format where the data contents is represented by instructions. But in this case the instructions are more like byte code than machine executable ones.

Efforts to make a legal distinction between types of data on the basis of a subjective interpretation of functionality with an assumption that this distinction determines machine structure make no factual sense. We have two extreme situations, one represented by the universal algorithms based on term rewrite systems and the other represented by PostScript. In one case we have programmatic functionality defined from non-instruction data which is constantly modified as the computation progresses. In the other case we have data represented by instructions with no functionality outside of the data representation. We have a spectrum of possibilities between these extremes with no way of distinguishing instructions from data which will result into functionality always going with instructions and never going with data.

**The Nature of Software Processes**

Sometimes software is not claimed as a machine patent. It may also be claimed as a process patent. For example, Gene Quinn explains How to Patent Software is a Post Bliski Era as follows:

> While it is true that the Federal Circuit has largely made “software” unpatentable, they did not prevent the patenting of a computer that accomplishes a certain defined task. Given that a computer is for all intents and purposes completely useless without software, you can still protect software in an indirect manner by protecting the computer itself, and by protecting a computer implemented process. You see, processes have been patentable since 1790, and despite the misinformed protestations by computer scientists and mathematicians, you can protect software because the software provides the instructions for the computer to operate, can be defined as a process or method like any other patentable process or method. Rather than describing the process as it would be done by a human actor, you simply define the process as it is done by a computer. If you know what you are doing it really is not that hard, at least conceptually. The inability to actually call the invention “software” does, however, mean that the disclosure needs to be much more verbose and perhaps even a bit circuitous.

Then I mus ask, what exactly is a process described in this manner? I exclude from this discussion the famous example is the industrial process for curing rubber which has been litigated before the Supreme Court in *Diamond v. Diehr*. I am discussing patents on the software itself, that is patents on computations.

There are at least three different processes to consider. There is the words of the patents. There are the machine instructions loaded in memory. There is the transistor activity when the instruction cycle executes. These three processes are different. The words of the patent use the full expressive power of legal English. They refer to the real world meanings of the bits. The instructions have no such capabilities. They describe the specific mathematical algorithm that the instruction cycle must execute. But the transistor activity is the execution of the instruction cycle itself which is a universal algorithm. This hardware process is the same for all computations.

This situation is even more complex when the programmer uses software universal algorithms or other algorithms such as regular expressions which replicate the behavior of other algorithms.

Software patents rely on the courts conflating all these processes, applying the law as if they were all the same thing. The words of the patent define the claimed process. Then when the court sees the computer is performing what is described with these words then this process is deemed "implemented". Whatever is actually implemented is not examined. The role of symbols and semantics is absent. The role of universal algorithms and the instruction cycle are off the radar screen.

This conflation of processes is best summarized in the above quoted brief of Microsoft, Phillips and Symantec to the Supreme Court:

> Purporting to analyze the patent-eligibility of software, as opposed to that of hardware, relies on an illusory distinction. The functionality of any digital device is the product of the same transistor activity, and it is the configuration of the pathways between those transistors that dictates their functionality.

The distinction between hardware and software patents is that software is manipulation of symbols. To conflate software with transistor activity is the same error as conflating marks of ink on paper with text carrying meanings. The letter is an abstraction which is recognizable when one sees its physical representation but it is not the same thing as the physical representation. This is why people are able to recognize that *abacus*, ABACUS and **A**b*a*C*u***s** are all different ways of writing a word for the Roman era computing device. The physical layouts of the typefaces are different but the letters are recognized as being the same.

In a computer the symbols are called bits. The digital computer alphabet has only two "letters" called 0 and 1. They are represented physically by a variety of means. Sometimes 0.5V means 1 and 0V means 0 but sometimes engineers use the reverse convention. Sometimes the chosen voltages will be different from 0.5V and 0V. Sometimes the bits are electric charges stored in capacitors. Sometimes they are the state of flip flop circuits. Sometimes they are the polarity of a magnetic field and sometimes they are cavities in an optical media. This diversity of representations is similar to the diversity of typefaces in written letters. Bits are abstractions which are different from their physical representation.

Bits are also abstractions different from their meanings. For example the letter "n" is a separate entity from the meanings of "notation", "binding" and "administration" even though all these words involve the letter "n". Similarly the digit "0" is an entity which is separate from a number, a birth date or the name of a city even though all these data elements are constituted of bits in computer memory, some of which may be "0".

A computation is a manipulation of the abstract symbols. It is represented physically by the activity of the computer which manipulates the physical bits. But still, a computation, as defined by an algorithm, is an abstract entity distinct from the computer and the transistor-implemented process by which it operates.

This distinction should be intelligible to anyone having used a pocket calculator. It is obvious that the calculations are not the calculator and the descriptions of the calculations are not a description of the calculator. This is the very principle which makes software such as Bochs possible. The

calculations done by the processor are not the processor. They may be carried out in software without having to build a specific circuit for them.

**Experiments and Demonstrations**

The argument so far requires to understand rather technical facts of technology and mathematics. I have been told that this sort of arguments is unintelligible to laymen, that too much expertise is required. This objection doesn't diminish the merits of the argument but the practical problem is real. How do we convince laymen of the truth of what is said?

This problem may be solved with demonstrations which prove the facts in a way that doesn't require learning too much about computers. I propose a series of experiments that will demonstrate visually some of the contentious points. These demonstrations may be shown to any audience of laymen. We may present them to judges and juries in a courtroom. We may present them to patent scholars in a conference. We may do private demonstrations to lawmakers and their staff. Perhaps they could be recorded on video and posted on the Internet.

*Programing an Unprogramed Computer*

We have two competing explanations of how functionality is introduced in a computer by means of program. One of them is the storeroom of parts theory of *Prater*. According to this explanation introducing a program in a computer changes the configuration of the computer circuit and turns it into a specific circuit. The other explanation is that the processor and the memory are the moving parts of a machine implementing a universal algorithm. The introduction of a program is supplying input to the universal algorithm and from a machine perspective involves nothing more than the normal work of these moving parts.

Here is a demonstration which aims at proving that a modern day general purpose computer is never unprogramed in practice. It is always programed. We don't introduce a program into a computer which could be described as a collection of disjointed components, a “storeroom of parts”. We introduce more programs into an already programed and working computer during the normal course of its operations.

The demonstration starts with an unprogramed computer as it gets out of factory. We progressively introduce in its memory more and more programs, demonstrating what is the state of the computer before it was programed and what the task of loading a program in memory entails. At each stage the audience may see that the “unprogramed” computer is actually programed, powered up and running. It is shown that we can't program a computer unless it already runs a program. Here is how it goes:

- We start with a computer with no operating system in its powered off state. We tell the audience that this computer is the closest thing to a totally unprogramed computer because this is a computer as it gets out of factory before we install any software. We show that when we turn it on it doesn't boot.

- We show that even when the computer is in this state it already includes a program called the BIOS. This program is burned into some nonvolatile memory and soldered into to computer circuitry during assembly in factory. Therefore even a totally unprogramed computer as it comes out of factory has a program in memory. The BIOS is guaranteed to be always present when the computer is powered up. We show this to the audience by making the computer running the BIOS user interface.

- Then we initiate the procedure to install an operating system. To install means to store an operating system in working order on the computer. A possible procedure is to run a program especially designed to install the operating system. We show that the BIOS is able to load in memory this installation program. Then we show that the computer executes the installation program. We explain to the audience that the outcome of this program is that

an operating system will be stored on the computer hard disk and will automatically start upon powering the computer up.

- This installation procedure is very long, so it is better to cut this part of the demonstration short. Instead we show another computer with the operating system installed. We tell the audience that the installation program has been run on this computer and we are showing them the result. We show that when the computer is powered on the BIOS loads the operating system in memory. This is programming the computer with the operating system and we show that the BIOS is required to perform this operation.

- Once the operating system is started this computer is still unprogramed in some sense. For example it doesn't run a spreadsheet and a web browser yet. We may further program the computer by double clicking on icons the usual manner. Then the operating system will further program the computer by loading a spreadsheet and a web browser in memory.

- This computer is still unprogramed in the sense that it doesn't run the interactive application written in JavaScript made available by a web site. JavaScript is a programming language which is typically used to extend the capabilities of web pages with enhanced user interfaces and dynamic features. The user using a web browser may access several web sites and see web pages as usual. It is only when the web site with the JavaScript application is accessed that this application is brought into memory and executed. This is programming the computer with a JavaScript program. Such programing happens behind the scenes when the user gets to a site designed in this manner during the normal course of web browsing. From the point of view of the user, nothing special happens. He is just visiting yet another web site as usual. The operating system and web browser must be running and fully functional for such a JavaScript program to be run.

- This computer is still unprogramed in the sense that the spreadsheet macros have not yet been loaded in memory. The spreadsheet program works fine. We can edit a spreadsheet with it. But when we open a spreadsheet with macros then they are loaded in memory and further programming occurs. This is programming the computer with macros.

- Then we show what a truly unprogramed looks like. We run a utility which destroys the BIOS. Then we power off the computer. The result is that the computer won't start anymore. It can't display the BIOS user interface. It can't start the operating system. It can't start an operating system installation program. The computer has been turned into an expensive doorstop because it can't be programed at all. The only way to fix this situation is remove the chip containing the defective BIOS and replace it with one containing a working BIOS. This is a physical operation altering the the computer circuit.

As we can see throughout this demonstration computers in working order are not unprogramed, not even when they get out of the factory without an operating system. Computers are shipped with a BIOS. The only time when programming a computer could be said to make a new circuit is when the memory chip containing the BIOS is soldered on the computer motherboard. Once this is done computers are programed computers which can be further programed during the normal course of their operations.

*Symbols and Procedures Manipulating Symbols*

We have two theories of how algorithms are implemented into a computer. The factually correct theory is that computers are symbols processors. They are machines for manipulating bits and algorithms are procedures for such symbolic manipulations. The legal theory is that circuitry is configured in such manner that pathways between transistors are altered to physically implement the algorithm. Symbols and meanings are absent from this explanation.

This demonstrations shows the symbols and their manipulation to the audience by showing them a debugger. This is a tool computer programmers use to troubleshoot programs and fix bugs. Here is how the demonstration goes.

- The demonstrator explains that with the tool called a debugger a programmer can see the bits as they are stored in memory. The demonstrator starts a debugger and show the display to the audience.
- The demonstrator explains that the information showed on screen is a human readable rendering of the bits as they are stored in memory. The bits are symbols and a human trained in reading them will know what they mean. The demonstrator shows how a programmer can inspect various locations in memory. He explain the audience that even though they can't read the meanings of the symbols a trained programmer will understand them.
- Then the demonstrator explains that a program is a procedure to modify the bits. The program is a series of instructions and each instruction instructs the computer of a modification of the bits which must be done.
- Then the demonstrator explains that under normal circumstances instructions are executed very fast. Depending on the speed of the computer, from thousands to billions of instructions per second are executed. But a debugger can slow the computer to a speed a human can cope with. The instructions may be executed one by one with a pause after each instruction to allow the programmer to see what has been done. This allows a programmer to check that the program works as expected. He may then find bugs where the wrong instruction is executed.
- Then demonstrator executes one instruction in front of the audience. He shows the audience that some of the bits have been changed as a result. This confirms what has been told, that a computer program is a procedure to manipulate the symbols, the bits.

*Separability of the Real World Meanings of Data*

We want to prove that the real world meaning of the data is separate from the action of the computer. The pocket calculator is used as an analogy to explain this phenomenon.

We may demonstrate this visually by running a calculator application on the computer. This application looks exactly like a pocket calculator and functions in exactly the same manner. This shows that the instructions in this program are separate from the real world semantics of data in exactly the same manner as the pocket calculator is separate from the real world interpretation of numbers.

Another demonstration of the same point will require an application designed for internationalization. In these applications all the textual information of the user interface elements like the titles of window panels, menu text and error messages must be stored separately in an ordinary text file. We may have several versions of this file where the same application code will display different text on the user interface. This is often used to maintain several versions of the application in different languages by having an English version, a Spanish version etc of this text file.

The demonstration requires two versions of the text file. One version contains the normal English text which is expected when running the application. The other version contains nonsensical garbage like %*9ds! which makes the user interface unintelligible. The demonstration goes like this:

- The demonstrator run the two versions of the application side-by-side, telling the audience that the same code is being run. There is no difference in the computer instructions being executed. The difference is in the text shown to the user. One application is ordinary English everyone understands. The other is gobbledygook, which makes the user interface pretty impossible to use.

- The demonstrator shows both versions of the file for the user interface text to let the audience know what the difference is. He explains that one version contains the intelligible text and the other contains the gobbledygook. But other than this change in the text the

two applications are identical. It should be apparent to the audience that the difference is not in the computer instructions. It is in a file containing pure textual data.

- Then the demonstrator carries the same tasks in both applications side by side, showing that they both work identically except for the text of the user interface. The audience will see how the application works by watching how the demonstrator interacts with the plain English version. They will see that the gobbledygook version by itself is impossible to use. But the demonstrator shows that we can know the meaning of the gobbledygook by referring to the plain English version just next to it. When the demonstrator makes abstraction of the gobbledygook and manages to interact with the user interface the application works just fine. The point is that in both applications the instructions carry out their tasks as intended.

- The demonstrator explains that for the computer the text being displayed is meaningless symbols. It doesn't matter to the computer program whether this information is meaningful or not. It only matters to the human users. The point is that the functionality of a meaningful user interface doesn't lie with computer executable instructions. It lies with the choice of the data being shown to the user.

Imagine that you are sued or threatened to be sued for infringement over a claim which attributes to a machine functionality which belongs to the text displayed by a user interface. Think of the Lodsys patent lawsuits. Could you use a demonstration like this one for your defense?

*Functionality which Changes as the Program Runs*

We want to prove that software functionality is something which can be produced on-the-fly through manipulation of data during the course of the execution of a program. This is very different from the legal explanation which assumes all functionality results from storing in memory instructions altering the machine structure according to the principles set forth in *WMS Gaming*.

This take the previously mentioned patent claim from Lodsys as a reference. We need an implementation which uses the time of day to decide on a course of action. When the count of seconds is even, it asks the user the question “What do you think of this program?” When the counts of seconds is odd, it asks the user the question “What do you think of Barack Obama?” This is an application which changes its functionality back and forth from customer feed-back to political opinion polling every second. If this program is run in front of the audience they will see the question will change from a run of the program to another.

The point is that functionality doesn't depend only on instructions. It may also depend on data because a test on data may determine which instructions are executed. Data routinely change dynamically and it is not machine structure according to the principles set forth in *WMS Gaming*.

*Difference Between Instructions and Data*

There are two competing explanations as to how functionality is produced from computer programming. One explanation is that the instructions from the computer instruction set determine the functionality when they are stored in memory. This is the legal explanation from *WMS Gaming*. When the instructions are introduced in a computer they impart functionality to the computer, turning it into a specific machine. The other explanation is that there is no real difference between instructions and data. Functionality may be implemented with any type of data when supplied to a suitably constructed algorithm whether or not the data is instructions from the instruction set.

I have two demonstrations for this one. The first demonstration shows that functionality may be produced with data which is not instructions in the instruction set. The other shows that these instructions are data and the instruction cycle is an algorithm separate from the hardware.

The first demonstration is of a bytecode interpreter such as those used to implement programming languages such as Python. Programs in this kind of languages are a special kind of data called bytecode. The program cannot be executed directly by the processor because bytecode is not instructions from the instruction set. We execute these programs with a different procedure. We must first start the bytecode interpreter which is a program made of instructions from the instruction set of the computer. Then the bytecode interpreter reads the bytecode data as input and does as the program must do.

The demonstration goes like this:

- The demonstrator first starts the bytecode interpreter for Python. He explains this is a program made of computer instructions which is loaded in memory. The computer is now programmed with the Python bytecode interpreter. This program displays a prompt asking for input. The audience sees this prompt and notices that input is expected.

- Then the demonstrator explains he will give input to the Python interpreter. He shows the files he uses as input and explains they don't contain machine instructions to the computer but they nevertheless are Python programs. When we try run these files directly as programs loaded in computer memory, they don't run. They may only be run indirectly by means of the Python interpreter.

- Then the demonstrator shows the effect of supplying one of the files as input to the interpreter. The audience sees the Python program is running.

- Then the demonstrator shows the effect of supplying different files as input to the interpreter. The audience sees different Python programs are running and each of them has a different functionality from the other Python programs.

- Then the demonstrator explains that for all these Python programs the instructions in the computer memory are the same.[60] They are always the instructions for the Python bytecode interpreter. If the instructions determined the functionality then the functionality would always be the same. But the demonstration shows that this is not the case. This is proof that data stored in the bytecode files is determining the functionality of the Python programs.

How do we explain this result? It is that programs such as bytecode interpreters are universal algorithms implemented in software. The functionality of the interpreter is the ability of carrying out every possible computation provided it is given the corresponding input data. This data is in the bytecode files.

Notice that universal algorithms are mathematical procedures which are separate from the real world semantics of the data. They may carry out all possible computations but they don't have the expressive power to assign a real world semantics to the data.

The second demonstration shows that the instruction set of the x86 platform has been implemented in software. This program is Bochs. The demonstration goes as follow.

- The demonstrator brings a computer based on a SPARC processor. He explains that this computer has a different instruction set than the one found in ordinary PC which is based on the x86 platform.

- The demonstrator tries running a program compiled for the x86 on the SPARC. Of course the audience sees that the program doesn't run. The same program is shown to run on an ordinary PC which has an x86 processor.

- Then the demonstrator starts the Bochs program on the SPARC computer. He explains that this software simulates an x86 computer. It allows to run programs made of instructions for the x86 without actually having an x86 computer processor.

- Then the demonstrator boots an operating system such as Windows or Linux on the SPARC with the help of Bochs. He explains that it is sufficient to simulate a PC with software to run PC software. There is no need to have the actual hardware. He shows that regular PC programs run within a Bochs simulation which wouldn't run on the SPARC absent of Bochs.

This demonstration shows that the x86 instruction cycle is an algorithm and that the x86 instructions are data processed by this algorithm. The implementers of algorithms have the choice between building dedicated circuits, in this case a x86 processor, or writing software for it, in this case the Bochs program.

This demonstration should prove the point even to an audience which is unwilling to believe that all algorithms are mathematical algorithms. It their understanding of the term”algorithm” is that it is a step-by-step procedure which may be patentable when it meets the requirements of patent law then they must conclude that the x86 instruction cycle is such a procedure. Then the conclusion that the instructions are data processed by an algorithm is unavoidable.

At this point the notion that instructions configure a general purpose computer into a specific circuit is not tenable because it is contradicted by the accumulation of facts from this series demonstrations.

*A Virtual Machine Project*

The next demonstration is to show that the execution of all software is a mathematical computation pursuant to a mathematical universal algorithm. I propose a virtual machine project.[61]

The idea is to write the mathematical formulas for the computation carried out by a virtual computer.[62] Virtual means the computer is not actually built. Some software will simulate the virtual computer by carrying out the computation of the formulas, like we saw with Bochs. Of course a real computer is used for this, but the formulas describe the computation. They don't describe the machine. This point is further emphasized when one observes that the real computer is different from the virtual one and the formulas for the virtual computer are chosen in such manner that they don't apply to the real one. The virtual computer is a pure mathematical abstraction.

Then we port some compilers, operating systems and applications to this virtual computer. As a result an entire software stack from operating system to applications will run over the virtual computer. This means the execution of all this software is pure mathematical calculations pursuant to the mathematical definition of the virtual machine.

The demonstration goes as follow.

- The expert demonstrator first shows the audience the mathematical definition of the virtual machine. Everyone sees the mathematical formulas and understand that this is mathematics. They will not understand the formulas but their eyes will tell them this text has the inimitable appearance of mathematical language.

- Then the expert demonstrator tells the audience the virtual machine is software that computes these formulas. If the expert is credible, and there is no reason he wouldn't be, the audience will trust him at his word.

- Then the expert demonstrator runs the virtual machine. He loads Linux and BSD distributions and shows the software is running in the virtual machine. Everyone sees the formulas do exactly what the expert says they are doing. Audience members will not understand the details of the mathematics and the technology, but they can trust their eyes to see that everything works as the expert says it should.

At this point everyone will see that running the software is a computation which is mathematically

defined. The point that all software is a mathematical algorithm is visible.

The same demonstration shows the point of the universal algorithm. The same series of mathematical formulas is used for all programs because they describe the computations done by the virtual computer as a whole. This is exactly what a universal algorithm is.

**An Actual Patent Claim as an Example – A Claim on Storing a Composite File Structure**

Let's look at how the argument plays out on an actual patent claim. I have chosen claim 10 from this patent titled Capturing and combining media data and geodata in a composite file.

This patent is about combining media data such as video and audio with geolocation data such as latitude and longitude in a single file. The claimed innovation is in the structure of a file designed to support the annotation of the media data with information on the location where the audio and video have been recorded. Claim no 10 defines the claimed file structure.

> 10. A computing device configured to store media data and geodata in a composite file structure in non-volatile memory, the composite file structure comprising:
>
> a body including:
>
> one or more media objects including media data captured at a plurality of geographic locations along a path in a geographic area, and
>
> one or more geodata objects including geodata identifying each of the plurality of geographic locations with a latitude value and a longitude value, each of the one or more geodata objects corresponding to the one or more media objects, wherein the media data objects and the geodata objects are interleaved in the body of the composite file structure; and
>
> a header describing contents of the composite file and including objects identifying the one or more media objects and the one or more geodata objects, wherein each of the geodata objects includes a stream number identifier and a media object identifier identifying a corresponding media object to which the respective geodata object relates;
>
> wherein the header includes parsing information to describe how to parse the body as a streaming file.

This claim is drafted as a machine claim. Which machine? It is one which is configured to store a certain composite file structure in non-volatile memory. This claim doesn't recite the structure of the machine. It recites its function, what it achieves instead of how it is done. I can think of three possible outcomes for this kind of claim drafting.

- The element of the claim so described is either well known or obvious. This informs the reader that this old or obvious element is part of the claimed invention but it should be expected that the point of novelty is elsewhere. Otherwise how could the patent be valid?
- This language is claiming as the invention all possible ways to build a machine which achieves the stated result regardless whether or not this particular way of building the machine has been invented by the inventor.
- This language attributes to a machine some functionality which doesn't result from the work of the machine. The lack of information on the machine structure hides that this is the case.

In the case of this claim, which is it? I take the third option. The claim is attributing to a machine

some functionality which is not the function of a machine. I believe this functionality wouldn't be considered patentable subject matter if the claim were not drafted to a machine as a software patent.

Remember what I have said earlier on the art of the programmer? I am going to repeat for clarity. When confronted to a real world problem, one of the programmer's tasks is to find out a representation of the elements of the problems in terms of machine symbols. These symbols are the bits, the 0s and the 1s stored in the computer. An algorithm is a procedure for manipulating symbols. It requires that we define an encoding which is used to represent symbolically the information processed by the algorithm. A composite file structure such as the one from this claim is an example of such an encoding.

Another task is to define procedures that constitutes the program.[63] Only this second task defines something which is machine executable. But the first task is an essential part of computer programming. It is a mistake to attribute the result of the first task to the operation of a machine because because an arrangement of symbols and their semantics is not a function of a machine.

The innovation in this claim is the result of the first task: the definition of the composite file structure. Here is an analogy to help laymen understand. Consider a claim like “a printing device configured to print a paper form comprising boxes for writing such and such data.” Ordinary printing devices can print any form no matter the arrangement and contents of boxes. Such a claim should be seen as an attempt to claim the form by calling it a patent on a printing device. A composite file structure is analogous to the paper form.

A file, like any data representation, is an arrangement of bits with a syntax and a semantics. The syntax is analogous to the arrangement of boxes in the paper form. There is a series of “places“ in the file where various element of information must be written. Please refer to this drawing from the patent to see the particular arrangement which is disclosed in the patent specification. The semantics is the meanings of the contents which must go within each place. It is the equivalent of the meanings of the written text which must go in each box in the paper form.

A mathematical theory suitable to define mathematically such data organizations is category theory.[64] This theory is applicable to the current task.[65] The theory of formal language is another applicable theory.[66]

We may verify this is not the function of a machine by considering what happens when the file is written on some optical disk such as a DVD. The file structure and its contents is on the disk but the disk is not a machine. If the word “store” from the claim is construed as “hold in storage” then the optical disk store the file. But if we insert the DVD in the bay of a DVD reader would the computer be “configured to store media data and geodata in a composite file structure“?

Another construction of the word “store” may be “write the data on a storage device”. While the word “store” denotes something a machine can do this still doesn't make the claimed innovation a machine function. We may see why by reviewing this list of possible implementations.

1. It could be a device with a program to copy optical disks. You insert a disk in one bay, a blank disk in another bay and the copy program will duplicate the disk. If the disk being copied happens to contains a composite file the computer is “configured” to store the composite file on non-volatile memory, which is the blank disk.

2. Or it could be a computer running an ordinary program which saves the contents of ordinary computer memory into hard disk. If the composite file is present in ordinary computer memory, which by definition is volatile, it will be stored into non-volatile memory by such an operation.

3. Or it could be a computer running an ordinary web browser. If we use the browser to download the composite file from a web page it may be saved on non-volatile memory.

4. Or it could be a computer running an email program. If someone sends us the composite file as an attachment to an email the email program may save it on non-volatile memory.

5. Or it could be the bare computer with only the operating system. If we use an ordinary file copy utility provided with the operating system to copy the composite file from hard disk to hard disk the file will be stored in non volatile memory.

You get the idea. Any ordinary file copy program will ”store media data and geodata in a composite file structure“ as long as the original file has this kind of contents. All these programs are old. All these programs have the ability to copy the bits regardless of their structure and regardless of their meanings. The claim reads on these copy programs provided the bits being copied are arranged in the claimed manner and carry the specified semantics. This is claiming a computer as a copy machine except that the copying is limited to a particular kind of contents, much like the printing device of the analogy is limited to printing a particular paper form.

This is an application of the principle that the semantics of data is not implemented by instructions. Even if one is to accept the legal theory of software functionality this claims doesn't read on a new machine. It reads on an existing machine with an additional limitation that the data has a specific file format and a specific type of contents. This limitation doesn't further specify the machine because it doesn't translate into new instructions.

This act of copying is a mathematical algorithm. We have a series of symbols, the bits. The procedure is to replicate the bits in order one after another, mechanically like a robot, in such manner that we obtain an identical copy of all the bits in the same order. This procedure meet the requirements for being a mathematical algorithm. It computes what is known in mathematics as the identity function which plays in important role in algebra, lambda-calculus and category theory.

There is another possible construction for the phrase ”store media data and geodata in a composite file structure“. It may mean read the media data and geodata from separate sources and write them together in the composite file structure. Then the verb “store “ no longer means a plain copy. It means the more complex operation of assembling the file structure from scratch from the individual data elements.[67] This construction has two problems. One is obviousness, the other is that it is also attributing to the machine functionality with is not the machine function.

Assembling the file from its data elements is a mathematical algorithm too. It is a series of bit copy operations, one copy operation for each data element, together with some book keeping to initialize the header elements to their proper values. These operations are executed mechanically, like robots. The mathematical task is to write the mathematically defined encoding of the data which I have described before.

Once the file structure is known this algorithm is obvious to a programmer. In the paper form analogy, once you have in hand both the form and the data which goes into the form, filling in the boxes is obvious. Someone may try claiming a process for taking the data and putting it into the boxes. Would this process be nonobvious on the basis that there is no prior art for filling a form like this particular one because all the prior art is about other forms or other types of data? For a programmer this is the kind of obviousness which applies to this claim construction. The innovation is in the file structure, not in the procedure to fill in the file with data.

Another issue is that the claim defines the encoding by its real world semantics. As was said previously, this semantics is not a function of the machine instructions because they don't have this kind of expressive power. Attempts to define the instructions of a program by means of the real word semantics of the data is attributing to instructions functionality they don't have.

The main evidence are the reference manuals documenting the instructions. It can be verified that instructions don't have the kind of expressive power by reading their specifications. But depending on the circumstances of the claim, there may another way to show this same point. Sometime the very same instructions will work equally fine on data which carry another semantics than the one

recited in the claim. When this happens we have proof that the meanings of the bits are not a function of the instructions because we can use the very same instructions on bits carrying other meanings.

Let's see more specifically how this idea applies to this claim. An element of the composite file structure is this one.

> one or more media objects including media data captured at a plurality of geographic locations along a path in a geographic area,

This is legalese saying that someone must be traveling with an audio or video recorder and records as he travels, then the media data so recorded falls within the metes and bounds of the claim. What happens when the user stands still when recording the video? What happens when the video is computer generated graphics like some Shrek movie? Then the media data has not been “captured at a plurality of geographic locations along a path in a geographic area”. But this is still video data in the same data format as video that is captured in the claimed manner. A program which inserts this video into the composite file structure won't tell the difference because it operates on the raw bits. It will copy the video data in its proper location no matter where the video comes from and how it has been recorded.[68] Which media data is given to the program is a choice of the user. This is not a function of the program and this is not a function of the computer running the program.

Here is another example. Another element of the composite file structure is this one.

> one or more geodata objects including geodata identifying each of the plurality of geographic locations with a latitude value and a longitude value, each of the one or more geodata objects corresponding to the one or more media objects, wherein the media data objects and the geodata objects are interleaved in the body of the composite file structure;

Latitudes and longitudes are numerical values. There is no difference between bits which represent latitude and longitude and bits which represent some other numerical quantities. In theory it is possible to use the exact same file structure, as defined by the series of places reserved for the various data elements, except that the numerical values stored in the places for for latitudes and longitudes are given a different real world interpretation. Then from the point of view of the program the file has the same structure but from the point of view of the metes and bounds defined by the claim the structure is different. A possible example might by a system for endoscopy. The media data is the video coming from the camera at the tip of the endoscope. But the coordinate system is not longitude and latitude. It is some other numerical quantities describing the location of the camera inside the human body. Then the programed file structure and the software used to combine the data into the composite file will be the same but the real world semantics will not be the one recited in the patent.

As a last point, I argue that software is a use of the mathematical language. Which use is it in this claim? First the syntax of the file is the syntax of mathematicians call a formal language. Then there is a real world semantics assigned to this language. This combination of syntax and semantics is used to record facts. It records in mathematical language the visual appearance and the sounds of a plurality of geographical locations together with the latitude and longitudes of these locations.

**Another Actual Patent Claim as an Example – A Claim on a System Architecture**

Let's look at another claim. I have chosen claim 1 from the same patent as the previous one.

1. A computer system for capturing and combining media data and geodata, comprising:

> a media data capture module configured to capture media data at a plurality of geographic locations along a path in a geographic area, and to store the media data

in a media data stream;

a geodata capture module configured to capture geodata identifying each of the plurality of geographic locations along the path at which the media data was captured with a longitude value and a latitude value; and

a multiplexing module for interleaving the media data and the geodata as respective one or more media objects and one or more geodata objects into a composite file, wherein the composite file includes a header describing contents of the composite file, the header including identifying objects that identify the one or more media objects including the media data, and the one or more geodata objects including the geodata, and a body including the one or more media objects and the one or more geodata objects, wherein each of the geodata objects includes a stream number identifier and a media object identifier identifying a corresponding media object to which the respective geodata object relates;

wherein the header includes parsing information to describe how to parse the body as a streaming file.

For the benefit of the readers which are not computer professionals let's start by explaining what the words “capture” and “capturing” mean. This is tech speak for recording the video and/or audio into a file made of bits.

Capturing may be seen as a simple on-going copy operation. We have a flow of numbers coming from a computer interface connected to microphone or video camera. These numbers represent numerically the video and/or audio information. These numbers must be copied somewhere safe as they are produced. The somewhere safe is the video and/or audio file.

The preceding paragraph only applies to uncompressed audio and video. Compressed audio and video is somewhat different. The amount of data produced during a recording session is very large and this is often inconvenient. This problem is solved by running some arithmetical calculations on the series of numbers which transform them into a smaller series of number. This reduces the amount of data to a more acceptable level.[69] These calculations are called compression. For compressed audio and video, capturing collectively refers to the reading of the series of numbers, the calculations for the compression and the storage of the compressed data into some safe location.

With this knowledge we see more clearly what is the claimed computer system for capturing and combining media data and geodata. Capturing is the operation of recording the bits from their sources. Combining is the operation of aggregating the media data and the geodata in a composite file. Combining is the same operation which has been discussed with claim 10 above. Both operations are manipulations of bits which means that both operations are manipulations of symbols. These operations are further limited in the claim in terms of the real world semantics of the bits. As with claim 10 this is problematic for the same reasons.

In the case of this claim the limitation on semantics is blatantly absurd. Consider this claim element:

> a media data capture module configured to capture media data at a plurality of geographic locations along a path in a geographic area

Imagine a digital video camera. It has an embedded computer system which controls the operations of the camera. It also has an embedded GPS receiver for capturing geodata. In this imaginary camera the computer captures both the video and the geodata and records this information in a composite file as described in the claim. How would one implement “configured to capture media data at a plurality of geographic locations along a path in a geographic area”? The obvious answer is to board the camera on a vehicle and record some video as it travels. This would

capture video “at a plurality of geographic locations along a path in a geographic area” by virtue of the travel. How is this different from configuring the same module to capture video data without a limitation on the geographic locations and just travel with the device?

This claim is attributing to the computer and its program a function which is achieved by means other than computing. It is well known in the computer profession that no amount of programming will make a computer travel. This is not a functionality provided by the instruction set of any computer.

This underlines the difference between the expressive power of the legalese dialect of English and the instruction set of a computer. One cannot presume that everything which is written in a claim automatically translate into machine executable instructions without verifying that this is the case. In particular the real world meaning of bits is easily expressible in a claim language but is never expressible in instructions.

Once the real world semantics of the data is filtered out the claim, what is left? It is the abstract computation running on the computer which is defined by the instruction cycle. This is why I say software is mathematics. The instruction cycle is a mathematical algorithm. The computer is a machine for carrying out mathematical computations and assigning a real world semantics to the bits is using mathematics as a language to describe and analyze the real world.

Assuming the courts disagree with this last assessment, what happens next? The claim states that the computer system is comprised of at least three modules, one for media capture, one for geodata capture and one for multiplexing. What is a module? This term is ambiguous. If software is considered then a module is a portion of a program. Programmers often divide complex programs into smaller parts to make the solution simpler and more manageable. Sometimes these parts are called modules. But a module may also sometimes be a hardware component. Which is it? Did the inventor invent both?

Algorithms may be implemented in software or hardware at the implementers choice. As far as the computation is concerned, both approaches will work but this doesn't mean the two approaches are equivalent. Circuits have features other than their computational functions. They vary in size, shape, weight, speed, manufacturing costs and power requirements among other things. A dedicated circuit is typically faster than software running on a generic computer of similar clock speed[70] but the dedicated circuit may carry out only the algorithm for which it has been built. Software has its own advantages. It is flexible in terms of functionality. One doesn't need to have a dedicated machine for each task. Also software may easily be upgraded if required.

These differences highlight the point that a computation does not define a circuit and it does not define the operating principles of a circuit. If patent law assumes that a definition of a computation implicitly define the circuit then the other characteristics of circuits are ignored. Someone who invents a better circuit to carry out the same computation faster or with less electrical power may still infringe on an existing patent. The one who owns a claim on the computation owns a claim on all the corresponding circuits regardless of whether or not he has invented the circuit.

Consider the embedded computer in a digital video camera. Capturing video and recording it in a file is the very purpose of this camera. It makes engineering sense to implement the three modules of this claim as a special purpose circuit. This decision will reduce the load on the embedded computer and may allow to use less powerful but also cheaper and less power hungry components. The result is a better, cheaper camera with a longer battery life. In this case a special purpose circuit is not equivalent to software because the advantages of dedicated circuitry matter. As the claim stand, if the word module is construed to include hardware modules this claim will cover the camera embedded computer system even though the inventor might not have invented any circuit at all.

The other construction for the word “module” is software. Then I would ask what is the construction for the phrase “configured to”? The claim is written to a computer system comprising modules. The modules are “configured to” do some functionality. What does “configured to”

means in this context?

Does it mean whenever the functionality occurs the module is automatically considered to be configured to perform it? Then the words “configured to” would be superfluous. One might as well recite the functionality without a limitation on configuration. Normally the words in a claim are expected to have some substantial meaning. If it says “configured to” then these words should not be ignored.

Another more reasonable construction is to follow the principles of *WMS Gaming*. Then the word “module” is construed as a series of instructions from the instruction set of the computer. The phrase “configured to” means that the instructions are so chosen that they instruct the computer how perform the stated functionality. Then the functionality is required to limit the module algorithm to one which limits machine structure in the sense of *WMS Gaming*.

This is where the knowledge of how the various programming techniques implement functionality is useful. Real world semantics is separate from the native hardware instructions. Also the program needs not use the hardware instructions. It may be directed to a bytecode interpreter or to a programming language based on a term rewrite system such as lambda-calculus. The program may also run within software like Bochs. These implementations would be non infringing under this construction because the instructions which are loaded in memory do not by themselves fall within the metes and bounds of the claim. Besides in these scenarios these instructions are old.

Another construction is that the module is data in memory which is used to produce the stated functionality. This is an admission that a computer program is data provided as input to the computer. This is an admission that all kinds of data, not just instructions, are used to produce functionality. This leaves open the question of how this data is machine structure. The principles of *WMS Gaming* are not applicable to this construction

I have a question here for friendly lawyers. Can we get from the courts a ruling that the principles of *WMS Gaming* must be used to determine whether a software implementation is machine structure which infringes on a software patents? This may possibly open the door to writing software which will not infringe on software patents drafted as machine patents. It may be possible to work around these claims with well chosen programming techniques.

**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

[Cooper 2004] Cooper, S. Barry, *Computability Theory*, Chapman & Hall/CRC, 2005

[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. 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

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

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

[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

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

[Stoy 1981] Stoy, Joseph E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, First Paperback Edition

[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]
It is also included in the anthology [Davis 1965]
Here is a biography of Alan Turing, from the MacTutor History of Mathematics Archive

[Walters 1992] Walters R. F. C., *Categories and Computer Science*, Cambridge University Press printing, originally published (c) 1991 by Carslaw Publications.

**Footnotes**

1 To be fair, not all legal decisions use this kind of analysis. For example *Cybersource Corporation v. Retail Decisions, Inc.* did not.
2 Actually the pathways between transistors are not physically modified at all. Transistors are semiconductors devices whose resistance to electrical current may be modified during the operation of the circuit. The voltage at the output of the transistor will vary according to the resistance. The switch metaphor may describe how transistors are used in a digital circuit but this is only a metaphor. The actual principle of operation is that the pathways are permanently etched on the circuitry and are never altered. What is changed is whether or not the transistors will apply voltage to these existing pathways. From a logical perspective the circuit operates like an array of switches turning on and off but what physically happens is the current flows according to the voltage levels set by the transistors. If machine structure is defined as the physical layout of the circuit which is etched on the semiconductor at the factory then this structure is never changed. We may talk about transistors establishing electronic paths only to the extent we assimilate the application of some voltage to an existing pathway as the “making of a new electronic path”.
3 The state on or off of transistors acting as switches is used to represent the values 1 or 0 of bits. Taking addition as an example, when adding numbers 13 and 25 to make 38, some transistors are turned on and off in the processor to represent the number 13, some transistors are turned on and off to represent the number 25 and the operation of addition is turning on and off more transistors for producing the bits of 38 out of the bits for 13 and 25. Transistors are turned on and off and pathways between transistors are established and torn down on the basis of both the values of data and the instruction of addition.
4 See [Taylor 1998] p. 55.
5 See [Cooper 2004] p. 4
6 See [Cooper 2004] p. 3. The paper by Alan Turing is [Turing 1936]. The book by Martin Davis is [Davis 2000]. See also Jack Copeland's A Brief History of Computing.
7 See under the heading *The Definition of “Algorithm” in Computer Science* in the previous article for the relationship between the mathematical and computer science definitions of algorithms.
8 See [Greenlaw 1998] p. 19
9 This is part of Kleene's definition of algorithm. See [Kleene 1967] p. 223.
10 A common representation is to use standardized voltage levels in circuits, often 0V means zero and 0.5V means one. This is by no means the only representation which is used.
11 See [Patterson 2009] p. C-4
12 See [Rogers 1987] pp. 1-2.
13 The distinction between function and algorithm may raise questions from laymen who think of mathematical computation as calculations on numbers. They may ask aren't the symbols only tools for representing numbers? Then why shouldn't we think of a mathematical algorithm as a procedure about the numbers? Mathematician has investigated this line of thought. Their answer is the notion of Gödel numbers which is a mathematical method for representing series of symbols as numbers. They found that from the perspective of mathematics it is immaterial whether we define a computation as a procedure on numbers or a procedure on symbols because every computation defined as an operation on symbols has a corresponding numerical computation operating on the Gödel numbers of the symbols. And conversely computations which are defined as operations on numbers are actually carried out by means of symbols. Therefore the difference between computations on numbers and computations on symbols is a matter of form and not substance. This idea will be further explained when I will discuss the Church-Turing thesis.
14 See [Taylor 1998] p. 58.
15 We may find the definitions of various models of computations in textbooks of computation theory. For example see the title of [Taylor 1998] *Models of Computation and Formal Languages*.
16 See the previous article the section titled *The Correlation with Pencil and Paper Calculations* which is found in appendix A for a discussion of how mathematical algorithms relate to pencil and paper calculations.
17 It is currently unknown whether there a physical phenomenon which may be used for computing functions which can not be computed in principle by an idealized human being. However according to the Church-Turing-Deutsch principle the laws of quantum mechanics

are such that every physical process may be simulated by a Turing machine.

18 Here is an example of a situation where such a mathematically precise definition of "algorithm" is needed. Sometimes the question arise in mathematics whether a specific problem may be solved at all. There are problems which are undecidable because if there is no effective method in the universe of mathematics which will solve them. When this is the case looking for a solution is a lost cause because a solution is not possible. Mathematicians want to be able to write mathematical proofs of whether or not a problem is undecidable. Once they have the proof it is undecidable they will stop looking for the solution. This kind of proof requires a mathematically precise formulation of what an algorithm is.

Below Howard Delong discusses what is needed to find out whether arithmetic is decidable. Given some arbitrary formula of arithmetic can we determine with an algorithm whether, yes or no, this formula is provable as a theorem? He explains the difficulty thus: (emphasis in the original) (See [Delong 1970] p. 186.)

> In order to prove the existence of a decision procedure for a system, one has to find an effective finite method for determining whether or not a formula is a theorem. For such a proof it is not necessary to define the general concept of an effective finite method. It suffices to merely exhibit a method which everyone accepts as effective (that is no inventiveness is needed beyond what is learned in the procedure) and finite (that is, anyone could finish (barring death, etc.) in a finite amount of time). For a proof that no decision procedure exists, however, there must be an accurate definition of the concept.
>
> An analogy will make this clear. To bisect an angle in plane Euclidian geometry is easy; one does not have to know everything that can be done with a straightedge and compass. All one has to do is to construct a bisector. In contrast, to show that it is impossible in general to trisect an angle requires that one have a precise definition of *all possible constructions with a straightedge and a compass*. Because the ancient Greeks lacked such a precise definition, they were never in a position to prove the impossibility of trisection. Hence, in general, impossibility proofs are more fundamental and more difficult than proofs of construction.
>
> If we are going to consider the possibility that arithmetic might not have a decision procedure, we must precisely define decision procedure and consequently, effective method. The purpose of an effective method is to determine the truth value of some given instance of a proof predicate for some system, which would include calculating functions. Our problem, then, is to know what an effectively calculable function is, and what an effectively calculable predicate is. In 1936 Alonzo Church proposed that we identify the intuitive notion of an effectively calculable function with the mathematical exact notion of a general recursive function and hence also the intuitive notion of an effectively decidable predicate with the mathematically exact notion of a general recursive predicate. This identification is called *Church's thesis* [. . .]

This is what the Church-Turing thesis is for, to know with mathematical precision what is a mathematical algorithm so mathematicians may use this understanding for answering questions of mathematics such as this one.

19 See [Kleene 1967] p. 232.

20 See [Minsky 1967] p. 111.

21 For example suppose a mathematician wants to show that every problem solvable by means of lambda-calculus may also be solved with a Turing machine. How will he proceed? This mathematician will take the list of the elementary operations of lambda-calculus and for each of them he will find a way to carry out the same manipulation of symbols with operations permitted by a Turing machine. Then he will verify that Turing machines will let these operations be combined in the same ways as they are in lambda-calculus. This is mathematical proof that every manipulation of symbols which is permissible in lambda-calculus can also be achieved with a Turing machine. The mathematician has shown that Turing machines are at least as powerful as lambda-calculus. Then the mathematician does

the reducibility analysis in the reverse direction, showing that every computation done by a Turing machine can be done in lambda-calculus. At this point he has shown that the two models are equivalent.

22 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.

23 The previous article under the section titled *Register Machines and their Variations* contains several references to publications containing this analysis. Highlights of this information are included in this article.

24 See [Hopcroft 1979] p. 166

25 See [Hopcroft 1979] p. 167

26 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. It is proven equivalent to a RASP with similar input and output capabilities and also to Turing machines.

27 This applies to those patents where it is the computation, the manipulation of symbols with possibly a real world semantics, which is the patented method. This is in opposition to things like an industrial process for curing rubber because the step of actually curing the rubber is not a manipulation of symbols.

28 Lambda-calculus has been proven to compute the exact same functions as Turing machines. Therefore these formulas also show that the computer computes Turing computable function.

29 See [Devlin 2000] p. 1.

30 See [Devlin 2000] p. 2.

31 Some of this evidence has been mentioned in the broad section of the previous article following the heading **Speech**. This whole book is dedicated to explaining the language of advanced mathematics to people with nothing more than high school knowledge in mathematics.

32 See [Cooper 2004] p. 4.

33 I have explained the phenomenon in more details in the previous article under the heading *The Notion of Formal System*. I am repeating some of this material here because of its importance.

34 See [Minsky 1967] p.219

35 See [Kleene 1967] pp. 199-200

36 There is abundance of evidence for this in textbooks of mathematical logic. See the previous article under the headings *The Notion of Formal System* and *The Connection of Computation with Language and Logic* for citations and references.

37 See [Minsky 1967] p.222

38 See [Devlin 2000] p. 8, and also take note of the title of this book: *The Language of Mathematics, Making the Invisible Visible*.

39 The details of this part of the mathematics of computing are the topic of numerous books. See for instance [Greenlaw 1998], especially chapters 2, 3 and 7. Another good reference is [Hopcroft 1979]. You may also see these definitions extracted from [Hopcroft 1979] pp. 1-2: (emphasis in the original)

> A “symbol” is an abstract entity that we shall not define formally, just as “point” and “line” are not defined in geometry. Letters and digits are examples of frequently used symbols. A *string* (or *word*) is a finite sequence of symbols juxtaposed. For example *a*, *b* and *c* are symbols and *abcb* is a string. . . . The empty string, denoted by ϵ is the string consisting of zero symbols.
>
> . . .
>
> An *alphabet* is a finite set of symbols. A (*formal*) *language* is a set of string of symbols from some alphabet. The empty set, $\emptyset$, and the set consisting of the empty string {ϵ} are languages.
>
> . . .

> Another language is the set of all strings over a fixed alphabet Σ. We denote this language by $\Sigma^*$. For example, if $\Sigma^* = \{a\}$ then $\Sigma^* = \{\epsilon, a, aa, aaa, \ldots\}$. If $\Sigma^* = \{0, 1\}$ then $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$.

This series of definitions is taken from a textbook on computation theory. They show that symbols and strings of symbols considered separately from their meanings are mathematical entities. In these definitions the formal languages are defined as set of strings of symbols which are constructed without any semantics. This doesn't imply that the symbols can't have meanings. It is just that the mathematics of raw text is independent from meanings. Please note the last example where the binary alphabet is used to define a language in this manner. Computers are using this binary alphabet.

40 See Stephen Kleene's definition of mathematical algorithm [Kleene 1967] p. 223 and also p. 226. The Supreme Court is giving a succinct but factually correct description of what is found in this definition. You may also find it in Appendix A of the previous article.

41 You will find a discussion of universal algorithms in the previous article under the headings *Register Machines and their Variations*, *Universality and Random Access Stored Program* and *Implication on Hardware Architecture*. The relationship between universal algorithms and computer hardware is documented under the headings *Implication on Hardware Architecture* and *Historical Connections Between Mathematical Logic and the Invention of Modern Computers*.

42 Some text books on computation theory presents the mathematical analysis of what is the minimal instruction set which is required to ensure the algorithm executing them is universal. See for example [Minsky 1967] pp.206-214.

43 Later the ENIAC was improved to allow to store programs in memory for execution. In this article I am referring to the earlier design which was manually programed by means of a plug board. I use the phrase “early ENIAC” to make clear I am referring to the earlier design.

44 Historically the first universal algorithm has been published by Alan Turing in 1936. See [Turing 1936]. He has proven that the *universal Turing machine* can compute every possible Turing-computable function because it can reproduce the behavior of every possible Turing machine when given a description of this Turing-machine behavior. By application of the Church-Turing thesis this leads to the conclusion that the universal Turing machine can compute every function which is computable by means of an algorithm. Other universal algorithms have been discovered since.

45 There are several other components such as the power supply and the input/output peripherals, for example keyboard, mouse, display and hard disks. The processor and memory are the components which are actively used in carrying out the computation.

46 This is my simplified layman friendly explanation for purposes of this article. Textbooks on computation theory provide the accurate scientific and mathematical justifications. You will find the references in the previous article under the headings *Abstract Machines*, *Register Machines and their Variations*, *Universality and Random Access Stored Program* and *Implication on Hardware Architecture*.

47 Examples of such manual are The Art of Assembly Language Programming and The PDP-11 Processor Handbook. These manual also document how the instructions are represented as bits.

48 Patterson and Hennessy describes this instruction set architecture as this. (emphasis in the original) See [Patterson 2009] p. 21

> 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.

Compare this which Mitchelmore and White concept of abstract-apart. Machine instructions are defined in their own terms, independently from the actual hardware implementations. And they are also defined in their own terms independently from the real world meanings of the bits. This makes the machine instructions abstract-apart both from real world

applications and hardware.

49 Here is how Hamacher, Vranesic and Zaky describe the instruction cycle. (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.

50 This is called self-modifying code. There are legitimate applications to such technique. This is also used by authors of computer viruses to make them hard to detect by anti-virus engines. See metamorphic code and polymorphic code. Perhaps these techniques will prove useful to avoid infringing on a software patent drafted as a machine patent if we get a ruling that self-modifying code doesn't make machine structure as per *WMS Gaming*. Ask your lawyer before trying this.

51 This is a fundamental principle of digital electronics. See Appendix C of [Patterson 2009] generally. Se also Chapter 3 generally for how bit manipulations are implementing arithmetic operations in hardware..

52 Adders circuits occur within processors as subcomponents. See [Patterson 2009] chapter 4 generally, especially figures 4.1 and 4.2.

53 See the brief p. 13

54 See the brief pages 10-13. In particular page 12: “ The role of software is simply to automate the reconfiguration of the electronic pathways that was once done manually by the human operators of ENIAC. ” See also footnote 4 page 12: “As an aid to comprehension, the reconfiguration is often conceptualized as the computer “running” the software or “executing” the software instructions. In reality, however, when stored as electrical charges, the ones and zeros of the binary code produce electrical currents that literally (but temporarily) reconfigure the electronic pathways running between transistors in the same way that human operators reconfigured the wiring of ENIAC by hand. ”

55 This section of the article is superfluous because the other arguments should be more than sufficient. But I can't help being thorough.

56 Regular expressions are part of the mathematical theories describing the mathematical properties of text. Several textbooks of computation theory documents them. For example they are the topic of [Hopcroft 1979] chapter 2 and [Taylor 1998] charter 9.

57 [Kluge 2005] is a reference manual describing several universal algorithms implementing lambda-calculus in this manner.

58 [Patterson 2009] distinguish between the control and data path functions of the processor. The instructions configure the control part and the data influences the data path. In both cases there are pathways between transistors which are configured and torn down. In both case this transistor activity is executing the instruction. In neither case it is programming the computer. See [Patterson 2009] generally.

59 It should be noted that these instructions will be understood by a computer in a very

different manner as they may be understood by a human. Most people will see meaning in the text. But from the point of view of the computer, assuming we have a serial printer directly connected to the computer, the instruction

> Print the text “Act 1, Scene 1”

is understood by the computer to mean the sequence:

> Print the letter A
> Print the letter c
> Print the letter t
> Print a white space
> Print the numeral 1
> Print a comma
> Print a white space
> Print the letter S
> Print the letter c
> Print the letter e
> Print the letter n
> Print the letter e
> Print a white space
> Print the numeral 1
> Send the printer a code representing the end of the current line of text

The computer attributes no meaning to the text. The computer just moves bits representing characters to the printer.

60 It is important to make sure the bytecode interpreter doesn't translate the bytecode into instructions before executing them. This is called Just-In-Time compilation. This would defeat the point of the demonstration. Some bytecode interpreters support Just-In-Time compilation and others don't. The demonstrator will need to be careful about which one he uses.

61 This is where my work stops. I don't intend to do this myself.

62 The PDF exhibit I have provided for the formulas corresponding to circuit elements may serve as an inspiration.

63 This is not an exhaustive list of everything a programmer must do, obviously, but these two tasks are necessary and they are sufficient to show that there is more to a program functionality than the instructions executed by a computer.

64 See [Walters 1992] for a reference with applications to computers. Category theory includes boolean algebra and language theory as sub-theories. It also provide neat ways to define a mathematical semantics to languages.

65 If you want the mathematical details, the structure disclosed in the specification at figure 2 is a series of what is called products in category theory. This particular series is definable as a a recursive structure in a Cartesian closed category.

66 See [Hopcroft 1979] for a reference manual written for computer scientists.

67 I think this construction is stretching the language of the claim beyond its breaking point but it is worth looking into it anyway. It is instructive and the draftsman of the patent could have written language directed to this construction.

68 Then someone may ask, where does the geodata comes from? If the video camera stands still, all the geodata values will be identical. They will all point to the location where the video has been recorded. In the case of synthetic video like the Shrek movie, the geodata may be fake. The program will work on fake data as well as it works on “real” data. The point is that there is nothing in the instructions of the program which require the data to carry the claimed semantics. The same program will work equally well on data with meanings outside of the metes and bounds of the claim.

69 On playback some other calculations will return the media stream to an uncompressed state prior to reproducing the sound or video.

70 Digital circuits usually have an internal clock which dictate their speed. We may make faster circuits either by making one with a similar design with a faster clock speed or by making one with a similar clock speed but with an otherwise more efficient design.