Does Programming a Computer Make A New Machine?

By PolR

If you ask this question you may receive a different answer depending on who you ask. If you ask a patent attorney he will answer that there is well established case law that says programming a computer in effect makes a new machine for purposes of patent law. But if you ask a computer programmer he will say that obviously, programing a computer doesn't make a new machine. The whole point of a programmable computer is precisely that there is no need to make a new machine for every individual program.

Which is it? Let's first take a look at what case law says and then compare it with the facts of computer science. They are not currently aligned.

Arguments That a Program Can Create a New Machine:

The landmark case which is usually mentioned in connection with this doctrine is *In re Alappat* (July 29, 1994):

We have held that such 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. *In re Freeman*, 573 F.2d 1237, 1247 n. 11, 197 USPQ 464, 472 n. 11 (CCPA 1978); *In re Noll*, 545 F.2d 141, 148, 191 USPQ 721, 726 (CCPA 1976); *In re Prater*, 415 F.2d at 1403 n. 29, 162 USPQ at 549-50 n. 29.

Three cases are cited in support of this holding. The first one, *In re Freeman*, contains nothing relevant to the purposes of this article, because it doesn't address the "new machine" concept. It is, instead, a case that makes a distinction between process algorithms and mathematical ones, a distinction that in fact does not exist in computer science.[1] Here are the relevant portions of the other two cases:

There is *In re Noll* (November 18, 1976),

There is nothing abstract about the claimed invention. It comprises physical structure, including storage devices and electrical components uniquely configured to perform specified functions through the physical properties of electrical circuits to achieve controlled results. Appellant's programmed machine is structurally different from a machine without that program. It thus broadly corresponds to the combination held to be statutory subject matter in claim 18 in *In re Bernhart*, supra.

Alappat also cited footnote 29 of *In re Prater* (August 14, 1969)

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.

These two cases make explicit references to changes in the physical structure of the computer as a justification that a new machine is made. We also find such a reference to physical structure in another case *Noll* is further citing:

In re Bernhart (November 20, 1969) (emphasis in the original)

There is one further rationale used by both the board and the examiner, namely, that the provision of new signals to be stored by the computer does not make it a new machine, *i.e.*, it is *structurally* the same, no matter how new, useful and unobvious the result. This rationale really goes more to novelty than to statutory subject matter but it appears to be at the heart of the present controversy. To this question we say that if 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 fact that these physical changes are invisible to the eye should not tempt us to conclude that the machine has not been changed.

This notion that physical changes in the computer make a new machine is also found in *WMS Gaming, Inc. v. International Game Technology* (July 20, 1999) (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]

Footnote 3 reads:

[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*.

Please note that *WMS Gaming* refers to *Bernhart* but these two cases state two technically different stories on which physical changes are making the new machine. The *Bernhart* story is that the memory elements are changed. In a modern computer this is electrical charges stored in capacitors because this is how current DRAM memory technology functions. The *WMS Gaming* story is about the opening and closing of interconnected transistors in the microprocessor. This is not the same thing as charging capacitors in DRAM.

The *Alappat* decision was written by Judge Rich. He was also on the panel of every case cited above. Judge Rich also wrote about 20 years before *Alappat* a dissenting opinion in *In re Johnston* (September 19, 1974) (emphasis in the original):

I am quite familiar with the legal doctrine that a new program makes an old general purpose digital computer into a new and different machine. This court has been through that many times and I am not denying the validity of this principle – which partakes of the nature of a legal fiction when it comes to drafting claims. My problem is that, knowing the *invention* to be a new program, I must decide whether it is patentable in any claimed form in view of *Benson,* whether claimed as a machine, a "machine system," or otherwise.

There is also this dissenting opinion from then Chief Judge Archer joined by Judge Nies (emphasis in the original, reference to a footnote omitted)

Thus, a known circuit containing a light bulb, battery, and switch is not a new machine when the switch is opened and closed to recite a new story in Morse code, because the "invent[ion] or discover[y]" is merely a new story, which is nonstatutory subject matter. An old stereo playing a new song on a compact disc is not a new machine because the invention or discovery is merely a new song, which is nonstatutory subject matter. The "perforated rolls [of a player piano] are parts of a machine which, when duly applied and properly operated in connection with the mechanism to which they are adapted, produce musical tones in harmonious combination." *White-Smith Music Publishing Co. v. Apollo Co.*, 209 U.S. 1, 18, 28 S.Ct. 319, 323, 52 L.Ed. 655 (1908). Yet a player piano playing Chopin's scales does not become a "new machine" when it spins a roll to play Brahms' lullaby. The distinction between the piano before and after different rolls are inserted resides not in the piano's changing quality as a "machine" but only in the changing melodies being played by the one machine. The only invention by the creator of a roll that is new because of its music is the new music. Because the patent law does not examine musical compositions to determine their relation to those that have gone before, the distinction between new and old music can never qualify for patent protection.

It is not the computer – the machine qua computer – that performs the [mathematic] function, but, rather, the [mathematic function] is attained only through "use" of the general-purpose computer. The general-purpose digital computer is itself a total and self-complete machine entity. Versatility in electronic data processing is its endowment, its reason for being, its stock in trade.

Digitronics Corp. v. New York Racing Ass'n, Inc., 187 USPQ 602, 640, 1975 WL 21112 (E.D.N.Y.1975), *aff'd on other grounds*, 553 F.2d 740, 193 USPQ 577 (2d Cir.1977)

What Does Computer Science Say?

The facts of computer science side with the dissenting opinions. Most computer professionals would say the technical explanations given in *Noll*, *Prater*, *Bernhart* and *WMS Gaming* are incorrect from a computer science perspective. From the point of view of physical machine structure, no new machine is made when a computer is programmed.

I am aware of the legal theory that a new machine is made because a new machine function is implemented. This is not the same explanation as the one found in the cases cited above but competent lawyers are telling me this explanation is legally correct. From the point of view of computer science, this explanation doesn't work any better than the other one. The technical evidence shows that programming a computer doesn't impart the computer with capabilities it doesn't already have absent the programming.

Judge Rich in his dissent in *Johnston* uses the phrase "legal fiction". I will leave to lawyers the issue of whether this phrase is appropriate here. The purpose of this article is to show that no actual machine is made in any technologically meaningful sense. The legal doctrine that programming a computer makes a new machine is as related to reality as a legislation on the value of the number pi.

Imagine a device equipped with a robotic arm. At the tip of this arm there is a pen, an eraser and a camera. The device may write information on paper with the pen. It may use the camera to read what is written and it decides what to do next based on what is being read. The device may erase what is written and overwrite it with new information. This machine is a general purpose computer. It solves problems by reading and writing information.

Real computers are not robotic arm devices. They are electronic devices built according to the stored program architecture. However the robotic arm metaphor helps understand the functions of the electronic components.

- *Main memory* is a component where information is written and read. This information is modifiable. It is possible to overwrite information with new information. Main memory has no computational capabilities. It is the equivalent of the paper in the robotic arm metaphor.

- The *CPU* or *processor* is the component which carries out the computation. It does so by reading and writing information from main memory. It is the equivalent of the device controlling the robotic arm.

- The *bus* is a communication component between the CPU and main memory. The CPU reads and writes information in memory by sending and receiving electronic signals through the bus. This is the equivalent of the robotic arm with a minor difference. The components playing the parts of the pen, eraser and camera are physically located in the

memory and not the bus as the robotic arm metaphor would suggest. This doesn't change the fact that the CPU computes by reading and writing information in memory through signals sent through the bus.

- *Input devices* allow to bring information from outside of the computer into the memory for computation. This is the equivalent to inserting sheets of written material into the robotic arm computer.

- *Output devices* allow to bring information outside of the computer for outside parties to see and use. This is the equivalent of taking a copy of the information written on paper in the robotic arm computer and giving it to an outside party.

There is an obvious parallel between a computer and a human using pencil and paper. There are differences too. A computer is faster and can handle more information than is practical for a human. A computer can also be embedded in other devices such as an anti-lock brake system to carry out tasks no live human can hope to do. These differences represent an expansion in speed, quantity and circumstances where computations may be practically carried out. They do not represent a change in the nature of what is a computation.

A more fundamental difference lies in the treatment of semantics. When presented with the word FOX a human will know which kind of brown furry animal this is. A computer will only recognize the letter F followed by the letter O followed by the letter X. This is a major limitation. How could computers carry out a computation if meaning cannot be used? The answer is given by mathematics. There is in the theory of computation a special kind of procedures called *algorithms* which are executed without using the meaning of the symbols. It is sufficient to recognize the symbols themselves. An example of an algorithm is sorting text in alphabetic order. If we want to place in order the words FOX and FROG we don't need to know which animals are being referred to. It is sufficient to notice the first letters F are the same and the second letter O comes before the second letter R. Therefore we have determined that FOX comes before FROG without referring to the meanings of the words.

[Update: The preceding paragraph describes the behavior of the computer at the hardware level. The computer is unable to interpret the meanings of symbols because there are no components of digital electronics with this capability which could be included in a computer. Therefore procedures requiring judgment calls based on meanings cannot be implemented in a computer. An example of such unimplementable procedure may be the application of a legal test relying on the discretion of the judge. Mathematical algorithms are how programs work around this limitation of hardware. - End update.]

A computer is a machine for processing binary symbols called bits. A computer may carry out a computation only if the procedure is an algorithm. If a procedure requires to use the meanings of the symbols it is not machine executable. The task of a programmer is to find an algorithm applicable to the problem he wants to solve. Then he will write a description of the algorithm in some programming language. This description is the computer program.

I am aware that this is not the definition of the word algorithm the law usually uses.[2] The legal definition is more or less synonymous with "process". But the definition from the theory of

computation is the one which is appropriate to the task of explaining the operating principles of a computer according to computer science. This is the definition which is used in this article and it is not synonymous with "process".

How Does a Computer Execute a Program?

How does a computer execute a program? Here again computation theory has the answer. Mathematicians have discovered a special kind of algorithm called "universal algorithms" – called that because they can carry out all possible computations. If a function may be computed by an algorithm, any algorithm, then a universal algorithm can also compute it.

CPUs are circuits built to carry out a particular universal algorithm called the instruction cycle. These circuits are designed to recognize sequences of bits called instructions which corresponds to steps in the computation. Each make and brand of CPUs are designed to carry out a limited number of instructions called the instruction set.[3]

The instruction cycle reads an instruction in main memory and executes it. Then it reads the next instruction and executes it. This cycle of successive reading and execution of instructions continues until the computation is completed.

The computer program is initially written by the programmer in a human readable language. This language is called source code. Some programming tools, typically a compiler, will translate the source code into instructions for the CPU. The result is executable code meant for machine execution. When the executable code is stored in the computer's main memory, the CPU may begin its execution.

This completes the explanation of the stored program computer. We may now examine how the structure of the computer relates to the functions of software.

How Computer Structure Relates to Software Functions

Storing executable code in memory doesn't make an electrical circuit dedicated to the functions of the software.

Storing executable code in main memory is like writing symbols on a piece of paper. No computational capability results. Main memory, like paper, is not a component capable of carrying out a computation. This capability belongs to the CPU writing in memory, or to the human who handles the pencil.

Introducing code in memory doesn't affect the CPU. The information stays in memory, unused, until the program execution begins. Then the CPU reads the first program instruction and execute it. Then it reads the second instruction and executes it. The CPU continues reading instructions one by one and executes them one by one until it is instructed that the end of the program is reached. This instruction cycle the same for every program.[4]

The number of instructions in the instruction set is limited. Here the phrase 'instruction set' doesn't mean the program. It means the list of instructions the CPU is capable of executing.

This list is found in technical manuals where the functionality of the CPU is documented.

The CPU has in its circuitry all the structure necessary to carry out all instructions in the instruction set. All programs are required to use only instructions from this set. There is no operation of reading in bulk an entire program from memory into the CPU because the instructions are read and executed one by one. No portion of the computer is ever configured to have a structure dedicated to the execution of a program because programs are executed one instruction at the time. From the point of view of the computer structure, there is no concept of an entire program. From the point of view of the circuit, there is only one instruction that matters, and it is the current instruction. The notion of a program is an abstraction separate from the circuit structure. This is why the CPU has the structure necessary to execute all programs.

The mechanical arm computer metaphor explains well what happens. We have a mechanical arm machine built to execute a universal algorithm. It is programmed by writing executable code on a sheet of paper and introducing this sheet in the machine. This doesn't change the structure of the mechanical arm machine and the paper by itself doesn't compute. The program is executed when the mechanical arm carries out the instruction cycle algorithm on this data.

A consequence of this observation is that we cannot conclude a new machine has been made on the basis that some circuitry dedicated to the software function has been configured because no such configuration occurs.

New functions for software may be implemented by imposing limitations on the inputs of an old algorithm.

The function of doubling a number is a simple example. We may multiply the number by two, or we may add the number with itself. Either way the same function is accomplished: the number is doubled.

In this example we have a choice of two algorithms, one which computes an addition and another which computes a multiplication.[5] If we build a pair of circuits, one for addition and one for multiplication, both are suitable but none is dedicated to doubling a number. This result is achieved by imposing a constraint on the inputs given to the algorithms. In one case the number must be added with itself; in the other case the number must be multiplied by 2.

Computers programmers know many algorithms which could be used by imposing restrictions on the inputs to compute new functions. For example, some algorithms may make make a decision by comparing data with the contents of a decision table. The contents of this table is not an algorithm, and it is not executable data. But if we change the contents, a different decision is reached. This may be used to implement several different software functions depending on the particular type of decision which must be reached. Another example are algorithms for processing regular expressions which may perform various text processing functions depending on input.

Computers are programmed in this manner, by requiring the input of a universal algorithm to be a specific program. While the source code of a program may literally express a new algorithm, the corresponding executable code is an input to the instruction cycle. It is not a

blueprint or the description of a new circuit implementing the new algorithm.

A consequence of this observation is that we cannot conclude the algorithm or the circuit implementing the algorithm are new on the sole basis that the software *functions* are new.

The functions of software may be represented in memory by data which are not executable CPU instructions.

This is a consequence of the previous point. Consider the decision table example. The function of this algorithm depends on the contents of the table, and this is not executable data.

Programmers are not obligated to use the native instruction cycle of the CPU as their universal algorithm. They often use this instruction cycle because this typically results in faster programs. But the alternatives are also frequently used.

Universal algorithms are algorithms. They may be written in software and compiled into executable code for the CPU. The computer so programmed may be used to execute software written to the alternative universal algorithm. A common case is virtual machines running bytecode. Programming languages such as Python are running on such virtual machines. Executable Python programs are not instructions for the CPU. They are bytecode instructions for the virtual machine. The CPU executes instructions for the virtual machine, which is interpreting Python bytecode.

Not all universal algorithms are instruction cycles executing on instructions. Other types of universal algorithms are also used sometimes. The family of *logical programming* languages such as Prolog rely on SLD-resolution. The family of *functional programming* languages rely on various implementations of beta-reduction strategies. These universal algorithms are not instruction cycles.

The consequence is that legal arguments about software should not assume the executable program is instructions directed to the CPU. They should not even assume programs are made of instructions at all. If they make these assumptions the arguments are not applicable to the full range of tools and techniques which may be used in software development.

Does a Clock That Ticks Become a New Machine?

Programs may be modified in memory while they are executed.

The contents of the memory of a computer is constantly modified as the CPU executes instructions, because memory is where all kinds of data resides, not just programs. The CPU carries out computations by constantly writing and overwriting data in memory. This may happen as often as billions of times per second on the current generation of computers. If your desktop clock changes from 11:59 AM to 12:00 PM, memory has changed (not to mention the counters that are changing constantly just to know that time has passed). Displaying new results from a search engine is changing the memory contents. Moving the mouse, typing on the keyboard are all operations that change the memory contents. And every computation a

program carries out similarly changes the memory contents.

All contents of memory may be modified including the program which is being executed. While it is usually considered bad practice to change a program during execution, this is technically possible and there are circumstances where this actually happens. For example the patent number RE38,104 which was litigated in the recent *Oracle v. Google* case is claiming such a programing technique. How do we distinguish a memory change which make a new machine from one which doesn't? There is no objective test because there is no basis *in technology* to make such a distinction.

The practice of avoiding changes to executable code during execution is applicable to programs using *imperative programing* languages. This prohibition is not applicable to all types of programming languages. *Functional programing* languages rely on some implementation of beta-reduction as the universal algorithm. In this case, the program is a data structure which must constantly (thousands or millions of times per second at least) be modified according to the rule of beta-reduction for the computation to progress. Constant incremental modification of the program is part of how this particular kind of universal algorithms works.6

These changes in memory cannot make a new machine because otherwise no machine would be able to carry out a computation until the end. As soon as the memory is modified it would no longer be the same machine. Think of a car riding on the road. The wheels are rotating, the engine is running. These are physical changes. Do the moving parts make a new car every time they move? No car goes anywhere by that logic, because a car constantly keeps becoming a different car while it runs. If a computer physically changes when programmed then so does a light bulb when it's turned on and off in a Morse code pattern. So does an internal combustion engine rotating. Are we going to argue a clock becomes a different machine every time it ticks? This is a physical change too. How many examples do we need before this notion is discredited?

These arguments are similar to arguing that shifting the lever controlling the reverser wheel of a steam engine transforms the steam engine into a new machine - *while it is travelling at over seventy miles an hour, under power!* This argument is obviously absurd, but it is entirely analogous to the idea that directing a general-purpose computer to execute a new program transforms the computer into a new machine. Instructions and data are the normal inputs of a stored program computer instruction cycle, in the same way that coal and water are the normal inputs of a steam locomotive. A computer is useless without a program to run and a set of data to act on; a steam locomotive will go nowhere without being fed coal and water first. And programs are brought into the computer memory while the computer is powered on and running, by clicking on an icon or selecting a menu item. Sometimes just visiting a web site will bring a JavaScript program in the memory without the knowledge of the user.

Physical changes which are required for a machine to perform its function are not changes in machine structure. They don't make new machines. Introducing a program in the memory of a computer is this kind of change.7

Programming a computer is a reversible operation.

It is worth noting that whether or not a program is self-modifying, writing a program in a computer memory is a temporary change. The memory will revert to its previous state when the program is erased, often by powering off the computer. Programs are also erased when the program is done executing and the memory is freed for other uses. It is not simply that the installation of a program changes the computer in ways not visible to our eyes. It is that the changes are temporary, sometimes fleeting, in nature, and this conflicts with the ordinary understanding of what is meant by the manufacture of a machine.

Why no new physical structure is made when programming a computer

At this point it must be clear that the previously stated arguments based on the physical structure of the computer which are found in case law are technically incorrect. No new machine structure is made when a computer is programmed.

The main memory of a computer is typically DRAM. Storing data in DRAM involves storing an electrical charge in capacitors. Other types of memory could be used as well: SRAMs are used in CPU caches, magnetic domains are used on magnetic hard disks etc. This is why bits are a symbolic abstraction different from their physical representations. But still, we do not err when we consider primarily the scenario where main memory is DRAM. Nowadays, this is the technology used to make most computers main memory.

How Case Law Gets the Technology Wrong

Prater is wrong because a stored program computer is not a storeroom of parts that can be wired into a different configuration by 'introducing a program'. When the program is introduced, it changes the patterns of voltages on the memory cell capacitors. The configuration of the computer wiring and all of the logic circuits remains fixed.

Bernhart and *Noll* are wrong for the same reason.

The footnote in *WS Gaming* is wrong. The notion of associating bits with transistor implemented switches is a mere teaching aid for beginners. The correct understanding will be acquired over time as we learn more about digital electronics. This is like first learning that a tomato is a vegetable and then, when the correct botanical knowledge is acquired, we learn that a tomato is a fruit and a berry. The wiring between the transistors is etched at the factory and never changes. There is no digital electronics equivalent of an ancient ENIAC patch panel in a stored program computer. Transistors work by changing voltages in wiring that never changes. Also, storing bits in DRAM does not changes voltages between transistors. The activity of transistors is *executing* the program. It is not *programming* the computer.

All of these cases make a more fundamental error. They ignore the actual operating principles of the stored program computer. They ignore that all programs are input to the instruction cycle. They ignore that from a technology perspective there is no difference between bits representing data and bits representing programs. They ignore that computers are unable to interpret the

meanings of symbols. They ignore that the meanings of symbols must be abstracted away by the programmer when he writes a program.

The correct operating principles of stored program computers has consequences. One of them I have already mentioned: programming a computer does not change the machine structurally in any technologically meaningful sense. But this is not the only consequence. There are also consequences on how functionality relate to the machine structure. Software is input, like any other data input. Functionality results from giving input to a known algorithm. But this explanation covers only the ability to implement the purely mathematical aspects of algorithms. There is no way to associate the non-mathematical meanings of the data with machine structure. Any aspect of the software functions which relates to the non-mathematical meaning of the data is not something that results from the machine structure.

The Legal Theory that New Computer Functionality Creates a New Machine is Technically Incorrect

It is possible to have new uses for old machines.

How do we distinguish between a function which makes a new machine from a new use of an existing machine? A truck may carry log wood. Then the function of the truck is to carry log wood. If the truck carries sacks of sand, its function is to carry sacks of sand. But this is not as a result a new truck. It's just carrying a new load. A similar point may be made about computers, because programs are very much like payloads. They are inputs for the instruction cycle or some other universal algorithm. We may have new software for old computers without making new computers in the same way we may have new uses for trucks without making new trucks.

A sewing machine is another example. One may change the types of threads and the directions and distances moved by the needle to cause the machine to do different things. The default configuration may let you stitch in straight lines. Another configuration may let you sew buttons on a garment. Or you may use a zigzag stitch. A sewing machine has a variety of functions which may be used in various combinations to obtain a broad range of desired effects. One may apply force to turn a straight stitch into a contour stitch. Or one may apply a decorative thread as in embroidery. None of these activities makes a new sewing machine.

When people believe software makes physical changes to the computer structure, they have an explanation for the new machine doctrine. They argue that new functionality is associated with the physical changes. But once this explanation is rejected, how could it be that new software functions cause the computer to be new? The simple, and obvious, truth is that the application of a device to a particular purpose does not transform the device.

It is possible to have new uses for old algorithms.

This is a different take on a similar theme. The relationship between function and algorithm is not always immediate and straightforward. As I have previously mentioned, a function may be implemented by giving some data as input to an existing algorithm, be it universal or otherwise. It is not because the function is new that the underlying algorithm is new. We may just have a

new input for an old algorithm. And if the algorithm is not new, even when one believes a new algorithm makes a new machine there is still no reason to believe the machine is new.

We may have new algorithms for old functions.

As a different but related point, it is possible for a programmer to develop a totally new algorithm and use it to carry out an old function by giving it a well-chosen input. If this old function is recited in someone else's patent, depending on how the claim is written, it may be possible that the old claim reads on the new algorithm. In this scenario the old claim would read on an algorithm the patentee has not actually invented.

The meaning of the data is not used by the computer.

A computer has no ability to read meaning in the symbols representing the data. This is why the computer may execute a procedure only if it is an algorithm. When the programmer writes his program, he must abstract away the meaning of the data in order to get a machine executable algorithm.

This imposes limits on what the functions of the computer could be. It may only act on the raw uninterpreted symbols. Once an answer is reached, the symbols must be outputted to an outside party for interpreting the meaning. Then, how about those patents that describe the functions of software in terms of the meaning of the data? On what reasonable basis could courts conclude these functions are new functions of the computer?

If we configure a printing press to print a new novel, for example, do we say it has a new function which is to print this specific novel? There is a mathematical function called the identity function which is defined by the formula $f(x)=x$. It produces as output the same value as the input it receives. An algorithm that implements this function is an algorithm that makes an exact copy of its input. A printing press may be seen as a machine dedicated to executing an algorithm of this kind. It makes an exact copy of the printed letters. Computers can replicate digital information exactly; they can act like printing presses. The relationship meanings have with the printing press is a particular example of the relationship meanings have with all algorithms. Then, why would a function defined in terms of the meanings of the bits be deemed a function of the computer? These machines are processing raw uninterpreted bits. They don't process meanings.

An analogy with the player piano is apt here. A player piano has no aesthetic sense. It doesn't appreciate the creative work in a Brahm's lullaby. There is no reason to believe a player piano is imparted the function of playing a Brahm's lullaby when it is given the corresponding piano roll. The player piano just does what it was designed to do, mechanically playing the sounds are they are recorded on the roll. Similarly the computer has no sense of the meanings of the data. It just mechanically manipulates the symbols according to the algorithm. Meanings have been abstracted away by the programmer when he wrote the program. He needed to, because if meanings are not abstracted away, the procedure is not machine-executable. Part of the creative process of the programmer is precisely to find an algorithm which computes correctly without having to use the meaning of the data. Like the beauty of the melody on a piano roll, the

meaning of the data can only be interpreted by an outside observer.

The function of a stored program computer is to carry out all the functions of all possible software.

There are technical manuals documenting the functionality of a CPU. These manuals describe the exhaustive list of all instructions the CPU can execute. The function of a CPU is to carry out all possible computations which are expressible by a combination of these instructions. This is clearly what CPUs are designed to do. This is consistent with the mathematical observation that the instruction cycle is a universal algorithm.

A CPU has all the structure necessary to execute the instruction cycle. This means it has all the structure to execute all programs written using these instructions. Introducing a program into memory does not add more structure to the computer. This is consistent with the notion that writing data in memory doesn't add structure. Deciding otherwise leads to logical inconsistencies like the one I have mentioned about programs which are constantly modified as they are executed.

From this perspective, introducing a program into a computer doesn't give the computer a functionality it doesn't already have. The computer is already capable of executing the program. The program itself may be new in the sense that it may have never been written before but the inherent capabilities of the machine are not new or changed.

Here is another logical inconsistency in the legal view. According to the legal theory, introducing software into a computer imparts the computer with the functions of the software. What if the software implements a universal algorithm? Let's say it is a Python virtual machine. Then the function being imparted is the ability to carry out all possible computations provided it is given the program as input. This is the function of a Python virtual machine. Then, if we apply the legal logic, it is possible for a programmer to make a machine with such functionality. He just has to introduce a Python virtual machine in the computer memory. Where is the reason to believe a hardware engineer can't make a circuit with similar functionality?

Here is yet another logical inconsistency. I argue that software is mathematics because the computer always executes a universal mathematical algorithm. Lawyers often reply that the mathematics of computing is a description of the computer, like the mathematics of mechanical engineering is a description of pulleys and levers. Even if this were correct, what would this mathematical description say about the function of the computer? -- It is that it carries out a universal algorithm. It can compute all the computable functions provided it is given a program as input. One can't deny this is the function of the computer while simultaneously arguing that the mathematics of computing is a description of the physical computer.

The function of a program is not a function of the underlying computer

According to the operating principles of a stored program computer, the functionality of software is implemented by a combination of two actions: (a) meanings related to the application are given to the bits and (b) data is given as input to an existing algorithm. Let's see what this is doing to the functional relationship between the program and the electronic

structure of the underlying computer.

1. Meanings related to the application are given to the bits.
   The computer is unable to process meanings. It can only process uninterpreted symbols. The hardware of a computer has no more relationship with the meanings of symbols than a printing press has with the meaning of the printed matter. Functionality defined in terms of meanings is not related to the electronic structure of the computer. The part of software functionality which concerns meanings is not the result of the operations of digital electronics.

2. Data is given as input to an existing algorithm.
   All data may be given as input to an algorithm to define functionality. In mathematics, this is a standard way to define new functions from known algorithms. All algorithms and all data could be and are used in this manner. For example, if the functionality is to double the number, would the number 2 count as something which brings new functionality to a multiplying circuit? If we are to double the numbers, one of the inputs must be 2. If we don't supply the number 2, the circuit will not double the number. If we use the number 3 instead of 2, the function would change to tripling a number. Similarly, all software has a relationship of this nature with the instruction cycle. It is data given as input. And it is common for programmers to implement functionality by controlling the inputs to known algorithms which are not the instruction cycle.
   Here we have a dilemma. Does all data given as input have the ability to change the function of the electronics? Could the bits for the movie *Citizen Kane* count as something which imparts the function of playing *Citizen Kane* when given to a video player? Or is it that some inputs impart functionality and some don't? How do we tell the difference? There is no objective criteria in technology to solve this riddle.

The concept of software imparting a new function to the computer has no basis in technology because: a) computers are unable to process meanings; and, b) the inherent capabilities of the machine encompass the range of all possible inputs, this includes all possible programs.

At this point is must be clear that the facts of technology do not support the legal theory that when software has new functionality then a new machine is made. I mention below a pair of additional points worthy of mention.

The Functional v. Non-Functional Data Theory
All non functional data may be represented by executable instructions

How do we define the distinction between functional (software) and non-functional (not software) data? This distinction has no basis in technology. As observed before, all types of data may be functional when used as input to a suitably chosen algorithm. Conversely algorithms may always be used to represent data which is normally viewed as non-functional.

For example, consider MIDI files for storing music. The files contained "events" which are instructions on how to generate a sound, say a note. A series of events in the file represents a tune. Playing a MIDI file is going through the events one by one and generating the corresponding sound. Can we say the function of a specific MIDI file is to play a specific tune?

The contents of this file can certainly be seen as an algorithm for playing the tune. Why can't we say the function of the MIDI file is to play a particular tune?

Another example is the PDF and PostScript file formats which are instructions for printing the visual appearance of a document on a page. A PostScript file is a printing algorithm for the document. Why can't we say the function of the PostScript file is to print this particular document?

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

This is an algorithm for printing Shakespeare's *Hamlet*. Similar algorithms may be written for all literary work.

Cloud computing -- where is the particular machine?

Cloud computing raises issues of its own. If a patent is written to software which requires a user's computer to interact with a sea of servers over the Internet, can we pinpoint a *particular* machine or apparatus that infringes a machine patent? How can a network of devices on the Internet be *particular* new machines?

Conclusion
For all the above reasons, case law conflicts with known technological facts. This is a correctable problem. After all, the same legislature that thought it could mandate legally the value of pi in conflict with its actual value, were persuaded, upon encountering more information, to table and hence kill off the unrealistic and ludicrous law. As the local paper quoted one legislator on the ill-advised bill, "the Senate might as well try to legislate water to run up hill as to establish mathematical truth by law."

When legal fictions contradict known technological truths, it doesn't change the technological truths. It merely puts the law outside of reality.

---

1 In his dissent, Judge Archer states: (see footnote 28, emphasis in the original)

The *Freeman* case cited by the majority did not hold that a general purpose computer when programmed becomes a special purpose computer and a "new machine" within § 101.

Freeman reads in part like this: (emphasis in the original, some footnotes omitted)

The fundamental flaw in the board's analysis in this case lies in a superficial treatment of the

claims. With no reference to the nature of the algorithm involved, the board merely stated that the coverage sought "in practical effect would be a patent on the algorithm itself." Though the board gave no clear reasons for so concluding, its approach would appear to be that every implementation with a programmed computer equals "algorithm" in the *Benson* sense. If that rubric be law, every claimed method that can be so implemented would equal nonstatutory subject matter under 35 U.S.C. § 101. That reasoning sweeps too wide and is without basis in law. The absence, or inadequacy, of detailed claim analysis in the present case is further illustrated by the conclusion that "the novelty resides in the program" when, as here, the claims recite no particular computer program. In the present case, it is not the claims but the *specification* that discloses implementation of the claimed invention with computer programs.

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." 409 U.S. at 65, 93 S.Ct. at 254, 175 USPQ at 674, 409 U.S. at 65, 93 S.Ct. at 254, 175 USPQ at 674 (emphasis added).

…

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.

The manner in which a claim recites a mathematical algorithm may vary considerably. In some claims, a formula or equation may be expressed in traditional mathematical symbols so as to be immediately recognizable as a mathematical algorithm. See, *e. g., In re Richman*, 563 F.2d 1026, 195 USPQ 340 (Cust. & Pat.App.1977); *In re Flook*, 559 F.2d 21, 195 USPQ 9 (Cust. & Pat.App.1977), *cert. granted sub nom., Parker v. Flook*, ___ U.S. ___, 98 S.Ct. 764, 54 L.Ed.2d 780 (1978). Other claims may use prose to express a mathematical computation or to indirectly recite a mathematical equation or formula by means of a prose equivalent therefor. See, *e. g., In re de Castelet, supra* (claims 6 and 7); *In re Waldbaum*, 559 F.2d 611, 194 USPQ 465 (Cust. & Pat.App.1977). A claim which substitutes, for a mathematical formula in algebraic form, "words which mean the same thing," nonetheless recites an algorithm in the *Benson* sense. *In re Richman, supra*, 563 F.2d at 1030, 195 USPQ at 344. Indeed, the claims at issue in *Benson* did not contain a formula or equation expressed in mathematical symbols. When considered as a whole, each of the claims in *Benson* did, however, recite in prose a formula for converting binary coded decimal numbers into binary numbers.

The "local positioning algorithm" described in appellant's specification is the order of steps in processing the hierarchical tree structure and spatially relating the various characters to be displayed. Appellant has thus used the term "algorithm" as a term of art in its broad sense, *i. e.,* to identify a step-by-step procedure for accomplishing a given result.[8]

The method claims here at issue do not recite process steps which are themselves mathematical calculations, formulae, or equations. Each of claims 8, 9, and 10 merely defines a new, useful, and unobvious process for operating a computer display system. The board, therefore, erred in its reliance on *Benson* as its sole basis for concluding that the present method claims are drawn

to nonstatutory subject matter.

Footnote 8 reads:

[8] The preferred definition of "algorithm" in the computer art is: "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." C. Sippl & C. Sippl, Computer Dictionary and Handbook (1972).

2 See *Freeman* in footnote 1 above for example.

3 This meaning of the phrase "instruction set" must not be confused with "computer program". I have seen patent attorneys calling programs "set of instructions". But in computer architecture the phrase "instruction set" refers to the collection of instructions which are permissible in a program because they are the ones a CPU can recognize. It does not refer to the program itself.

4 Compare this with what the court have said in *WMS Gaming*: (emphasis in the original)

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]

This refers to footnote 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*.

The fallacy is three-fold.

The first error is that the instructions are executed one at the time. The so-called electrical paths last no longer than the time required to execute one instruction. Then they are replaced by other paths for the next instructions. Modern computers may execute billions of instructions per second. The so-called electrical paths are very transient in nature.

The second error is that "electronic switches" and "electrical paths" are a metaphor which must not be construed literally. In a microprocessor all the actual electrical paths are permanently etched at the factory and can never be changed afterwards. The transistors change the voltages in these wires. The current may flow or not flow according to the changes in voltages as if switches were opened and closed but transistors are not actually switches and there is no creation of actual electrical paths that were not previously there.

The third error is this transistor activity is the execution of the computer program. It is not the

act of programming the computer. The computer is programmed by storing the instruction in main memory. This operation must be completed before the first instruction of the program is executed.

5 Most people will think of addition and multiplication as single indivisible steps. However these procedures have some complexity and require multiple steps to handle the individual digits, especially when the numbers are large. The computation of an addition or a multiplication is carried out by means of an algorithm and the portions of computer circuitry for such calculations are hardware implementations of these algorithms. Perhaps the complexity and algorithmic character of ordinary arithmetic is more apparent when one considers the long division algorithm. Other arithmetic algorithms are multiplication algorithms and also Russian peasant multiplication. When the numbers are represented by binary numerals, this is computer bits, algorithms suitable for binary arithmetic must be used.

6 Several such universal algorithms are decribed in this book: Kluge, Werner, *Abstract Computing Machines, A Lambda Calculus Perspective*, Springer-Verlag Berlin Heidelberg 2005.

7 Please compare this with the explanation from *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. The fact that these physical changes are invisible to the eye should not tempt us to conclude that the machine has not been changed.

I have read this quote as referring to main memory. From the technical perspective it is clearly incorrect for the reasons stated in the article. What if we construe the phrase "memory elements" as referring to a computer hard disk?

A hard disk is a device where information is read and written. It is much slower than main memory. It is not directly connected to the CPU so it cannot be used as a source of instructions for the instruction cycle. A hard disk is a peripheral device which is accessed through input and output operations.

According to the stored program computer architecture programming the computer is introducing the program in main memory. Programming the computer is not installing or storing the program on the hard disk. Programs on a hard disk are not directly executable because the CPU cannot read the instructions directly from the hard disk. In order to execute a program on hard disk the user must first issue a command to bring it into the main memory. The user may, for example, double click on an icon or select an item in a menu. Then the operating system will locate the program on the hard disk and bring it into main memory for execution. This operation is not a simple copy of the bits. Some transformations may have to be made, like linking the program with libraries stored in memory and resolving physical addresses. A program on hard disk is not in its final executable form until it has be loaded in the computer main memory.

The task of putting a program on hard disk where the operating system may locate it and bring

it into memory on the user's behalf is called *installing* the software. This task is different from the operation of programming the computer as defined above. Programs may also be removed from the hard disk. The install utility which comes with most software often allows to *uninstall* the program, this is to remove it from the hard disk.

Like main memory, hard disks are writable devices which are constantly updated as the computer operates although at a much slower rate. Treating changes in hard disk as the making of a new machine is like treating changes in main memory as the making of new machines. This is conflating the normal operation of the machine with changes in its structure. But in the case of a hard disk there is no possibility to argue that the changes affect the computation the computer is able to carry out because programs cannot be executed directly from the hard disk.