What Does “Software Is Mathematics” Mean? - Part 1

Software Is Manipulation of Symbols

by PolR

You probably have heard computer professionals say that software is mathematics. You've certainly read it on Groklaw more than once. But is it true? What does that statement mean? I want to show you why it's true, and I will also answer some typical criticisms.

Mathematics is a language[1]. Software is mathematics because, according to the principles of computation theory, the execution of the software is always a mathematical computation according to a mathematical algorithm. That is, the execution of a computer program is the computer making utterances in mathematical language.

Some people think the sentence "software is mathematics" means software is described by mathematics, and then they say that everything could be described by mathematics. In that sense, they say, everything would be mathematics. If we push that logic to its conclusion nothing would be patentable, because mathematics is not supposed to be patentable subject matter. They say this is an absurd result contrary to law.

This is not what we mean when we say software is mathematics. If we want, we can describe software with mathematics. Sometimes we do. This is not the reason we say software is mathematics. We say the execution of the software *is the utterance* in mathematical language as opposed to the meaning of the utterance. An utterance *is mathematics*. The meaning of the utterance is *described by mathematics*. This is one important difference between these two phrases as I hope to now show you.[2] This point is related to the distinction between three types of mathematical entities: formulas, algorithms and computations.

I'll show you that using $E=mc^2$.

The famous equation $E=mc^2$ is a mathematical *formula*. It is text written with mathematical symbols in mathematical language. This *is* an utterance in mathematical language. It analogous to what we call a (declarative) sentence in English grammar. This formula *is mathematics*.

The meaning of this formula is a is a law of nature, actually a law of physics. It is a statement relating the mass of an object at rest with how much energy there is in this object. The energy of the object and its mass are *described* by the utterance in the language of mathematics. The object, its mass and its energy, are not mathematics. They are described by mathematics because they are what the formula means.

The formula implies a procedure to compute the energy when the mass is known. Here it is:

1. Multiply the speed of light c by itself to obtain its square c^2.
2. Multiply the mass m by the value of c^2 obtained in step 1.
3. The result of step 2 is the energy E.

This kind of procedure is known in mathematics as an *algorithm*. The formula is not the algorithm. This *procedure* is the algorithm. Someone with sufficient skills in mathematics will know the algorithm simply by looking at the formula. This is why it is often sufficient to state a formula when we want to state an algorithm.

This algorithm is a procedure equivalent to making a logical deduction. Suppose we know the mass, How can we know the energy? We should be able to deduce it from the relationship stated by the formula. The algorithm is how we proceed with this deduction. When we use an algorithm to solve a problem we actually use a form of logic.[3] Algorithms too are part of the language of mathematics.

The task of carrying out the algorithm is called a *computation*. When carrying out the algorithm

with pencil and paper we have to write mathematical symbols, mostly digits representing numbers but also other symbols such as the decimal point. These writings too are utterances in mathematical language. In the example, the meaning of the utterances are numbers representing the speed of light, its square, the mass and the energy of an object. Carrying out the algorithm is mathematics because it is making these utterances.

But what if we use a machine? We may use a digital device such as a pocket calculator or a computer to carry out the calculation. Then the symbols are no longer marks of pencil on paper. They are bits in some digital electronics circuit. This too is utterances in mathematical language, but now the machine and not the human is making the utterances. According to the mathematical principles underlying the stored program architecture the execution of all computer programs is the execution of a mathematical algorithm, therefore this execution is a mathematical computation. This is what "software is mathematics" means.[4]

This was an overview of the key ideas. Let's now elaborate.

Mathematics Is a Language

Some people, I am sure, will wonder what I mean when I say mathematics is a language. For them mathematics is about mathematical subject matter like numbers, geometric figures or abstract set theory.[5] This view is correct but mathematics is more than this. Serious mathematical work requires to write symbols on paper or bits in a computer. The linguistic aspect is unavoidable. On the other hand these symbols are pointless without their mathematical meanings. This is really like the two sides of the same coin. One cannot exist without the other. In this sense mathematics is a language even though mathematics is also the study of mathematical subject matter.

There is also the issue of foundations. There is a branch of mathematics called mathematical logic. This is where mathematicians define the foundations of their discipline. What is a mathematical formula? What is a theorem? What are the criteria of logic that must be met for a mathematical proof to be valid? These are some of the foundational questions answered by mathematical logic. There is also a branch of mathematics called the theory of computation. This is actually a subbranch of mathematical logic. This is where mathematicians define what is an algorithm and what is a computation according to an algorithm. If you go read these definitions in textbooks from competent authors you will find they are all elements of the language of mathematics. The definitions expressly refer to the symbols, their arrangement in syntax and their semantics. All of mathematics ultimately rely on these foundations. In this sense, mathematics is indeed a language.

Symbols Are Abstract Ideas

Languages are written with symbols, typically letters, digits and punctuation marks. Mathematical language adds to this list various mathematical symbols.

There is a difference between the symbols and their physical representations. Think of letters for example. We could say letters are marks of ink on paper but we would be wrong. If we use a pencil they are marks of lead on paper. Or when you use a computer they are arrangements on pixels on the screen. If you walk down a city street you may see on buildings neon signs and carvings in stone. Symbols are abstract ideas. We can recognize them when we see their physical representations. But still, the symbols are not the representations.

Bits are symbols. Like letters they can be represented multiple ways. In pathways between transistors they are voltages. In main memory they are electrical charges stored in capacitors. On hard disks they are magnetic moments. On some optical disks they are cavities in the surface. There are many other forms bits can take. Like letters, bits are abstract ideas.

Symbols Need Not Be Watched by Humans to Have Meanings

Some people expect symbols to be something visible to humans. This view is too strict. Symbols

need not be visible to the human eye to carry their meanings.[6] While bits are not normally visible they may be indirectly observed by a programmer using debugging tools if he wishes so. Then the programmer may read them and understand their meanings.

An extreme example are the cuneiform tablets from the ancient Mesopotamian civilizations. Thousands of these tablets were buried in the sands of the Middle-East for centuries, unknown to all men living during this period. The ancient languages were forgotten. But still, archaeologists were able to find the tablets and decipher them. What happened to their meanings in all these years where no live human neither knew the language nor were even aware that there were tablets buried there? The meanings didn't disappear. They were patiently waiting for the archaeologists.

Something similar happens to Groklaw comments. After they are typed and the commenter clicks on the “submit” button, a series of electronic adventures begins. The bits are sent over the Internet and must go through countless communication wires and equipments before reaching the Groklaw database in some data center. Then the comments are stored in a database kept on magnetic storage. When a reader clicks on the link to display the comment, more electronic adventures occur as the comment is transfered across the Internet from the Groklaw database to the user's screen. The comments are unobserved when they are stored in the database and they are unobserved when they travel across the Internet. But, when they reach destination, the comments still have their meanings.

Computations Don't Process Electrons, Computations Process Symbols

Let me tell you some fundamentals of digital electronic and the underlying principles of mathematics. I hope this will clarify the difference between hardware and symbols.

Bits are from a two-symbol alphabet, written as 0 and 1. The relevant branch of mathematics is boolean algebra which is a part of logic. The bits stand for truth values, 0 stands for *false* and 1 stands for *true*. Boolean algebra also recognizes three operators, *and*, *or* and *not*, which could be applies to truth values. These operators correspond to the ordinary operations of logic of the same name.

Let me describe some algorithms for each boolean operators.

The *and* operator accepts two arguments. It is is *true* when both arguments are *true*. It is *false* otherwise. The corresponding algorithm is:

- Read two symbols as input.
- If both symbols are 1 then the answer is 1.
- Otherwise the answer is 0.

The *or* operator also accepts two arguments. It is *true* when either or both arguments are *true*. It is *false* otherwise. The corresponding algorithm is:

- Read two symbols as input.
- If both symbols are 0 then the answer is 0.
- Otherwise the answer is 1.

The *not* operator reverses the truth value of its sole argument. The corresponding algorithm is:

- Read one symbol as input.
- If the symbol is 1 then the answer is 0.
- Otherwise the answer is 1.

There are many more operations permitted in boolean algebra. They are all computed by combination of the three elementary operators. Everything in boolean logic can be done by assembling multiple instances of these three simple algorithms into a bigger algorithm.

Please look at these algorithms attentively. You will see that they are all operations on the symbols alone. They can be executed without referring to their meanings. All that is needed is to identify whether the symbols are 1 or 0 and act accordingly. There is no need to decide whether they correspond to any truth value. This is typical of all mathematical algorithms. Mathematicians have defined their criteria for what is a mathematical algorithm as opposed to some other kind of procedure. This observation is a consequence of some of these criteria.[7]

Boolean operators may be implemented by means of digital electronics precisely because there is no need to refer to the meanings of the bits. If electronic signals are treated as representation of symbols the activity of the circuit is the mirror image of the algorithms.

For example, let's assume an engineer decides that a voltage of 0V means the bit 0 and a voltage of 0.5V means the bit 1. He also decides he will not use other voltages. He can build circuits called logic gates which correspond to the boolean operators.

The AND gate implements the boolean *and* operator:

- The circuit takes two voltages as input on two incoming wires and produces one voltage as output on one outgoing wire.
- If both incoming voltages are 0.5V the outgoing voltage is 0.5V.
- Otherwise the outgoing voltage is 0V.

The OR gate implements the boolean *or* operator:

- The circuit takes two voltages as input on two incoming wires and produces one voltage as output on one outgoing wire.
- If both incoming voltages are 0V the outgoing voltage is 0V.
- Otherwise the outgoing voltage is 0.5V.

The NOT gate implements the boolean *not* operator:

- The circuit takes one voltage as input on one incoming wire and produces one voltage as output on one outgoing wire.
- If the incoming voltage is 0.5V the outgoing voltage is 0V.
- Otherwise the outgoing voltage is 0.5V.

Please take the time to check that the operations on the voltages are indeed the mirror image of the algorithms. When voltages are interpreted as bits as decided by the engineer, the action of the circuit is to carry out operations of boolean logic. More complex algorithms may be implemented by assembling multiple logic gates in a more complex circuit. This is one of the principles of digital electronics. Circuits manipulate symbols by manipulating their physical representations as voltages.

Now ask yourself, what happens if the engineer reverses the convention? What happens if he decides that 0V means 1 instead of 0? Of course this implies that 0.5V would mean 0 instead of 1. Would the circuits still be the mirror image of the algorithms? The answer is “yes”, except that the AND gate is now computing the *or* operator while the OR gate now computes the *and* operator. The NOT gate still computes the *not* operator. In short, the AND and OR gates have swapped their roles. Please look again at the gates and see for yourself.

What does this mean? It means the bits are not the voltages. One cannot tell which algorithm is computed by a circuit just by looking at what the circuit does to the voltages. The same circuit will implement two different algorithms depending on how the voltages are interpreted as symbols.

As I said before, bits are symbols. They are abstract ideas. They can be represented by any voltages an engineer may chose. The pair 0V and 0.5V is a common industry standard but other standards may exist. The pair 0V and 0.35V is another common one. Bits may also be represented

by something which is not voltages. In main memory they are electric charges in capacitors.[8]

Some people think a computation is the electronic process, the transistor activity manipulating electricity which occurs inside the circuit. They are wrong. A computation is about processing bits. It is about the symbols. It is not about voltages or electrons.

Bits Must Be Organized Into Syntax

On cannot randomly put together a bunch of symbols and expect them to mean something. Symbols must be grouped together according to rules of syntax. Something like $E=mc^2$ is a mathematical formula while *=e)Ru+* is not. This is partly because $E=mc^2$ complies to the rules of mathematical syntax while *=e)Ru+* does not.

Similarly bits in a computer must be organized according to rules of syntax. Often they are grouped to represent numbers. There are several ways to do this.

One possibility is called “unsigned integers”. Under this convention bits represent natural numbers 0, 1, 2, 3 . . . up to the maximum quantity of numbers permitted by the available quantity of bits. If we have 8 bits at our disposal unsigned numbers are in the range 0 to 255. Another possibility is called “2s-complement[9] format” which allows negative numbers. The same 8 bits in 2s-complement format can represent numbers in the range -128 to 127.

Notice how the two syntaxes represent different ranges of numbers. This means that sometimes, the same series of bits may mean two different numbers depending on the choice of syntax. For example 11111111 means 255 as an unsigned integer and it means -1 in 2s-complement format. It is not possible to tell which it is just by examining the bits because they are the same. To tell the difference we must know which syntax the one who wrote the bits has used.

This is a key idea. The one who writes the bits gets to choose the syntax. If you don't know which one he chose you can't read the bits.

This is another reason why data is not the same thing as its electrical representation. Let's suppose an engineer aligns 8 voltages in a row and tell you which voltage corresponds to a 0 or a 1. He asks you to read the number. Can you read it? You can't unless the engineer tells you his choice of syntax. Just knowing the voltages and the bits they stand for isn't good enough.

This ambiguity may also apply to circuits. For example there are circuits called adders for adding binary numbers. The relevant algorithms for addition are such that the same adder which adds unsigned numbers also correctly adds numbers in 2s-complement format.[10] For example when this circuit adds 10000000+01111111 resulting into 11111111 it could mean either 128+127=255 in unsigned integer format or -128+127=-1 in 2s-complement format. It is not possible to know which it is from an examination of the bits and the circuit structure alone. We must also know which convention on the representation of numbers is used.

The circuit is identical. The bits are identical. The meaning is different because the syntax is different. The user of the circuit gets to choose whether he adds unsigned integers or 2s-complement numbers because the same circuit does both. The computation is not the electronic activity of the circuit. It is an abstract idea involving symbols and a choice of syntax to represent meanings.

There are lots of rules of syntax for various types of data. There are some for floating point numbers. They are needed when the computation requires fractions. Other rules are encoding for characters such as ASCII or Unicode. There are standards for very complex data like video and audio files. Many of these rules are international standards. Other are defined by the authors of particular programs for internal use.

Data Has Meaning

Symbols have meanings. In the case of boolean algebra, 0 means *false* and 1 means *true*. In the case of arithmetic series of bits mean numbers. Sometimes the numbers encode letters and the text has meanings like legal briefs or contracts. The text has meaning. Sometimes the bits are stored in files or databases. This data also has meaning.

Mathematical symbols may have simultaneously two types of meanings. There is the abstract mathematical meaning, like truth values and numbers. Then they may be some non mathematical interpretation given to the abstract mathematical meaning. For example a number may be a count of apples in the grocery inventory, or it may be the speed of a rocket in flight.

Like symbols and syntax, the meanings of the symbols is not physical component of the circuit doing the computation. When you use a computer to maintain the inventory of a grocery there are bits representing information about lemons and other food. These bits are descriptions of the food, they are not the food. The lemons are not electronic components of the machine.

Data and Computations Are Contents

To sum up what we have seen so far, we have symbolic language defined by a series of syntactical and semantical relationships as set forth below:

1. There is some physical substrate, often elements of a physical computer, which represents symbols.
2. There are conventions of syntax on how the symbols are organized.
3. When symbols have the proper syntax, they have meaning in mathematics, such as boolean or numerical values.
4. The mathematical values may be used, alone or in combination, to represent other entities such as letter of the alphabet or the complex structures found in files and databases such as video and database records.
5. Then the mathematical language may also be given some non mathematical interpretations.

All these relationships are defined by various conventions. There is the convention which identifies which voltage represents which bit. There are conventions on the format of numbers. There are conventions on how to use numbers to represent letters, digits and other characters. There are conventions on file formats and data structures. There are many more conventions, this is not an exhaustive list.

These conventions come from diverse sources. Some of these conventions are defined by the computer engineers who have designed the computer components. Others are industry standards. Some more are part of culture, like the meaning of English words we find in textual information. Other conventions are defined by the programmer when he defines the data used by its program. Regardless of the source all these conventions are intangible. They are neither physical elements of circuitry nor physical phenomenons. They are elements of knowledge which are required to be able to read and write the data.

Where do algorithms fit in this picture? Algorithms are independent from hardware and meanings. They lie in the middle, they are about the symbols and their syntax.

Algorithms, in the mathematicians' sense of the word, are methods for manipulating the symbols. A computation is the manipulation carried out according to the algorithm. Symbols are not something physical like voltages as the discussion of boolean gates and voltages has shown. Computations and algorithms are not hardware processes because they manipulate the symbols and not the voltages.

An algorithm must be machine executable. It must be the type of procedure which could be executed by logic gates which may react only to voltages representing symbols.[11] No interpretation of the meanings is possible. The gates can't do that. The meanings of the symbols is understandable to someone who can read the bits but it is not used by the computing circuit when

carrying out a computation. Therefore an algorithm doesn't depend on the meanings to be executed.

If an algorithm is neither a hardware process nor an operation on meaning what are they? Algorithms are manipulations of the uninterpreted symbols and their syntactic arrangement.

Please think of a legal brief. You need a physical stack of paper with marks of inks. Or you need electrons for your electronic file. But the brief is not a stack of paper with marks of inks and it is not electrons in a file. Like a legal brief, algorithms and computations are not physical entities. They are contents. They require a physical representation but still remain different from the representation.

But unlike a legal brief, algorithms are limited to the symbols and their syntax. They must be executable without having to interpret the semantics. On this respect algorithms are more like a printing press which is able to print the letters without referring to their meanings. But unlike the printing press, algorithms are not machines because they are not physical entities.

How Computer Hardware Carries Out Computations

With this background information, we may ask how does a computer carries out a computation? We have already seen part of the explanation in the discussion of the boolean gates. When the voltages are used to represent bits the gates implement boolean operators. If we give an algorithm to a competent engineer, he will find an arrangement of gates that will carry out the corresponding computation. When the engineer etches the gates on an integrated circuit he makes a dedicated circuit because this circuit can carry out only the computation for which it has been designed. But a general purpose computer can carry out any computation as required by the software. Making a general purpose computer requires something more.

Mathematicians have discovered a special category of algorithms called universal algorithms. Several universal algorithms are known.[12] Each of these algorithms can compute every function which is computable provide we give them a corresponding program as input. In effect, any universal algorithm can emulate the behavior of every other algorithm. This is why we call them universal. This is like a glorified Swiss army knife but for computing. A single algorithm suffices to serve the purposes of all of them.

Dedicated circuits for universal algorithms may be implemented with boolean gates. This circuit can only compute the universal algorithm for which it has been designed. This is exactly like a dedicated circuit for any other algorithm but with one difference. The chosen algorithm is universal. This makes the circuit a programmable general purpose computer. The reason is that the universal algorithm it will carry out any computation when it receives the corresponding program as input. The task of writing such a program and giving it to the computer is called “programming the computer”.

Please note that a program is data. It is made of bits. This is like giving a video as input to a video player. In both cases we are giving data as input to an algorithm.

Most modern general purpose computers are built according to a common engineering pattern called the stored program computer architecture. These computers are dedicated to carrying out a universal algorithm called the instruction cycle.[13] A program for this algorithm is a series of instructions. Each instruction corresponds to an operation which must be performed whenever the instruction is executed.

A computer has many parts. Two of them stand out as the most important in the execution of instructions.[14] The first one is main memory. This component is exactly what its name implies. It is a container for information, a place where bits can be read and written. The instructions must be stored in the computer main memory before they can be executed. The other important computer component is called the processor, or the CPU.[15] This component too is what its name implies. It reads the instructions from memory and carries out the corresponding operations.

The instruction cycle works as follows:[16]

1. The CPU reads an instruction from main memory.
2. The CPU decodes the bits of the instruction.
3. The CPU executes the operation corresponding to the bits of the instruction.
4. If required, the CPU writes the result of the instruction in main memory.
5. The CPU finds out the location in main memory where the next instruction is located.
6. The CPU goes back to step 1 for the next iteration of the cycle.

As you can see, the instruction cycle executes the instructions one after another in a sequential manner. In substance the instruction cycle is a recipe to “read the instructions and do as they say”.[17]

This is an algorithm in the sense mathematicians give to this word. It is a series of steps for performing a task by manipulating symbols which meets the requirements mathematicians have defined for a procedure to be a mathematical algorithm. We say software is mathematics because the execution of all computer programs is the execution of a universal mathematical algorithm. In this sense software is making utterances in the language of mathematics.

Please recall the objection mentioned at the beginning of this article, that some people think we say software is described by mathematics. You should now see more precisely why we are not arguing that. The instruction cycle is not a description of the computer. It is what the computer does. We are arguing that software *is* mathematics because what the computer does is a mathematical computation according to a mathematical algorithm.

Sources

The mathematical model of computation corresponding to the stored program computer is called the RASP or Random Access Stored Program. It has been documented in chapter 1 of [Aho 1974]. This is how mathematicians and theoretical computer scientists state in the mathematical language the algorithm implemented as the instruction cycle.

The RASP is a universal version of a family of algorithms called register machines. This family is described in [Taylor 1998] chapter 5.

I have explored how the algorithm of the instruction cycle can be stated in the language of lambda-calculus. This statement is more detailed because it covers a wider range of features commonly found in modern computers. This file can be downloaded here (PDF).

Some people think that the presence of random elements bring an algorithm outside the scope of mathematics. This is incorrect. Mathematics, including probability theory, is able to handle randomness. In particular there is in mathematics random mathematical processes called stochastic processes. Physicists routinely use probability theory to state some of the laws of quantum mechanics. In computer science a normal deterministic algorithm can be transformed into a randomized algorithm it a source of random numbers is treated as an input from which data is read. Then the procedure of a probabilistic algorithm is not different from a deterministic one, the difference is in the input. This is precisely how operating systems such as Linux treat hardware number generators. From a software perspective they are systems files from which random data is read.

The semantics of some programming languages has been defined mathematically. The text of program implemented in one of these languages must be the description of a mathematical algorithm as defined by the language semantics. An example of such a language is Standard ML. The definition of the language is found in [Milner 1997]. See [Milner 1991] for a commentary and [Reppy 2007] for the mathematical definition of I/O and concurrent programming libraries. Another example of a programming language with a mathematically defined semantics is Coq. This language is documented in [Bertot 2004].

More mathematical references may be found in previous articles published on Groklaw:

- An Explanation of Computation Theory for Lawyers
- 1 + 1 (pat. pending) — Mathematics, Software and Free Speech
- A Simpler Explanation of Why Software is Mathematics

A Remark on Algorithms and Abstract Ideas

Mathematical algorithms are a subcategory of procedures for manipulating symbols. The instruction cycle is also a procedure for manipulating symbols.

Some legally skilled people are telling me that all procedures for manipulating symbols are abstract ideas. This is interesting. The question of whether a particular computation is a mathematical algorithm may be superfluous. As soon as we show a procedure is manipulating symbols we don't need to determine whether this manipulation is mathematical. We already have proof that it is an abstract idea.

I think this circumstance may be used to develop a test for when a patent involving software is patent eligible. Case law from the Federal Circuit suggests that the legal difficulty is to find a workable definition of some key terms.

The Federal Circuit had problems agreeing on what the term “mathematical algorithm” means. From *in re Warmerdam*:

> One notion that emerged and has been invoked in the computer related cases is that a patent cannot be obtained for a "mathematical algorithm." *See In re Schrader,*, 22 F.3d 290, 30 USPQ2d 1455 (Fed.Cir.1994) and cases discussed therein. That rule is generally applied through a two-step protocol known as the *Freeman-Walter-Abele* test, developed by our predecessor court, the first step of which is to determine whether a mathematical algorithm is recited directly or indirectly in the claim, and the second step of which is to determine whether the claimed invention as a whole is no more than the algorithm itself. *See Schrader, 22 F.3d at 292, 30 USPQ2d at 1457*.
>
> 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.

Trying to define what is an abstract idea doesn't seem to be working either. From *MySpace, Inc. v. GraphOn Corp.*:

> When it comes to explaining what is to be understood by "abstract ideas" in terms that are something less than abstract, courts have been less successful. The effort has become particularly problematic in recent times when applied to that class of claimed inventions loosely described as business method patents. If indeterminacy of the law governing patents was a problem in the past, it surely is becoming an even greater problem now, as the current cases attest.
>
> In an attempt to explain what an abstract idea is (or is not) we tried the "machine or transformation" formula—the Supreme Court was not impressed. *Bilski*, 130 S.Ct. at 3226-27. We have since acknowledged that the concept lacks of a concrete definition: "this court also will not presume to define `abstract' beyond the recognition that this disqualifying characteristic should exhibit itself so manifestly as to override the broad statutory categories of eligible subject matter. . . ." *Research Corp. Techs., Inc. v. Microsoft Corp.*, 627 F.3d 859, 868 (Fed.Cir.2010).
>
> Our opinions spend page after page revisiting our cases and those of the Supreme Court,

> and still we continue to disagree vigorously over what is or is not patentable subject matter. *See, e.g., Dealertrack, Inc. v. Huber,* ___ F.3d ___ (Fed. Cir.2012) (Plager, J., dissenting-in-part); *Classen Immunotherapies, Inc. v. Biogen IDEC*, 659 F.3d 1057 (Fed.Cir.2011) (Moore, J., dissenting); *Ass'n for Molecular Pathology*, 653 F.3d 1329 (Fed.Cir. 2011) (concurring opinion by Moore, J., dissenting opinion by Bryson, J.); *see also In re Ferguson*, 558 F.3d 1359 (Fed.Cir. 2009) (Newman, J., concurring).
>
> This effort to descriptively cabin § 101 jurisprudence is reminiscent of the oenologists trying to describe a new wine. They have an abundance of adjectives—earthy, fruity, grassy, nutty, tart, woody, to name just a few—but picking and choosing in a given circumstance which ones apply and in what combination depends less on the assumed content of the words than on the taste of the tongue pronouncing them.

I think the concept of manipulation of symbols is well defined. It should be possible to develop a workable legal test on this basis. An obvious possibility is to modify the *Freeman-Walter-Abele* test which has been rejected in *Warmerdam*, testing for manipulations of symbols instead of algorithms. Another possibility is to make a literal application of *Mayo v. Prometheus* “to transform an unpatentable law of nature into a patent-eligible *application* of such a law, one must do more than simply state the law of nature while adding the words “apply it.” ” This alternative test would compare the utility of the claim taken as a whole with the utility of the manipulations of symbols taken alone, adding the words “apply it”. There must be a substantial difference between the two, otherwise otherwise the claim is not patent eligible according to *Mayo*.

I don't have the legal skills to determine whether these tests are legally appropriate. I bring them up as examples of what may be considered. I think the courts should be able to apply this kind of tests because the fundamental problem with the definition of terms is solved.

The Federal Circuit has granted an en banc review of CLS Bank International v. Alice Corporation. The questions are:

> a. What test should the court adopt to determine whether a computer-implemented invention is a patent ineligible "abstract idea”; and when, if ever, does the presence of a computer in a claim lend patent eligibility to an otherwise patent-ineligible idea?
>
> b. In assessing patent eligibility under 35 U.S.C. § 101 of a computer-implemented invention, should it matter whether the invention is claimed as a method, system, or storage medium; and should such claims at times be considered equivalent for § 101 purposes?

I think people submitting amicus briefs should explain why a test based on manipulations of symbols would work.

It goes without saying that symbols must have a physical representation. This doesn't make a procedure to manipulate symbols something less abstract. The digital equivalent of processes for pushing a pencil and making marks of lead on paper should not be patent eligible. Arguing that this is not patenting mathematics in the abstract is a charade. There is no way of carrying out a mathematical computation without actually writing the symbols. Therefore it should not matter whether the manipulation of symbols is claimed as a method, a system, a storage medium or any other physical device. It should not be patent eligible in all these scenarios.

The meanings given to the symbols should make no difference. As long as the procedure manipulates symbols their meanings will never confer patent eligibility. I have a riddle that may help convince you why this is a desirable result.

Please take a pocket calculator. Now use it to compute 12+26. The result should be 38. Now give some non mathematical meanings to the numbers, say they are counts of apples. Use the calculator to compute 12 apples + 26 apples. The result should be 38 apples. Doe you see a difference in the calculator circuit? Here is the riddle. What kind of non mathematical meanings

must be given to the numbers to make a patent eligible difference in the calculator circuit?

This is the kind of question the Federal Circuit is asking about computers. When I read case law about section 101 patentable subject matter I see the court analyze the meanings of the bits to determine whether the invention is abstract. But at the same I see the court working from a legal theory where a software patent is actually a hardware invention. Can't they see this is a contradiction? This is the point of the riddle. The meanings of the bits is not a hardware component of the machine and it is not influencing the steps of the computation.

Activities like input, output and mental steps of reading, writing and thinking about the meaning of symbols should not confer patent eligibility to a manipulation of symbols. These steps are just more manipulations of symbols.

Data gathering where nothing more is achieved but obtaining the data which will be manipulated should not confer patent eligibility to a manipulation of symbols. The data must come from somewhere. There is no way to apply a manipulation of symbols without gathering data somehow.

Procedures for manipulating symbols have obvious Free Speech implications. Think of science fiction stories where a robot-like machine communicate with human beings using human language. Software is evolving to make this sort of human-machine interactions a reality. Such conversations should not be patentable because a machine is involved. Here is another riddle. Suppose a human is interacting with a computer. What kind of man-machine interface will make the conversation patentable assuming the same conversation could be held using human language? What are the implications of the answer on First Amendment rights?

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

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

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

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

[Epstein 1989] Epstein, Richard L., Carnielli, Walter A., *Computability Computable Functions, Logic and the Foundations of Mathematics*, Wadsworth & Brooks/Cole, 1989

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

[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

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

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

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

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

Footnotes

1 There are actually several mathematical languages because each branch of mathematics has its own requirements for notation. But for simplicity let's say it is a language.
2 There is more to mathematics than utterances in mathematical language. Mathematical entities like numbers and geometrical figures are also mathematics. The article mentions the utterances in mathematical language because this is the part of mathematics the sentence “software is mathematics” refers to.

Some people sometimes try to draw a contradiction between *described by* mathematics and *is* mathematics. They argue that when something is described by mathematics it is not mathematics because the things being described is not the description. There is no such contradiction. It is possible and common to describe entities like numbers, geometric figures and algorithms with mathematical formulas. For example a circle centered at the origin with a radius of unit 1 is described by the equation $x^2+y^2=1$. Therefore mathematical entities can both be mathematics and be described by mathematics.

Similar situations occur outside of mathematics. For example it is possible to describe the syntax of an English sentence using English. Then the sentence is both English and described by English.
3 This is a fact of mathematics. One mathematically precise statement of the connection between algorithms and logical deductions is called the Curry-Howard correspondence. Another mathematically precise statement of this connection is the resolution algorithm used in mathematical logic. This discovery had led to the development of logic programming. For the more familiar family of imperative programming languages, the mathematically precise statement of this connection is called Hoare logic.
4 There is more to software than program execution. But according to patent law, the aspect of software which is patented is the program execution. We say “software is mathematics” as a slogan to remind people that the aspect of software which is patented is entirely a mathematical computation.
5 I am sure our mathematician friends will have something to say about what is mathematical subject matter. I chose to keep this discussion simple. If you need to explore the question I suggest you read [Devlin 2000], especially the prologue. This book discusses what is mathematics in a language easily understandable by non mathematicians.
6 Bits are copyrightable whether or not they are watched by humans.
7 See [Kleene 1967] p. 223, where Stephen Kleene defines the notion of algorithm as mathematicians see it. You will find several requirements. Among them there is this one:

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

If the meanings of the symbols are ambiguous it is impossible to execute the algorithm in this manner. Resolving the ambiguity is an intervention that requires insight or ingenuity. This requirement would not be met. On the other hand if the symbols are unambiguous, for purposes of executing an algorithm mechanically, like robots, their meanings are superfluous. Why should we need the step of noticing the symbol means *true* when we already know it is 1? There is no reason. The algorithm works just as fine when we act on the symbols alone like these boolean algorithms do.

This logic applies to all algorithms in the sense mathematicians give to this word. This point has been noticed by Richard Epstein and Walter Carnielli. See [Epstein 1989] p. 70, where they describe a series of models of computations used to define classes of algorithms:

> What all of these formalizations have in common is that they are all purely syntactical despite the often anthropomorphic descriptions. They are methods for pushing symbols around.

8 This assumes the memory technology is DRAM. Other technologies may represent bits differently.
9 This is pronounced *two's complement*.
10 See [Hamacher 2002] page 368.
11 I am referring to the criteria mathematicians have defined for a procedure to be an algorithm in

the sense of mathematics. See footnote 7 above. While mathematicians have not stated their criteria in these terms they are equivalent in practice with requiring that algorithms be machine executable manipulation of symbols. In the case of digital electronics this means the algorithm must be computable by a circuit made of logic gates.

12 A theoretically important universal algorithm is called the universal Turing machine. Other universal algorithms are used in practice for programming purposes. Instruction cycles are the preferred universal algorithm for imperative programming which is the most widely used programming paradigm. In logic programming algorithms such as SLD resolution are used to resolve problems described in logical terms using Horn clauses. See [Ben-Ari 2001] chapter 7 and 8 for a complete discussion. In functional programming various implementations of normal order β-reduction are used. See [Kluge 2005] for a book dedicated to several implementations of normal order β-reduction. It should be noted that normal order β-reduction is a universal algorithm which modifies its program as the computation progresses. It cannot be assumed the program is always unchanged by the computation as is usually (but not always) the norm in imperative programming.

13 Other universal algorithms exist as indicated in footnote 12. However the instruction cycle is the chosen one when building a stored program computer.

14 Main memory is the equivalent of a pad of paper in a pencil and paper calculation. The CPU is the electronic equivalent of whoever handles the pencil. We may view the stored program computer as an automated version of a human carrying out a pencil and paper calculation, but much faster.

Other components of stored program computers are the peripheral devices for input and output and communication components called buses to interconnect everything and also other components like power supply, casing etc.

15 The acronym CPU stands for Central Processing Unit.

16 This is a simplified explanation for readability. Here is how [Hamacher 2002] p. 43 describes the hardware implementation of an 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.

17 Every stored program computer contains a hardware implementation of this algorithm. This is not the only possible way to implement it. Program like a virtual machine for a Python bytecode interpreter are software implementations of the instruction cycle. There are also universal algorithms which are not instruction cycles. These algorithms too may be implemented in software. Ultimately the software universal algorithms are executed by the hardware instruction cycle.

Programs intended to software implementations of some universal algorithm are not instructions

for the hardware instruction cycle. This is a point to remember because some people think all software are instructions for the CPU. Sometimes they build legal arguments based on this idea. These arguments are often wrong because they don't take into account all the possible forms software can take.