An Explanation of Computation Theory for Lawyers,

by PolR

Published by Groklaw November 11, 2009
http://www.groklaw.net/article.php?story=20091111151305785

I am a computer professional with over 25 years of experience. I have a Master's degree in computer science and no legal expertise beyond what I acquired reading Groklaw. Yet I have developed an interest in understanding US patent law and how it applies to software. I have read a few court cases and legal briefs. Based on my professional knowledge I can say that in some precedent-setting cases the legal understanding of computers and software is not technologically correct.

I see the situation like this. The authors of legal briefs and court rulings have enough of an understanding to feel confident they can write meaningful arguments on the topic. But yet they do not understand computers and software well enough to reach technically correct conclusions. The unfortunate result is legal precedents that do not connect with reality. Computers don't work the way some legal documents and court precedents say they do[1].

I think I can identify the root cause of this. It seems that nobody has ever taught the legal community the basic concepts of computation theory. At the very least I don't find any evidence that computation theory is known in the cases and legal briefs I have read so far. Lawyers and judges know about modern electronics in general and computers in particular. They know about code and the compilation process that turns source code into binary executable code. Computation theory is something different. It is an area of mathematics that overlaps with philosophy[2]. Computation theory provides the mathematical foundations that make it possible to build computers and write programs. Without this knowledge several of the fundamental principles of computer science are simply off the radar and never taken into consideration.

The fundamentals of computation theory are not obvious. A group of the greatest mathematicians of the twentieth century needed decades to figure them out. If this information is not communicated to lawyers and judges they have no chance to understand what is going on and mistakes are sure to happen. All the errors I have found result from this omission.

The purpose of this text is to help fill the gap. I try to explain all the key concepts in a language most everyone will understand. I will underline the key legal questions they raise or help answer as they are encountered. I will conclude the article with examples of common mistakes and how the knowledge of computation theory helps avoid them.

Let's illustrate the importance of computation theory with one of the legal issues computation theory will help resolve. Consider the following list of statements.

- All software is data.
- All software is discovered and not invented.
- All software is abstract.
- All software is mathematics.

If my understanding of the US patent law is correct, whether or not any of these statements is true could determine whether or not software is patentable subject matter. The resolution of this issue has serious consequences to the computer electronics and software industries.

When you know computation theory, you know without a shred of a doubt that each of these statements states a fact that is grounded in well-established mathematics. If you don't know

computation theory, these statements will probably look to you like debatable issues.

Effective methods

Historically, modern computation theory started in the 1920s and 1930s with research on "effective methods". In a totally untypical manner, the definition of this mathematical concept is written in plain English as opposed to some mathematical notation that would be impenetrable to laymen. Here it is[3]:

> A method, or procedure, M, for achieving some desired result is called 'effective' or 'mechanical' just in case
>
> 1. M is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);
> 2. M will, if carried out without error, always produce the desired result in a finite number of steps;
> 3. M can (in practice or in principle) be carried out by a human being unaided by any machinery save paper and pencil;
> 4. M demands no insight or ingenuity on the part of the human being carrying it out.

The terms 'effective' or 'mechanical' are terms of art in mathematics and philosophy. They are not used in their ordinary, everyday meaning. The phrase "effective method" is a term of art. This term has nothing to do with the legal meaning of "effective" and "method". The fact that these two words also have a meaning in patent law is a coincidence.

An effective method is a procedure to perform a computation that is entirely defined by its rules. The human that carries out the computation must follow the rules strictly; otherwise the answer he gets is not the one that is intended. Examples of effective methods were taught to us in grade school when we learned how to perform ordinary arithmetic operations like additions, subtractions, multiplications and divisions. We were taught procedures that are to be carried out with pencil and paper. You have to perform these operations exactly how they were taught, otherwise the result of the calculation is wrong.

This definition was written in the early part of the 20th century, before the invention of the electronic computer. This was before the word "algorithm" came to be used in computer science. Effective methods are precursors to computer algorithms. The definition works from the assumption that you need a human to do a computation[4]. Clause 3 explicitly mentions that the computation must be one that it is possible for a human to carry out with pencil and paper. Aid from machinery is explicitly forbidden. In effect this definition spells out what kind of human activity would qualify as an algorithm, but without using that terminology. The term "effective method" was used instead.

Of course this concept has since been superseded with the modern concept of computer algorithm. Mathematical discoveries from the 1930s have shown the human requirement may be dropped from the definition without changing the mathematical meaning. Effective methods and computer algorithms are the same thing, except that computers are allowed to execute algorithms without running afoul of the definition. This is known as the Church-Turing thesis. This is one of the most important results of computation theory and theoretical computer science. A big part of this text will be devoted at explaining exactly what this thesis is, how mathematicians discovered it and what is the evidence we have of its truth.

There are a few fine points in this definition that deserve further explanations. Clause number 3 states

that an effective method can be carried out "in practice or in principle" by a human being. This language is there because mathematics is about abstract concepts. You don't want an effective method to stop being an effective method when a human runs out of pencils or paper. You don't want the method to stop being an effective method when a human dies of old age before being done with the calculation. This is the sort of thing that may happen if, for example, you try to calculate a ridiculously large number of decimals of π[5]. By stating the method may be carried out "in principle" the definition makes abstraction of these practical limitations.

Clause number 4 states that no demands are made on insight or ingenuity. This clause implies that all the decisions that are required to perform the calculation are spelled out explicitly by the rules. The human must make no judgment call and use no skill beyond his ability to apply the rules with pencil and paper. The implication is that the human is used as a biological computing machine and serves as a reference for defining what a computation is.

Effective methods and the Church-Turing thesis are relevant to an issue that has legal implications. What is the relationship between an algorithm executed by a computer and a human working with pencil and paper? Abstract methods made of mental steps are not patentable subject matter[6]. When is software such a method? When is it a physical process? With knowledge of computation theory and the Church-Turing thesis, lawyers and courts will be able to tap on the discoveries of mathematics to answer these questions with greater accuracy.

Formal Systems

Why were mathematicians so interested in effective methods in the 1920s? They wanted to solve a problem that was nagging them at the time. Sometimes someone discovered a theorem and published a proof. Then some time passes, months, years or decades, and then someone discovered another theorem that contradicted the first and published a proof. When this happens, mathematicians are left scratching their heads. Both theorems cannot be true. There must be a mistake somewhere. If the proof of one of the theorems is wrong then the problem is solved. They revoke the theorem with a faulty proof and move on. But what if both proofs are sound and stand scrutiny?

These kinds of discoveries are called paradoxes. They are indicative that there is something rotten in the foundations that define how you prove theorems. Mathematical proofs are the only tool that can uncover mathematical truth. If you never know when the opposite theorems will be proven, you can't trust your proofs. If you can't trust your proofs, you can't trust mathematics. This is not acceptable. The solution is to look at the foundations, find the error and correct it so paradoxes don't occur any more. In the early twentieth century, mathematicians were struggling with pretty nasty paradoxes. They felt the need to investigate and fix the foundations they were working on[7].

The relationship between algorithms and mathematical proofs is an important part of computation theory. It is part of the evidence that software is abstract and software is mathematics. The efforts to fix the foundations of mathematics in the early twentieth century are an important part of this story.

One of the most prominent mathematicians of the time, David Hilbert, proposed a program that, if implemented, would solve the problem of paradoxes. He argued that mathematical proofs should use formal methods of manipulating the mathematical text, an approach known as formalism. He proposed to base mathematics on formal systems that are made of three components.

1. A synthetic language with a defined syntax
2. An explicit list of logical inference rules
3. An explicit list of axioms

Let's review all three components one by one to see how they work. Mathematics uses special symbols

to write propositions like a+b=c or $E=mc^2$. There are rules on how you use these symbols. You can't write garbage like $+=^{\%}5/$ and expect it to make sense. The symbols together with the rules make a synthetic language. The rules define the syntax of the language. Computer programmers use this kind of synthetic language every day when they program the computer. The idea of such a special language to express what can't be easily expressed in English did not originate with computer programmers. Mathematicians thought of it first[8].

How do you test if your formula complies with the syntax? Hilbert required that you use an effective method that verifies that the rules are obeyed. If you can't use such a method to test your syntax then the language is unfit to be used as a component of a formal system[9].

Inference rules are how you make deductions[10]. A deduction is a sequence of propositions written in the language of mathematics that logically follows. At each step in the sequence there must be a rule that tells you why you are allowed to get there given the propositions that were previously deduced. For example suppose you have a proof of A and separately you have a proof of B. Then the logical deduction is you can put A and B together to make "A and B". This example is so obvious that it goes without saying. In a formal system you are not allowed to let obvious things be untold. All the rules must be spelled out before you even start making deductions. You can't use anything that has not been spelled out no matter how obvious it is. This list of rules is the second component of a formal system.

It was known since the works of philosopher Gottlob Frege, Bertrand Russell, and their successors that all logical inference rules can be expressed as syntactic manipulations. For example, take the previous example where we turn separate proofs of A and B into a proof of "A and B" together. The inference consists of taking A, taking B, put an "and" in the middle and we have "A and B". You don't need to know what A and B stand for to do this. You don't need to know how they are proved. You just manipulate the text according to the rule. All the inferences that are logically valid can be expressed by rules that work in this manner[11]. This opens an interesting possibility. You don't use a human judgment call to determine if a mathematical proof is valid. You check the syntax and verify that all inferences are made according to the rules that have been listed[12]. This check is made by applying an effective method[13]. You may find a computerized language for writing and verifying proofs in this manner at the Metamath Home Page.

If there is no human judgment call in verifying proofs and syntax, where is human judgment to be found? It is when you find a practical application to the mathematical language. The mathematical symbols can be read and interpreted according to their meanings. You don't need to understand the meaning to verify the proof is carried out according to the rules of logic, but you need to understand the meaning to make some real-world use of the knowledge you have so gained. For example when you prove a theorem about geometry, you don't need to know what the geometric language means to verify the correctness of the proof, but you will need to understand what your theorem means when you use it to survey the land.

Another place where human judgment is required is in the choice of the axioms. Any intuitive element that is required in mathematics must be expressed as an axiom. Each axiom must be written using the mathematical language. The rules of logic say you can always quote an axiom without having to prove it. This is because the axioms are supposed to be the embodiment of human intuition. They are the starting point of deductions. If some axiom doesn't correspond to some intuitive truth, it has no business in the formal system. If some intuitive truth is not captured by any axiom, you need to add another axiom[14].

The list of axioms are the third component of a formal system. Together the syntax, the rules of inferences and the axioms include everything you need to write mathematical proofs. You start with

some axioms and elaborate the inferences until you reach the desired conclusion. Once you have proven a theorem, you add it in the list of propositions you can quote without proof. The assumption is that the proof of the theorem is included by reference to your proof. And because all the rules and axioms have been explicitly specified from the start and meticulously followed, there is no dark corner in your logic where something unexpected can hide and eventually spring out to undermine your proof.

To recapitulate, effective methods play a big role in this program. They are used (1) to verify the correctness of the syntax of the mathematical language and (2) to verify that the mathematical proofs comply with the rules of inferences. The implication is that there is a tie between formal methods and abstract mathematical thinking. If you consider the Church-Turing thesis, there is a further implication of a tie between computer algorithms and abstract mathematical thinking. This article will elaborate on the nature of this tie.

Let's return to Hilbert's program. When you make explicit all part of the mathematical reasoning in a formal system, the very concept of mathematical proof is amenable to mathematical analysis. You can write proof about how proofs are written and analyze what properties various formal systems have[15]. Hilbert intended to use this property of formal systems to bring the paradoxes to a permanent end. Hilbert's program was to find a formal system suitable to be the foundations of all of mathematics that would meet these requirements:

1. The system must be consistent. This means there must be no proposition that could be proven both true and false. Consistency ensures there is no paradoxes.
2. The system must be complete. This means every proposition that could possibly be written in the system could be either proven true or proven false. Completeness ensures that every mathematical question that could be formulated with mathematical language could be solved within the confine of the system.
3. The system must be decidable. This means that there must be an effective method to find at least one actual proof of a proposition when it is true or at least one actual refutation of the proposition when it is false[16]. Decidability ensures that when confronted with a mathematical problem we always know how to solve it.

If the formal system is amenable to mathematical analysis then, according to Hilbert, it would be possible to find a mathematical proof that the system meet the requirements of consistency, completeness and decidability[17]. Completing such a program would be a formidable intellectual accomplishment. All of mathematics would be contained in a system based on axioms, inference rules and effective methods. Every mathematical question would be solved by carrying out a decision method where human judgment plays no part. In this approach, the role of human intuition is limited to the selection of the axioms.

From Effective Methods to Computable Functions

Hilbert's program was successful in part and failed in part. Useful formal systems that appear to be free of paradoxes were developed[18]. This is the successful part. The failed part is there is no formal system that can be used as the foundation of the entirety of mathematics and is also consistent, complete and decidable. Such formal systems are not possible. Theorems that show that these properties are mutually incompatible have been found. But before the mathematicians reached this conclusion, substantial research was done. It is in the course of this research that the fundamental concepts of computation theory were developed. Ultimately these findings have led to the development of the modern programmable electronic computer[19].

When working on Hilbert's program, mathematicians were struggling with another nagging issue. Hilbert's program called for every mathematical concept to be defined using a special mathematical

language. How do we write the definition of effective method in such a language? The reference to a human being calculating with pencil and paper is not easily amenable to this kind of treatment. This is why the original definition is written in plain English. Some specific effective methods could be defined mathematically. Effective methods in general were something a mathematician would know when he sees one but couldn't be defined with mathematical formulas.

The work on Hilbert's program is what led to the discovery of how to define mathematically the concept of effective methods. Several different but equivalent definitions were found. To reflect the definitions being used, effective methods are called "computable functions" when we refer to the mathematical definitions. When effective methods are considered for use with a computer instead of being carried out by a human, they are called "computer algorithms". When translated in a programming language for use on an actual computer, they are called "computer programs".

These multiple definitions of computable functions led to very different methods of how to actually make the calculations. A human being that would use one of these system would write on paper something very different from someone using the other system. But still they would all do the same calculations. The reason is that algorithms are abstractions different from the specific steps of how the calculation is done[20].

For example, consider a sorting algorithm written in a programming language like C. You give the program a list of names in random order and the program prints them sorted in alphabetic order. The program may be compiled for Intel x86, for Sun SPARC, for ARM or any other make and brand of CPUs. The binary instructions generated by the compiler for one CPU will be very different from the instructions for the other CPUs because this is how CPUs are made. If you trace the hardware instructions that are actually executed with a debugger, you will verify that the execution of the code is different from CPU to CPU because of the difference in the instructions. But still it is the same C program that executes. The names will be printed out in sorted order on all CPUs. The abstract sorting algorithm is the same on all CPUs.

This difference between specific instructions and abstract algorithm has its basis in computation theory. By the time you will be done reading this article you will know what that basis is.

There are three of the definitions of computable functions that stand as the most important ones. The Church-Turing thesis states that computable functions as defined by these definitions are the same functions as those one may compute with effective methods.

1. Kurt Gödel's recursive functions
2. Alonzo Church's lambda-calculus
3. Alan Turing's Turing machines

One needs to know all three definitions to understand computation theory. All three definitions played a role in the discovery that Hilbert's program is impossible to accomplish. All three definitions bring considerable insights as to what an algorithm is. All three definitions are equivalent to both effective methods and computer algorithms. Together they explain why the work of a human with pencil and paper and the work of a computer manipulating bits follow the exact same abstract processes.

Now let's look each of these three definitions one by one and flesh out the most legally relevant concepts of computation theory.

Kurt Gödel's Recursive Functions

In mathematics the word "function" is a term of art. A function is a correspondence between a pair of mathematical objects that is such that the second object is determined by the first. For example the area of a circle is a function. When you know the circle, you can find out the area. Addition is a function.

You know a pair of values[21] and you add them up to get the result. Often functions are associated with a method or formula that state how to calculate the value of the function.

This terminology issue being settled, let's go back to the main topic. Kurt Gödel wasn't looking to define computable functions. This happened almost by accident. He was trying to do something very different and a few years later other mathematicians found out the implications of what he did.

Kurt Gödel was working on the Hilbert program. He was working on a formal system called "Peano arithmetic" that defines the arithmetic of positive integers. This is the numbers 0, 1, 2, 3 .... to infinity. The basic principle of Peano arithmetic is to base the entirety of arithmetic on the ability to count. This is a foundation made of two basic components. The first one is the number zero (0). The other one is the successor function that increases a number by one. The successor of 0 is 1, the successor of 1 is 2 etc. But the language of Peano arithmetic is too rudimentary to write numbers this way. It can only write the successor by appending an apostrophe. Zero is written 0, but 1 is written 0' and two is written 0" and you continue like this appending one apostrophe every time you increment by one. You can identify a number by counting the apostrophes that are appended to 0.

Why are mathematicians bothering to do this? It is because this is research on the foundations. The goal is to find out what are the most minimal concepts that could serve as the basis of arithmetic. Do we need addition? Do we need subtraction? Do we need Arabic numbers? Or can we do everything with zero and successors alone? If the foundational concepts are kept very minimal, it reduces the risks that some complexity hides paradoxes that will pop up after a decade or two of research. It also makes proofs about the formal system simpler.

Remember the Hilbert program. There is a requirement to analyze the formal system to prove consistency, completeness and decidability. If the formal system is complex, these proofs will be hard to write. The hope is to make these proofs easier by keeping the formal system simple.

Gödel was studying a method of defining arithmetic functions called recursion. When you write a recursive definition[22] you must specify two rules.

1. You define the value of the function for the number zero
2. Assuming you know the value of the function for an arbitrary number n, you define how you compute the function for n'.

With these two rules you know how to compute the value of the function for all integers. Rule 1 tells you the value of the function for 0. Then you use rule 2 to compute the value of the function for 1 given that you know the value for 0. Then you proceed to compute the value for 2 and then 3 etc. As an example here is how you define addition using recursion.

1. m + 0 = m
2. m + n' = (m + n)'

For the sake of example assume you want to compute 5+3. Starting with 5=0"" and 3=0''' the steps go as follow:

5 + 3 = 0"" + 0'''	Translation into the Peano arithmetic language
0"" + 0''' = (0"" + 0'')'	by rule no 2
(0"" + 0'')' = (0"" + 0')''	by rule no 2
(0"" + 0')'' = (0"" + 0)'''	by rule no 2
(0"" + 0)''' = (0"")'''	by rule no1
(0"")''' = 0""""	because the parentheses can be removed at this point
0"""" = 8	Translate back into Arabic numbers

This illustrates how recursion works as an effective method. Every time you define something using recursion, you actually define an effective method to perform the calculation.

This example also shows why no one will want to actually make additions in this manner. The process is way too cumbersome. The goal is not to provide an alternative to the usual methods of doing arithmetic. The goal is to provide a foundation that is sound and free of paradoxes.

The purpose of the recursive definition of addition is to provide a reference. Other methods to add numbers are allowed but only if we can prove a theorem that proves these methods will yield the same answer as the recursive definition. Without the theorem we have no guarantee that the alternative calculation will yield the correct answer. The point of the theorem is to provide the guarantee that the procedures we use to calculate additions rest on the foundations.

Formal systems work like an edifice. You have a small number of concepts and associated axioms and rules of inference that provide a foundation. But the foundation is not used directly because it is too minimal to be usable in a practical situation. You must build theorems and theorems over theorems until you have a body of knowledge that is usable. Part of the task required by the Hilbert program was to build the edifice to show the proposed foundations are suitable for the intended purpose. This is what Gödel's concept of recursion is contributing to.

Gödel Numbers

When working on recursion, Gödel got a revolutionary insight. He noticed something useful about prime numbers.

I suppose you remember prime numbers from your high school arithmetic. But if you don't let's recapitulate. A prime number is a number greater than 1 that can be divided evenly only by 1 or itself. There is an infinite sequence of prime numbers: 2, 3, 5, 7, 11, 13 ... etc. Every number can be factored into the product of prime numbers. Here are a few examples.

- $12 = 2^2 3^1$
- $1125 = 2^0 3^2 5^3$

When writing a prime number factorization it is customary to omit the multiplication symbols. They are implicit. $2^2 3^1$ actually means $2^2 \times 3^1$.

Look at the exponents. Sometimes you need to multiply the same prime number multiple times to get the result. $2^2 3^1$ actually means 2 x 2 x 3. An exponent of 1 means you multiply one times. An exponent of 0 means you don't need to multiply the prime number. $2^0 3^2 5^3$ actually means 3 x 3 x 5 x 5 x 5.

What if you assign numbers to an alphabet? For instance suppose you assign a=1, b=2 until z=26? Then you can use prime numbers to encode text into numbers. You can turn "Pamela" into something like $2^p 3^a 5^m 7^e 11^l 13^a$. Replace the letters with their corresponding numbers and the text becomes a number. You can also do the reverse. When you want to read the text you factor the number into primes and then turn the exponents back into letters. This works because there is only one way to factor a number into prime numbers; therefore a number can only translate into one text. There can be no ambiguity.

Numbers that are used to encode text in this manner are called Gödel numbers.

Ordinary text editing operations people can do with pencil and paper can also be defined using recursion. For example you can use arithmetic to put $2^g 3^r 5^o 7^k$ and $2^l 3^a 5^w$ together to make $2^g 3^r 5^o 7^k 11^l 13^a 17^w$. This corresponds to the use of pencil and paper to put "grok" and "law" together to make "groklaw". Arithmetic operations that manipulate the exponents of prime numbers are the mirror

images of the symbolic manipulations on text.

This is time to go back to formal systems. Put this idea together with what we have just said on prime numbers. We get that formal systems themselves have a mirror image in arithmetic. This is the case for Peano arithmetic. There the logic goes:

- All the symbols can be translated into numbers.
- The effective methods that test when a mathematical proposition made of symbol is syntactically correct can be translated into a recursive function that makes the same test on exponents of prime numbers.
- The effective method that tests when a proof is made according to the inference rules can also be translated into a recursive function that makes the same test on exponents on prime numbers.
- All the axioms can be translated into exponents on prime numbers by translating their symbols.

Then the entire formal system has been translated into arithmetic[23]. This is called the "arithmetization of syntax". It is one of the greatest mathematical discoveries of the twentieth century. It has many far reaching consequences[24], some of them are germane to patent law.

Symbols and Reducibility

Gödel numbers are evidence that the symbols of the mathematical language need not be visual patterns written on paper. They can also be something abstract like the exponents of prime numbers. Or if we take a more modern look, they can be 0s and 1s stored as electromagnetic signals in electronic devices. In all cases the symbols have the same meaning and when we use them to write mathematical statements they allow to write the same logical inferences. This observation raise more issues. What do we mean, "have the same meaning"? What do we mean "allow to write the same logical inferences"? This asks for a human intelligence to look at the symbols and make such determination. This is in apparent conflict with the principle of a formal system where the validity of mathematical proofs is verified without using human judgment calls.

Gödel's work brings a solution to this apparent conflict. Syntactic translations between the different representations of the formal system can be defined mathematically. Do you remember when Hilbert pointed out that formal systems may be subject to mathematical analysis? Here we are. Syntactic translations are such mathematical analysis. Once you have shown mathematically that there is a translation you can conclude that whatever the meaning is in the original it will be preserved in the translation. This is called reducibility[25]. We say syntax is reduced to arithmetic when a translation from the syntax to arithmetic is available.

Reducibility is a concept that should be important to patent law. Abstract ideas are not patentable subject matter. In order to prevent abstract ideas from being indirectly patented, there are exceptions in the law that will not allow patenting an "invention" made of printed text based on the content of the text. The implication of Gödel's discovery is that the symbols that represent an abstract idea need not be printed text. The symbols can be something abstract like exponents of prime numbers. They can be something physical like electromagnetic signals or photonic signals. Symbols can take a variety of forms[26]. Does the law exempt abstract ideas from patenting only when they are written in text or does it exempt all physical representations? In the latter case what is the test that can tell a physical representation of an abstract idea from a patentable physical invention? Mathematicians know of such a test and it is reducibility.

Enumerability

Gödel numbers are also used to define computable enumerability. This is a term of art for a mathematical concept of computation theory. Imagine a list of all numbers that have a specific

properties. For example these lists are computable enumerable:

- The even numbers: 0, 2, 4, 6 ...
- The odd numbers: 1, 3, 5 ,7 ...
- The prime numbers: 2, 3, 5, 7, 11 ...

Computable enumerability occurs when you can test with a recursive function whether the number has the property or not. You build your list by running all the numbers 0, 1, 2 in sequence and run them against the test. The first number that passes the test is the first on the list, The next number that passes the test is the next one etc.

Thanks to Gödel numbers, it is possible to define an arithmetical test for the syntax of the definition of a recursive functions. Then we can enumerate all the possible definitions. More specifically each recursive function corresponds to the Gödel number of its definition. What we enumerate is the Gödel numbers and this is the same thing as enumerating the definitions themselves because the Gödel numbers can be translated back into the corresponding text.

Now consider the Church-Turing thesis that computer programs are the same thing as recursive functions. This is not obvious now, but by the time we are done with this article all the mathematical evidence will have been presented to you. Would this be of consequence to the patentability of software? Developing a computer program that fulfills a specific purpose is mathematically the same thing as finding a number that has some specific properties. Can you patent the discovery of a number?

Another way to look at it is that there is a list of all the possible programs. We can make one by listing all possible programs in alphabetic order[27]. Any program that one may write is sitting in this list waiting to be written. In such a case, could a program be considered an invention in the sense of patent law? Or is it a discovery because the program is a preexisting mathematical abstraction that is merely discovered?

This idea that programs are enumerable numbers is supported by the hardware architecture of computers. Programs are bits in memory. Strings of 0s and 1s are numbers written in binary notations. Every program reads as a number when the bits are read in this manner. Programming a computer amounts to discovering a number that suits the programmer's purpose.

Alonzo Church's Lambda-Calculus

Lambda-calculus is the second definition of effective method that uses mathematical language, recursive functions being the first. I follow the plan of explaining each of these definitions one by one and now is the turn of lambda-calculus.

It is the brainchild of Alonzo Church. Like Gödel he was doing research in the foundations of mathematics but he used a very different angle. Instead of working on the arithmetic of positive integers he was trying to use the abstract concept of mathematical function as his foundation.

Functions occur in all areas of mathematics: arithmetic, geometry, boolean algebra, set theory etc. Church was concerned with the properties of functions that belonged to all types of functions independently of the branch of mathematic. When Peano arithmetic uses zero and the successor function as the most elementary concept, lambda-calculus gives this role to the operations of abstraction and application[28].

What is abstraction? Do you remember in high school how shocking it was when the teacher showed you for the first time an algebraic formula that mixed up letters and numbers? How could you add a letter and a number like in x+7? For most students this concept is hard to grasp. People can understand 3+7 or 9+7 but x+7? What kind of number would be the result of such addition? The point that was

hard to learn is abstraction. The point of writing x+7 is not to calculate an answer. The point is to define a pattern, the act of adding 7 to something. The pattern must be applicable to any number whatever it is. This is the purpose of using a letter. The formula doesn't want to refer to a specific number. The letter marks the place where a number belongs but this number is unknown and must not be specified. This finding of a pattern is abstraction.

Application is the reverse of abstraction. It is what you do when you know that x is the number 5 and that at last you can turn x+7 into 5+7.

To put it simply, abstractions are when you make functions by stating how they are calculated. Application is the actual use of the rule of the function.

The task Church set to himself is to develop a formal system based on abstraction and application. These two operations require the exercise of human intelligence. A formal system requires rules that can be followed without the use of human ingenuity. How do you reconcile the two? Church found a solution that involved breaking down abstraction and application into more elementary steps.

He handled abstraction with a syntactic trick. He required that the variable being abstracted is explicitly identified with the Greek letter λ. The expression x+7 would have to be written λx.x+7. Then you no longer need to use intelligence to know that it is an abstraction and that x is the marker for what is being abstracted. This information is written in the formula.

Application is expressed by juxtaposition. If you decide that x is 5 you would write (λx.x+7)5. Parentheses are used to group symbols when how they read may otherwise be ambiguous. This example means the abstraction x+7 is applied to the value 5. At this point the formal system would want you to look at the variable after the λ, in this example it is x, and you would replace it with the expression it is applied to, 5 in this example. Then (λx.x+7)5 is transformed into 5+7. This operation is called beta-reduction[29]. This is a fancy name Church has cooked up to mean what a modern word processor would call a search and replace operation.

This technique allows multiple levels of abstraction. For example consider the concept of addition. It can be expressed as an abstraction of both arguments λyλx.x+y. An expression like this would capture the very concept of addition independently of the values being added. Also each value can be applied separately. For example (λyλx.x+y)7 becomes after beta reduction λx.x+7. Functions like λyλx.x+y that produce another abstraction are called high order functions. A complete computation would require to apply beta reduction repeatedly until no abstractions are left. For example (λyλx.x+y)(7)(5) would turn into 5+7 after two beta reductions.

The combination of the λ notation and the beta reduction operation allowed Church to define his lambda-calculus. That is a formal system that captures the properties of abstraction and application. The repeated use of beta reduction until no abstraction is left is an effective method that is used to perform all computations in this system.

There is a twist. My examples used arithmetic operations because every one understands elementary arithmetic. This helps make the concepts clear. But the goal of Church was to study functions independently of any branch of mathematics. He did not include arithmetic operations in lambda-calculus[30]. Surprisingly this doesn't matter. Abstraction and application alone are sufficient to let you count. And then you can use recursion to reconstruct arithmetic from this basis.

Here is how you may represent numbers in lambda-calculus.

1. λsλx.x
2. λsλx.s(x)
3. λsλx.s(s(x))

4. λsλx.s(s(s(x)))

Remember Peano arithmetic when we identified the numbers by counting the apostrophes? Now we count the occurrences of s. The writing system is different but the idea is the same.

If you find yourself scratching your head at stuff like λsλx.s(x) wondering what it actually means this is normal. You need to be skilled at lambda calculus to make sense out of this kind of formula. I am not giving you a course in lambda calculus. There is no way you can learn how to read this language from the explanations I am giving you.

The point I am making is there is a translation from the concepts of 0 and successor numbers to other concepts that belong to lambda-calculus. The result of the translation is less important than the fact that there is a translation at all. The reason is that if there is a translation then mathematicians may invoke the principle of reducibility. The implication is that any computation[31] that can be expressed in the language of Peano arithmetic can also be expressed with the language of lambda-calculus.

The existence of a translation is why mathematicians consider that recursive functions and lambda-calculus are equivalent. All computations that can be done by means of recursion can be translated into an equivalent computation made with lambda calculus. And conversely the language of lambda-calculus can be transformed into arithmetic by means of Gödel numbers. There is no computation that one of the languages can do that the other can't do.

There is a fundamental difference between Gödel numbers and this translation. Gödel numbers are translating the symbols of the formal language one by one. Therefore you have a mirror image of the text of the language hidden in the exponents of prime numbers. The translation into lambda-calculus is different. It translates not the symbols but the numbers themselves into functions. You have a mirror image of the numbers hidden into an enumeration of functions.

This technique of translation is called modeling. The mirror image in lambda-calculus of Peano arithmetic is called a model.

Imagine a geometric pattern like a circle. It remains a circle whether it is a wheel, a plate or a jar cover. You can see the pattern is the same by comparing the shapes. But how do you define "the same shape" mathematically? You don't eyeball the object and call them the same. You verify that all of these objects have a boundary where every point is at the same distance from the center. This is the mathematical definition of a circle and everything that meets the definition is a circle.

Now imaging the whole formal system is an abstract pattern made of relationships between formulas. The axioms and inference rules of the formal system are the equivalent of the definition of the circle. Everything that obeys these same rules is matching the definition of the pattern. The translation is proof that the model obeys the rules of Peano arithmetic. Therefore the abstract pattern that is characteristic of the arithmetic of positive integers is found within lambda-calculus[32].

Models work a bit like a mock up. In the olden times, engineers designing a car would build a mock up before building a real car. The examination of the mock up helped them finding faults in their design before incurring the expense of making the real thing. The model of arithmetic is a mathematical mock up except that unlike the car mock up, the model has the full expressive power of the original.

The Church-Turing Thesis

Alonzo Church was so impressed by the power of Gödel numbers that he argued every effective method would be reproducible by some recursive function based on a Gödel number encoding of the effective method. This idea implies that recursive functions, lambda-calculus and effective methods are equivalent concepts.

This is known as the Church Thesis. It didn't convince everyone when it was first formulated. A more convincing formulation had to wait until the discovery of Turing machines. Nowadays a revised version of the Church Thesis that adds Turing machines to this list of equivalent concepts is accepted by mathematicians under the name of the Church-Turing thesis.

A Universal Algorithm

In Peano arithmetic each recursive function makes a different effective method. The rules to carry out the computation are provided by the definition. In lambda-calculus the situation is different. There is only one effective method that is used for all computations. This effective method is to keep doing beta reduction until there are no applications left that could be reduced.

Since lambda-calculus has the capability to perform every possible computation (in the extensional sense) then one algorithm is all one needs to perform every possible computation[33]. The difference between one computation and another is in the data that is supplied to the algorithm. This is one of the reasons we argue that software is data.

Lambda-calculus can be and has been implemented as a programming language. The language LISP is a prominent example.

This raises an interesting possibility. Suppose I am sued for infringement on a software patent. I may try defending myself arguing I did not implement the method that is covered by the patent. I implemented the lambda-calculus algorithm which is not covered by the patent. And even if it did, lambda-calculus is old. There is prior art. What happens to software patents then?

Anyone that argues in favor of method patents on software must be aware that the current state of technology permits this possibility.

Extensionality

Consider two of these idealized humans that are required to carry out effective methods. One of them is carrying out, say, a multiplication using Peano arithmetic. The other does the same multiplication using lambda-calculus. If we would compare their writings on paper, they will be different. The formal languages are not the same. the symbols are not the same and one follows the rules of recursion directly while the other follows them indirectly through repeated beta reduction. How can it be said the two computations are the same?

This is a very important question because it speaks to what constitutes "is the same" from a mathematician's perspective. For example when one argues software is mathematics, the argument is that the computation done by the computer is the same thing as a mathematical computation, therefore this is mathematics that is being done by the machine. Such argument can be understood only when one knows the mathematical meaning of "being the same" computation.

This question can be answered two different ways. There is the "intensional" definition and the "extensional" definition[34].

The intensional definition considers the details of how the computation is done. If you change the details of the computation, from the intensional perspective you get a different computation even though the two computations always produce the same answer. By this perspective recursive functions and lambda-calculus are different.

The extensional definition ignores the details of the calculation and considers only the equivalence by means of reducibility. If the two calculations can be matched with a translation from one language to another, then they are the same from the extensional perspective. From the extensional perspective recursive functions and lambda-calculus are equivalent because every calculation made in one system

is matched by a calculation made in the other system..

The law has a similar[35] concept when it makes a difference between ideas and expressions of ideas. For example consider I write a program. Some outside party has written a similar program. This party sees my program and thinks I infringe on his proprietary rights. There are two scenarios depending on whether he makes his claim according to copyright law or patent law.

For claims of copyright infringement the text of the source code matters. If I wrote my program independently and I can prove the texts are different, I don't infringe on his rights. This is an intensional point of view.

For claims of patent infringement then differences in the text of the source code won't matter. It won't even matter if the code is written in a different programming language. If my program uses the same method that is covered by the patent according to whatever legal test of "same method" is applicable, I will infringe. This is an extensional perspective.

Which of the two perspectives is correct? It depends on the degree of abstraction that is wanted. The intensional perspective is the less abstract. It is the perspective you need when you study the properties of the written expressions of the symbols. It may be used to answer questions like how many steps are required to produce an answer or how much space in memory is required to store the symbols.

The extensional perspective is more abstract. It is the one you need when you are concerned with the expressive power of the language. It answers questions like: whether you can say the same things in both languages or how to tell when two expressions in different languages have the same meaning.

The intensional vs extensional dichotomy explains the situation of the C sorting program that is the same when run on different hardware architectures. When you compile the code, you produce different instructions depending on which hardware architecture is targeted. These difference are significant only when you look at the code from the intensional perspective. When you take the extensional perspective, all executables produce the same output. Therefore they are the same. The consequence is that software is abstract. The details of the instructions don't matter[36].

Denotational Semantics

C is a programming language, not a mathematical language. How do we define its meaning mathematically? This is done using a technique called denotational semantics[37].

For example consider the statement i=i+1 that could be written in many programming languages such as C. It looks like an equation but it is not an equation. Here the letter i is called a variable. Imagine a box with a label. The box contains information. The variable is the label for such a box. The statement i=i+1 means open the box labeled i, get the number that is in the box, add 1 to it and put it back in the same box. If there is no number in the box your program has a bug and it will fail. The question is what is the mathematical meaning of this operation.

Suppose you run your program in your favorite debugger[38]. You can stop the execution of the program just before the i=i+1 statement. Then you can use the debugger to inspect the memory of the computer. What do you see? There are many variables in use. Each of them is a labeled box containing some information. You may have a variable called "FirstName", another called "LastName", another one called "Salary" and one named "HiringDate". The memory is assigning each of these names a value. This is a mathematical function. It doesn't happen to be one you calculate because the values are explicitly stored and not computed with a formula or anything. It is a mathematical function nonetheless.

Now use the debugger to execute the i=i+1 statement and stop the execution again. What do you see in

the memory? None of the variables have seen their values changed except i which has been incremented by one. The memory is a new mathematical function that is identical to the previous one except for the change in the value of i. This is the meaning of the statement. When you consider the content of the memory as a function, the statement is a high order function that changes this function into another function.

This is how denotational semantics works. Lambda-calculus is good at handling high order functions. You build a mathematical model of the memory content of the computer using lambda-calculus. Then each statement in the language can be translated into a high order function that manipulates this model. The result is a definition of the mathematical meaning of the program.

This programming statement is discussed in a brief to the Supreme Court of the United States by Professor Hollaar and IEEE-USA. When arguing that software is not mathematics, the brief states[39]:

> But in most instances, the correspondence between computer programs and mathematics is merely cosmetic. For example, the equation $E = MC^2$ expresses a relationship between energy and matter first noted by Einstein, while the computer program statement E = M * C ** 2 represents the calculation of M time C raised to the second power and then assigning the result to a storage location named E. It is unfortunate for purposes here that the early developers of programming language made their calculation-and-assignment statements look like mathematical equations so that they would seem familiar to scientists and engineers. But the common programming statement I = I + 1, which increments the value stored in location I, is essentially nonsense as a mathematical equation. Similarly, a computer program is a series of calculation-and-assignment statements that are processed sequentially, not a set of simultaneous mathematical equations that are solved for their variables.

This statement from the brief is incorrect. The correspondence between a statement in a high level language such as C and mathematics is not defined by syntactic similarity. It is defined by denotational semantic. There is another correspondence that involves machine instructions. This correspondence will be discussed when we reach the topic of Turing Machines.

Programming Directly from Mathematical Languages

If programming languages have a mathematical meaning what stops programmers from writing their programs using a mathematical language in the first place? The answer is it can be done. Languages such as LISP in the functional family of languages implement lambda-calculus and can be used to write programs using a mathematically defined notation. Programmers prefer languages in the imperative family[40] of languages because the resulting programs is closer to the hardware architecture of the CPU and gives them more control on how the algorithm will execute. This situation may change. The trend is to increase the number of CPUs that are put on a chip. In presence of multiple CPUs imperative language becomes more difficult to use. The more CPUs, the harder it gets write a program that runs well on the computer. Functional languages are more amenable to work well in this kind of environment.

There is another approach. One may take advantage of the relationship between algorithms and logical proofs. Consider this parable.

> This is 1963. It is the heart of the Cold War. The head of CIA rushes into the office of his main scientific advisor.
>
> CIA Head: We have received a report from James Bond. The Soviets have developed a new

> formula that makes our strategic defense obsolete. It will cost us billions to upgrade. But the report says it could be disinformation, that the new theory may be something phony to make us spend billions uselessly. We cannot afford to guess. We need to know. Can you use a computer to find out if the formula is true of false?
>
> Chief Scientist: Sure I do. I can write a program that prints "True" whatever the input is and another one that prints "False" whatever the input is. One of the two programs is bound to provide the correct answer. Therefore I can use a computer to print the solution you seek.

There are two major schools of mathematicians. The classical school would say the answer of the chief scientist is logically correct although inappropriate. One of the two programs does indeed print the correct answer and the chief scientist can indeed write it. Therefore the chief scientist has literally answered the question that has been asked.

The intuitionistic school will protest that the chief scientist's answer is illogical. You can't argue you can write the requested program until you can tell which program is the one you seek.

The parable has been designed to make it sounds like the intuitionistic point of view is the correct one. In this context the whole point of the question is to know which of the two programs provides the answer. I wanted to show that intuitionistic logic has merits. Such logic requires building a tie between a proof that a problem is solvable and the construction of an effective method that actually solves the problem. Their point is that if you can't actually solve the problem, how can you argue a solution exists?[41]

If you work out this tie using the language of lambda-calculus you get something called the Curry-Howard correspondence. It is a mathematical translation that takes a proof and turn it into an algorithm in the language of lambda-calculus or vice-versa. The implication is you can write a mathematical proof and use an automatic process to extract the built-in algorithm and compile it into an executable program. Or conversely you write a program and the compiler verifies that it is bug-free by extracting a proof of its correctness from the code or reporting an error if the proof doesn't work out. This is a topic for research. The mathematical principles are known but their application is not ready for use outside of the research laboratory yet.[42] Despite this limitation some specific programs that have been verified in this manner have been developed.[43]

A question for patent law is what do you do when this research comes to fruition? At this point there will be no actual difference between the work of computer programmer and the work of a mathematician. How would you patent software without directly patenting mathematics?

The merging of programming and mathematical proofs has industrial applications. [See Campbell for an example.] If you can prove a program is mathematically correct you verify that its logic will perform according to the intent of the programmer. This eliminates a large quantity of bugs.[44]

Several formal verification methods exist outside of the Curry-Howard correspondence. One of the most popular is called Hoare logic[45]. It is a formal system where every statement in a programming language corresponds to a rule of inference. Programmers annotate their code with mathematical propositions and verify that the rules of inferences are obeyed. If they do, the program is proven mathematically correct.

Alan Turing's Turing Machines

Turing machines are the third of the major mathematical definitions of computable functions. They were proposed specifically to resolve an open question with the first two definitions. Alonzo Church

put forth the thesis that Gödel numbers were powerful enough to reproduce every effective method that could be. But at the time the evidence was intuition. It looked like Gödel numbers have that power but how do you prove it? Alan Turing's work brought convincing evidence.

The definition of effective method requires a human being doing the work with pencil and paper. Turing proceeded from an analysis of what such a person is able to do. He eliminated all redundant possibilities until he was left with what is essential.[46] Let's look at this elimination process.

Do we need to write things in columns or otherwise arrange symbols in two dimensions? This is not necessary. One can do the same work by writing everything on the same line and visualizing how it fits two-dimensionally. It is inconvenient, but since the task is to find out the essential minimum, writing things in two dimensions can be done away with. This suggests that regular sheet paper is not necessary. The paper may be in the form of a long tape where symbols are written sequentially. The human winds the tape back and forth as he works with the information.

How long the tape must be? Our idealized human who carries out the effective method never runs out of time and paper. The tape has to be infinitely long to reflect this capacity.

A human often needs to refer to information already written. How fast does he need to shuffle paper to find out what he wants? Moving the tape one symbol at the time is sufficient. If you need to go farther away you can repeat moving the tape one symbol until you get where you want to be.

When reading the information, how many symbols do you need to look at at the time? The answer is one. If you need to look at more symbols you can examine them one by one.

When faced with making a choice, on what basis would our person make his decision? Choices are made on the basis of the symbols being read and on his state of mind. He must track where he is in the execution of the effective method. This tracking is done by changing his state of mind. Then, when faced with a choice, the human will consider his state of mind and the symbol being read and make the next move as is required by the method. This implies we don't need to delve into human psychology or the biology of the brain. All we need to know is (1) that there are states of mind and (2) that the human changes his states as he tracks where he is in the execution of the effective methods.

How many states of mind are required? Only a finite number is required. They can be inventoried by analyzing the rules of the method and finding the junctures where choices are made. This means that if we write down the inventory of states of mind we may represent them with a finite number of symbols.

These considerations are summarized in the list of capabilities below. They are the minimal requirements to perform any effective method.

- There is an infinitely long tape where symbols can be recorded sequentially.
- The tape has a "current" position where a symbol can be read. This is the current symbol.
- A symbol may be written at the current position overwriting any symbol already there.
- The tape may be moved one symbol position either to the left or to the right. It may also stand still.
- The human carrying out the method has a number of distinct possible states of mind.
- The states of mind are finite in number and may be represented with a finite number of symbols.
- The human changes his state of mind based on the previous state and the current symbol on the tape.

The implication is that it is possible to describe every effective method in these terms. The rules for carrying out the method are written as a list of quintuples. This is a list of line items. Each of them is made of five elements of information.

1. The current state of mind that is required for the rule to apply.
2. The current symbol that is required for the rule to apply.
3. The symbol that is written on the type when the rule applies. If this is the same symbol as the one mentioned in item 2, then this is a do-nothing operation.
4. Whether the tape moves right, left or stands still after writing the symbol if the rule applies.
5. The state of mind one must adopt after the application of the rule.

This process is transforming the broad scope of effective methods into something that has a standardized form. You analyze the original method rules, translate them into a suitable list of quintuples, and you are all set.

Now imagine that our idealized human carrying out the method chooses to use an idealized typewriter instead of pencil and paper. The typewriter writes on an infinite tape that can be moved left or right one symbol at a time at the press of a key. The user may see the current symbol or overwrite it with a new symbol by the press of a key. There is a blackboard on the wall where the list of quintuples is written. The user uses his current state of mind and the current symbol to select the applicable quintuple, and then he performs the actions dictated by the quintuple. He repeats this process until it is over. This human is doing the same calculation as if he were carrying out the original effective method.

Now imagine you could endow the typewriter with the ability to read the symbols and track its internal state by itself. The quintuples could be mechanized, allowing the typewriter to carry out the calculation all by itself, without the help of an actual human. What you get when you do this is a Turing machine.

The Evidence for the Church-Turing Thesis

Now we have closed the loop. Effective methods are reduced to Turing machines. But by definition a Turing machine is an effective method because the human typing on the keyboard of the idealized typewriter might just as well use pencil and paper. Therefore the equivalence between effective methods and Turing machines is a two-way street.

Turing machines can be translated into recursive arithmetic with Gödel numbers. Recursive arithmetic can be translated into lambda-calculus with the method we already saw. Turing found a set of rules for a Turing machine that can do the beta reduction algorithm of lambda-calculus. The cycle is complete. All three definitions of computable functions may be reduced to each other. The argument in favor of the Church-Turing thesis is now complete.

Quite a lot of evidence of the Church-Turing Thesis has been found beyond what has been presented here. An article on the Church-Turing Thesis on the Turing Archive summarizes it as follow:

> Much evidence has been amassed for the 'working hypothesis' proposed by Church and Turing in 1936. Perhaps the fullest survey is to be found in chapters 12 and 13 of Kleene (1952). In summary: (1) Every effectively calculable function that has been investigated in this respect has turned out to be computable by Turing machine. (2) All known methods or operations for obtaining new effectively calculable functions from given effectively calculable functions are paralleled by methods for constructing new Turing machines from given Turing machines. (3) All attempts to give an exact analysis of the intuitive notion of an effectively calculable function have turned out to be equivalent in the sense that each analysis offered has been proved to pick out the same class of functions, namely those that are computable by Turing machine. Because of the diversity of the various analyses, (3) is generally considered to be particularly strong evidence. Apart from the analyses already mentioned in terms of lambda-definability and recursiveness, there are analyses in terms of register machines (Shepherdson and Sturgis 1963), Post's canonical and normal systems (Post 1943, 1946), combinatory definability (Schonfinkel 1924, Curry 1929, 1930, 1932),

> Markov algorithms (Markov 1960), and Godel's notion of reckonability (Godel 1936, Kleene 1952).

This same article also makes a point of what the Church-Turing thesis does *not* say. The thesis says Turing machines are equivalent to computations made by human using effective methods. It doesn't say they are equivalent to any possible computation made by a machine. The reason is that the idealized typewriter argument doesn't discuss equivalence with other machines, and therefore we don't have a basis to reach such a conclusion.

The article provides a list of published articles where mathematicians have constructed abstract computing machines that are not equivalent to Turing machines. These calculations cannot be done by a human and are not effective methods. Nobody has constructed a physical device that implements any of these machines. It is unknown whether you can build a physical computing device that can perform a computation a Turing machine can't do.

The Working Principle of the Modern Computer

What if we design an effective method that directs the idealized human to read the quintuples of a Turing machine and do as they tell him to do? What if we make a Turing machine out of this method? You get a universal Turing machine. Here "universal" means that it can do the work of all other Turing machines.

There are two lists of quintuples here. The universal Turing machine uses a list of quintuples to define its activity like any other Turing machine. If you supply this machine with a tape where the quintuples of another machine are written, the universal machine will emulate the behavior of the other machine.

This is a major discovery in the history of computing as the Stanford Encyclopedia of Computing points out:

> Turing's construction of a universal machine gives the most fundamental insight into computation: one machine can run any program whatsoever. No matter what computational tasks we may need to perform in the future, a single machine can perform them all. This is the insight that makes it feasible to build and sell computers. One computer can run any program. We don't need to buy a new computer every time we have a new problem to solve. Of course, in the age of personal computers, this fact is such a basic assumption that it may be difficult to step back and appreciate it.

This is the point where computation theory stopped being the exclusive province of mathematicians and became relevant to engineers. The question was how do you build an actual universal Turing machine?[47] Today we know the answer. We use electronics. The tape corresponds to the computer memory. The table of quintuples corresponds to the CPU instruction set. The CPU itself has the ability to change states. The Turing machine you want to emulate is programming data that you load in memory for execution. When we assemble the pieces, we have a modern digital computer.

When making an actual computer, engineers had to cross the line that separates the abstract world of the mathematician from reality. The Turing machine is an abstraction suitable to provide a foundation. Do you remember what we said about foundations when discussion Peano arithmetic? They are not meant to be used directly. You need to build an edifice on top of them to obtain a usable result. The engineers had to build the edifice.

Moving a tape one symbol at the time is inefficient. You need to be able to get the information directly. You need to access memory randomly.

There are many types of storage devices with different characteristics: RAM, ROM, Flash, magnetic

disks. optical disks etc. You need to mix and match them as required. Where the Turing machine needed one storage device, the tape, in real life computer need many.

The Turing machine can do only one operation: it writes one symbol on the tape based on the appropriate quintuple. This is too rudimentary for real-life computing. An actual computer needs an elaborate instruction set.

You get the idea. The Turing machines provided the starting point but engineers had to elaborate on the concept. The result is what is now known as the Von Neuman architecture.[48]

Why do we still think computers are Turing machines after these engineering changes? The answer depends on if you look at an intensional definition or an extensional definition. Mathematicians didn't stop working and reducibility didn't go away. They studied the mathematical effect of changes in the architecture of the Turing machine.[49] What if the machine has more than one tape? What if the tapes are replaced with random access memory? The answer is these machines can be reduced to Turing machines. You don't bring into existence new computations by doing these changes. From an extensional perspective the modified machines perform the same computations as a plain Turing machine.

There is one difference that has an effect. The idealized human never runs out of time and paper. This is why the Turing machine has an infinite tape. But in the real world, there are physical limits. Memory is not infinite, and time is limited. This restricts the computations that are possible. A real-life computer is an approximation of a Turing machine. Whether or not the calculations it carries out corresponds to one performed by a Turing machine depends on whether or not this calculation will fit within the physical constraints. If it doesn't fit, the calculation will not be carried out to completion even though the Turing machine has the theoretical ability to do it. But when a computation fits within the constraints, it is always the same as one that is done by a Turing machine.

The Nature of Software

Let's recapitulate. A universal Turing machine is one that, when given the quintuples of another Turing machine, will do as instructed. This principle is implemented in an actual computer as follows:

- The tape of a Turing machine corresponds to the memory of a computer.
- The read/write capability of a Turing machine corresponds to the memory bus of the computer.
- The changing states of the Turing machine corresponds to the changing states in the CPU electronics.
- The quintuples of a universal Turing machine correspond to the CPU electronics.
- The quintuples of the other Turing machine correspond to the computer program.

The last bullet is an important one for patent law. It says software is the information you need to give to the universal Turing machine to make it perform the desired computation. This simple fact has consequences.[50]

- Software is data.

  This is because information is synonymous with data. When you look at a physical computer, software is stored in a data file like any other form of data. It is loaded in memory and stored in exactly the same chips of RAM as data. The CPU manipulates the software with the same instructions as it uses to manipulate other forms of data. The fact that software is data is the mathematical principle that makes it possible to build and sell modern digital computers.

- Software is abstract.

This is a consequence of how Turing machines are constructed. A computer program is equivalent to the work of an idealized human being carrying out an effective method. By definition this work is an abstraction written down on paper. Another way to look at it is that a computer program is equivalent to some Turing machine. If we apply the discoveries of Alan Turing, we never need to physically build this Turing machine. It is sufficient to find a description of it and pass it to a universal Turing machine. The only device we need to actually build is the universal Turing machine. The computer is the universal Turing machine. The software is a description of an abstract machine that is never physically implemented.

- Software is mathematics.

  This is because all the instructions the CPU is able to execute are mathematical operations. Physical computers are implementations of universal Turing machines. Universal Turing machines are part of mathematics. Therefore, whatever they do is mathematics. Another way to look at it is software is the description of some abstract Turing machine. Since all Turing machines are part of mathematics software is mathematics

- Software is discovered as opposed to invented.

  This is a consequence of the fact that software is abstract and software is mathematics. You can't invent abstract ideas and mathematics. You can only discover them.

What is Not Computable?

Computable functions are defined through an exercise in reducibility. Whatever is reducible to a Turing machine is computable. The flip side of the question is, what cannot be reduced to a Turing machine?

Alan Turing discussed this question in his doctoral thesis.[51] He studied what he called an oracle machine, which is a Turing machine augmented with the capability to ask a question to an oracle when the computation reaches a certain point. Then the oracle answers the question and the computation resumes, taking this information into consideration.

The oracle could be anything.[52] For example, imagine our idealized human typing on his idealized typewriter doing calculations about sunspots. Every now and then, he picks up the phone and asks an astronomer the percentage of the surface of the sun that is currently covered with sunspots. The astronomer looks into the telescope and provides the answer. Then our idealized human resumes typing on the idealized typewriter carrying out the rest of the computation. The astronomer is the oracle. Observing the sun is not a computation even if the result of the measurement is a number. Therefore the whole process including the astronomical observation is not something that could be done by a Turing machine alone.

A Turing oracle machine brings the concept of computations relative to the oracle. This means part of the process is a computation and part is not. It is possible to draw a line between the two by sorting out what is the oracle and what is the Turing machine.

Consider that instead of a human being typing at the typewriter, we have a computer that interfaces directly to the telescope. It takes photographs of the sun and computes the surface of the sunspots from the photos. This computer is doing mathematical calculations relative to the photographs, but taking the picture is not a mathematical operation. This kind of integration with non-computational devices is where the boundaries of computation theory lie.

This very issue has been contemplated by the Supreme Court of the United States in two celebrated cases, Parker v. Flook and Diamond v. Diehr.

In *Parker v. Flook* the court ruled on how mathematical algorithms should be treated by patent law:

> Respondent's process is unpatentable under § 101 not because it contains a mathematical algorithm as one component, but because once that algorithm is assumed to be within the prior art, the application, considered as a whole, contains no patentable invention. Even though a phenomenon of nature or mathematical formula may be well known, an inventive application of the principle may be patented. Conversely, the discovery of such a phenomenon cannot support a patent unless there is some other inventive concept in its application.

In *Diamond v. Diehr* the court looked at the flip side of the situation:

> In contrast, the respondents here do not seek to patent a mathematical formula. Instead, they seek patent protection for a process of curing synthetic rubber. Their process admittedly employs a well-known mathematical equation, but they do not seek to preempt the use of that equation. Rather, they seek only to foreclose from others the use of that equation in conjunction with all of the other steps in their claimed process. These include installing rubber in a press, closing the mold, constantly determining the temperature of the mold, constantly recalculating the appropriate cure time through the use of the formula and a digital computer, and automatically opening the press at the proper time. Obviously, one does not need a "computer" to cure natural or synthetic rubber, but if the computer use incorporated in the process patent significantly lessens the possibility of "overcuring" or "undercuring," the process as a whole does not thereby become unpatentable subject matter.

These two cases address this question: when is an invention an application of a mathematical algorithm as opposed to being the algorithm itself? With knowledge of computation theory, we may say the answer must have to do with the interactions with the real world. Everything that is the symbolic manipulations done by a CPU or equivalent devices is a mathematical computation. But when the device interacts with the outside world, whether or not you have a patentable invention, according to those cases, depends on the nature of this interaction. This is what we learn from Turing's oracle machine.

Why Lawyers Need to Know Computation Theory

What is a mathematical formula when implemented in digital electronics? What is an application of the formula as opposed to the formula itself? These questions have confounded lawyers for decades for good reason. These are hard questions. But they are the questions they have to answer when software patents are discussed.

For this type of question, guidance from mathematicians is appropriate. This is why patent lawyers should learn computation theory. This is especially important in software because the relevant mathematical concepts are hard. It took decades for the best mathematicians of the first half of the twentieth century to figure them out. Their findings are among the greatest intellectual achievements of their time. It is appropriate to stand on the shoulders of giants because this is not the sort of thing anyone can figure out by himself merely by pondering dictionary definitions and thinking things through. But many courts have been trying to do just that in software patents cases. For example consider this sentence from In re Alappat:

> 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 a single sentence the court tosses out the fundamental principle that makes it possible to build and

sell digital computers. You don't need to create a new machine every time you perform a different computation; a single machine has the capability to perform all computations. This is what universal Turing machines are doing.

In the 1940s, the ENIAC used to be programmed by physically reconnecting circuits. This design was awkward. Engineers took pains to eliminate this need and designed the next generations of computers by using the ideas of universal Turing machines as guidance. The explicit goal was to make sure you don't need to make new circuitry, much less make a new machine, when you program the computer. This history is documented on the Stanford Encyclopedia of Philosophy.[53] See also this article on Groklaw.

How could the court make such a holding? It is because it didn't know about computation theory. And I suspect it didn't know because no one ever put an explanation of computation theory into the court record.

Let's look at another example. This one is from Application of Walter D. BERNHART and William A. Fetter. It discusses a patent on a software algorithm to plot lines based on Cartesian coordinates. In this case, several issues where computation theory is relevant show very clearly. The first one is about determining what is a method made of mental steps:

> Looking first at the apparatus claims, we see no recitation therein of mental steps, nor of any element requiring or even permitting the incorporation of human faculties in the apparatus. These claims recite, and can be infringed only by, a digital computer in a certain physical condition, i. e., electro-mechanically set or programmed to carry out the recited routine. The claims also define the invention as having plotting means for drawing lines or for illustrating an object. When such functional language is used in a claim, 35 U.S.C. § 112 states that "such claim shall be construed to cover the corresponding structure, material, or acts described in the specification and equivalents thereof." The specification here mentions only mechanical drafting machines. The claims therefore cover, under section 112, only such mechanical drafting machines and their equivalents. We know of no authority for holding that a human being, such as a draftsman, could ever be the equivalent of a machine disclosed in a patent application, and we are not prepared to so hold in this case. Accordingly, we think it clear that applicants have not defined as their invention anything in which the human mind could be used as a component.
>
> ...
>
> In the case now before us, the disclosure shows only machinery for carrying out the portrayal process. In fact it is the chief object of the invention to eliminate the drudgery involved in a draftsman's making the desired portrayals. Accordingly, a statutory process is here disclosed. Looking then to method claim 13, we find that it in no way covers any mental steps but requires both a "digital computer" and a "planar plotting apparatus" to carry it out. To find that the claimed process could be done mentally would require us to hold that a human mind is a digital computer or its equivalent, and that a draftsman is a planar plotting apparatus or its equivalent. On the facts of this case we are unwilling so to hold. We conclude that the method defined by claim 13 is statutory, and its patentability must be judged in light of the prior art.

Imagine how different this passage would have been if the court knew about Turing machines and their equivalence with effective methods. Would the idealized human being typing on his idealized typewriter count as proof that the claims are made of mental steps? Or would a court hold that the very

same steps are mental steps only when they are carried out by an actual human? In this case, the court clearly looked at the problem from an intensional perspective and took note of the difference between actual humans and electronic devices. But what if the extensional perspective is the correct one? This is the way mathematicians would look at it. Without knowledge of computation theory, the court has no way even to take the mathematical view into consideration.

This *Bernhart* case is really fascinating. It is a concentrate of some of the hardest issues where computation theory is relevant. Here is another extract that speaks to what constitutes a symbol:

> Nor are the "printed matter" cases, cited by the board, *supra*, controlling as to these apparatus claims either on the facts or in principle. On their facts, those cases dealt with claims defining as the invention certain novel arrangements of printed lines or characters, useful and intelligible only to the human mind. Here the invention as defined by the claims *requires* that the information be processed not by the mind but by a machine, the computer, and that the drawing be done not by a draftsman but by a plotting machine. Those "printed matter" cases therefore have no factual relevance here.

What if the judge had factored in the equivalence of a Turing machine and effective methods? He would have understood that the information stored in memory is symbolic in nature. He would have understood that there is no difference between a mental step of a human being and the computations of the computer. He would have understood that the symbols written in computer memory have the same meaning as those written on paper. Besides, some humans can read information meant to be processed by machines. There are tools like debuggers that have been made expressly for this purpose.

Let's take this from another angle. What kind of files contain instructions that cause the computer to perform calculations? Here are a few examples. This list is by no means complete:

- Executable binaries: these files contain instructions that are executed directly by the CPU. These files are the result of the compilation of source code. When legal cases discuss the programming of a computer, they think almost exclusively of these files. I never see an analysis of how the case reasoning would extend to other kinds of instructions.
- Interpretable code: these files contain programs in human-readable form that may also be executed directly by the computer without an intermediate compilation step. This requires a special program called an interpreter that reads the code and executes the instruction. Several programming languages are interpreted. This court didn't consider the possibility that the patented algorithm could be implemented in such a language. Would such code infringe? But in this case it is an invention made of a particular arrangement of characters that is intelligible to the human mind. Would this change the assessment of the court regarding the factual relevance of the "printed matter" cases?
- PDF files: these files contain instructions to be executed by a virtual machine that are specialized to the visual rendering of written documents. How do the printed matter court cases apply to patents written to algorithms made of the execution of such instructions?

The distinction between symbols readable by humans and machine-readable information is artificial. Computation theory teaches us that symbols remain symbols whether they are ink on paper or recorded in electronic form. In both cases they can be used to implement the same abstract processes.

Let's take this same idea from still another angle. Suppose a court determines that software in its executable binary form is patentable because it is turned into something physical. This is an argument we see frequently. For purpose of this discussion it doesn't matter if the court thinks the physical something is a machine, a process or whatnot. The question is what happens to the patentability of code in forms other than executable binaries?

- The court doesn't want to patent data like a novel or music. Therefore the physical something that makes binaries patentable will not be sufficient to make data patentable.
- The court doesn't want to patent textual data in a PDF file no matter how novel and nonobvious the text may be. Such files are made of executable instructions. Therefore it must take more than the presence of novel and non obvious instructions that are executed to make something patentable.
- How about patenting source code in textual form? This will raise free speech issues. How can you discuss code if you can't write the text without infringing? The text of source code should not be patentable.
- How about code written in an interpreted language? This code is never compiled into executable binaries. the only executable binary is the interpreter. This is the same code that runs for all programs. It does not implement the patented algorithm and by itself it doesn't infringe. The interpreter implements the language. The text of the source code (which is not patentable) is given to the interpreter. The interpreter reads the text and does what it is told. All data is manipulated by the interpreter on behalf of the program. On what basis should this code be patentable? The physical something that is the basis of patentability never comes into existence. The interpreter doesn't infringe. The data is not patentable. The source code is not patentable. Executing instructions doesn't make patentable something that isn't.

This analysis is meant to communicate that software is about manipulating symbols. It is not about physical circuitry or processes. Whatever perceived circuitry is used to claim software is patentable, programmers have the means to make an end-run around this alleged circuitry. It is possible to arrange the manipulation of the symbols in such a way that the circuitry that is the alleged basis of patenting software never comes into existence.

We have already encountered a theme like this when we discussed the beta reduction algorithm of lambda-calculus. One algorithm can do the work of every other algorithm. We can make sure the method patents are never infringed because the claimed methods are never physically implemented. Interpreted languages are built on this basis.

From the same *Benrhart* case, there is a section that is reminiscent of the "software makes a new machine" from *Alappat*:

> 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. If a new machine has not been invented, certainly a "new and useful improvement" of the unprogrammed machine has been, and Congress has said in 35 U.S.C. § 101 that such improvements are statutory subject matter for a patent. It may well be that the vast majority of newly programmed machines are obvious to those skilled in the art and hence unpatentable under 35 U.S.C. § 103. We are concluding here that such machines are statutory under 35 U.S.C. § 101, and that claims defining them must be judged for patentability in light of the prior art.

These physical changes are the normal operation of the computer. It would probably not have occurred to this judge to rule that turning the steering wheel to the left is an improvement to a car that is

patentable subject matter and could be barred from patenting only because it is old and obvious. But this is a physical change that is applied to the car, and this change does give the car the ability to turn left. It is a useful improvement of the car based on the logic of this paragraph.

Machines have moving parts. Moving the moving parts as they are intended to be moved is not an invention. What are the moving parts of a computer? This information is in the definition of the Turing machine. The moving parts include the ability to read and write symbols in memory. More, the operating principle of the computer requires that programs are stored in memory and are modifiable by the computer as the execution of the program progresses. The changes to the memory are not an improvement of the computer. They are the operation of the computer as it is intended to be operated. The knowledge of the Turing machine enables one to make this determination.

What if we hold the other view? Suppose we agree with this court that writing something in memory is a patentable improvement of the computer? Then we get absurdities. Modern computers change the state of their memory billions of times per second. Do we have billions of improvements per second, each of them being patentable subject matter? How does one perform due diligence to avoid patent infringement then?

Looking only at memory changes that concern programming instructions doesn't help much. Such instruction is data held in memory and may be changed as fast as the speed of memory permits.

Let's take a look at some of the instances where the actions of a user may cause a new program to be loaded in memory.

- You click on a menu item.
- You visit a web site and Javascript is embedded in the page.
- You visit a web site and a Flash or Java applet is downloaded.
- Your operating system downloads a security patch.
- You use the add/remove application menu option of your Ubuntu system to get application from the web.
- You open a PDF file. The PDF document is a series of instructions written for execution by a virtual machine.
- You open a text document that contains macros.
- You open a spreadsheet that contains formulas. Solving formulas in a spreadsheet counts as an algorithm, doesn't it?
- You use a program that generates and executes (or interprets) code on the fly. This includes many circumstances when your program accesses a relational database using SQL, because this language does not specify the algorithm in the code. The algorithm is determined dynamically by the language interpreter. Since SQL is the dominant standard for databases this situation covers most programs that use databases.

From a user perspective these activities are the normal operation of the computer. It is not reasonable to ask the users to conduct a patent search for due diligence against infringement every time they perform one of these actions. This cannot be the correct interpretation of patent law.

Software is Mathematics

The preceding section was a series of case studies explaining why lawyers should know computation theory. Now it is time to return to the questions that started this section. What is a mathematical formula when implemented in digital electronics? What is an application of the formula as opposed to the formula itself? We already know the answer. Software is mathematics. Everything that is executed by a CPU is mathematics. The boundaries of mathematics lie in the interaction of the computer with the

outside world.

This idea is a hard one to understand unless you know computation theory. When you don't know computation theory, all sorts of misconceptions occur. For example patent attorney Gene Quinn has written an argument against software being mathematics. Computation theory is conspicuous by its absence in this article. He is an eloquent man. Someone who doesn't know computation theory will likely be convinced. If you bring computation theory into the picture, however, his argument falls apart.

I think this quote summarizes the core of his position:

> I think the disagreement between me and my critics is largely due to the way patent law views mathematics, which seems admittedly quite different from the way that many computer programmers view mathematics.
>
> It is impossible to argue that software code does not employ mathematical influences, because it does. I think Dijkstra's explanation that a "programmer applies mathematical techniques" is completely true and accurate. Having said this, the fact that mathematical techniques are employed does not as a matter of fact mean that software is mathematical. Under the US patent laws you cannot receive a patent that covers a mathematical equation or a law of nature. You can certainly use mathematical equations and laws of nature as the building blocks to create something that is new and nonobvious that is patentable. So even if software used mathematical equations there would be no prohibition against the patenting of software under a true and correct reading of the US patent laws.

If I paraphrase the argument, Quinn thinks there are some mathematical equations or methods on one side and the programmer uses them to write software on the other side. Then he says "Look, the maths are over there. This is not the same as the software over here." Of course when couched in these terms, software is different from mathematics. For example if someone write a program simulating chemical reactions, the laws of chemistry are mathematically defined and they are different from the software. Or similarly if one uses mathematical influences to reason about the code, the influences are not the code.

The error is that the mathematics of computation theory is not the mathematics Quinn is pointing to. You can't understand what is meant by "software is mathematics" unless you understand computation theory.

What does it take for the law to find as a matter of fact that software is mathematical? Do the opinions of mathematicians and computer scientists trained in mathematics counts? Or only lawyers?

Since Quinn mentioned Dijkstra, let's explain what are the mathematical methods he advocated.[54] He wanted the programmers to use Hoare logic to formally prove the programs before they are compiled and run for the first time. As we have said, Hoare logic is a formal system wrapped around an imperative programming language. Here is what Hoare himself says in the introduction of the seminal paper where he first proposed his logic:

> Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some

> extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

When one writes programs using this method, the program is a by-product of writing a mathematical proof. Software is not "employing mathematical influences" to use the words of Quinn. The statements in the program are providing the applicable inference rules that are used to write the proof. When you are done with the proof, all that is left to do to get actual running code is insignificant post-solution activity. You take the code that has been proven and compile it.

When software is written is this manner, it is as mathematical as anything could be. But if this is not sufficient to convince a court, how about software developed using the Curry-Howard correspondence? This improves on the above methodology in that there is no difference between the text of a mathematical proof and the text of the program. You can compile the proof directly to get executable code.

I will admit not all software is developed in such manner. In fact most programmers never use such methods. Does the way software is developed count for patent infringement determination? Do we need to make a difference between software developed mathematically which is not patentable and software which is developed using a non-mathematical method which is patentable? Perhaps once it were decided that formal verification techniques bring the code into a patent-free haven, these methods would grow more popular. The industry could avoid billions of dollars in patent infringement liabilities. A side-effect would be great improvements in the reliability of software due the the drastic reduction in the number of bugs. But in such scenario what would be the point of software patents?

We don't need to consider such hypotheticals. The activity of the computer itself is mathematics no matter how the source code is written. There the argument goes:

- Is the activity of a human carrying out an effective method mathematics? Mathematicians think so.
- Is the activity of the same human typing on an idealized typewriter performing the same effective method mathematics? This is the same work using a different tool.
- If we automate the idealized typewriter to work by itself without a human, is it doing mathematics? Again this is the same work being done.
- If we implement the idealized typewriter in actual real-world electronics, is the computer doing mathematics? You get the picture here. If the human at the start of the chain is doing mathematics, so does the computer.

We may also ask, how about denotational semantics? It is a translation of the code into the language of lambda-calculus. It doesn't matter whether the programmer uses a mathematical methods or not when developing software. The source code has a mathematical meaning regardless of how the developer wrote it.

Now contrast the above with this quote from Quinn:

> Returning to the matter of software generally, the logic employed by the computer programmer is certainly of a mathematic nature, but that is quite different than saying that the code created by the programmer is a mathematical algorithm. Now I am speaking in generalities here, but what I am trying to address is whether all computer software should be considered unpatentable subject matter. I see absolutely no justification for all software to be considered unpatentable subject matter because it is simply not correct to say that software code is the equivalent of a mathematical equation or a mathematical algorithm.

> Employing the same logical structure is certainly wise, and complies with best practice standards for programming, but at the core computer software directs. The code is a series of instructions written using mathematical logic as its foundation. In the patent arena this does not and cannot mean that the patenting of software is the equivalent of patenting mathematics. It merely means that the instructions are written in a language and format that are heavily influenced by mathematics.

This whole paragraph betrays a complete lack of knowledge of computation theory. From an extensional perspective, software is data that represents the quintuples of a Turing machine. This is a mathematical entity. Or if you prefer to take an intensional view and consider the specifics of the hardware architecture, the instruction set of the CPU is a machine of some sort based on a computational model that is different from but still reducible to a Turing machine. This machine is universal in the sense that it can perform any computation provided it is given a symbolic description of this computation. Software is this description. Both the universal machine and the description of the computation are mathematical entities. The patentable physical devices are the actual CPU as it is etched in silicon and the computer, including memory, buses, motherboard etc. When the computer computes, it is doing mathematical work exactly like a human being working with pencil and paper.

We can find the root of Quinn's misconceptions in this quote:

> The influence mathematics has on programmers is largely or perhaps even solely related to work-flow. Mathematics teaches us how to manage a problem by going through the steps to solve the problem in a predictable and traceable manner. By employing the same type of thoughtful, step-by-step approach to writing code the programmer can manage write segments of code and tie everything together into an overall structure that will deliver the desired functionality.

Contrast this view with the use of Hoare logic or Curry-Howard correspondence. Contrast this view with denotational semantics. Contrast this view with the fact that the computer itself is executing an effective method. The role of mathematics in software is not limited to work flow and debugging. Without knowledge of computation theory there is no way to understand how deep the rabbit hole goes.

There is one more interesting argument in this quote:

> I just cannot reconcile in my mind why employing lessons learned from mathematics would, could or ought to render software unpatentable. If you take this logic to its extreme you are left with absolutely nothing being patentable, which I suspect is the goal of some, but certainly not all, of those who would rather software not be patentable subject matter. Again, it is undeniable that mathematics is the backbone to virtually everything. Virtually everything can be explained using mathematical models. If you combine mathematics and physics there is virtually no mechanical, electrical or chemical phenomena that cannot be explained in a descriptive way. So does that mean that mechanical, electrical and chemical inventions ought not to be patentable?

Here we are looking at a two-way street. It is correct that we can describe just about everything mathematically. It is correct that such descriptions don't make anything unpatentable. But we can also use something physical to describe mathematics. In such case patenting the physical representation is patenting mathematics. We don't patent mathematical textbooks. Quinn's error is he is looking at one side of the street without considering the other. This is understandable considering the fact he is not looking at the same mathematics as we are discussing.

When we want to invoke this argument to dismiss the use of mathematics and claim the physical reality is patentable, we have an obligation. We need to make sure we correctly understand what part of mathematics is involved, what is the corresponding physical entity, and how they relate. If mathematics describes the physical entity, then the physical entity may be patentable. But what if things go the other way round? In the case of software the analysis goes like this:

- When a human being writes something mathematical with pencil and paper, does the text describes mathematics? Or does the mathematics describe the text? The text describes mathematics of course.
- When the human uses a typewriter to write the same mathematics, the written text still describes the maths. The maths don't describe either the text or the typewriter.
- If we automate the typewriter to execute an effective method, the relationship between the text and the typewriter doesn't change. The text describes the maths. The maths don't describe the automated typewriter nor the text.
- If we build an electronic device that does the same calculations as the automated typewriter but with symbols encoded electronically instead, does their meaning change? Of course not, the symbols still mean the mathematics. The maths don't describe the computer or anything physical in the computer. The maths don't describe the software either.

As an concluding remark, did you notice how I have used reducibility to track where the meaning of symbols lies? This is exactly the sort of thing mathematicians have used reducibility for. This sort of thing is exactly why mathematicians prefer an extensional view over an intensional one to track abstract concepts across different representations.

References

Cases

Application of Walter D. BERNHART and William A. Fetter. 417 F.2d 1395

Bilski v. Kappos, Amicaus Brief of Professor Lee A. Hollaar and IEEE-USA.

Diamond v. Diehr, 450 U.S. 175, 182 (1981)

In re Alappat, U.S. Court of Appeals Federal Circuit, July 29, 1994

Parker v. Flook 437 U.S. 584 (1978)

On the Web

[Agda] Agda: an interactive proof editor

[Campbell] MacGregor Campbell, Universal kernel code to keep computers safe, from New Scientist,

[Cayenne] Cayenne hotter than Haskell

[CLISP] GNU CLISP home page.

[Compcert] Compcert compilers you can *formally* trust

[Coq] The Coq proof assistant

[Dictionary] Dictionary of Algorithms and Data Structures

[Dijkstra] On the cruelty of really teaching computer science (manuscript) (html)

[Epigram] Home page of the epigram project

[Groklaw] Correcting Microsoft's Bilski Amicus Brief -- How Do Computers Really Work?

[Metamath] Metamath Home Page

[Quinn] Groklaw Response: Computer Software is Not Math

[Turing Archives] The Turing Archives for the History of Computing. This site is maintained by Jack Copeland, professor at the University of Canterbury, New Zealand

[Stanford] The Stanford Encyclopedia of Philosophy

In Print

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

[Gödel 1986] Gödel, Kurt, 1986, *Collected Works*, vol. 1, Oxford: Oxford University Press.

[Hoare 1969] Hoare, Charles Anthony Richard, An Axiomatic Basis for Computer Programming, Communications of the ACM, October 1969 pp. 580

[Kleene 1950] Kleene, Stephen Cole, Introduction to Metamathematics, D. Van Nostrand Company, 1950

[Soare 2009] Turing Oracle Machines, Online Computing, and Three Displacements in Computability Theory, *Annals of Pure and Applied Logic* , to appear in volume of Proceedings of Computability in Europe, Siena, 2007 meeting. Available on-line [Link] In a private email Dr Soare informs me this has been published from Publisher Elsevier. However I don't have the actual publication title. I have used the on-line version.

[Stoy 1977] Stoy, Joseph E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. The MIT Press, 1977

[Turing 1936] Turing, Alan, On Computable Number with and Application to the Entscheidungsproblem, Proceeding of the London Mathematical Society, ser. 2, vol 42 (1936), pp. 230-67. Correction: vol 43 (1937) pp. 544-546. This paper could be ordered from the publisher here [Link] This paper is available on-line here [link]

[Turing 1939] Turing, Alan, Systems of logic based on ordinals, Proceeding of the London Mathematical Society, vol 45, Part 3 (1939) pp. 161-228. This paper is available on-line here [Link]

Footnotes

1 I believe the rulings in *Benson*, *Flook* and *Diehr* by the Supreme Court of the United States. all reached conclusions that I consider to be technically correct.

2This is why the Stanford Encyclopedia of Philosophy is one of the authorities I will quote. See [Stanford]

3 From the article on the Church-Turing Thesis at the Turing Archives for the History of Computing. See also [Turing Archive] from the references.

4 Prior to the invention of the electronic digital computer, the word "computer" referred to humans doing calculations. See the Brief History of Computing at the Turing archives [Turing].

5 The decimal expansion of the number π has an infinite number of decimals. No matter how fast we

calculate them, the calculation will never end. Therefore the definition of what constitutes a mathematical calculation must not be contingent on the practical limitations of a human doing the calculation.

6 This text was written before the Supreme Court ruling on *Bilski v. Kappos*. I don't know if the ruling will affect the accuracy of this sentence.

7 Some of this history is summarized in chapter III of [Kleene 1950]. Another account that goes way over my head in the mathematical details may be found in the article on Paradoxes and Contemporary Logic of the Stanford Encyclopedia of Philosophy. See also the article on Russell's paradox.

8 Martin Davis traces this idea back to Leibnitz (1646-1716). See [Davis 2000]

9 In this section on formal systems I dig deeper into the connection between algorithms and abstract thinking. Computation theory has very close ties with the very definition of what is a mathematical proof, and I am exploring some of that here. I am laying some of the ground work I need to explain why software is discovered and not invented, why software is abstract and why software is mathematics. T

10 Inference is a fancy word that here means deduction.

11 A complete list of these rules can be found in chapter IV section 19 of [Kleene 1950]

12 This means mathematicians can't argue their theorem with a mesmerizing tap dance and hope nobody will follow up and check it, as sometimes lawyers seem to do in litigation. Other mathematicians will never look at how plausible the argument may sound. They will check each inference one by one and make sure every single one of them follows the rules. As soon as they find a single inference that doesn't clearly follow the rules, the whole proof is ruthlessly rejected.

13 The consequence of this fact is there is no dispute among mathematicians on whether a theorem is proven or not. They check the proof and either it passes the test or it doesn't. This fact may be put to productive use by lawyers. In any issue where mathematical truth is relevant, theorems that are brought into the court record will not be disputed by anyone competent to do so.

14 This is why you have axioms for arithmetic, other axioms for geometry, yet other axioms for set theory etc. The intuitive truths at the foundation of any mathematical discipline are different. The axioms are chosen to reflect these truths.

15 This was called by Hilbert metamathematics which means mathematics about mathematics. See [Kleene 1950]

16 This requirement further expands on the role effective methods play in Hilbert's program. However this part of the program cannot be done. There are theorems that show that effective methods to solve all mathematical questions in a consistent formal system suitable to be the foundation of all of mathematics are not mathematically possible. However it is possible to develop algorithms of a more limited scope that will automatically discover the proof of some theorems. According to the article on Automated Reasoning of the Stanford Encyclopedia of Philosophy, these algorithms have reached the point where they can be used by researchers to attack open questions in mathematics and logic and to solve problems in engineering. This is further evidence of the tie between computer programs and abstract thinking.

17 There is a risk of circular reasoning in this logic. If the mathematical analysis of the formal system is itself subject to paradoxes, then the proofs envisioned by Hilbert would be unreliable and nothing would be really solved. This is why Hilbert further required that the proof of consistency and completeness must be done through finitary means. The reason is all the paradoxes that plagued

mathematics during Hilbert's time were located in parts of mathematics that were dealing with infinity in one way or another. Hilbert argued the proofs of consistency, completeness and decidability should only use parts of mathematics that are considered safe from paradoxes and therefore could be relied on.

18 See chapters IV to VIII of [Kleene 1950] for the development and analysis of a formal system sufficient to form the basis of the arithmetic of integers. Most of mathematics can be based on a formal system called the Zermelo-Fraenkel set theory.

19 A very readable account of this history can be found in [Davis 2000]

20 This is where dictionary definitions of algorithm are lacking. For example the Dictionary of Algorithms and Data Structures [Dictionary] defines algorithm as "a computable set of steps to achieve a desired result." Literally the definition is correct. But it doesn't capture nuances like degrees of abstraction such as the one illustrated in the example of the C sorting program. Lawyers working from dictionary definitions without knowledge of computation theory have no chance of really understanding algorithms.

21 In mathematics a pair of mathematical objects is also a mathematical object.

22 This explanation is based on the simplest form of recursion called primitive recursion. Effective methods are equivalent to recursion in its full generality. See [Kleene 1950] for an complete explanation. See also the article on Recursive Functions in the Stanford Encyclopedia of Philosophy.

23 Gödel has provided a full translation of a formal system called Principia Mathematica in his seminal paper *On formally undecidable propositions of Principia Mathematica and related systems.* The original was in German and published in 1931. An English translation is available in [Gödel 1986] pp 144-195.

24 Two of these consequences are the famous Gödel undecidability theorems. They proved that Hilbert's program is impossible to fulfill because when a formal system is powerful enough to serve as the foundation of Peano arithmetic, then consistency and completeness are mutually incompatible. A system that is consistent cannot be complete and a system that is complete cannot be consistent. These theorems are outside the scope of this text.

25 Reducibility is a term of art in computation theory.

26 It can be said that the bits in computer electronics are numbers. Textual symbols are represented using numeric encodings like ASCII or Unicode. The original Gödel numbering system is not friendly to modern electronics and more convenient encodings were developed. This doesn't change the substance of the argument being made. The bits are symbols that represent something abstract.

27 We can make such a list by listing all series of one symbol A, B .. Z, then all series of two-symbols AA, AB ... ZZ and then series of three symbols, four symbols etc., and we filter out those series of symbols that are not valid programs according to the rules of syntax of the programming language. Whatever program someone writes, it must be on this list.

28 Abstraction and application are terms of art specific to lambda-calculus.

29 There is also an alpha-reduction that let you rename a variable. For example (λx.x+7) may be rewritten (λy.y+7) without changing the meaning of the abstraction.

30 Nowadays this is called "pure" lambda-calculus. There are variants of lambda-calculus that augment lambda-calculus with operators from other branches of mathematics.

31 This translation is provided in detail in chapter 5 of [Stoy 1977] among other places. Note that I said the computations can be translated. This means the recursive functions are translated, and this is the point that is of interest to computation theory. The method I am referring to doesn't have the capability

to reproduce predicate calculus which is an integral part of Peano arithmetic. Therefore this part of Peano arithmetic is not included in the model.

32 For those of you who want a taste of how modeling works, I give some more explanations. A model required to reproduce all the base concepts and then demonstrate the translations obeys the same logical rules as the original.

We have seen how zero is translated. We also need a translation for the successor function; this is the apostrophe in Peano arithmetic. The translation is λzλsλx.s(zsx). Why this cryptic formula? It is because when you apply it to one of the formulas that are used to translate numbers and perform beta reduction you will obtain the translation of the next number. Let's see an example of how it works.

The formula below applies the successor to zero. After beta reduction you should get 1.

(λzλsλx.s(zsx))(λsλx.x)

Now we need to perform beta reduction on z to resolve the application.

λsλx.s((λsλx.x)sx)

Within the parentheses, we have a pair of successive applications (λsλx.x)sx.. In situation like this the rules of lambda-calculus say you must perform the two beta reductions one at a time. The first substitution requires you to replace s with s in λx.x. But there is no s in there. This is just a do-nothing operation that removes what is unnecessary.

λsλx.s((λx.x)x)

The next beta reduction requires to replace x for x. Again this is a do-nothing operation the removes more that is unnecessary.

λsλx.s(x)

This result is indeed -- the intended translation for the number 1.

As you can see, these kinds of models are not intended to provide an efficient method to do arithmetic. They are intended to study the expressive power of formal languages and gain understanding of how abstract concepts relate to each other.

33 There exist an infinite number of algorithms that may serve this purpose. As noted above, the native algorithm of lambda-calculus is too inefficient to be put into practical use. It can be improved into something that can be used in practice. Languages such as LISP make use of the improvements.

34 I have found this distinction in Soare [2009] p 13. This is a slightly different concept from extensionality as used in set theory.

35 I write "similar" concept and not "identical" concept. The law will consider issues like derivative works that have no equivalent in mathematics.

36 This fact contradicts the point of view of those that want to patent software on the basis that transistors and other signals are changing state in memory. This is an intensional detail that is absent from both the extensional properties of the software and the extensional test for patent infringement. If the test for infringement is extensional, then the invention subject matter must be defined extensionally.

37 For a more detailed explanation, see for example [Stoy 1977]

38 A debugger is a programming tool that lets a programmer verify the execution of a program and inspect the computer memory as it executes. Its purpose is to help find bugs and understand the error so it can be corrected.

39 See *Bilski v. Kappos*, amicus brief of Professor Lee A. Hollaar and IEEE-USA.

40 Imperative languages are made of statements that translate directly into instructions that can be executed by a CPU. These languages are well suited for programs meant to be executed into single CPU computers. They are popular because single CPU computers have dominated the market for so many years. Much of the understanding of programming that lawyers have developed is based on compiled imperative languages. This understanding often doesn't apply when other types of languages are considered. Languages can also be interpreted, or they can be something that is not imperative at all. It is dangerous to make case law based on compiled imperative languages without considering how the underlying logic apply to other types of languages.

41 The classical view is you are allowed to argue by contradiction. You assume a solution doesn't exist and prove a contradiction. Therefore some solution must exist somehow, even if you don't actually know what it is. Intuitionistic logic disallows this form of reasoning.

42 Examples of research in this direction are Coq, Agda, Cayenne and Epigram.

43 The Compcert compiler is an example. From the Compcert page: "The Compcert verified compiler is a compiler for a large subset of the C programming language that generates code for the PowerPC processor. The distinguishing feature of Compcert is that it has been formally verified using the Coq proof assistant: the generated assembly code is formally guaranteed to behave as prescribed by the semantics of the source C code.".

44 Namely it eliminates all bugs that are due to the actual code not matching the intent of the programmer as expressed in the proof. It doesn't eliminate the bugs due to the programmer intending to do the wrong thing, it does not eliminate the bugs due to the programmer proving something that doesn't mean what he thinks it means, and it does not eliminate problems due to the hardware not working as expected. But still this is a very large category of bugs that is gotten rid of.

45 This method has been first proposed by C.A.R. Hoare in [Hoare 1969]

46 An account of this story can be found in [Davis 2000] chapter 7. Turing's original analysis is found in [Turing 1936] section 9.

47 A summary of this history can be found in the Stanford Encyclopedia of Computing. This is also the topic of chapters 7 and 8 of [Davis 2000]. The Turing Archive brief history of computing reports the following:

> Turing's computing machine of 1935 is now known simply as the universal Turing machine. Cambridge mathematician Max Newman has remarked that right from the start Turing was interested in the possibility of actually building a computing machine of the sort that he had described (Newman in interview with Christopher Evans ('The Pioneers of Computing: an Oral History of Computing, London Science Museum, 1976)).
>
> During the Second World War, Turing was a leading cryptanalyst at the Government Code and Cypher School, Bletchley Park. Here he became familiar with Thomas Flowers' work involving large-scale high-speed electronic switching (described below). However, Turing could not turn to the project of building an electronic stored-program computing machine until the cessation of hostilities in Europe in 1945.

See also from the same article Turing's work with Colossus and ACE.

48 The Stanford Encyclopedia of Philosophy describes the contribution of John Von Neuman like this:

In 1944, John von Neumann joined the ENIAC group. He had become â€˜intrigued' (Goldstine's word, [1972], p. 275) with Turing's universal machine while Turing was at Princeton University during 1936–1938. At the Moore School, von Neumann emphasised the importance of the stored-program concept for electronic computing, including the possibility of allowing the machine to modify its own program in useful ways while running (for example, in order to control loops and branching). Turing's paper of 1936 (â€˜On Computable Numbers, with an Application to the Entscheidungsproblem') was required reading for members of von Neumann's post-war computer project at the Institute for Advanced Study, Princeton University (letter from Julian Bigelow to Copeland, 2002; see also Copeland [2004], p. 23). Eckert appears to have realised independently, and prior to von Neumann's joining the ENIAC group, that the way to take full advantage of the speed at which data is processed by electronic circuits is to place suitably encoded instructions for controlling the processing in the same high-speed storage devices that hold the data itself (documented in Copeland [2004], pp. 26–7). In 1945, while ENIAC was still under construction, von Neumann produced a draft report, mentioned previously, setting out the ENIAC group's ideas for an electronic stored-program general-purpose digital computer, the EDVAC (von Neuman [1945]). The EDVAC was completed six years later, but not by its originators, who left the Moore School to build computers elsewhere. Lectures held at the Moore School in 1946 on the proposed EDVAC were widely attended and contributed greatly to the dissemination of the new ideas.

Von Neumann was a prestigious figure and he made the concept of a high-speed stored-program digital computer widely known through his writings and public addresses. As a result of his high profile in the field, it became customary, although historically inappropriate, to refer to electronic stored-program digital computers as â€˜von Neumann machines'.

49 Some of this analysis is described in the Stanford Encyclopedia of Computing.

50 These consequences are in addition to the other arguments supporting the same conclusions that are scattered throughout this article.

51 See [Turing 1939]

52 Turing assumed the oracle would be some kind of mathematical function. Given a physical computer it is easy to extend the concept to forms of input that are outside the scope of pure mathematics. Computers can be connected to physical measurement devices.

53 See also [Davis 2000] chapter 8.

54For instance see On the cruelty of really teaching computer science. (html) See [Dijkstra] for a link to a scan of the original manuscript.

Before we part, I would like to invite you to consider the following way of doing justice to computing's radical novelty in an introductory programming course.

On the one hand, we teach what looks like the predicate calculus, but we do it very differently from the philosophers. In order to train the novice programmer in the manipulation of uninterpreted formulae, we teach it more as boolean algebra, familiarizing the student with all algebraic properties of the logical connectives. To further sever the links to intuition, we rename the values {true, false} of the boolean domain as {black,

white}.

On the other hand, we teach a simple, clean, imperative programming language, with a skip and a multiple assignment as basic statements, with a block structure for local variables, the semicolon as operator for statement composition, a nice alternative construct, a nice repetition and, if so desired, a procedure call. To this we add a minimum of data types, say booleans, integers, characters and strings. The essential thing is that, for whatever we introduce, the corresponding semantics is defined by the proof rules that go with it.

Right from the beginning, and all through the course, we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification. While designing proofs and programs hand in hand, the student gets ample opportunity to perfect his manipulative agility with the predicate calculus. Finally, in order to drive home the message that this introductory programming course is primarily a course in formal mathematics, we see to it that the programming language in question has not been implemented on campus so that students are protected from the temptation to test their programs. And this concludes the sketch of my proposal for an introductory programming course for freshmen.