-----Original Message-----
From: Pamela Jones
Sent: Sunday, February 10, 2013 12:42 PM
To: SoftwareRoundtable2013
Subject: Groklaw's Comment to Topic 1 (improving clarity of claim boundaries that define the scope of patent protection for claims that use functional language)

Gentlemen:

Thank you for expressing a desire to work more closely with the software community. Groklaw represents a large chunk of the software community, being a membership-produced journalistic website, one that covers legal events of particular interest to the technical community, especially to Free Software and Open Source software developers.

We are applying open-source principles to research and journalism to the extent that they apply. Our community us predominantly made up of software developers, but our membership includes those with a technical background in other fields, including mathematicians and physicists, CEOs of startups, developers of apps for mobiles, others with legal and paralegal training, as well as journalists, educators, and many end users who care enough about operating systems to want to volunteer to make Groklaw able to report effectively on patents, which we have been doing for a decade.

So, when we saw your request for comments on Topic 1, on how to improve clarity of claim boundaries that define the scope of patent protection for claims that use functional language, naturally we wanted to respond. I attach a copy of our comment, which was written by many members as a group work, and then opening up the draft to the world for their comments. By incorporating all the suggestions and comments from both sources, I believe it now adequately reflects widely held beliefs in the software developer community about means plus function claiming and some ways to improve them. We hope you find it helpful.

Pamela Jones
Founder and Co-Editor, Groklaw
http://www.groklaw.net

Groklaw's Response to the USPTO on the First Topic

Establishing Clear Boundaries for Claims That Use Functional Language

The USPTO has launched a Software Partnership initiative to enhance the quality of patents. It has invited the public to comment on three topics. This is the response of Groklaw to Topic 1, "Establishing Clear Boundaries for Claims That Use Functional Language".

Summary of the response

Claims on the functions of software without an accompanying algorithm result in granting rights to inventions the patentee has not invented.

This view is similar to a legal analysis by Mark Lemley.[1] Lemley argues based on legal considerations; Groklaw reaches this conclusion on technical grounds.

The functions of software should always be explicitly accompanied by a corresponding algorithm unless the algorithm is known from the prior art. Similarly, algorithms should be detailed up to the point where there is prior art for the implementation of all elements used to describe the individual steps. This rule should be applicable to all forms of claiming because the problems of functional claiming without an accompanying algorithm occur no matter how the claim is drafted.

We disagree with Lemley when he says functional claiming is the source of most of the ills of software patents. We believe that patents on abstract ideas are also problematic. It is possible to write claims on abstract ideas using very specific terms, so invalidating vague or overly broad patents will not solve the problems caused by claims on abstract ideas.

This response is divided in three parts followed by an addendum providing a supplementary comment:

- A. Permitting recitations of software functions without specifying their accompanying algorithms leads to grants of rights to inventions the patentee has not invented.
- B. Algorithms should be detailed up to the point where all functional elements have corresponding algorithms which are either explicitly disclosed or known in the computer programming art.
- C. Providing sufficient structure for functional language will reduce needless litigation but is not a substitute to section 101 subject matter analysis.
- Addendum. Observations on the practical ineffectiveness of patent disclosure in promoting innovation in computer programing.

A. Permitting recitations of software functions without specifying their accompanying algorithms leads to grants of rights to inventions the patentee has not invented.

This response documents this issue from the point of view of the mathematical theory underlying computer science. It shows why the problems of functional claiming occur no matter which form of claim language is used.

The key concept is the distinction mathematicians draw between a function and an algorithm. Hartley Rogers explains:[2] (emphasis in the original):

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

---

1 See Mark Lemley, Software Patents and the Return of Functional Claiming

2 See Rogers, Hartley Jr., *Theory of Recursive Functions and Effective Computability,* The MIT Press, 1987 pp. 1-2

A mathematical function is a correspondence between one or more input values and a corresponding output value. For example the function of doubling a number associates 1 with 2, 2 with 4, 3 with 6 etc. Non-numerical functions also exist. A function is not a process. There is no requirement that the function must be computed in a specific manner. All methods of computation which produce the same output from the same input compute the same function.

A method for computing the function is called an "algorithm". There is a series of steps to be executed starting with the input and ending when the output is produced. The ordinary procedures of arithmetic for adding and multiplying numbers using pencil and paper are examples of algorithms. In mathematics, nonnumerical algorithms for processing nonnumerical data also exist. Despite the similarly sounding words, a software function is not the same thing as a mathematical function. However the two concepts are closely related. If we look at the principles of mathematics underlying computer programming the functions of software are described with mathematical functions.[3] The methods used to perform the functions of software are implemented using mathematical algorithms. As indicated by Hartley Rogers, the same function may have several algorithms. For example we may double a number by multiplying it by 2, or we may add it to itself. Both ways the number is doubled.

Another example of a function with many algorithm for it is sorting. A sorting function takes as input a list or array of values in some random order and its output is a list or array of the same values sorted according to some predetermined order. A large number of sorting algorithms is known.

When the algorithm for a software function is not provided, **the claim will read on all algorithms for this function**. This implies the claim may read on methods the patentee has not invented. This shows the mathematical principles underlying computer science are supporting Lemley's description of the problems caused by functional claiming.[4]:

> While experienced patent lawyers today generally avoid writing their patent claims in means-plus-function format, software patentees have increasingly been claiming to own the function of their program itself, not merely the particular way they achieved that goal. Both because of the nature of computer programming and because of the way the means-plus-function claim rules have been interpreted by the Federal Circuit, those patentees have been able to write those broad functional claims without being subject to the limitations of section 112(f). They have regained the ability to claim ownership not of what they built, but of what it does. They claim to own the function itself.

There is another problem with functional claiming that Lemley has not identified. The same algorithm may be used to compute multiple functions. For example, an algorithm for multiplication may double, triple or quadruple a number. Several well-known algorithms in computer science are routinely used in such a versatile manner. An example is an algorithm for recognizing regular expressions, which may be used for a wide range of text processing functions. Therefore it is possible that a software function in a claim is actually an obvious use of an old algorithm. This circumstance will be easier to identify if we require that a software function is accompanied with the corresponding algorithm.

And there is still another problem with functional claiming. Sometimes the exact same algorithm implements multiple software functions when *applied to the same input*. An example may be a routine to draw the shape of a three-dimensional parabolic surface. This routine may be mentioned in a claim as a method for drawing the form of different physical devices, like radio-frequency antennas, mirrors for telescope or directional microphones. The mention of the object will make each of these claims sound like they are reciting different functions, but in reality the objects may have the same shape and dimensions. Then exactly the same code is used on the same numeric

---

3 See for instance textbooks on denotational semantics for one method of writing such descriptions. An example of such textbook is Stoy, Joseph E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, First Paperback Edition.

4 See Lemley *supra*, pp.2-3.

data. Nothing new has been invented. Once again this circumstance will be easier to identify if we require that a software function is accompanied with the corresponding algorithm.

There is a fourth problem with functional claiming. The knowledge of the functions of the software is not a guarantee that the patentee has a working algorithm in hand. Some functions are notoriously hard to program. For example cryptography relies heavily on the difficulty of finding algorithms for large integer factorization that are fast enough to be used in practice. The extreme cases are the undecidable problems. These are functions for which we have a mathematical proof that no corresponding algorithm can possibly exist. A patent may in theory recite a function like these, but its disclosure doesn't enable anyone to implement a corresponding algorithm. In fact, without disclosure of a specific algorithm, one has reasonable grounds to doubt the patentee has a practical way to implement the function claimed as his possession.

Whether the claim reads on a non-existant, new, yet to be discovered algorithm or on an obvious use of an old algorithm, it reads on an invention the patentee has not invented. In all cases, this is a problem.

The cause of these problems resides in part in the mathematical principles underlying computer programming. As Lemley points out, they occur no matter which form of claim language is used. It should not matter whether or not the claim is written in traditional 112(f) means-plus-function format. Either way, a claim element described in functional language without the corresponding algorithm will result in the same problem.

B. Algorithms should be detailed up to the point where all functional elements have corresponding algorithms which are either explicitly disclosed or known in the computer programming art.

This section addresses the degree of detail which should be required to meet the sufficient description requirement.

For example, if a claim recites the function of adding numbers, there is no requirement to describe a method for addition, because this method is well-known in the computer programming art. But when a claim recites a function for which no algorithm is known in the prior art, an algorithm must be explicitly disclosed.

An algorithm is usually described by elaborating it with an increasing degree of detail in a hierarchical manner. The rule for sufficient description should be applied at all levels of the hierarchy.

Here is how it would work: An algorithm is made of steps. Each step is itself described in functional terms. Therefore each step is itself a functional element for which an algorithm must be provided. This more granular algorithm is also made of steps described in functional terms. Algorithms for the more granular steps must also be provided. This elaboration of steps into more granular algorithms must continue until it is known from the prior art how to implement all individual steps. Then the algorithms for all functional elements are either explicitly disclosed or already known. This rule for what is sufficient structure should be applied for both purposes of enablement and defining the boundaries of claims. Anything less permits incomplete disclosure. Also, if some steps are allowed to be described in functional terms without disclosing a corresponding algorithm, we just move the problems of functional claiming from the main functions of software to the functions of individual steps. This doesn't accomplish much. Functional claiming will persist because applicants will cleverly write patent language that uses this loophole.

C. Providing sufficient structure for functional language will reduce needless litigation but is not a substitute to section 101 subject matter analysis.

Providing this degree of detail in the structure for functional elements will help reject vague or overly broad patents. The approach proposed by Groklaw has an additional benefit. It should create a presumption that any functional language recited without a corresponding algorithm is

within the prior art, otherwise the claim is invalid as indefinite. This presumption may help curtail baseless litigation with summary judgment motions on invalidity under sections 101, 102 or 103 because there will be fewer material issues of fact that could be disputed.

This is a highly desirable result but this doesn't resolve all the problems with software patents. In particular, Section 101 analysis is still required to determine whether a claim is drawn to an abstract idea. For example consider a claim on a computer programmed to compute the location of points on a non-vertical straight line in plane geometry according to the well-known formula $y=mx+b$. This claim is on a method to compute the y-coordinate of a point when the x-coordinate is known.

> A computer programmed for computing the y-coordinate using a method comprising:
> 1. a step of identifying the value x of the x-coordinate, and;
> 2. a step of multiplying the x-coordinate with the slope of the line m to obtain an intermediate result mx, and;
> 3. a step of adding the intermediate result mx with the value b of the y-coordinate at the origin to obtain the value y of the y-coordinate.

This is clearly a claim on an abstract mathematical algorithm. This claim is not vague and it is not overly broad. It doesn't use functional claiming. It is invalid because it is abstract. It is also invalid because the formula is old, but how about similar claims where the formula is new and nonobvious? They can only be invalidated using section 101.

This observation is typical. In mathematics an algorithm must be given at a great degree of specificity, otherwise it is not an algorithm in the sense mathematicians give to this term. Boolos, Burgess and Jeffrey explain:[5]:

> The instruction must be completely definite and explicit. They should tell you at each step what to do, not tell you to go ask someone else what to do, or to figure out for yourself what to do: the instructions should require no external source of information, and should require no ingenuity to execute, so that one might hope to automate the process of applying the rules, and have it performed by some mechanical device.

Any attempt to equate the abstract idea exception with a prohibition of vague or overly broad claims is inconsistent with the notion that mathematical algorithms *are* abstract ideas. Also we believe that many non-mathematical abstract ideas may be described in specific terms. Software development is an incremental activity. A complex software may use thousands of ideas. Each of these ideas is a basic building block which could be used in a large number of different pieces of software. Each of these ideas may be the subject matter of one or more patents. Patents on abstract ideas are as damaging as patents on vague or overly broad claims.

Lemley advocates to take section 112(f) seriously. He argues that vague and overly broad claims resulting from functional claiming are the cause of most if not all the ills affecting software patents.[6] (footnote omitted)

> It is broad functional claiming of software inventions that is arguably responsible for most of the well-recognized problems with software patents. Writing software can surely be an inventive act, and not all new programs or programming techniques are obvious to outside observers. So some software inventions surely qualify for patent protection. Even if there are too many software patents, the patent thicket and patent troll problems won’t go away if we simply reduce the number of software patents somewhat. And while the lack of clear boundaries is a very real problem, the most important problem a product-making software company faces today is not suits over claims with unclear boundaries but suits over claims that purport to cover any possible way of achieving a goal. The fact that there are lots of

5 See Boolos George S., Burgess, John P., Jeffrey, Richard C., *Computability and Logic*, Fifth Edition, Cambridge University Press, 2007, page 23.

6 See Lemley, *supra*, page 3.

> patents with broad claims purporting to cover those goals creates a patent thicket. And while the breadth of those claims should (and does) make them easier to invalidate, the legal deck is stacked against companies who seek to invalidate overbroad patent claims.

While Groklaw agrees that functional claiming is indeed a serious problem that must be fixed it is our position is that claims on abstract ideas are also problematic. If the patent system successfully curtails the use of vague and overbroad claims, patent trolls may shift their activity to claims drawn to abstract ideas. A patent on an idea which may potentially be widely used may be well-suited to patent trolling. Please recall that non-practicing entities may be profitable even when their targets don't infringe on claims. They only need to threaten litigation to extract undeserved settlements. Their business model doesn't require vagueness and breadth of claiming.

Patents on abstract ideas may also be used for other forms of abuse of the patent system, like building patent thickets that impede innovation for anti-competitive purposes.

Addendum. Observations on the practical ineffectiveness of patent disclosure in promoting innovation in computer programing.

We wish to bring to the USPTO's attention some observations on how patent disclosure helps or hinders innovation. These observations should be seen in light of the patent quid pro quo. Patentees are granted exclusive rights for a limited time in exchange for the disclosure of the invention. In the case of software patents, the quid pro quo doesn't work in practice according to the theory, with the result that patent law's ability to promote innovation in software is at least impaired. In our view, software patents are toxic to innovation.

Disclosure is intended to enable a person with ordinary skills in the art to make and use the invention. Most developers are not good at reading legal language, and patents seem to be written so as to obfuscate what the patent actually covers, so for best results, disclosure should be written in a language developers can read. Software programmers would welcome a greater use of textual and graphical notation systems known in the art, such as C-like pseudo-code or XML-like schemas for textual notation and Unified Modeling Language (UML) for graphical notation. We see the request for comments on topic 3 as a positive development.

For developers the preferred form of disclosure is source code for well-written working programs.[7]

Please note that the patent system is not the only incentive for disclosure available in the market. Parties skilled in managing a relationship with a community of developers will use an open development model where information is freely shared. One example is the free and open source (FOSS) development model. Another example is the IETF standard development model.

In the FOSS development model, disclosure of the source of working programs is the norm. Programmers need to read, use, modify and distribute the software in order to contribute modifications. For many businesses, this is R&D which is available at no monetary cost. From the point of view of programmers, this incentive leads to a superior form of disclosure when compared to the patent system.

Examples of software inventions developed and disclosed in this manner are the Perl and Python programming languages, the Linux operating system kernel, and the Coq proof assistant.

The IETF standard development process is different. They write detailed specification documents called RFCs (Requests for Comments). These documents are shared and updated until the participants generally agree these specifications are proven to work with actual implementations.[8] This process has been summarized with the phrase “We reject: kings, presidents, and voting. We

7 The program must be well-written. Obfuscated code is useless.

8 See RFC 2026 for the current version of the RFC development process. RFC2555 is an historic account describing how the RFC process has been used to invent and disclose the core Internet protocols.

believe in: rough consensus and running code." [9] Although this development process doesn't explicitly require the disclosure of source code the publication of reference implementations is a common practice.[10] It is required that a proposed standard must have two independent interoperable working implementations in order to be adopted.[11]

All the core communication protocols of the Internet are developed and disclosed in this manner.

Other inventions which have been disclosed in source code form are the web browser and the web server.[12]

If the goal is to maximize the ability of the patent system to promote innovation, then the source code of a well- written working program should be a required element of disclosure. Programmers are used to source code disclosure from sources other than patents. They will compare the quality of the knowledge they receive from patents with these other sources and they will decide whether patents are worth searching and reading on this basis. Should developers determine patents are not useful enough to be worth the trouble they will not read them, and this undermines the very purpose of the patent quid pro quo. In this scenario the disclosure function of patents is ineffective.

Unfortunately current case law does not favor a strong form of disclosure. For example, Fonar Corporation v. General Electric Corporation:

> As a general rule, where software constitutes part of a best mode of carrying out an invention, description of such a best mode is satisfied by a disclosure of the functions of the software. This is because, normally, writing code for such software is within the skill of the art, not requiring undue experimentation, once its functions have been disclosed. It is well established that what is within the skill of the art need not be disclosed to satisfy the best mode requirement as long as that mode is described. Stating the functions of the best mode software satisfies that description test. We have so held previously and we so hold today. Thus, flow charts or source code listings are not a requirement for adequately disclosing the functions of software.

See also Northern Telecom v. Datapoint Corporation:

> The computer language is not a conjuration of some black art, it is simply a highly structured language . . . . The conversion of a complete thought (as expressed in English and mathematics, i.e. the known input, the desired output, the mathematical expressions needed and the methods of using those expressions) into a language a machine understands is necessarily a mere clerical function to a skilled programmer.

These cases place patent practice at the low end of the range of possible degrees of disclosure. Not only source code is not disclosed, but the algorithm is not available either. Only the functions

---

9 See Andrew L. Russell, Rough Consensus and Running Code and the Internet-OSI Standards War (PDF).

10 An example of a reference implementation is found in RFC1321 Appendix A. Reference implementations may also be incorporated by reference. For example, RFC 5905 includes the reference, "This document includes material from [ref9], which contains flow charts and equations unsuited for RFC format. There is much additional information in [ref7], including an extensive technical analysis and performance assessment of the protocol and algorithms in this document. The reference implementation is available at www.ntp.org." This same RFC 5905 also includes a skeleton program with code segments in appendix A.

11 See RFC 2026 *supra*, section 4.1.2: A specification from which at least two independent and interoperable implementations from different code bases have been developed, and for which sufficient successful operational experience has been obtained, may be elevated to the "Draft Standard" level.

12 The W3C consortium hosts the first web page ever published. It includes instruction on how to obtain the source code for web browsers and web servers.

of the software are required to be provided.

From the perspective of a programmer, these cases eviscerate the usefulness of disclosure. The functions of most software inventions can be defined after a few brainstorming sessions. Turning these functions into a working implementation is still a lot of hard work. When only the functions are known, the programmer is still required to do the bulk of this work.[13] The functions of existing software can usually be seen just by watching the program in action. Developers may watch over the shoulder of a user, or they may inspect the computer internals with debugging tools. Disclosure of the algorithm should be mandatory. Disclosing the functions of software without the corresponding algorithm doesn't disclose any trade secret. Therefore the patent quid pro quo is not actually implemented.

At a strict minimum, the functions of software must always be accompanied by the disclosure of the algorithm unless the algorithm is known from the prior art. This will make the disclosure requirements consistent with the requirements of section 112(f) as we propose in sections A and B of this comment. But for best results source code should be required as well.

The lack of the algorithm and source code are not the only problems with *Fonar* and *Northern Telecom*. From a programmer's point of view, these decisions are factually erroneous.[14] It is not true that “normally, writing code for such software is within the skill of the art, not requiring undue experimentation, once its functions have been disclosed.” Sometimes the algorithm can be easily derived from the disclosure of the software functions and sometimes it can't. This depends on which functions are recited in the patent.

An example of algorithm which is easily derived from the disclosed functions is an arithmetical calculation corresponding to the disclosure of a suitably chosen mathematical formula. Someone with sufficient knowledge of mathematics can figure out the calculation to be performed just by looking at the formula.

Examples of algorithms which cannot be easily derived from a statement of the functions are large integer factorization and undecidable problems. These example were mentioned above. No practical algorithm for large integer factorization is known.[15] Disclosing large integer factorization as a function doesn't enable anyone to write a working program because this is a famous unsolved problem. Once a solution to this problem is found, if ever, then this kind of disclosure may refer to this discovery as prior art. For the time being there is no such prior art.

If an undecidable problem is disclosed as the functions of software, then the situation is totally hopeless. No algorithm at all may ever be written for this function because we have a mathematical proof that none exist.[16] In this case no invention corresponding to this so-called disclosure can ever be made.

The proper test for when the disclosure of the functions suffices to let a programmer write the software is whether the corresponding algorithm is obvious to a person having ordinary skills in the art once the functions are known. If the algorithm is not obvious, then it is not within the ordinary skills in the art to write this program. We are in presence of one of the functions for which algorithms are hard or perhaps impossible to find. The reader of the patent is forced to reinvent

---

13 For the sake of making a comparison, we note that this obligation does not follow from the disclosure commonly available from collaborative development processes. In the case of FOSS, the programmer has access to the source code of a working program he can immediately use. In the case of IETF the programmer can usually refer to a reference implementation. This is why the FOSS and IETF development models lead to a superior form of disclosure.

14 We are not presenting an argument for overruling these cases. We are just noting the errors.

15 Some algorithms that work for some (but not all) large integers have been found. These algorithms partially solve the problem. Other algorithms that work for all integers are known but they are way too slow for practical use. Finally an algorithm is known for quantum computer that should be practical to use but we don't know yet how the build the computer able to run it.

16 Examples of undecidable problems are found at this page on the wikipedia.

from scratch an undisclosed portion of the invention. This is yet another reason why we believe that the best procedure is to leave the algorithm unspecified only when it is known from the prior art, as indicated in section B of this response.

There is still another problem with disclosure. For a computer programmer reading patents is legally dangerous. He risks becoming liable for treble damages for willful infringement. Many computer programmers don't read patents for this reason. The whole point of disclosure is to enable skilled artisans to reproduce the invention. This cannot happen with software when computer programmers don't read patents.

There is a deeper reason behind this situation. Mark Lemley points out how difficult it is for job creating corporations using or selling actual software products to clear all software patent rights on their own products. See Lemley *supra*, page 24. (footnotes omitted)

> Because computer products tend to involve complex, multi-component technology, any given product is potentially subject to a large number of patents. A few examples: 3G wireless technology was subject to more than 7000 claimed “essential” patents as of 2004; the number is doubtless much higher now. WiFi is subject to hundreds and probably thousands of claimed essential patents. And the problem is even worse than these numbers suggest, since both 3G wireless technology and WiFi are not themselves products but merely components that must be integrated into a final product. Some industry experts have estimated that 250,000 patents go into a modern smartphone. Even nominally open-source technologies may turn out to be subject to hundreds or thousands of patents. The result is what Carl Shapiro has called a “patent thicket” – a complex of overlapping patent rights that simply involves too many rights to cut through.

A software developer can't clear all patent rights to his own software because there is no practical way to do so. This is true of all non trivial software. A software developer must choose between taking the risk of being sued by some rights holder or give up developing. If he chooses to take the risk of being sued then any action that exposes him to treble damages is recklessly increasing this risk. In these circumstances not reading patents is a sensible decision. Many software developers report that their employers forbid them from reading patents precisely for this reason.

The inability to clear all patent rights on actual software makes the patent system toxic to all job-creating organizations and individuals using and writing software. These parties can never be assured they own all rights to their own software properties. This impacts individual developers, communities like FOSS projects, nonprofit and commercial entities. Patents are rights to exclude. If all patent rights are not cleared, then any rights holder may sue and seek to enforce his exclusive rights. All software developers and users must live with this sword of Damocles constantly hanging over their head. The only practical alternative is to stop using and developing software because clearing all the rights is not a realistic option. This is a much worse problem than ineffective disclosure. Crippling an entire industry with the inability to clear all rights to one's own property is harmful to innovation and it actively promotes litigation.

This is not a situation that can be fixed through requiring better disclosure. Even in a best case scenario where every patent fully discloses all aspects of the invention in the most informative manner, this information will never be valuable enough to outweigh the costs of being found liable for treble damages. The monetary damages at risk are just too high. The disclosure function of patents will remain ineffective for as long as this situation persists. The ideal solution would be to enable software owners to clear all rights to their own properties.

In conclusion, the USPTO should examine how disclosure actually works in practice and compare their findings with how legal theory says it should work. Gaps will be found that damage the ability of software patents to promote innovation. Some of the gaps make software patents actually harmful to innovation. These gaps must be corrected. For best results the USPTO should consider requiring the source code of a well-written working program as part of the disclosure.