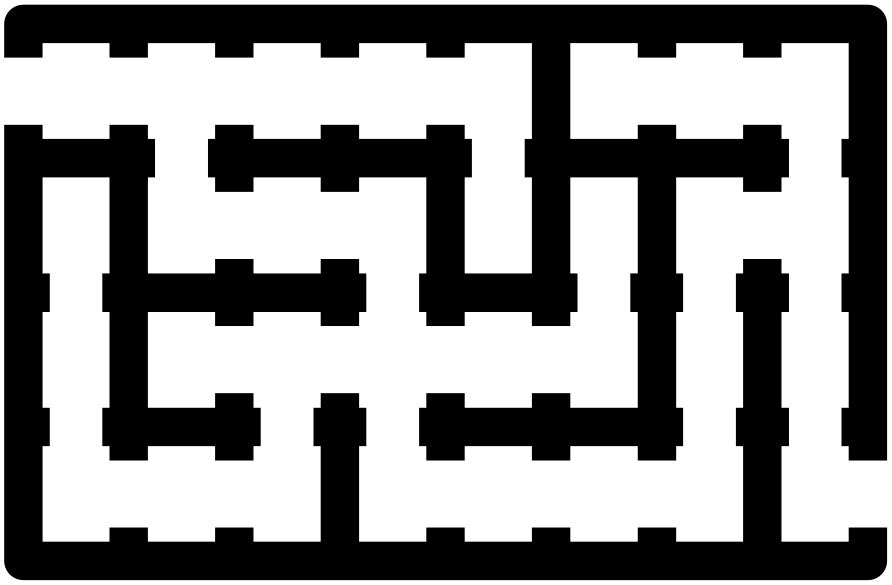
The Libre Labyrinth

Navigating the Maze of Free and NonFree Licenses



Version: 1 Jan 2008

Table of Contents

FLOS, A Federation of Colonies

The world of Free, Libre, Open Source (i.e. FLOS) is a somewhat fractious world. The fact that folks can't even agree on what to call what they support points to a fundamental problem. In the beginning, there was All Rights Reserved. And then there was ShareWare and freeware (which you might be able to share, but you probably couldn't charge money for, and might need to register to get the full privileges), the Free Software Foundation came out with the General Public License (which wasn't a Public domain license), and groups like MIT and BSD came up with their own licenses (which were more like Public Domain licenses).

Free Software had to differentiate itself from freeware by the motto “Free as in Speech, not free as in beer”. This tried to focus on the fact that Free Software was about liberty. (The Free Software Foundation had it's own Manifesto, to give you an idea of its alignment) And some folks had the idea of referring to Free Software as Libre Software just to clarify it was about liberty not about money. Some folks thought the whole Free/Libre thing went a little too far over the top, and was a little bit of a turn off to the corporate world, so they repackaged it with the label “Open Source”. Someone came up with an Open Source Definition (OSD). More recently, the rise of DRM and tivoization caused quite a stir as different groups argued whether having an anti-tivoization clause in a license still qualified it as Free, Libre, Open or not.

In the mean time, another organization called “Creative Commons” came along, which wasn't so concerned about Freedom (or whatever you called it) as it was interested in coming up with ways that artists could leverage their copyrights to accomplish different things. They came up with some licenses that are completely Not Free. And they came up with some licenses that are Free, some met the Open Source Definition, some did not satisfy the folks at Debian, and the Free Software Foundation at first disassociated itself from Creative Commons as being not enough Free as in Speech.

Meanwhile, things like an Anti-DRM clause to a license might spark off a dozen flamewar threads between various factions all arguing over whether or not an Anti-DRM is Free or Open or Non Free or whatever. And one of the recurring problems with such discussions is that it's entirely subjective. There really isn't a common objective language that is used to describe what various licenses are intended to do.

This, fundamentally, is a constitutional problem. Licenses tell you what is and is not allowed. They are like laws enacted by congress. A constitution is a set of rules that indicate what congress is committed to. And while the Free Software Foundation has its Manifesto, and the Open Source Foundation has the Open Source Definition, and Debian has it's guidelines, none of them really describe things in a common language, in an objective manner. What, exactly, does “free” mean anyway? It's like the US constitution that says the State must have “probable” cause. And the immediate question is “what exactly is probable and improbable?” The next question is how does “probable” in the US constitution translate to some other word in some other country's constitution.

Every Free, Open Source, Libre organization has it's own constitution, essentially. Some have manifestos, some have definitions, some have guidelines. But all of them require translations back and forth to understand what one is talking about compared to another.

This document is an attempt to describe the various licenses in a common and objective way. Rather than rely on subjective terms like “Free” or “Libre”, the various licenses will be described based on a common set of functionality, and list whether or not a license contains that function.

What are these measures? They start as a list of the basic rights granted an author for copyright: the rights to Copy, Distribute, and Create Derivative works. But then the list expands and refines various rights as needed. But in the end, it remains a list of various rights that one can use to describe what any particular Free, Libre, or Open Source license will allow.

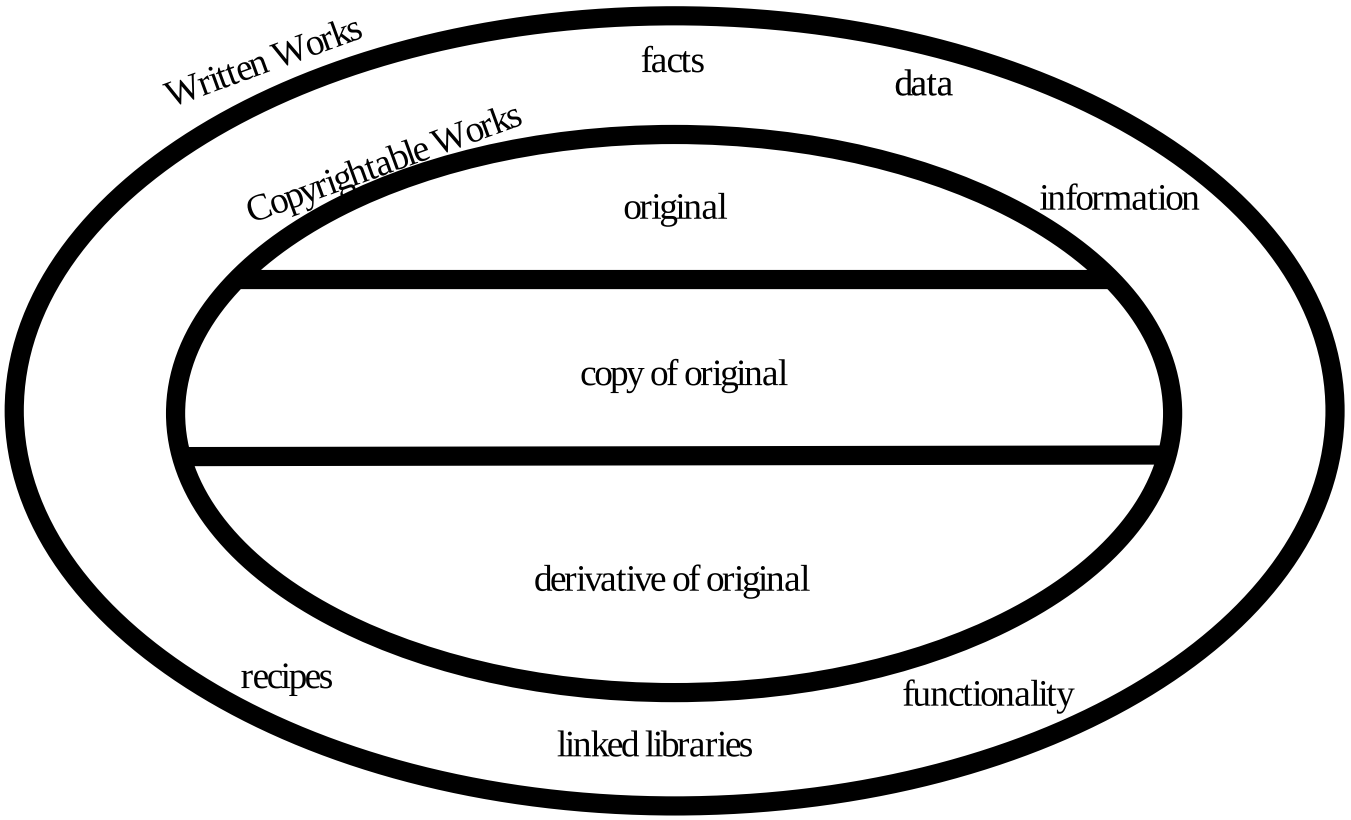
The goal is to create an objective way to not only describe what each license allows and does not allow, but also to do so in a way that allows side by side comparison of different licenses from different “constitutions” to be compared without entailing a whole bunch of legalese translations or wading through a bunch of politically motivated speech. All the terms such as “free” and “open” and “libre” were all chosen for politically motivated reasons. Some wanted to emphasize freedom, some wanted to play down freedom and emphasize quality code. So, one of the first things we can rule out using as a descriptor are the terms Free, Libre, or Open Source.

Once we've described some Free, Libre, Open Source licenses using this common, objective, list of measures, then it is hoped that we might possibly find some common themes among the different licenses, and maybe even come up with a single “constitution” that they can all agree on. That might be asking a bit much. But if nothing else, having a common, objective way to describe a bunch of different licenses would be useful in and of itself.

Venn Diagram of Written Works and Copyright Works

Licenses function first and foremost through copyright law. Copyright law deals with written Works of various sorts. Not all Works are copyrightable, though, so the first thing that needs to be established is that the Writings covered by Copyright is a subset of all possible writings.

Copyright law deals with three basic rights, the right to Copy, Distribute, and Create Derivative Writings. This creates three subsets within Copyright, which are also shown in the Venn Diagram below.



The basic requirement of what qualifies for copyright coverage is that the work be the result of some human self expression. A science fiction novel would be inside the coverage of copyright law. A list of countries and their capitals is a list of raw data and does not contain any human self expression. Software source code is made up of text files created by people. Source code generally contains some human expression of how the program should function and as such is covered by copyright law. Taking a library of software functions and linking them into another program is a purely functional operation and is outside the realm of copyright law.

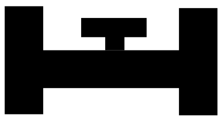
This is not the complete Venn Diagram to describe licenses. The final diagram will expand on this basic diagram a number of times. This diagram merely makes the first distinction between Written Works covered by copyright law and Written Works not covered by copyright law.

Venn Diagram with Doors

By the end of this document, the goal is to produce a single, but complete, Venn Diagram of all the sets and subsets that describe all the different aspects that affect the different licenses. One Venn Diagram will approximately describe all these possible distinctions.

A license can't change these subsets, it can only change which subsets are allowed to be accessible to different people. This access, these allowed transitions, is represented by taking the Venn Diagram and adding "doors" to indicate how transitions work. These doors will show how works in one area can or cannot be moved into different areas.

A door looks like this:



The two thick vertical lines represent the door frame. The thick horizontal line represents the door itself that can swing open. The smaller "T" shape on the door represents the location of the door knob. You can only open these doors from the side that has a door knob on it. If you're on the side without a door knob, you don't have access to the other side.

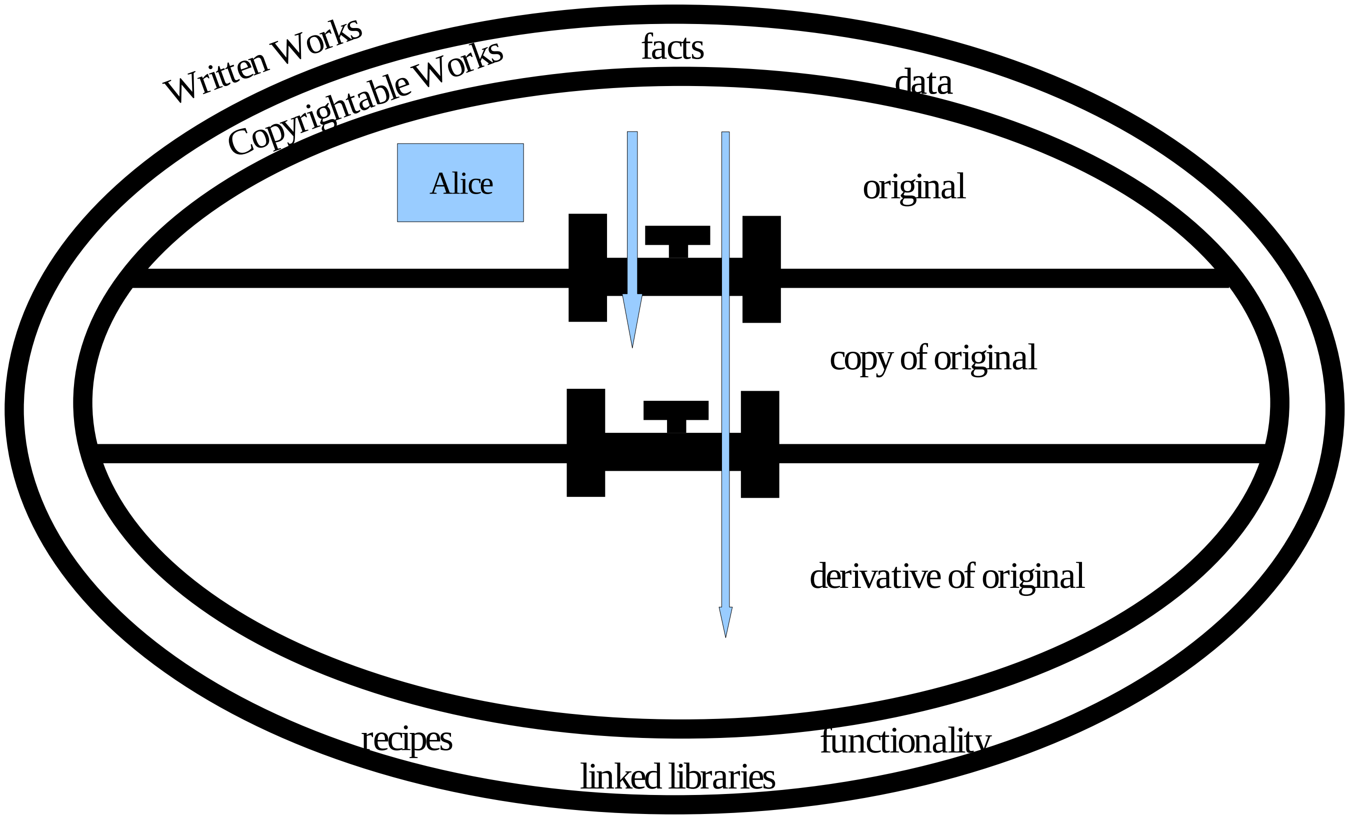
Labyrinths Instead of Venn Diagrams

Venn diagrams show sets and subsets and that's about it. What we're trying to describe is not only sets and subsets and overlapping sets, but also allowed transitions from one set to another, doors and passageways, and where different people might be located on these diagrams. For this reason, I don't refer to these diagrams as Venn Diagrams, but as Labyrinths. Yes, it's a bit of a stretch, especially for the first diagrams, but by the time we're finished, you'll find that you are in a maze of twisty passages, all alike.

Labyrinth of All Rights Reserved

Copyright automatically grants the author the exclusive right to Copy, Distribute, and Create Derivative works. These rights are granted as soon as the author puts their creative expression in some fixed medium. If the author (Alice) chooses to keep these rights exclusive to themselves, they might distribute copies of the work with the notice “All Rights Reserved”.

Below we show what a work licensed All Rights Reserved might look like:



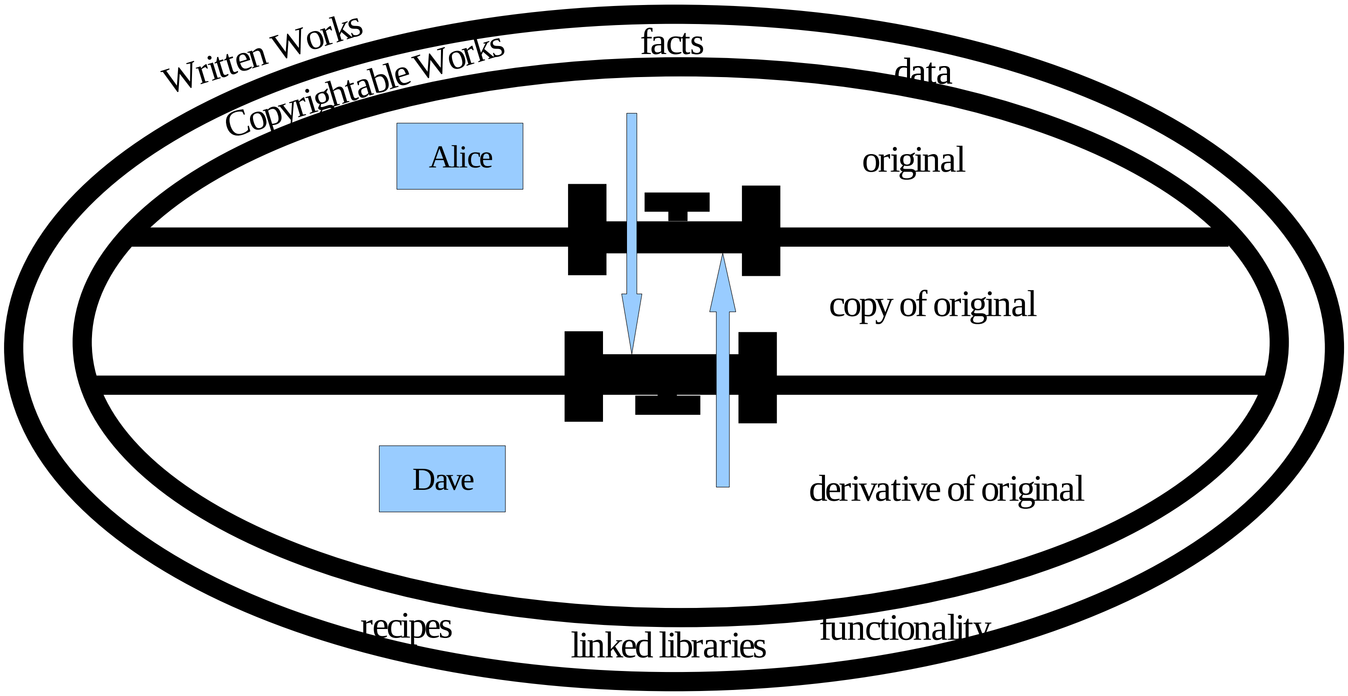
The position and direction of the doors shows that only the person who has rights to the original work can access the right to make copies or to create derivative works. Only Alice can give someone the right to make copies, distribute them, and to create derivative works. And until she licenses those rights or transfers those rights, Alice is the only person allowed inside the Copyright circle for her work.

It's a fairly simple labyrinth right now, but we'll be adding many more passageways and doors before we're finished.

Labyrinth of a Public Domain Style License

A person might create a work and then choose to give up their exclusive rights to their work and give those rights to the public. Or, when the copyright to a work expires, the exclusive rights to the work expire and become public domain and anyone may exercise the rights.

This diagram shows the labyrinth for a work licensed under a Public Domain license:



The position and direction of the doors is intended to indicate that if Alice has a work and licenses it under a Public Domain license, Dave can access the rights to create a derivative of Alice's original and Alice may not access the right to Dave's new work. The door is one-way.

Dave can license his derivative under an All Rights Reserved license if he chooses, and his derivative becomes it's own “original” work in a new labyrinth. But to Alice, it is a derivative of her work, but because she licensed her work Public Domain, that allowed Dave to license his derivative All Rights Reserved, and Alice is unable to access Dave's derivative of her work.

It is possible that Dave will choose to license his derivative under a Public Domain license as well. This is essentially Dave choosing to open the door between himself and Alice and giving her access. But he is not required to do this by the Public Domain style licenses. The doors remain in place, its up to Dave to choose whether he opens them or not.

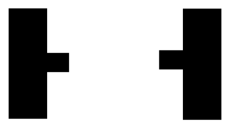
This is true whether the work was licensed under a Public Domain Style License or whether the work entered the Public Domain because the Copyright expired.

Copyleft Doors

An example of our one-sided, one-way, doors are the doors on the old-fashioned refrigerators. Older refrigerators have doors with handles on the outside that you have to pull or turn to unlock the door and open it. Because the handle is only on one side, you can't open the refrigerator from the inside. This leads to a dangerous situation when a child playing around these old refrigerators gets trapped inside. These tragedies led to various requirements to either chain the door so it can't be closed completely, or to remove the door completely so it can't easily become a trap.

A copyleft license is any license where any derivative must be licensed exactly the same as the original. What this does is take the doors off the hinges and remove any potential one-way traps.

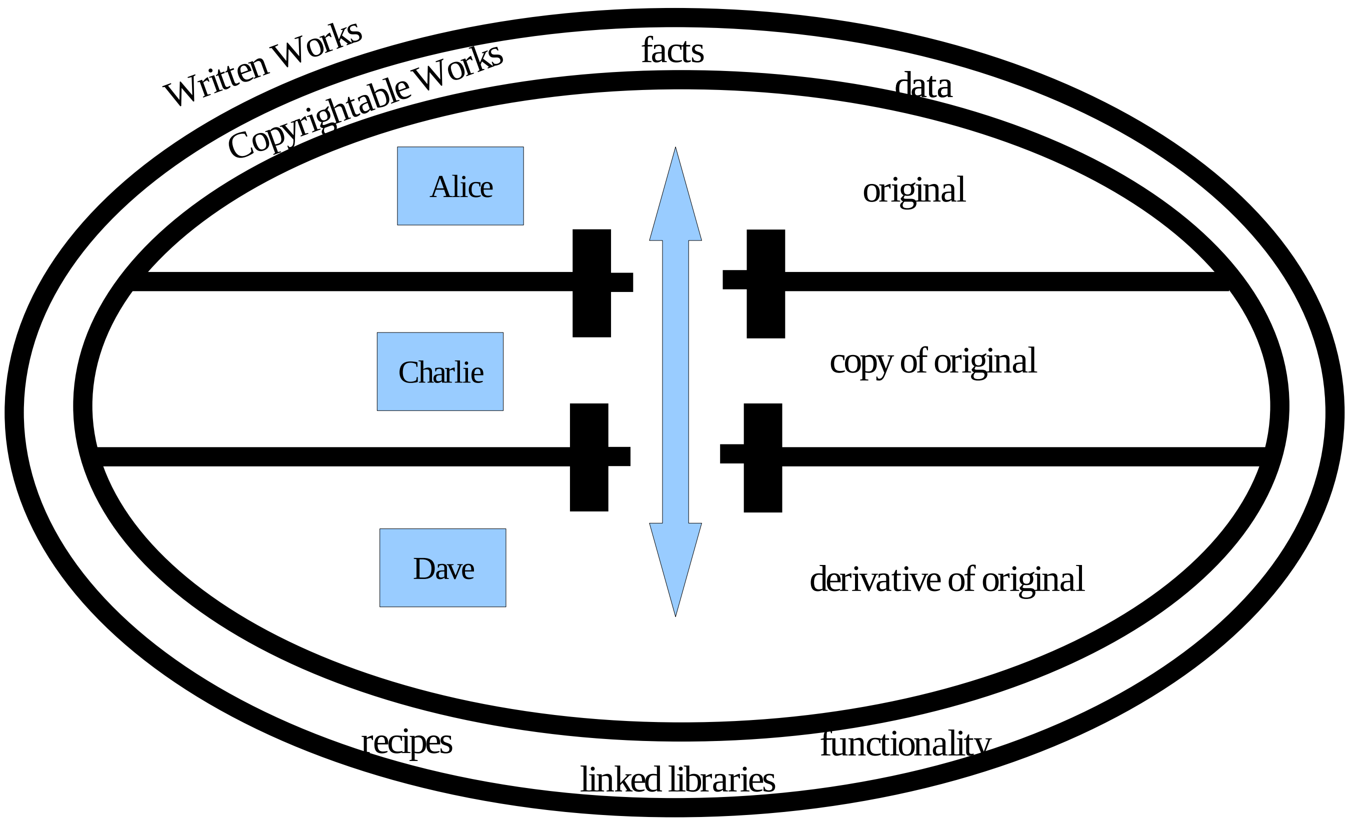
Such a   door is represented graphically as follows:



Note that it is possible to have different copyleft licenses by placing these two-way doors at different places in the Venn Diagram, creating different Labyrinths.

Labyrinth of Generic Copyleft License

The following diagram represents a generic copyleft labyrinth:



Note that in the above diagram, Alice creates the original work and licenses it under some rudimentary Copyleft license. This license grants Charlie and Dave the right to access the original work and either copy or create derivative works. However Alice's license requires that any copy or derivative have the exact same license that the original had, guaranteeing that Alice can access the copies created by Charlie and the Derivatives created by Dave.

There are no trap doors behind which Charlie or Dave can take Alice's original work and create something that Alice cannot access herself. Compare this to the one-way trap doors that exist in a Public Domain style license.

Unfortunately, there are a number of subsets that we haven't shown yet, so no copyleft license is actually this simple, because copyright isn't this simple. We need to distinguish the other copyright subsets and then show how each license allows or does not allow transitions from one set to another.

More Subsets

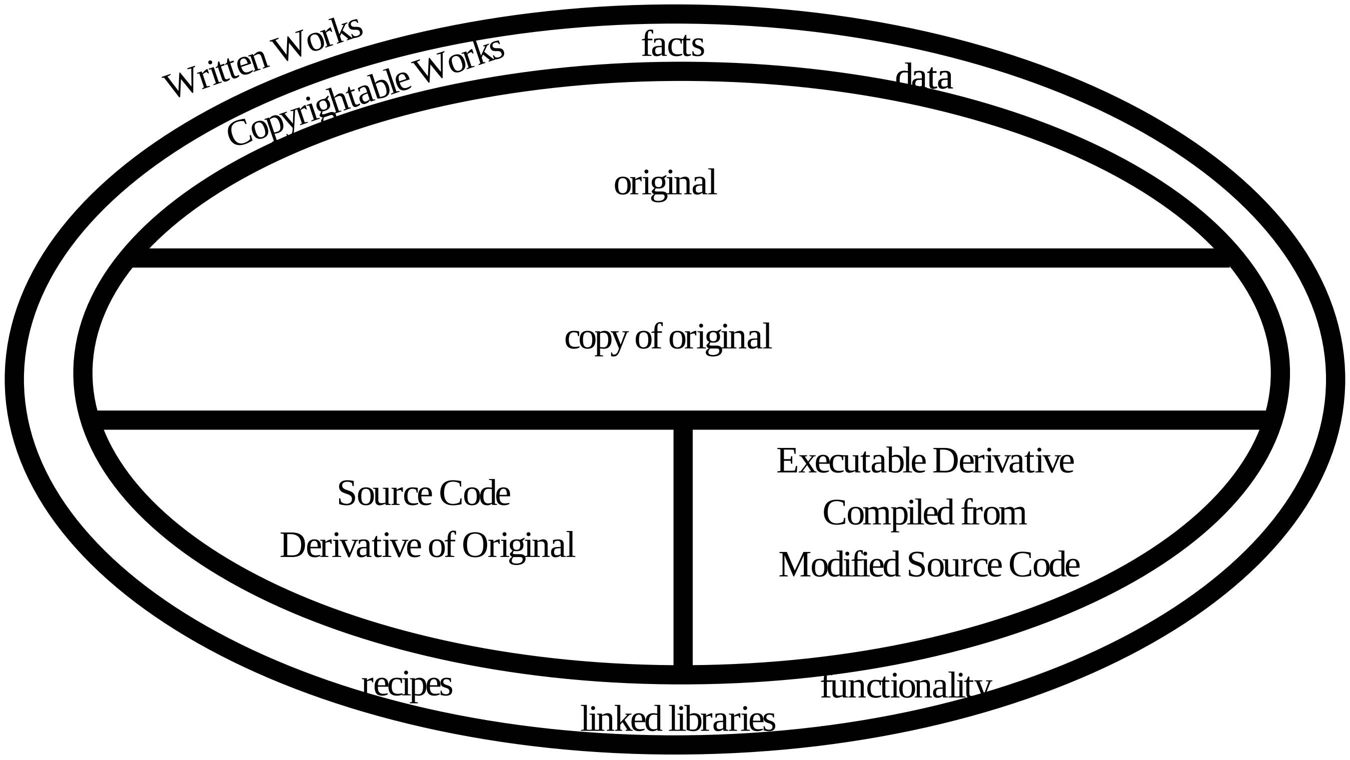
The diagram above breaks the set into the three legal rights of copyright: The original work, copies of the original work, and derivatives of the original work. However, more distinctions are needed to completely describe all the different possible subsets of a work.

Executables as Derivative

An executable is a piece of software that has been compiled from source code. The source code is some text based file that humans created to describe what the software should do. A compiler is then used to transform this source code into an executable that the computer can run. Compilation is naturally a one-way process, which is to say, you can easily convert human readable source code into an executable, but you can't easily convert an executable into human readable source code.

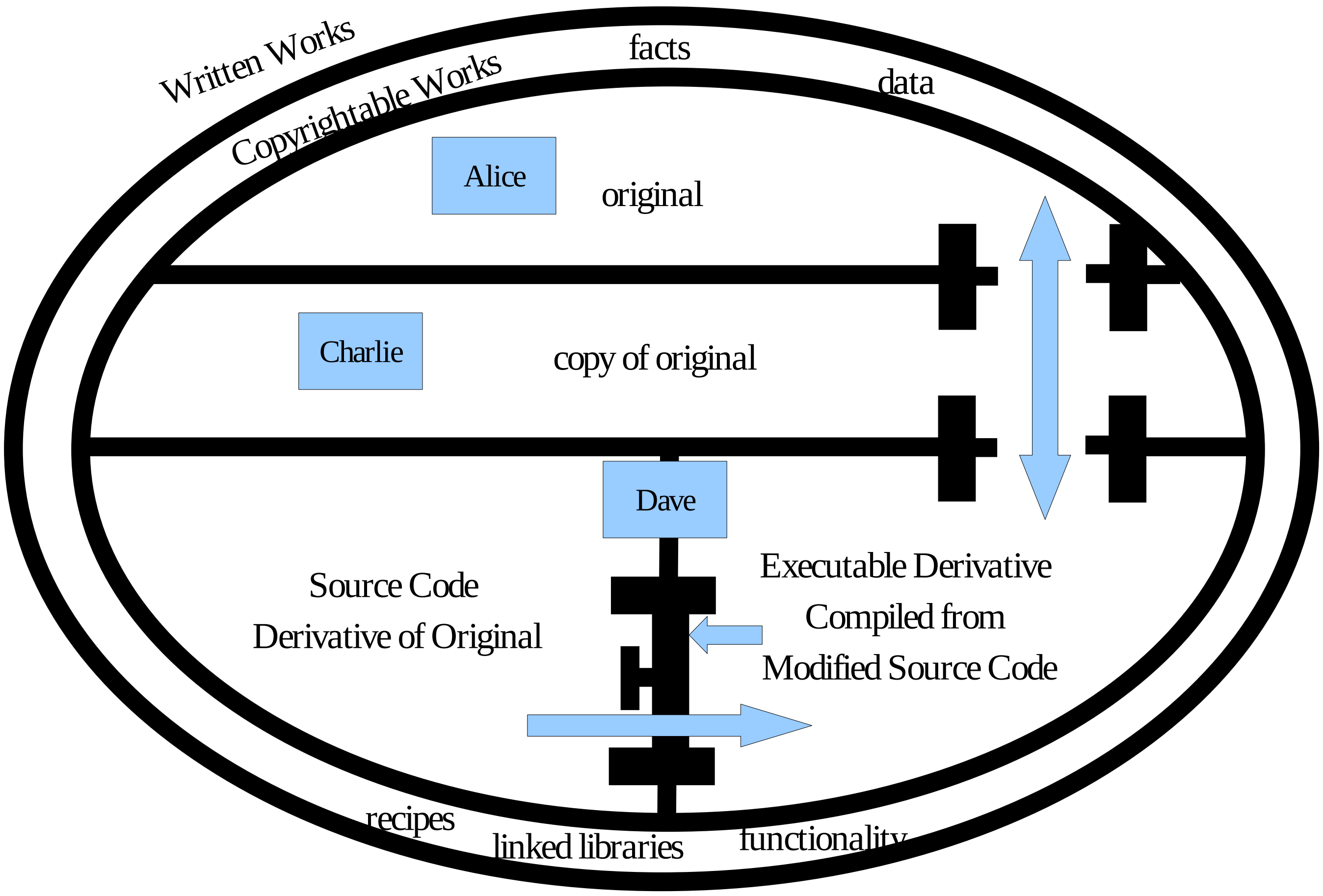
Because compilation is a one-way process, it is a wing in the “Derivative” room in the labyrinth. Some licenses have a source code requirement, which means that if someone creates an executable, they are required to keep source code with the executable under certain circumstances. Some licenses do not have a source code requirement, which means that someone could take content, modify it, compile it, distribute the executable, while keeping the modifications private. This becomes a one-way trapdoor.

Here is the basic Venn Diagram showing how the subset of “derivatives” is broken down into further subsets.



Labyrinth Showing Generic Copyleft with No Source Code Requirement

The diagram below shows what happens when Alice uses a Basic Copyleft License with no Source Code requirement:



Dave is able to access Alice's source code through the Generic Copyleft License. Dave then modifies the source code, creates an executable, and distributes the executable. Our generic copyleft license requires that all derivatives that are distributed be available under the same license, which means the executable is Copyleft. But the source code wasn't distributed, so it remains private to only Dave. Alice and Charlie will be denied access to Dave's source code.

Examples of different ways this might happen in real life:

The Creative Commons-Share Alike license is a Copyleft license, but it has absolutely no Source Code requirement of any kind. People like Dave can take the original works, create derivatives and are in no way obligated to make their own source code available to anyone else.

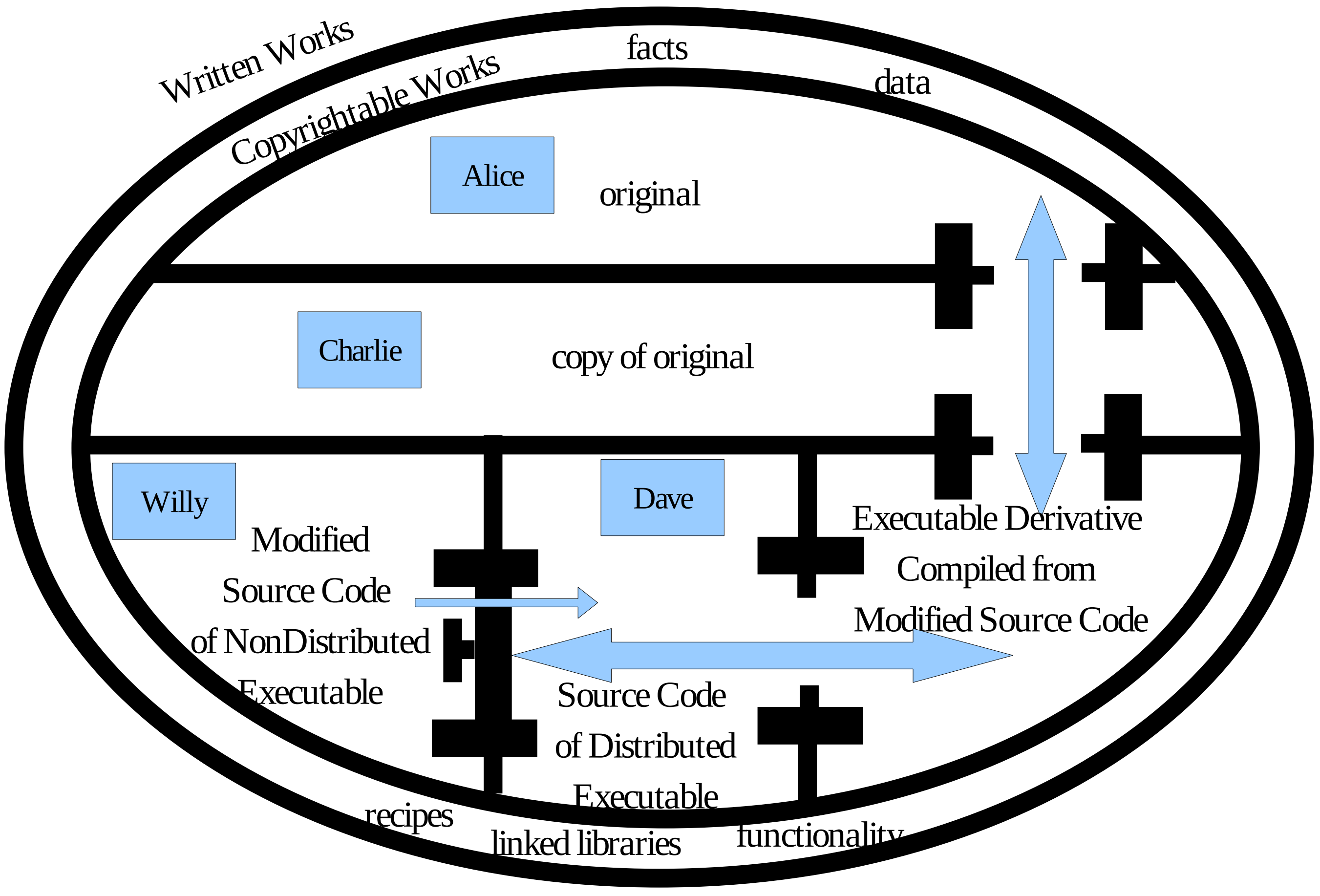
The GNU-GPL has a source code requirement that is attached to distributed binaries. If Dave creates a derivative and distributes the binary, he must also distribute the source code used to create that binary, or make the source code available to the public in some way, and that source code must be under the same license as everyone else.

However, since the GNU-GPL requires distribution to activate the source code requirement, it is possible for Dave to create a derivative, create an executable, present the executable as a web interface but not distribute the executable itself, and the source code requirement will not be triggered.

The Apple Open License has a source code requirement that attaches to all derivatives, even private derivatives that are not distributed. This is intended to prevent people from keeping source code private by using ways that avoid distributing the binary executable.

Labyrinth of Generic Copyleft License with Source Code Requirement Attached to Distribution

The diagram below shows what happens when we modify the generic copyleft license and add a source code requirement that is activated by distribution of the Executable.


Written Works
facts
data
Copyrightable Works
Alice
original
Charlie
copy of original
Willy
Modified Source Code of NonDistributed Executable
Dave
Executable Derivative Compiled from Modified Source Code
Source Code of Distributed Executable
recipes
linked libraries
functionality

In this example, Dave creates a derivative of the source and compiles it. He then distributes the executable. The source code requirement triggers, and forces Dave to also distribute his modified source code under the same copyleft license.

On the other hand, Webguy Willy, who created a derivative of the source code and compiled it into an executable, did not Distribute the executable, thereby avoiding triggering the source code requirement of the license. Willy can access Alice's and Charlie's and Dave's source code, but only Willy can access his source code.

Willy may be able to use the executable for something like a website. The executable is not distributed, it remains on the hosting computers where it is executed. Outside people may go to Willy's website, but they won't get a copy of the executable, they will get the output generated by the execution of the executable. This will not trigger the distribution clause of a source code requirement.

Also, different possibilities are available with the definition of "distributed". A license could declare that an entity such as a corporation may maintain private derivatives within the company. Derivatives might be distributed from members of that corporation, but the license might not require this to trigger the source code requirement. Employee's might be required to sign Non Disclosure Agreements, which would then require that even though they received a copy of this derivative work, they are not allowed to take that derivative or its source code outside the company. The GNU-GPL currently allows for organizations to maintain private copies through Non Disclosure Agreements within the organization. However, the GNU-GPL prohibits Non Disclosure Agreements when the work is distributed outside an organization.

Just a little bit confusing.

The point is that when you see the word "distributed" in a Venn Diagram here, it is referring to a possible source code requirement that is triggered by "distribution", and at that point, you need to keep in mind what exactly the license defines as "distributed" in order to determine if the source code requirement is triggered.

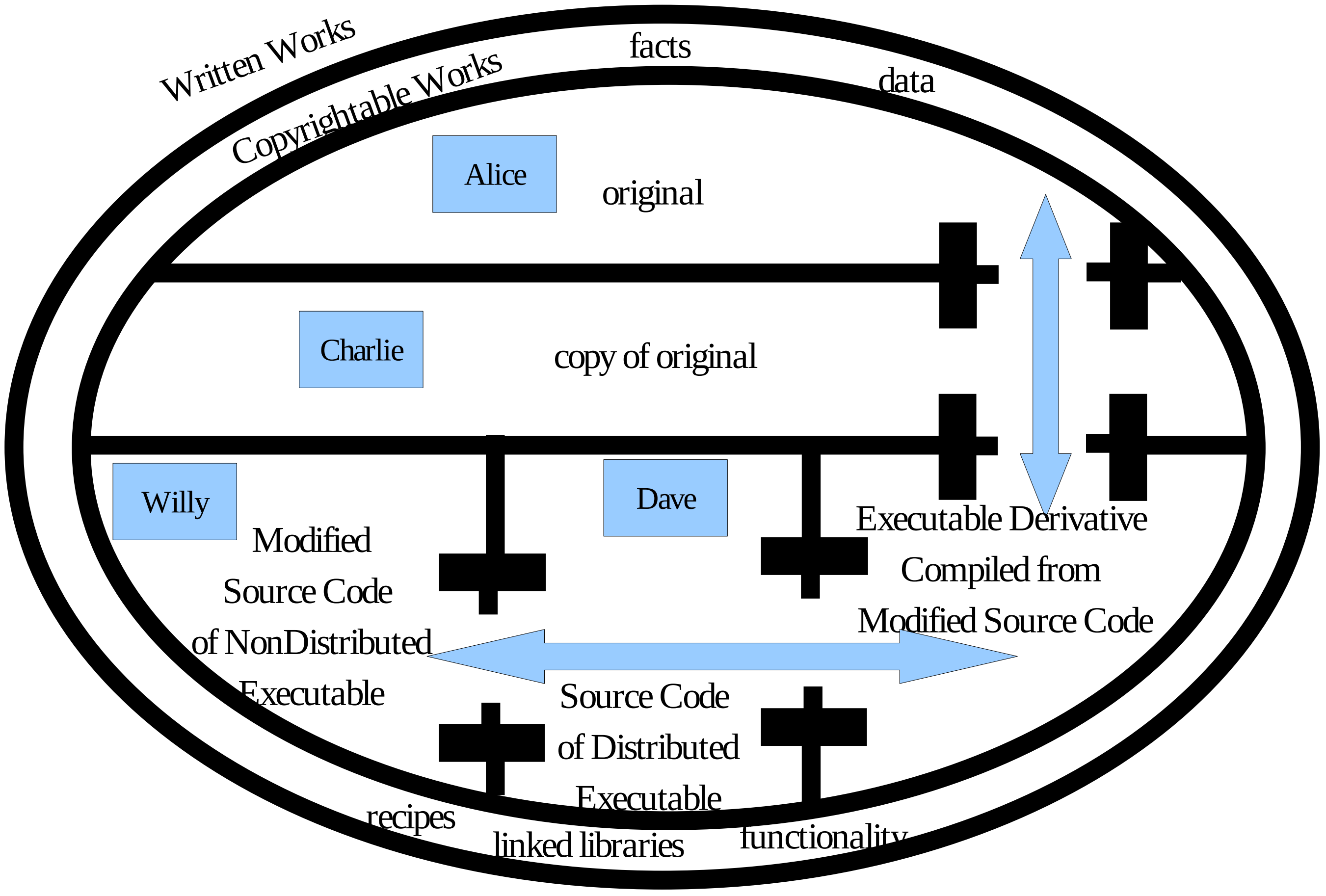
Furthermore, with a source code requirement triggered by distribution, the license may only require that the person or organization receiving the binary should also receive the source code. Alice created the original Copyleft code. Willy creates a derivative and distributes it to his trusted friend Trent. Trent can demand that Willy give him the source code. But Alice cannot legally demand the modified source code from Willy or Trent. Willy and Trent can decide to maintain the modifications to themselves. Willy cannot force Trent to keep the modifications private through a Non Disclosure Agreement (unless Trent works for Willy or similar). So if Trent wants to give the modifications to Alice, he can do so. But Alice cannot force either one of them if she didn't receive the modified executable.

A license may allow Dave to distribute the source code on his website, rather than require Dave include a copy of the source code with all distributions. This is sometimes done because the cost of distributing source code can be high, and a lot of people aren't interested in source code, and wont' make a fuss if they don't get it. Although Alice could get a copy this way, Dave could conceivably provide the source through a login only to people who received the executable, and then leave it up to them to decide if they give it to Alice. Generally speaking, if you distribute copylefted source to any number of individuals, it should be considered beyond your control to restrict. But depending on the exact wording of a specific license, the source code may or may not be publicly available to everyone.

A final possibility would be that a copyleft license require that all modifications be sent back to the original author. This is generally frowned upon. The issues are more logistical than anything else. Trying to designate who is the “official” maintainer, especially over a long period of time, may make it difficult to meet the requirements of the license. And may make some lawsuit-nervous folks even more nervous.

Labyrinth of Generic Copyleft License with Universal Source Code Requirement

If a license has a universal source code requirement attached to all derivatives, whether they are distributed or not, then the room that was closed off and occupied only by Willy becomes accessible by everyone. The diagram below shows a generic Copyleft license with a universal Source Code requirement.



Written Works
Copyrightable Works
facts
data
Alice
original
Charlie
copy of original
Willy
Dave
Modified Source Code of NonDistributed Executable
Executable Derivative Compiled from Modified Source Code
Source Code of Distributed Executable
recipes
linked libraries
functionality

Even though Willy doesn't distribute his executable, he is still required by the license to make his derivatives available to the rest of the community.

A copyleft license with a universal source code requirement doesn't have the issue of who is considered an “organization” and at what point can Non Disclosure Agreements be used, and at what point can they not be used. Any derivative must be made public.

There are still different possible implementations for this. The license might require source code be given to anyone who receives a copy of an executable. However, a special case must be made for derivatives which are not distributed. Returning modifications to the original author has the same issues as with the “triggered” source code license. Therefore, the generally accepted way to deal with derivatives that are not distributed is to make the source code publicly available on a website for some period of time.

Derivative Limited by some Functional Boundary

Another copyleft option is for the source code requirement to be active only within the functional boundaries of the original source code, rather than considering any derivative, including compiled or linked works, to be derivatives.

Copyright law gets a little bit fuzzier when we start talking about functional boundaries. Some functional boundaries do not fall inside the copyright boundary. Linking into library functions is not considered a derivation under copyright law, so a license may not be able to restrict the linking into a library of software functions even if it wanted to. Copyright law also allows for the functionality of software to be replicated without it qualifying as a Derivative. This can even occur down to the subroutine level or procedural call level. Different operating systems can provide the same functional calls without it falling into a matter of Copyright.

The point is that any copyleft license with a source code requirement may trigger the source code requirement on (1) any distributed derivative, (2) any derivative whether distributed or not. And the copyleft license may also define derivative to mean (A) the maximum definition of derivative recognized by copyright or (B) a subset of derivatives limited by some functional boundary.

This creates a number of permutations:

(1A) distributed derivative, maximum derivative: GNU-GPL

(1B) nondistributed derivative, maximum derivative: Apple licence

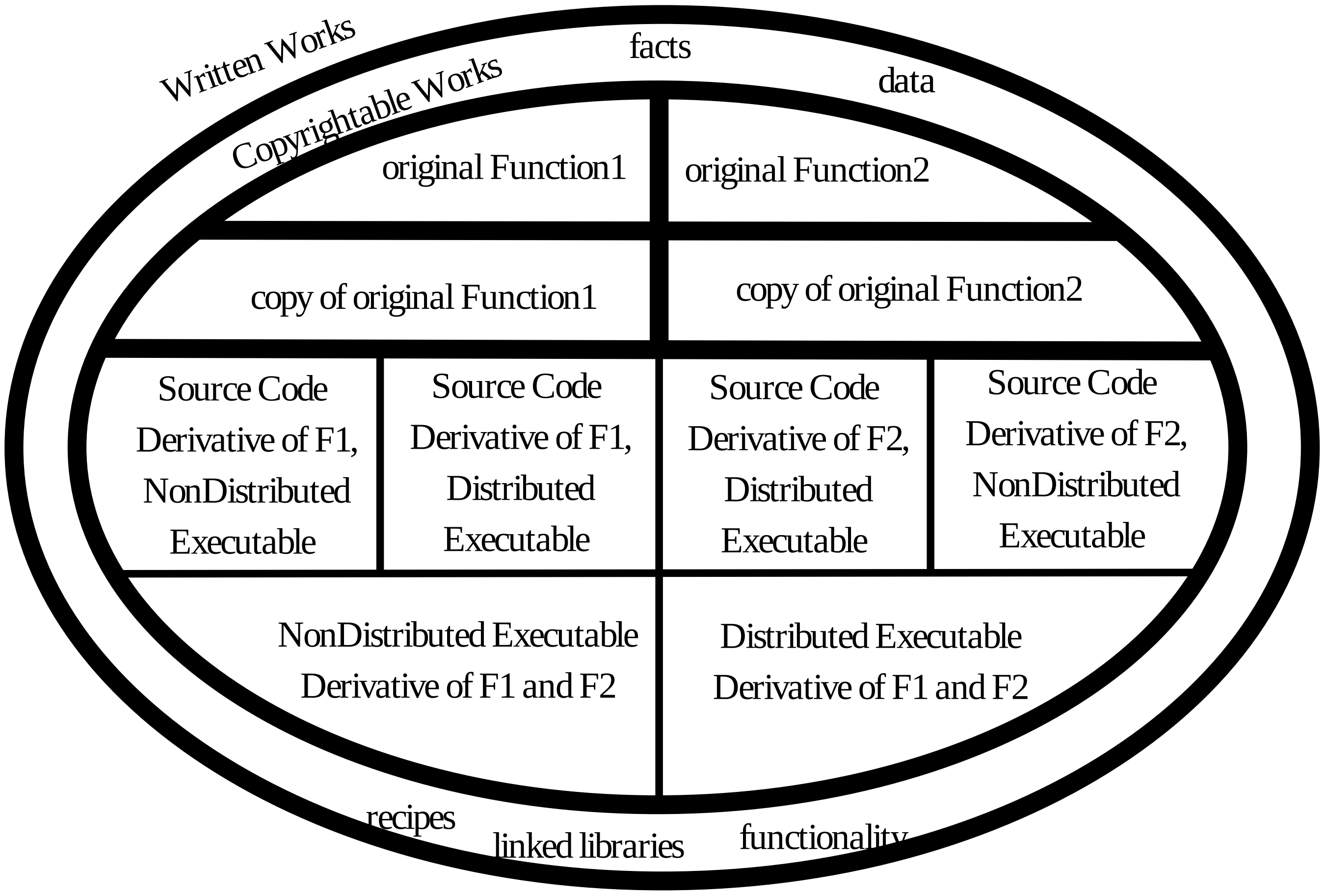
(2A) distributed derivative, functional derivative: GNU-LibraryGPL

(2B) nondistributed derivative, functional derivative: ???

Note that “maximum derivative” is simply whatever the law and the courts decide will qualify as a derivative. Linking into object files sometimes qualifies as a derivative, and sometimes does not. A “functional derivative” simply means that regardless of what the laws and the courts say is a derivative, the license is only concerned with derivatives of the work inside the functional boundary. Any work that links into this work through a functional boundary would not be considered a derivative by that license.

There is currently no license I know of for case (2B), but it would seem to be the condition best suited for an Open Hardware license. This will be discussed much later.

Mapping these permutations into a Venn Diagram requires more subsets of “derivatives”. We need to differentiate the different sets of functionality. Function1 and Function2 show two different functions, which should be enough for showing the different ways that copyleft and non-copyleft works can be combined. And we need to differentiate between distributed and non-distributed triggers. Here is the Venn diagram showing functional boundaries in derivatives:

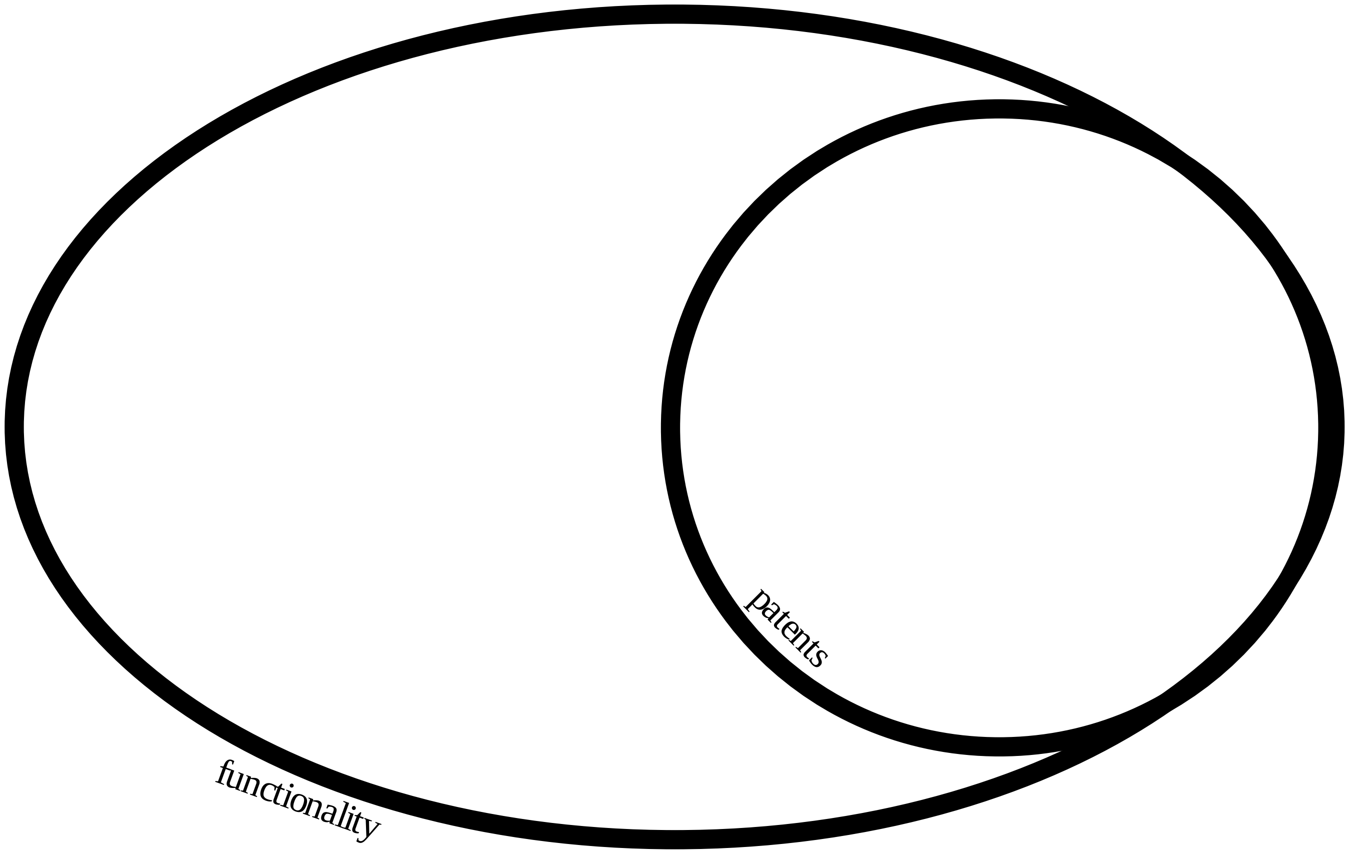

Written Works
facts
data
Copyrightable Works
original Function1
original Function2
copy of original Function1
copy of original Function2
Source Code Derivative of F1, NonDistributed Executable
Source Code Derivative of F1, Distributed Executable
Source Code Derivative of F2, Distributed Executable
Source Code Derivative of F2, NonDistributed Executable
NonDistributed Executable Derivative of F1 and F2
Distributed Executable Derivative of F1 and F2
recipes
linked libraries
functionality

Note that we removed the object files from the diagram. An object file can be considered an odd subset of the source code derivative blocks, if need be. Object files generally aren't as important as source code or executables. Source code can be modified by humans. Executables can be executed by computers. Object files are really an intermediate stage that can't do either, without further modification.

Also, keep in mind that while the above diagram uses the term “executable”, that subset can be any sort of work generated by a machine from user created works, and the final work makes it difficult to get back to the raw, original works. This could be a source-code to executable translation. But it could also be a file defining and describing a three dimensional object which is then translated to a two-dimensional image. It might be multimedia files, music files, any raw files that are combined into some final work.

The “Functional” Set and It's “Patent” Subset

There is a whole 'nother set of rooms for our Venn Diagrams or Labyrinths. It is a set called “functionality”, with a subset called “patents”. All patents must be functional.



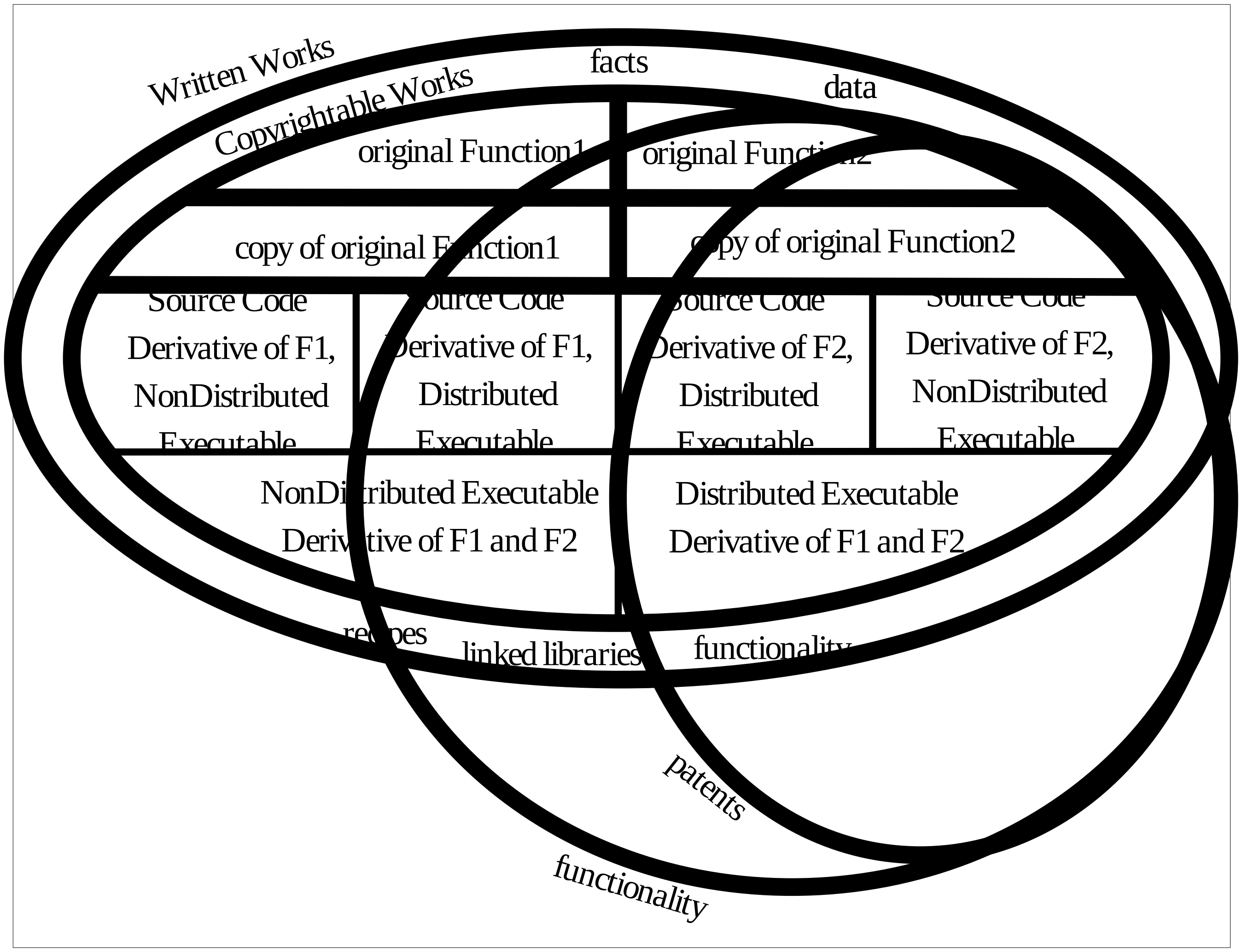
Functionality would include something as mundane as a real life, physical mouse trap. If you were to invent a new mouse trap that was significantly different than any previously existing mouse trap, you might be able to patent it.

Functionality also includes stranger things such as the execution of software on a computer. And it is currently possible to patent software, therefore, it is possible to execute some generic software that is simply “functionality”, or it is possible to execute some specific software that might be patented.

But software is also covered by copyright. Which means that there is some subset of functionality and patents that is also a subset of the copyright set.

Venn Diagram of Written Works, Functionality, Copyrights, and Patents

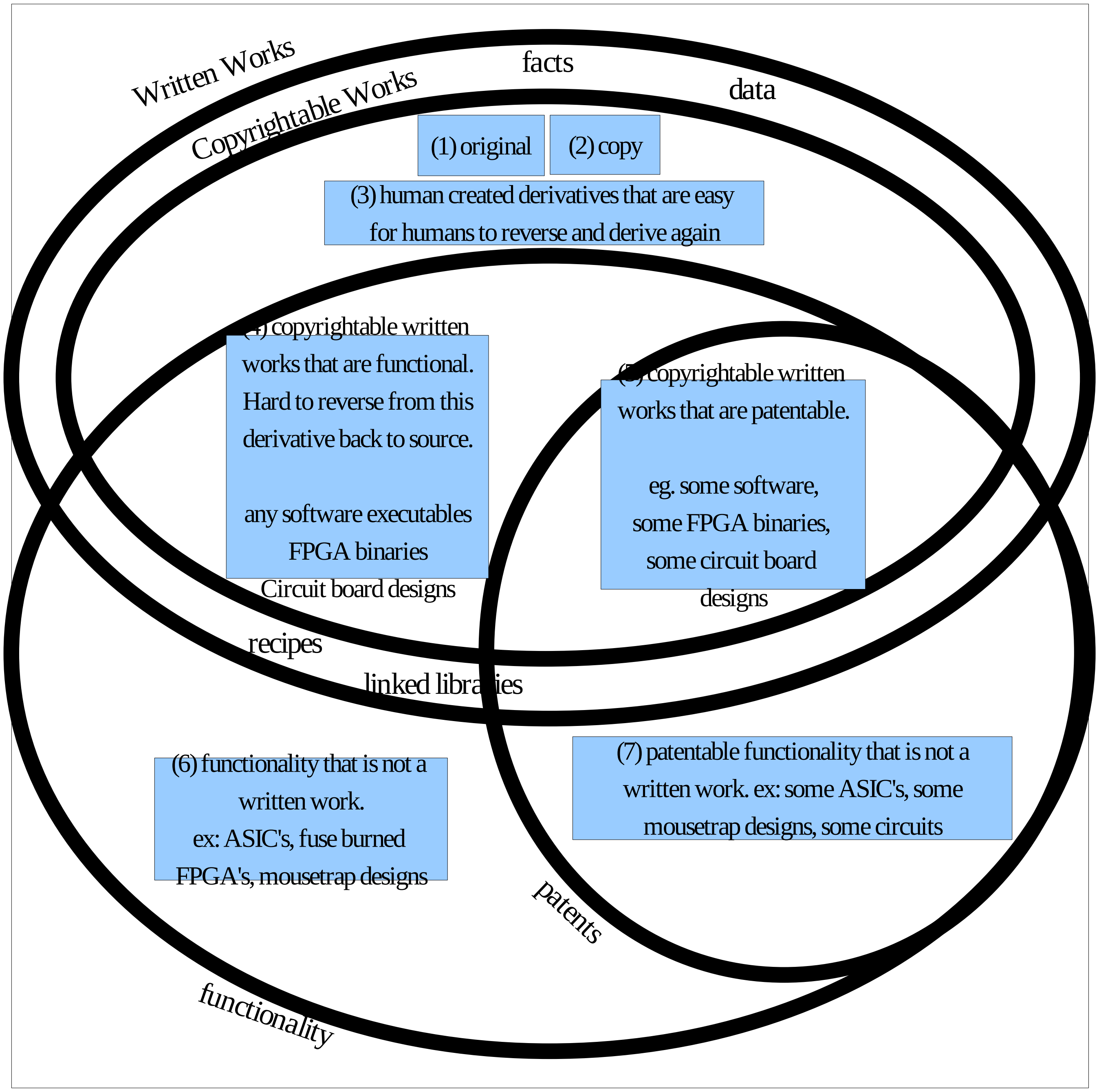
The diagram below shows the relationships between written works, functionality, copyrights, and patents.


Written Works
Copyrightable Works
facts
data
original Function1
original Function2
copy of original Function1
copy of original Function2
Source Code Derivative of F1, NonDistributed Executable
Source Code Derivative of F1, Distributed Executable
Source Code Derivative of F2, Distributed Executable
Source Code Derivative of F2, NonDistributed Executable
NonDistributed Executable Derivative of F1 and F2
Distributed Executable Derivative of F1 and F2
recipes
linked libraries
functionality
patents
functionality

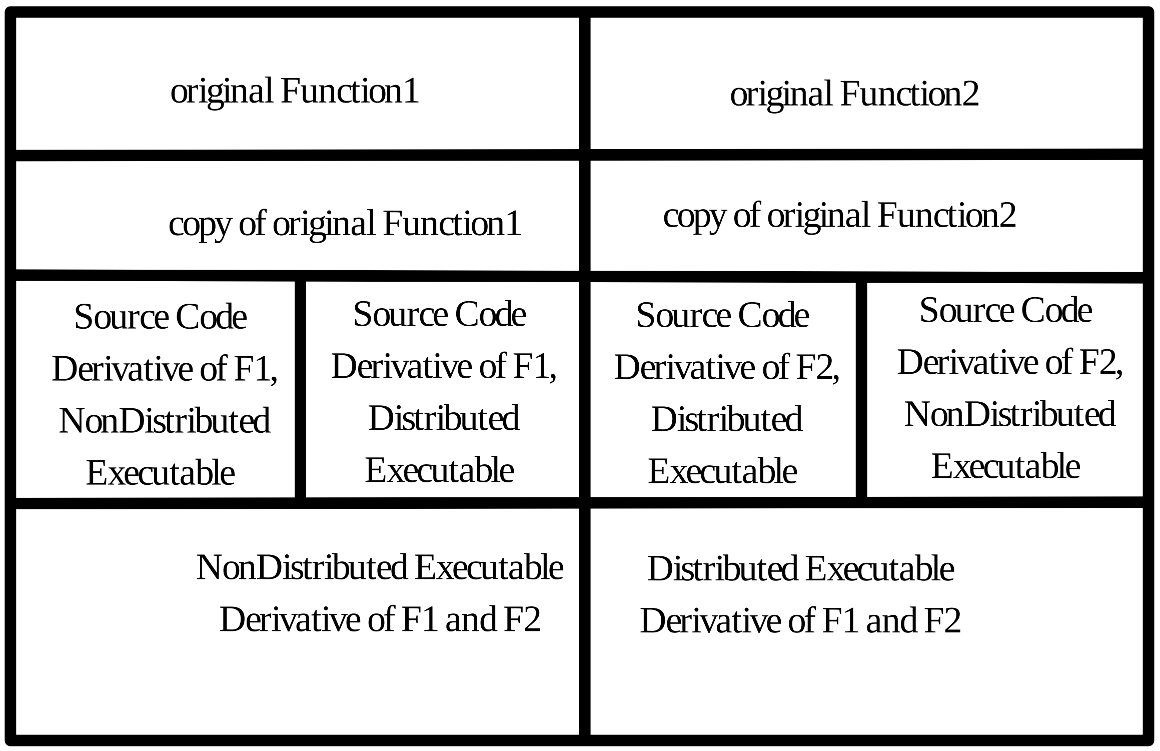
The Venn Diagram is starting to get a little complicated. First, some clarifications. The different functions (f1 and f2) are more for software and aren't applicable to every copyrightable work. Some copyrightable works aren't functional and aren't patentable. Art, such as paintings or statues or similar pieces are strictly aesthetic in nature, rather than functional. Not all written works that are functional are also patentable. Software is mainly functional, but not all software qualifies for a patent. If a piece of software does get patented, that patent applies a monopoly to any source code that implements the patented functionality. A patented file compression algorithm could be expressed a number of different ways and perform the same functionality. The patent on that algorithm would restrict any version of source code that implemented that algorithm, regardless of expression.

Venn Diagram of Written Works, Functionality, Copyright, Patents, and f1/f2 overlay

The different functions in the previous diagrams, Function1 and Function2, are independent of whether either function is patentable. Both functions might be patentable. Neither function might be patentable. Or one of the two functions may be patentable. To clarify this, the connection between different functional aspects of a work will be removed from inside the Venn Diagram and placed outside the diagram as an “overlay” that can be inserted anywhere in any of the subsets of “functionality”. Here is the Venn diagram:


Written Works
Copyrightable Works
facts
data
(1) original
(2) copy
(3) human created derivatives that are easy for humans to reverse and derive again
(4) copyrightable written works that are functional. Hard to reverse from this derivative back to source.
any software executables
FPGA binaries
Circuit board designs
(5) copyrightable written works that are patentable.
eg. some software, some FPGA binaries, some circuit board designs
recipes
linked libraries
(6) functionality that is not a written work. ex: ASIC's, fuse burned FPGA's, mousetrap designs
(7) patentable functionality that is not a written work. ex: some ASIC's, some mousetrap designs, some circuits
patents
functionality

Here is the function1 and function2 overlay:


original Function1
original Function2
copy of original Function1
copy of original Function2
Source Code Derivative of F1, NonDistributed Executable
Source Code Derivative of F1, Distributed Executable
Source Code Derivative of F2, Distributed Executable
Source Code Derivative of F2, NonDistributed Executable
NonDistributed Executable Derivative of F1 and F2
Distributed Executable Derivative of F1 and F2

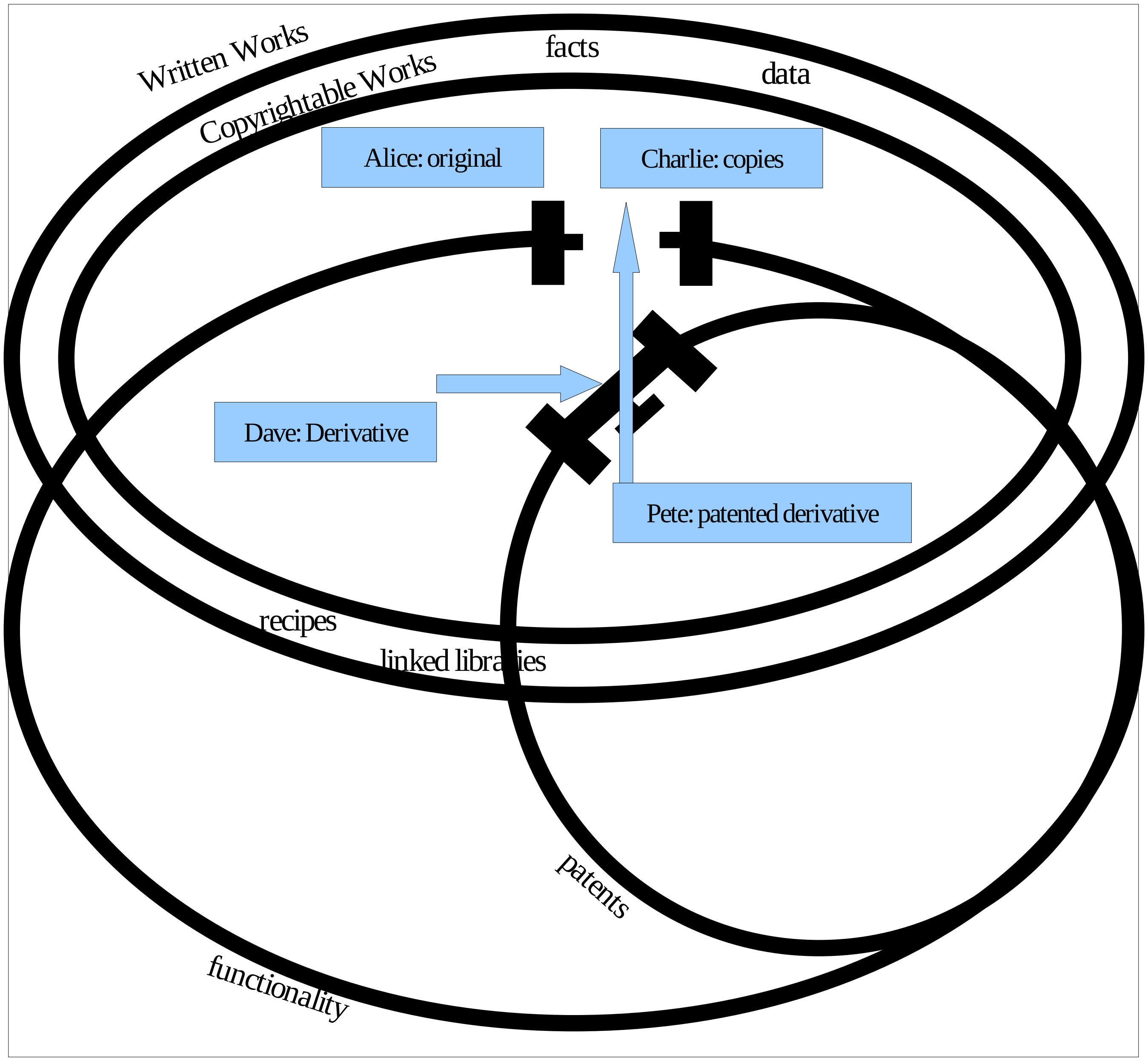
This overlay may be inserted into the Venn Diagram in subsets labeled 4, 5, 6, or 7. Sections 4 and 5 would probably refer to software functionality that is copyrightable. Sections 6 and 7 might refer to a hardware design in a written work and then that written work is converted into some purely functional implementation, which would not be a copyright 'derivative'. Hardware related issues will be described in more detail later, but just keep in mind that this overlay can be inserted in 4 different places in the Venn diagram shown on the previous page.

Patents Create One-Way Doors

The problem with patents is that our generic copyleft license only dealt with copyright law. It said all copies and derivatives of the original work must be licensed under the same license as the original work.
That's all well and good, except if the work also happens to contain a patent as part of the work. This comes up most notably in software projects. And if you were to place some content under a generic copyleft license, and that work also contained a patent, then the patent would act as a one-way trap door.

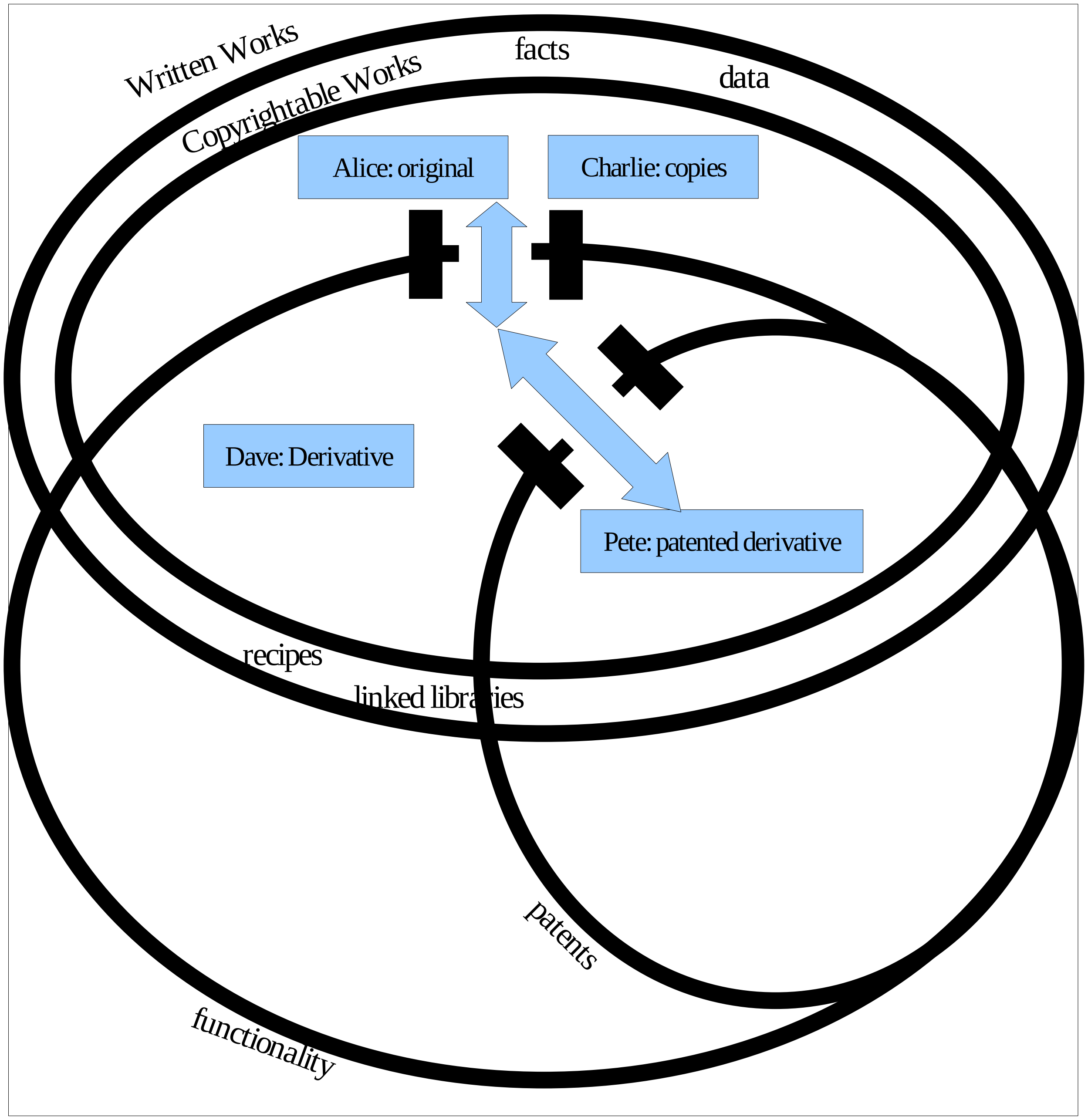
Labyrinth of Generic Copyleft License with no Patent Clause

The diagram below shows a generic copyleft license with no patent clause. Alice creates the original work and places it under the generic copyleft license. Charlie can make copies. And Dave can create derivative works. Pete comes along and creates a derivative of Alice's work that contains a patent. Pete can access Alice's original and Dave's derivative, but Alice and Dave can't use Pete's patented derivative.



Labyrinth of Generic Copyleft License with Patent Protection Clause

The generic copyleft license can remove the trapdoor provided by patent law by inserting a patent clause of some kind. The patent clause might say no one can patent the work or derivatives of the work., which would be like installing a door with no doorhandle so no one can access the patent subset. Or the license might say that people can patent derivatives, but those patents must licensed freely available to everyone else in the community, which effectively takes the door off the hinges. If you take a derivative into the patent “room”, you can't close the door behind you to monopolize your version.



Digital Rights Management (DRM)

Digital Rights Management is any functionality added to a design that attempts to control the viewer's rights to the content. This functionality may be implemented in software (section 4 and 5), or in hardware (section 6 and 7), or a combination of both. It is generally recognized that no DRM system can be uncrackable because it has to give the user encrypted content and the hardware and software to decrypt it. What makes DRM a serious hurdle for FLOS projects is the Digital Millenium Copyright Act (DMCA) which makes it illegal to circumvent DRM, even for lawful purposes.

Copyright law and Fair Use might say that the owner of a music Compact Disc would be allowed to legally copy that CD to the person's MP3 player. If the music industry doesn't like that, they might try to release the music with DRM embedded in it such that it makes it difficult for the average user to make a copy for their automobile, and another copy for their MP3 player.

DRM can be applied anywhere within the above Venn Diagrams and can attempt to control all rights, even the rights the user would normally be legally granted via the law and Fair Use and so on. DRM attempts to either block off all the rooms in the Venn Diagrams from the users, or close off some of the rooms.

DRM is similar to patents in the notion that it creates these rooms with one-way doors that allows a person to create some derivative, put DRM on it which operates outside the generic copyleft license, and that DRM prevents others from entering the room, from accessing the derivative, or accessing the function.

In FLOS projects, it is generally agreed that DRM should not be allowed to be used to prevent the basic rights of copy and create derivative works, or to circumvent source code requirements if they exist. Where the disagreement occurs, and the different licensing schemes arise, is whether a FLOS license should allow DRM to be restrict some function outside of basic copyright law.

This most common example of this is called Tivoization.

Tivoization

Tivoization is an interesting twist in the history of Libre Licensing.

In 2006, a company named Tivo came out with a Digital Video Recorder (DVR) which was essentially a small computer running linux. The DVR hooked up to your television and allowed users to record television shows to the computer's harddrive and play them back, instantly skip commercials, smart programming, and several other features.

The problem was that TiVo had some specialty hardware built into the TiVo box itself that would only allow TiVo's particular derivative of Linux to run on the hardware. Even if you paid for a TiVo box and had the know-how to modify the Linux operating system, and downloaded your modified version of linux to the TiVo box, the TiVo hardware would detect that it was a non-TiVo derivative and refuse to run it.

TiVo had followed the source code requirement of GNU-GPL v2, and provided the source code of their version of Linux, but they only sold systems that would guarantee that no one could use that source code to create their own derivatives that would run on the same hardware. TiVo felt they had followed the letter of the license, and had therefore not violated it.

Some came out against TiVo's behaviour. and supported modification of the GNU-GPL to add an anti-TiVoization clause, which disallowed people from using DRM to control who could run the software.

Others thought that it was OK for TiVo to use DRM hardware to control what software ran on their hardware platform. This group thought that the GNU-GPL license should only restrict itself to copyright law and software, and shouldn't get into issues of hardware.

In the end, the GNU-GPL was modified to version 3, which contains, among other things, an anti-tivoization clause.

But the debate over tivoization still remains somewhat divided in the FLOS community. Some would argue that DRM and the DMCA shouldn't be used to create a private room in the Venn Diagram that the rest of the community cannot enter themselves, that if a company like TiVo wants to use Linux, then that company should design their hardware so that users can run their own versions of linux on that hardware. Others argue that an anti-Tivoization clause will scare off other companies that would use Linux if they could use DRM to create their own private room, that it would be better to allow a company like TiVo to have their private room rather than to not have any company using Linux at all. These folks then argue that corporate competition would push some companies to come up with non-DRM or non-Tivoized versions of hardware.

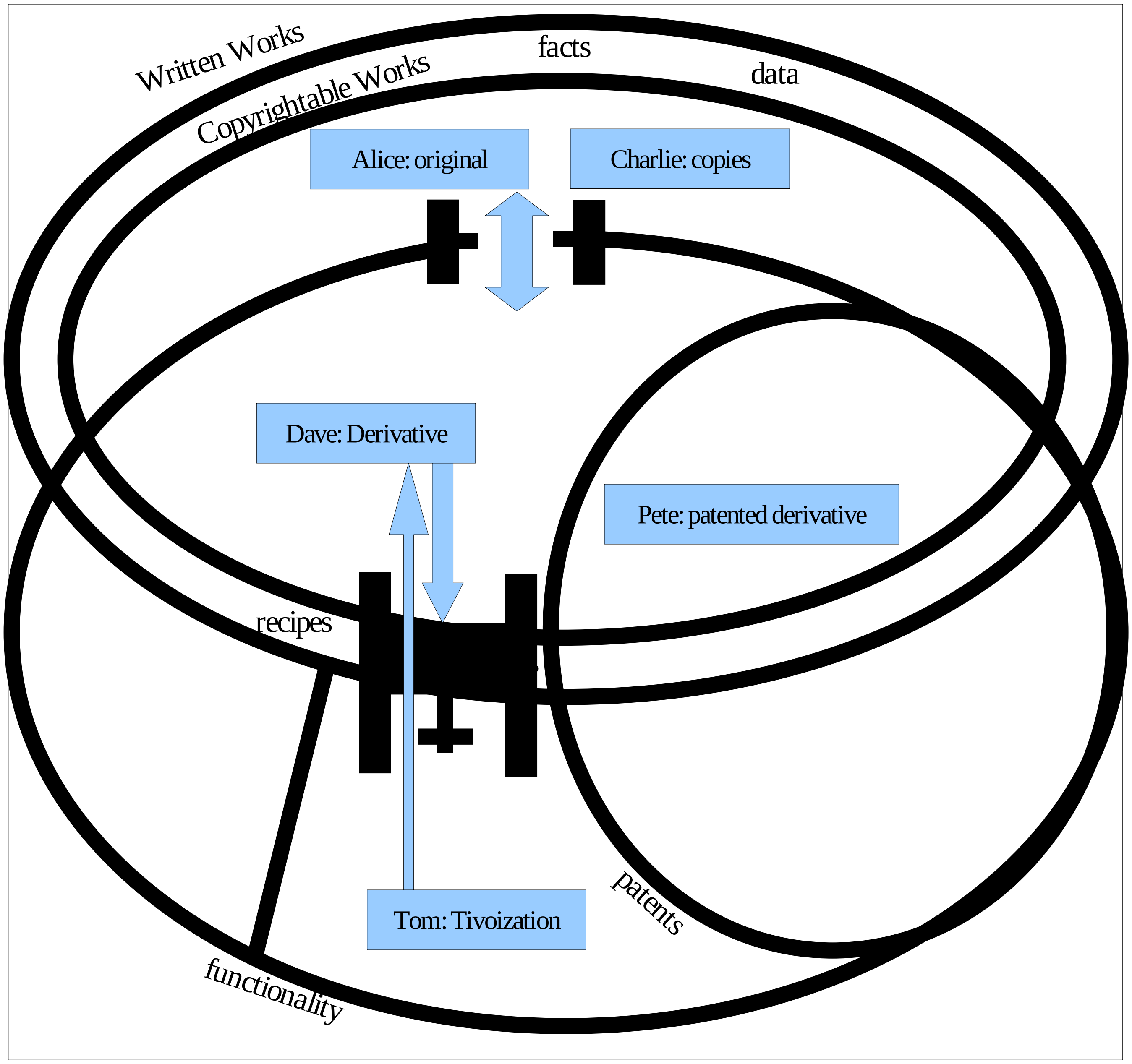
The argument seems to come down along the lines of how you look at what the priority of the license should be. Who is more important? (1) the community that created the original works or (2) the downstream users who may create derivatives. If you think the emphasis should be on the original developers, you probably support an anti-tivoization clause to prevent the downstream developers from creating a private room that the original authors can't enter. If you think the emphasis should be on the downstream developers and whatever they might create (even if it means the original creators can't use it), then you probably support the use of DRM to create private rooms because such private rooms will obviously encourage some folks to use FLOS works that might not have used it if those private (proprietary) rooms were not available.

Whatever your reasoning, there are basically two choices. Either you have a copyleft license with no anti-tivoization clause, or you have a copyleft license that prevents tivoization. These two options are shown below.

There are some uses of DRM that do not involve attempting to create a private room. Digital signatures, which is sometimes used in DRM, can also be used to electronically verify identities and other purposes which aren't attempting to create private, proprietary rooms. The GPL-V3 contains an anti-tivoization clause, but also allows certain uses of digital signatures and related functions.

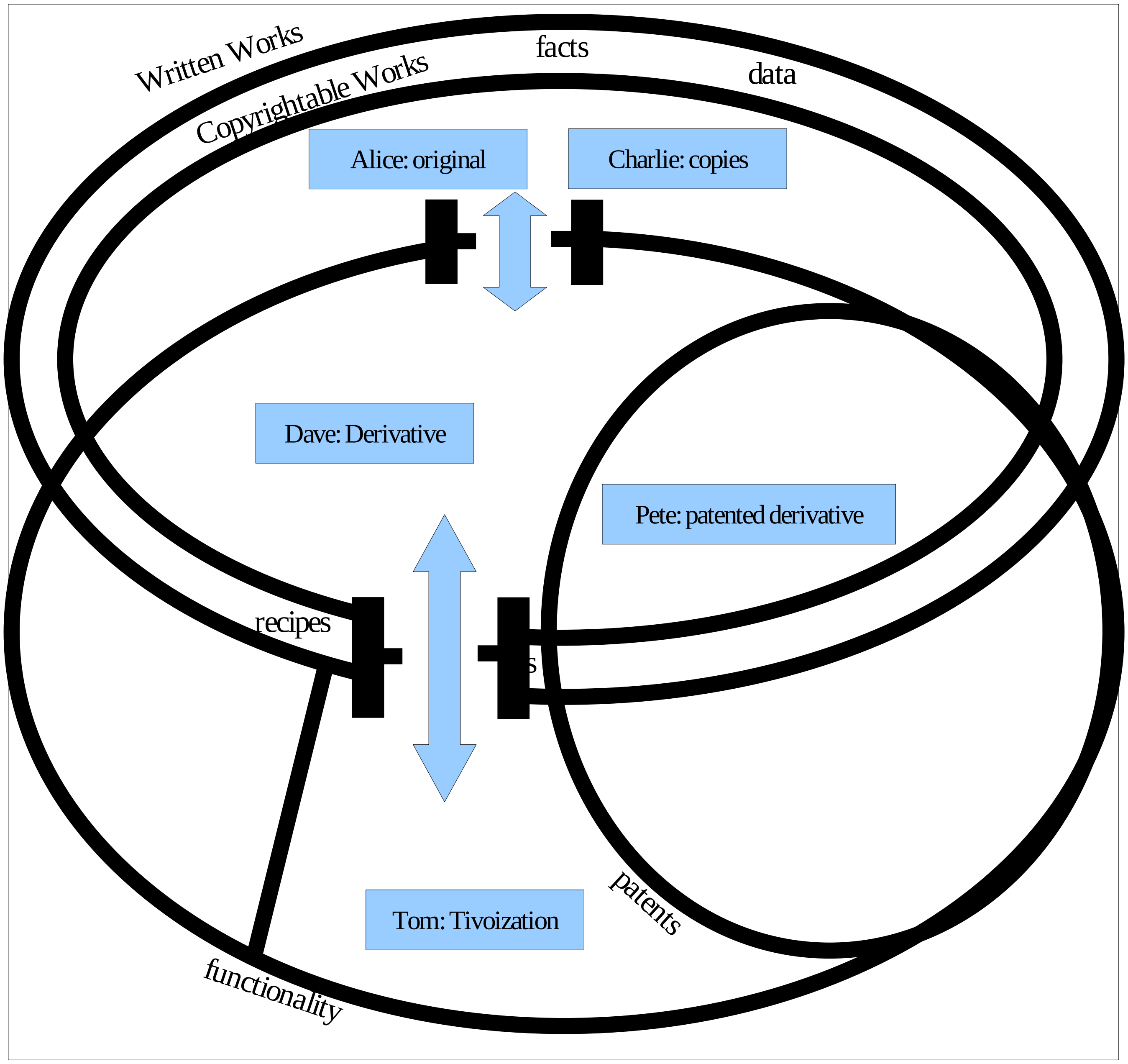
Labyrinth of a Generic Copyleft License with no Anti-Tivoization Clause:

The diagram below shows what happens when someone uses functionality outside basic copyright law to restrict how the work is used. Tom creates some Tivoized functionality that allows him to access Alice and Dave's version and pull it into the functional subset, but prevents Alice and Dave from entering the same room as Tom.



Labyrinth of a Generic Copyleft License with an Anti-Tivoization Clause:

A copyleft license could either completely disallow tivoization of any kind, or could allow the use of DRM as long as the DRM isn't used to create a private room within the labyrinth. (Note the similarities with possible patent solutions.) The diagram below shows a generic copyleft license that allows DRM as long as it isn't used to create a private room.



Open Hardware Licensing

If a group of hardware folks wanted to create an open hardware project and chose to use a copyleft software license such as the GNU-GPL, they would find differences between software and hardware would affect what kind of leverage they would actually get. While software projects operate for the most part under copyright law, hardware projects do not.

The issue is that while you might have code that describes some hardware project, and that code might look vaguely similar to source code in a Linux project, once you “compile” hardware source code, you may not be in the copyright circle at all.

Chips are generally coded up using a Hardware Description Language (HDL), with the most common language probably being Verilog. Verilog looks a lot like basic C code, but has a number of features to describe the parallelism that occurs in hardware designs. The verilog source code for a chip is converted to a lower level description of gates in a process called “synthesis”, which is similar to the software concept of “compiling”. But rather than producing an executable, a gate level description of the chip is produced, which contains a description of all the transistors in the chip, and how they are wired together. This description gets converted into “masks”. The fabrication plant prepares a layer of the chip by chemically coating it with some light sensitive material, then placing the mask over the chip and shining light through it. This process is repeated layer by layer until a working chip is created.

The result of fabrication is a purely functional piece of hardware: a piece of semiconductor material with doping to make transistors, and some kind of conductor to act as interconnecting wires. A chip.

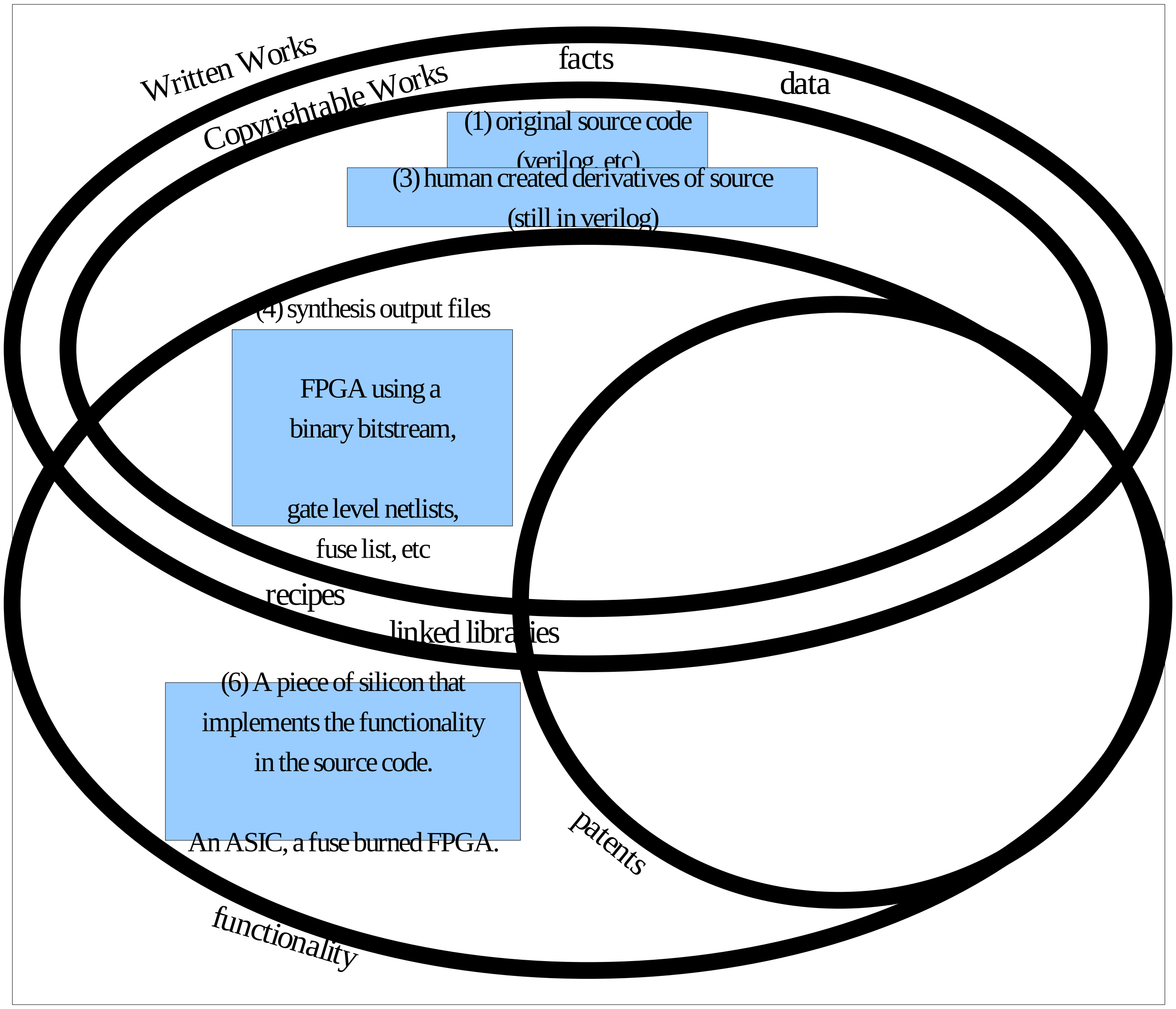
That chip is outside the “copyright” circle and inside the “functional” circle.

What this means is that if Alice uses something like the GNU-GPL for a hardware project and Dave creates a derivative of Alice's work, Dave may be able to distribute his end product in the form of a piece of silicon which isn't a copyright “derivative” and therefore Dave never triggers the source code requirement of the GNU-GPL.

If Dave creates a derivative of the verilog code and distributes the functionality in the form of a bitstream used to program an FPGA, then that bitstream may still be considered a derivative of the source code and would invoke the source code requirement. But a fuse-programmed FPGA would probably not count as a derivative, since blown fuses would be functional. If Dave created a special ASIC with his derivative functionality, then that ASIC would not be considered a copyright derivative, either, since it would be purely functional.

Venn Diagram of Hardware Process

The below diagram shows the different stages of the hardware process from source code to final silicon. It highlights how the final product of the process may occur outside of the concept of a “derivative” and therefore not trigger the source code requirement of a copyleft license.


Written Works
Copyrightable Works
facts
data
(1) original source code (verilog, etc).
(3) human created derivatives of source (still in verilog)
(4) synthesis output files
FPGA using a binary bitstream,
gate level netlists, fuse list, etc
recipes
linked libraries
(6) A piece of silicon that implements the functionality in the source code.
An ASIC, a fuse burned FPGA.
patents
functionality

If an Open Hardware project wants to leverage the benefits of a Source Code requirement, it will need to address the fact that while software executables are still copyright derivatives of their source code, silicon chips are not copyright derivatives of their source code.

Proposed Open Hardware License

An open hardware license would be useful for having some sort of license that would provide a source code requirement that would work when the copyright derivatives of source code is usually not distributed. The source code requirement would be “always on” like the Apple Public License.

Because a source code requirement that is “always on” would always infect the entire design of an ASIC, open hardware works would more likely be adopted if that source code requirement was limited to functional boundaries, rather than the maximum definition allowed for “derivative” as defined by the law and courts. So, the open hardware license will limit the copyleft nature of the license to functional boundaries, similar to the GNU-LGPL.

A patent protection clause would be recommended. And it would be limited to the functional boundary. This would allow a block to be used in an ASIC that contains a patent in an unrelated function without the user having to make the patent available to everyone. It would also protect the original function from being modified with a patent inside the function. A downstream user may add patented functionality inside the original functional boundary, but that patent must be freely licensed back to the community for that function.

An anti-tivoization clause would be recommended. This would apply to any software or hardware anywhere in the process (from the creation of the chip itself, to firmware uploads in the end product being done by the user), but would be limited in how they apply to the original functional boundaries. DRM could be used in some unrelated functional block, as long as it isn't used to prevent users from modifying the original functional blocks. An FPGA might use some sort of digital signature logic to control what bitstreams are loaded. This would be allowed for blocks outside the functional boundary of the original open hardware block, but would not be allowed to be used on anything inside the functional boundary of the open hardware blocks.

The “labyrinth” for an open hardware license is shown later in this document.

Side by Side Comparison of Different Real World Licenses

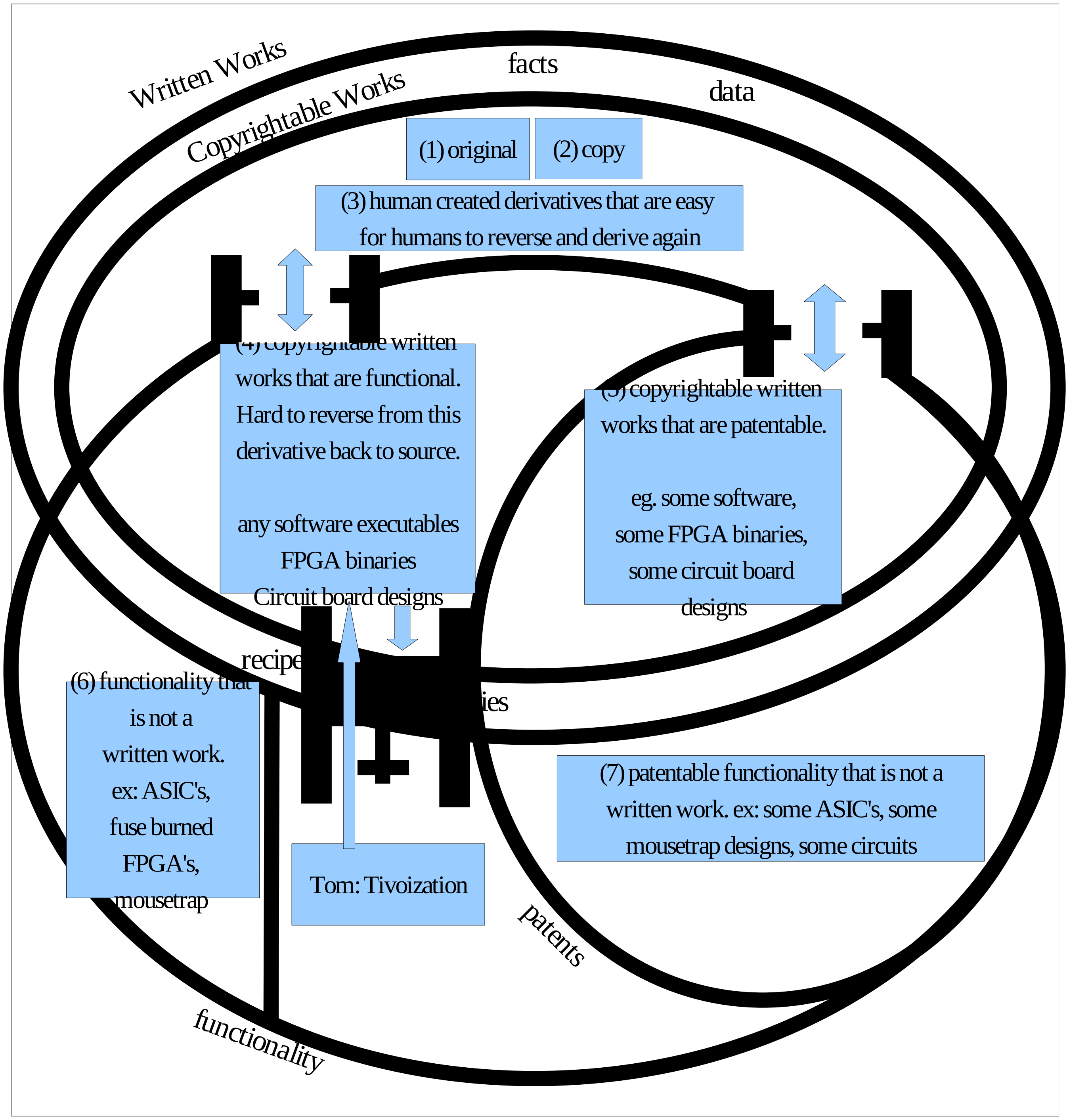
So far, we've only shown Venn Diagrams or Labyrinths for “generic” licenses. This was because the different sets and subsets of the Venn Diagram needed to be distinguished first and understood in terms of a generic license. What this produced was a single Venn Diagram to display all the different sets and subsets, along with an accompanying “overlay” which indicates when the source code requirement is triggered (if that license has a source code requirement)

We can now use this single Venn Diagram for each real world license and place different doors in different locations to indicate what transitions that license allows and how. We can also use that single overlay and check off which items will trigger the source code requirement.

(page inserted so licenses start on even page number)

Labyrinth of GNU-GPL v2

The GNU-GPL version 2 is a copyleft license with a source code requirement that applies to the widest definition of “derivative”. It has a patent clause that allows patents only if the patent is freely available to the community that created the original source code. It does not have an anti-tivoization clause.

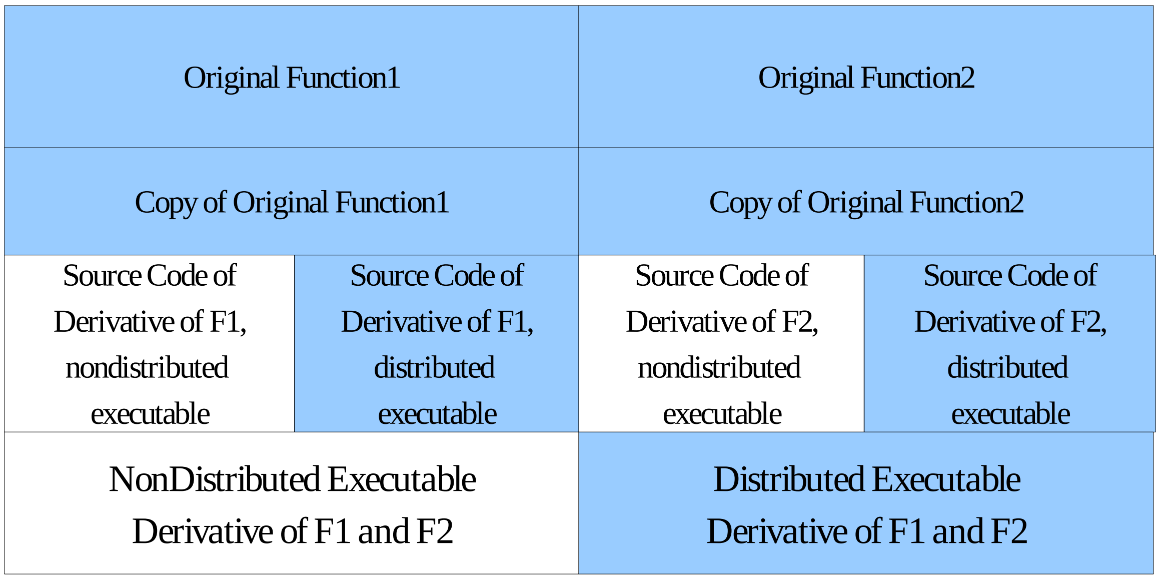


Written Works
Copyrightable Works
facts
data
(1) original
(2) copy
(3) human created derivatives that are easy for humans to reverse and derive again
(4) copyrightable written works that are functional. Hard to reverse from this derivative back to source.
any software executables
FPGA binaries
Circuit board designs
(5) copyrightable written works that are patentable.
eg. some software, some FPGA binaries, some circuit board designs
recipe
ies
(6) functionality that is not a written work. ex: ASIC's, fuse burned FPGA's, mousetrap
Tom: Tivoization
(7) patentable functionality that is not a written work. ex: some ASIC's, some mousetrap designs, some circuits
patents
functionality

Labyrinth overlay of GNU-GPL v2

The GNU-GPL uses the widest definition of "derivative" allowed by law. This means that "library" functions can be linked into another program without being considered a derivative, but non-library functions that are used are considered a derivative of those non-library functions, triggering the source code requirement.

The overlay below is colored blue for the sections that apply to "derivative" and therefore have the copyleft license applied to it.

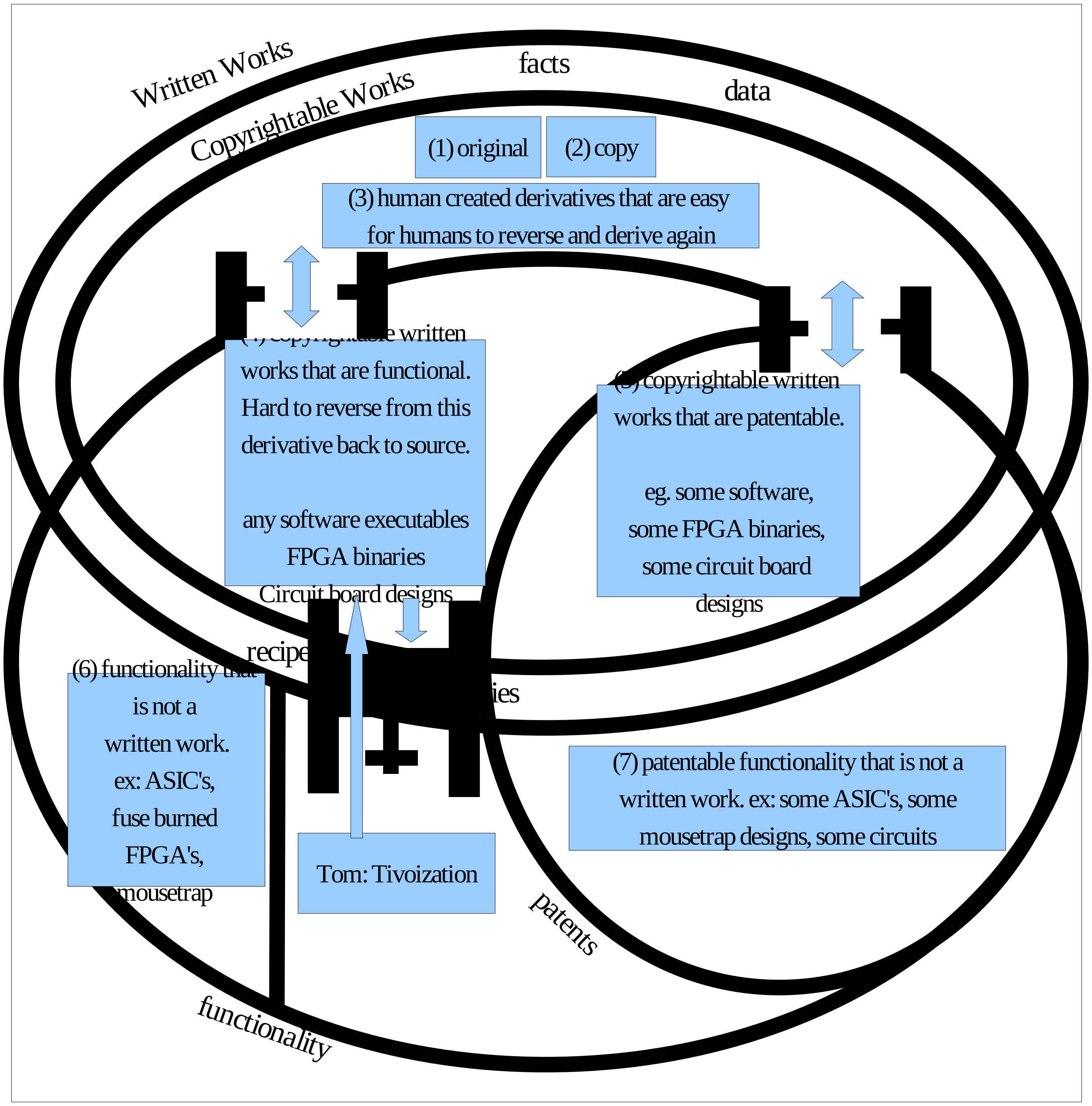


Original Function1
Original Function2
Copy of Original Function1
Copy of Original Function2
Source Code of Derivative of F1, nondistributed executable
Source Code of Derivative of F1, distributed executable
Source Code of Derivative of F2, nondistributed executable
Source Code of Derivative of F2, distributed executable
NonDistributed Executable Derivative of F1 and F2
Distributed Executable Derivative of F1 and F2

Labyrinth of GNU-LibraryGPL

The GNU-LibraryGPL version 2 is a copyleft license with a source code requirement that applies to a functional boundary. It has a patent clause that allows patents only if the patent is freely available to the community that created the original source code. It does not have an anti-tivoization clause.

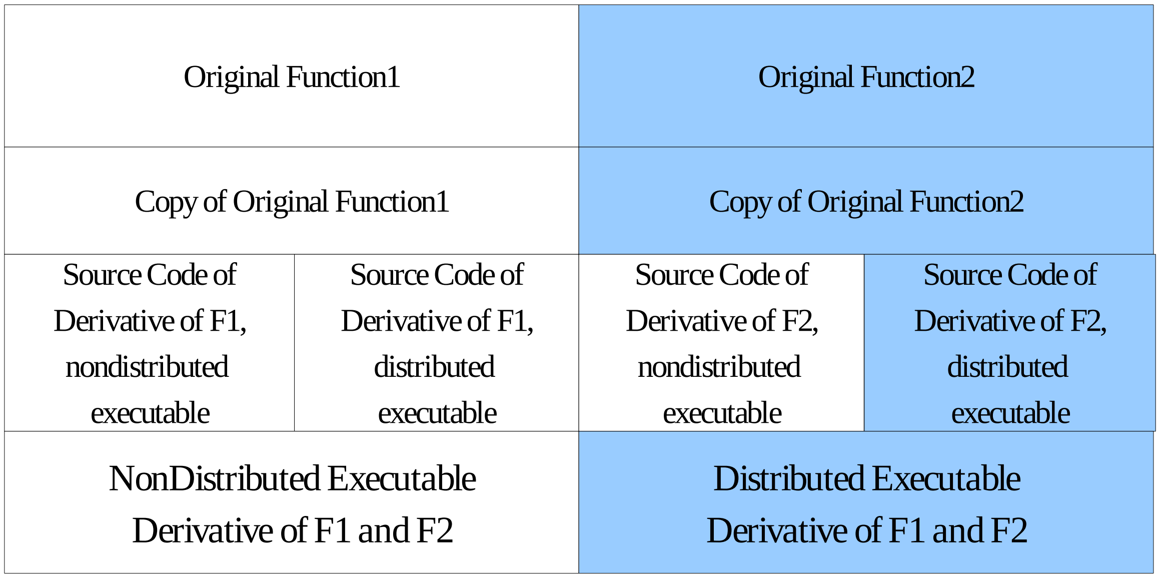
The GNU-LGPL has the same labyrinth as the GNU-GPL. The difference is only in the overlay.


Written Works
facts
data
Copyrightable Works
(1) original
(2) copy
(3) human created derivatives that are easy for humans to reverse and derive again
(4) copyrightable written works that are functional. Hard to reverse from this derivative back to source. any software executables FPGA binaries Circuit board designs
(5) copyrightable written works that are patentable. eg. some software, some FPGA binaries, some circuit board designs
recipe
ies
(6) functionality that is not a written work. ex: ASIC's, fuse burned FPGA's, mousetrap
(7) patentable functionality that is not a written work. ex: some ASIC's, some mousetrap designs, some circuits
Tom: Tivoization
patents
functionality

Labyrinth overlay of GNU-LGPL

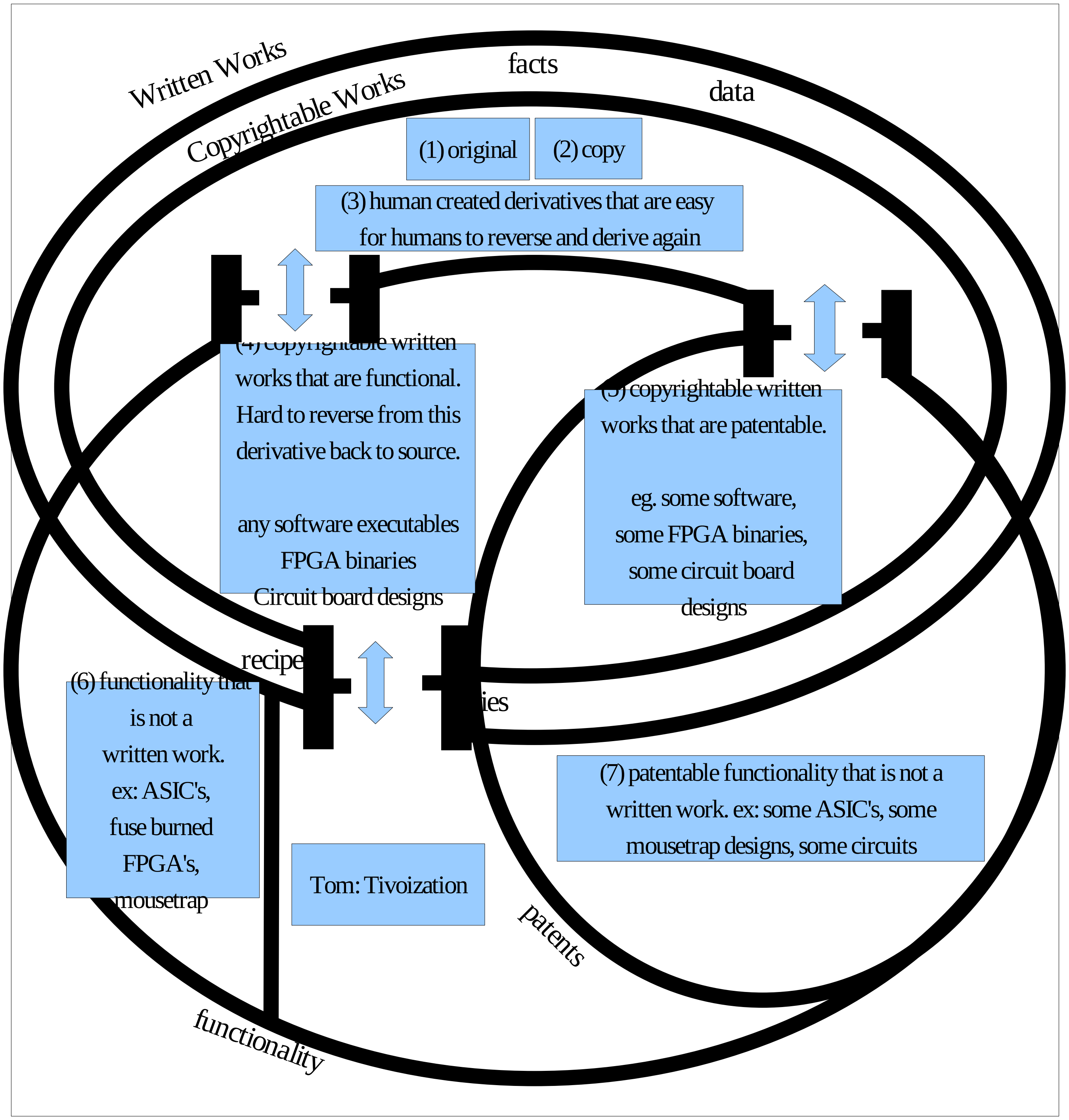
The GNU-LGPL limits its source code requirement to a functional boundary. This means that if F2 is GNU-LGPL, that function can be linked with a non-copyleft piece of software (function1) without triggering the source code requirement of function1.

The overlay below is colored blue for the sections that apply to “derivative” and therefore have the copyleft license applied to it.


Original Function1
Original Function2
Copy of Original Function1
Copy of Original Function2
Source Code of Derivative of F1, nondistributed executable
Source Code of Derivative of F1, distributed executable
Source Code of Derivative of F2, nondistributed executable
Source Code of Derivative of F2, distributed executable
NonDistributed Executable Derivative of F1 and F2
Distributed Executable Derivative of F1 and F2

Labyrinth of GNU-GPL v3

The GNU-GPL version 2 is a copyleft license with a source code requirement that applies to the widest definition of “derivative”. It has a patent clause that allows patents only if the patent is freely available to the community that created the original source code. It also has an anti-tivoization clause.

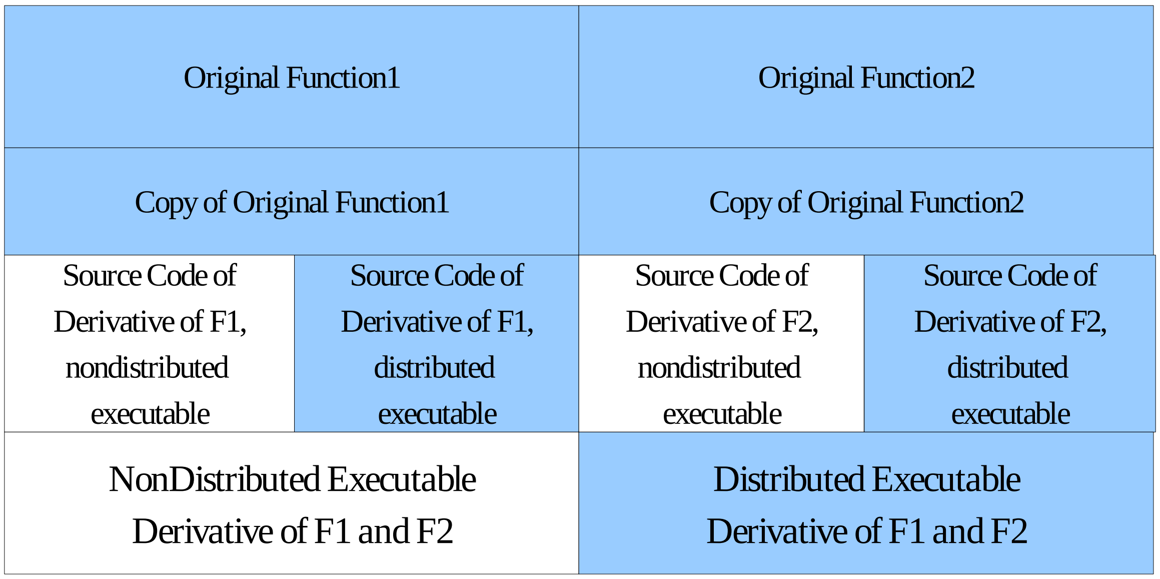

Written Works
Copyrightable Works
facts
data
(1) original
(2) copy
(3) human created derivatives that are easy for humans to reverse and derive again
(4) copyrightable written works that are functional. Hard to reverse from this derivative back to source.
any software executables
FPGA binaries
Circuit board designs
(5) copyrightable written works that are patentable.
eg. some software, some FPGA binaries, some circuit board designs
recipe
ies
(6) functionality that is not a written work. ex: ASIC's, fuse burned FPGA's, mousetrap
(7) patentable functionality that is not a written work. ex: some ASIC's, some mousetrap designs, some circuits
Tom: Tivoization
patents
functionality

Labyrinth overlay of GNU-GPL v3

The GNU-GPL uses the widest definition of “derivative” allowed by law. This means that “library” functions can be linked into another program without being considered a derivative, but non-library functions that are used are considered a derivative of those non-library functions, triggering the source code requirement.

The overlay below is colored blue for the sections that apply to “derivative” and therefore have the copyleft license applied to it.

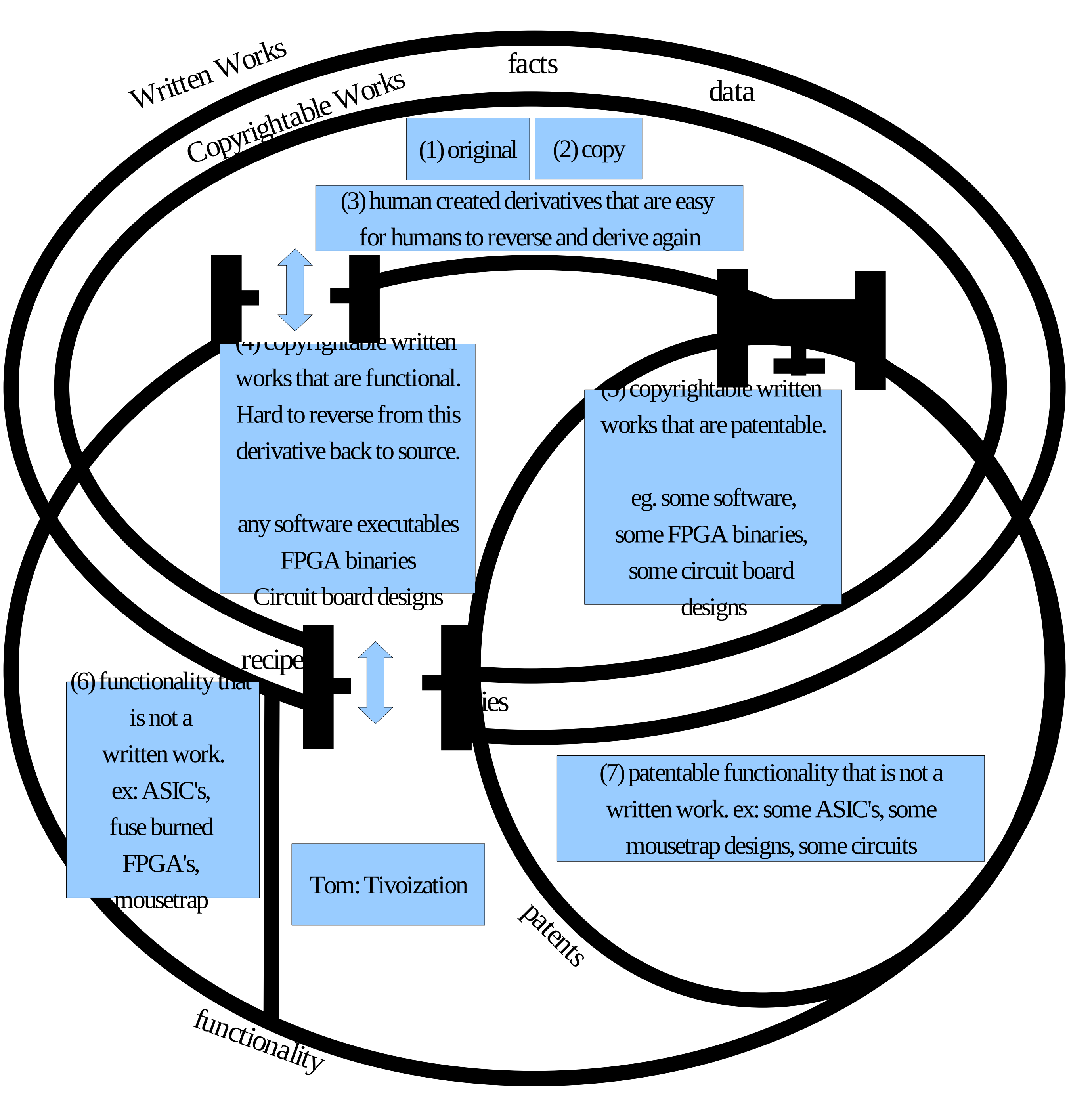
Note this is the same overlay as it was for GNU-GPL v2



Original Function1
Original Function2
Copy of Original Function1
Copy of Original Function2
Source Code of Derivative of F1, nondistributed executable
Source Code of Derivative of F1, distributed executable
Source Code of Derivative of F2, nondistributed executable
Source Code of Derivative of F2, distributed executable
NonDistributed Executable Derivative of F1 and F2
Distributed Executable Derivative of F1 and F2

Labyrinth of Creative Commons Share Alike

The Creative Commons Share Alike license is a copyleft license with no source code requirement and no patent protection clause. It contains a sort of anti-tivoization clause. Creative Commons does not recommend the CC-SA license for software projects.



Written Works
Copyrightable Works
facts
data
(1) original
(2) copy
(3) human created derivatives that are easy for humans to reverse and derive again
(4) copyrightable written works that are functional. Hard to reverse from this derivative back to source.
any software executables
FPGA binaries
Circuit board designs
(5) copyrightable written works that are patentable.
eg. some software, some FPGA binaries, some circuit board designs
recipe
ies
(6) functionality that is not a written work. ex: ASIC's, fuse burned FPGA's, mousetrap
(7) patentable functionality that is not a written work. ex: some ASIC's, some mousetrap designs, some circuits
Tom: Tivoization
patents
functionality

Labyrinth overlay of CC-SA

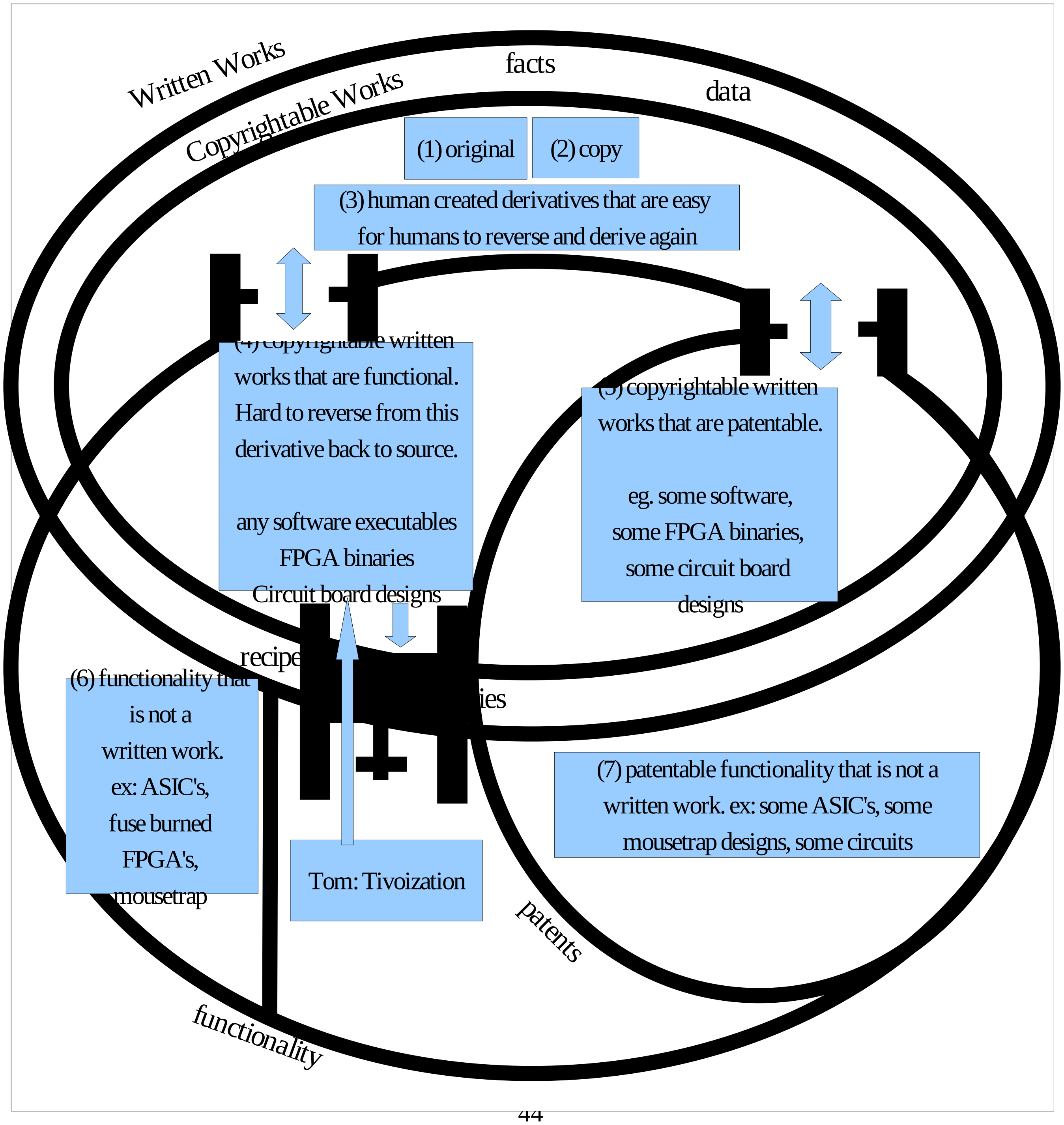
The CC-SA license has no source code requirement. Only the work distributed is required to be licensed CC-SA. No source files of any distributed work needs to be provided.

The overlay below is colored blue for the sections that apply to “derivative” and therefore have the copyleft license applied to it.

Original Function1		Original Function2	
Copy of Original Function1		Copy of Original Function2	
Source Code of Derivative of F1, nondistributed executable	Source Code of Derivative of F1, distributed executable	Source Code of Derivative of F2, nondistributed executable	Source Code of Derivative of F2, distributed executable
NonDistributed Executable Derivative of F1 and F2		Distributed Executable Derivative of F1 and F2	

Labyrinth of Apple Public Source License

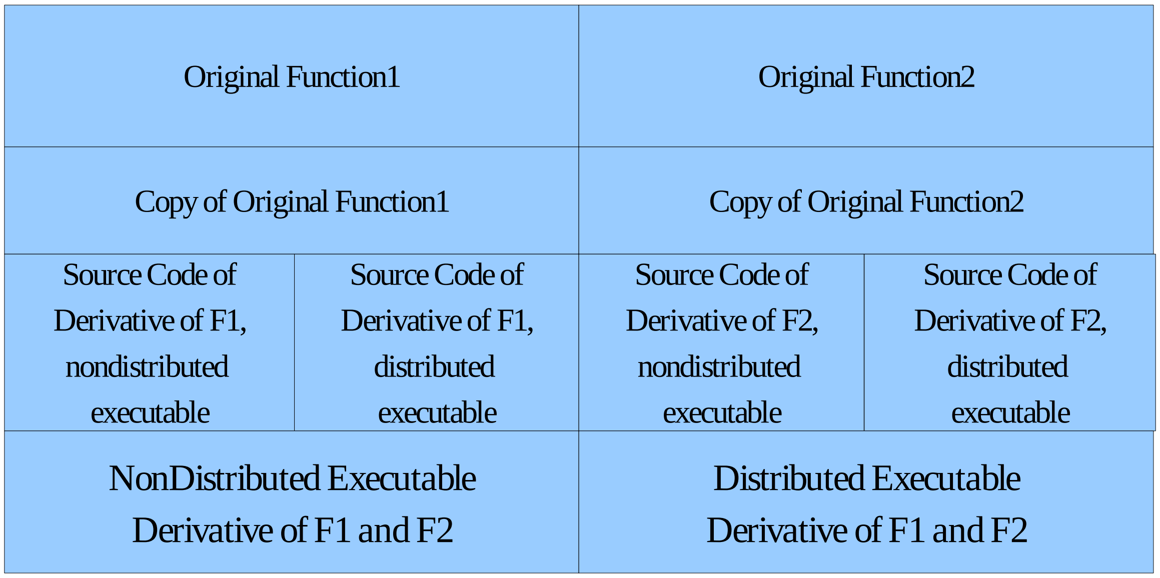
The Apple Public Source license is a copyleft license. It contains a source code requirement that uses the widest legal definition of “derivative”. The source code requirement is triggered by any modification, without requiring the modification to be distributed. The license contains a patent protection clause. The license does not contain an anti-tivoization clause.


Written Works
facts
data
Copyrightable Works
(1) original
(2) copy
(3) human created derivatives that are easy for humans to reverse and derive again
(4) copyrightable written works that are functional. Hard to reverse from this derivative back to source.
any software executables
FPGA binaries
Circuit board designs
(5) copyrightable written works that are patentable.
eg. some software, some FPGA binaries, some circuit board designs
recipe
ies
(6) functionality that is not a written work. ex: ASIC's, fuse burned FPGA's, mousetrap
(7) patentable functionality that is not a written work. ex: some ASIC's, some mousetrap designs, some circuits
Tom: Tivoization
patents
functionality

Labyrinth overlay of Apple Public Source License

The Apple Public Source License has a source code requirement that uses the widest legal definition of “derivative”. The source code requirement is triggered by any modification, without requiring the modification to be distributed.

The overlay below is colored blue for the sections that apply to “derivative” and therefore have the copyleft license applied to it.

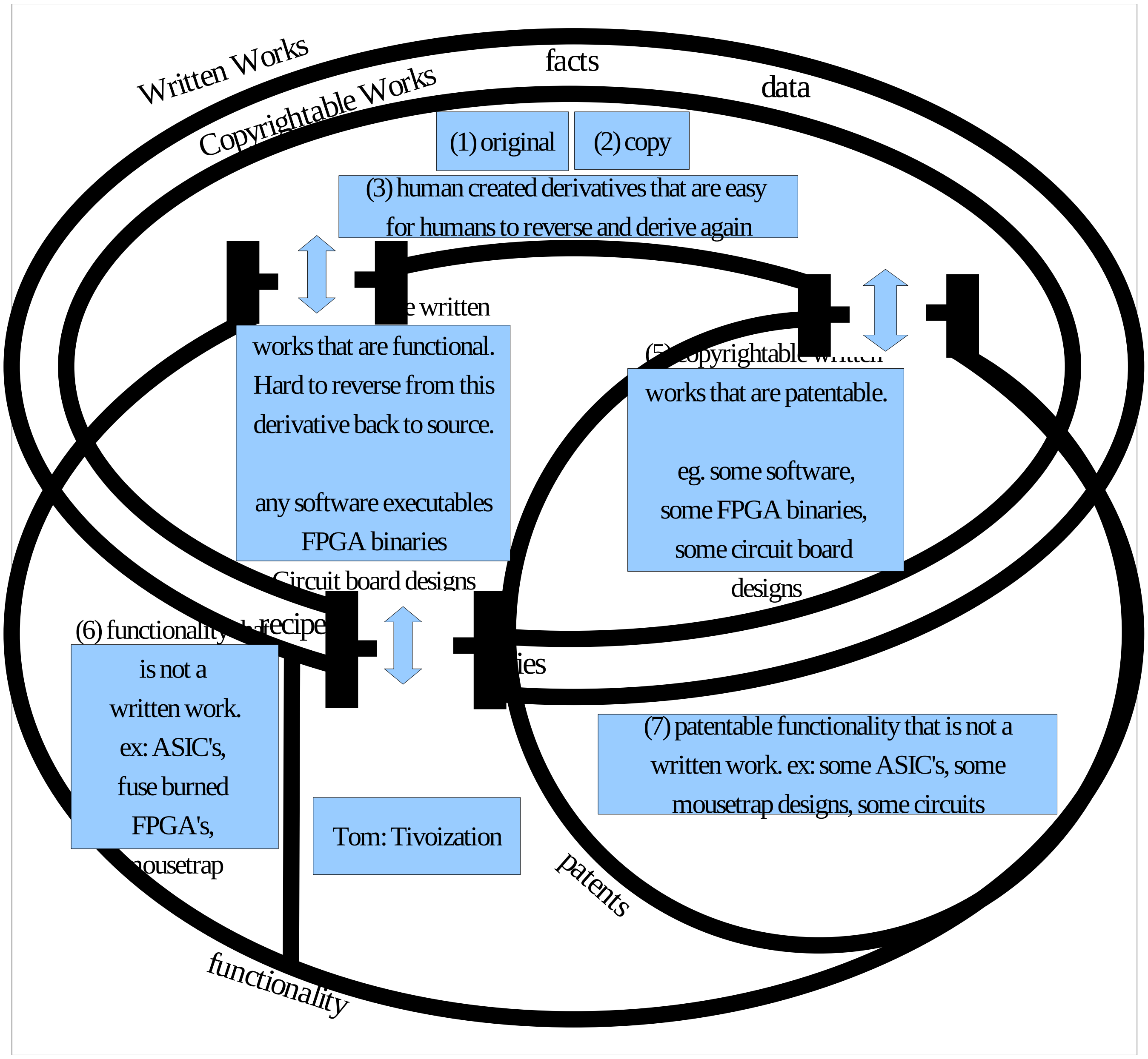


Original Function1
Original Function2
Copy of Original Function1
Copy of Original Function2
Source Code of Derivative of F1, nondistributed executable
Source Code of Derivative of F1, distributed executable
Source Code of Derivative of F2, nondistributed executable
Source Code of Derivative of F2, distributed executable
NonDistributed Executable Derivative of F1 and F2
Distributed Executable Derivative of F1 and F2

The overlay is all blue, meaning that the source code requirement is always triggered.

Labyrinth of Proposed Open Hardware License

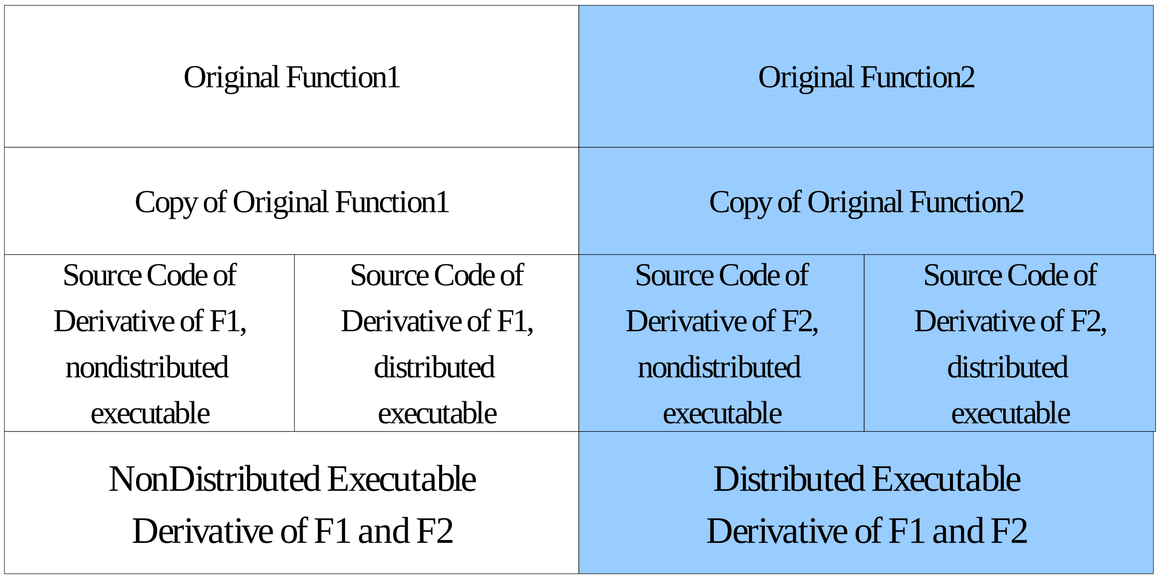
The proposed open hardware license is a copyleft license with a source code requirement that applies to a functional boundary and applies whether the work is distributed or not. It has a patent clause that allows patents only if the patent is freely available to the community that created the original source code or if the patent is outside the functional boundary of the original work. It has an anti-tivoization clause that applies to the entire application, not just the functional boundary.


Written Works
Copyrightable Works
facts
data
(1) original
(2) copy
(3) human created derivatives that are easy for humans to reverse and derive again
e written works that are functional. Hard to reverse from this derivative back to source.
any software executables FPGA binaries Circuit board designs
(5) copyrightable written works that are patentable.
eg. some software, some FPGA binaries, some circuit board designs
recipe
ies
(6) functionality that is not a written work. ex: ASIC's, fuse burned FPGA's, mousetrap
Tom: Tivoization
(7) patentable functionality that is not a written work. ex: some ASIC's, some mousetrap designs, some circuits
patents
functionality

Labyrinth Overlay of Proposed Open Hardware License

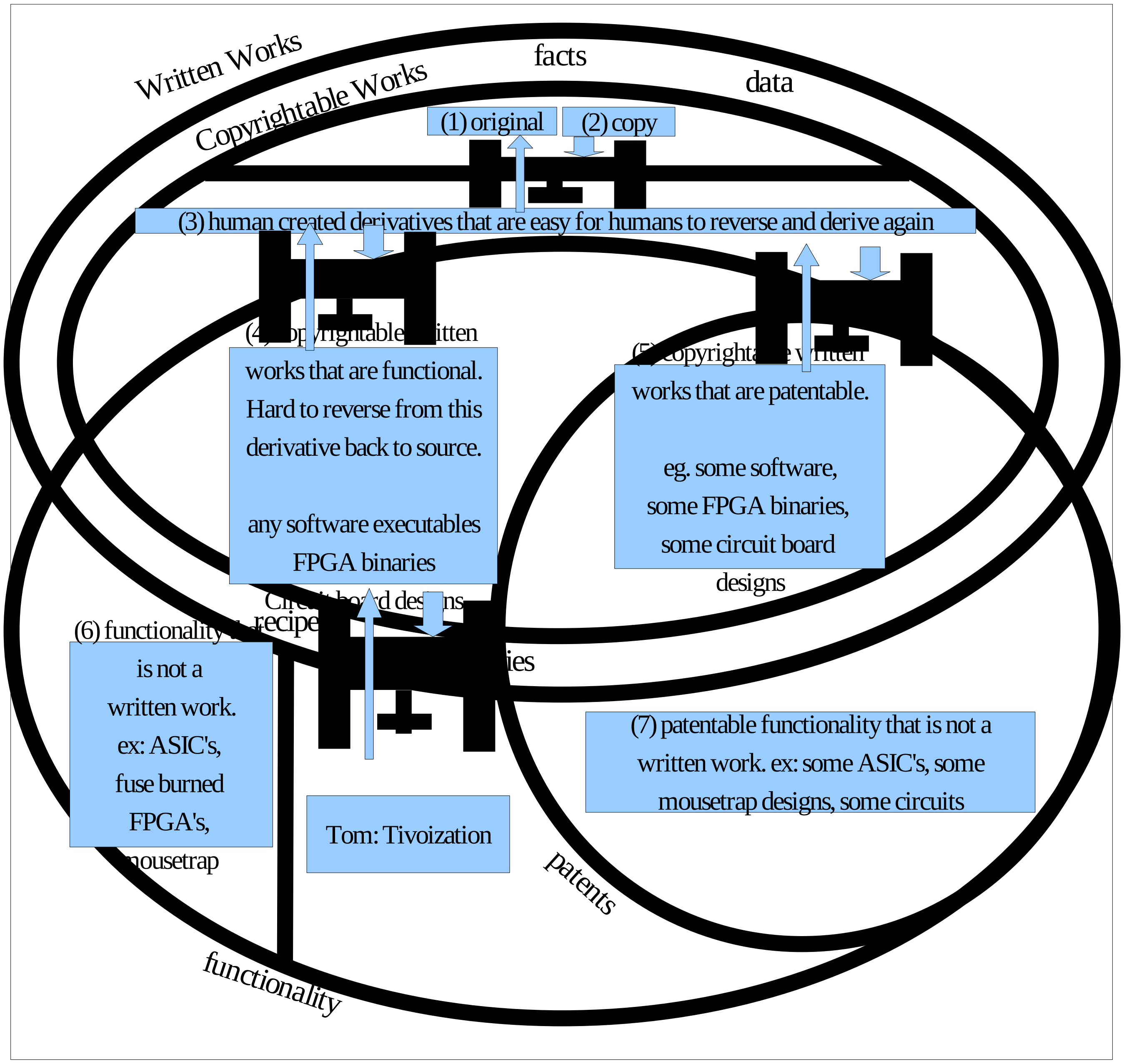
The proposed open hardware license limits its source code requirement to a functional boundary. This means that if F2 is an open hardware function, that function can be linked with a non-copyleft work (function1) without triggering the source code requirement of function1. The source code requirement is “always on”, however. Meaning that any derivative, whether it is distributed or not, must have its source code made available to the whole public.

The overlay below is colored blue for the sections that apply to “derivative” and therefore have the copyleft license applied to it.


Original Function1
Original Function2
Copy of Original Function1
Copy of Original Function2
Source Code of Derivative of F1, nondistributed executable
Source Code of Derivative of F1, distributed executable
Source Code of Derivative of F2, nondistributed executable
Source Code of Derivative of F2, distributed executable
NonDistributed Executable Derivative of F1 and F2
Distributed Executable Derivative of F1 and F2

Labyrinth of Public Domain Style License

The labyrinth for a Public Domain style license is shown below. Such a license would only be required to apply to the original work. Derivatives could be taken private. Patents could be used to close off functionality from the rest of the community. Tivoization could also be used to hold the work private and to prevent the community from using a hardware or software platform for their own derivatives. Even source code derivatives (3) can be closed off from the community who created the original (1).


Written Works
Copyrightable Works
facts
data
(1) original
(2) copy
(3) human created derivatives that are easy for humans to reverse and derive again
(4) copyrightable written works that are functional. Hard to reverse from this derivative back to source. any software executables FPGA binaries
(5) copyrightable written works that are patentable. eg. some software, some FPGA binaries, some circuit board designs
Circuit board designs
recipe
ies
(6) functionality that is not a written work. ex: ASIC's, fuse burned FPGA's, mousetrap
(7) patentable functionality that is not a written work. ex: some ASIC's, some mousetrap designs, some circuits
Tom: Tivoization
patents
functionality

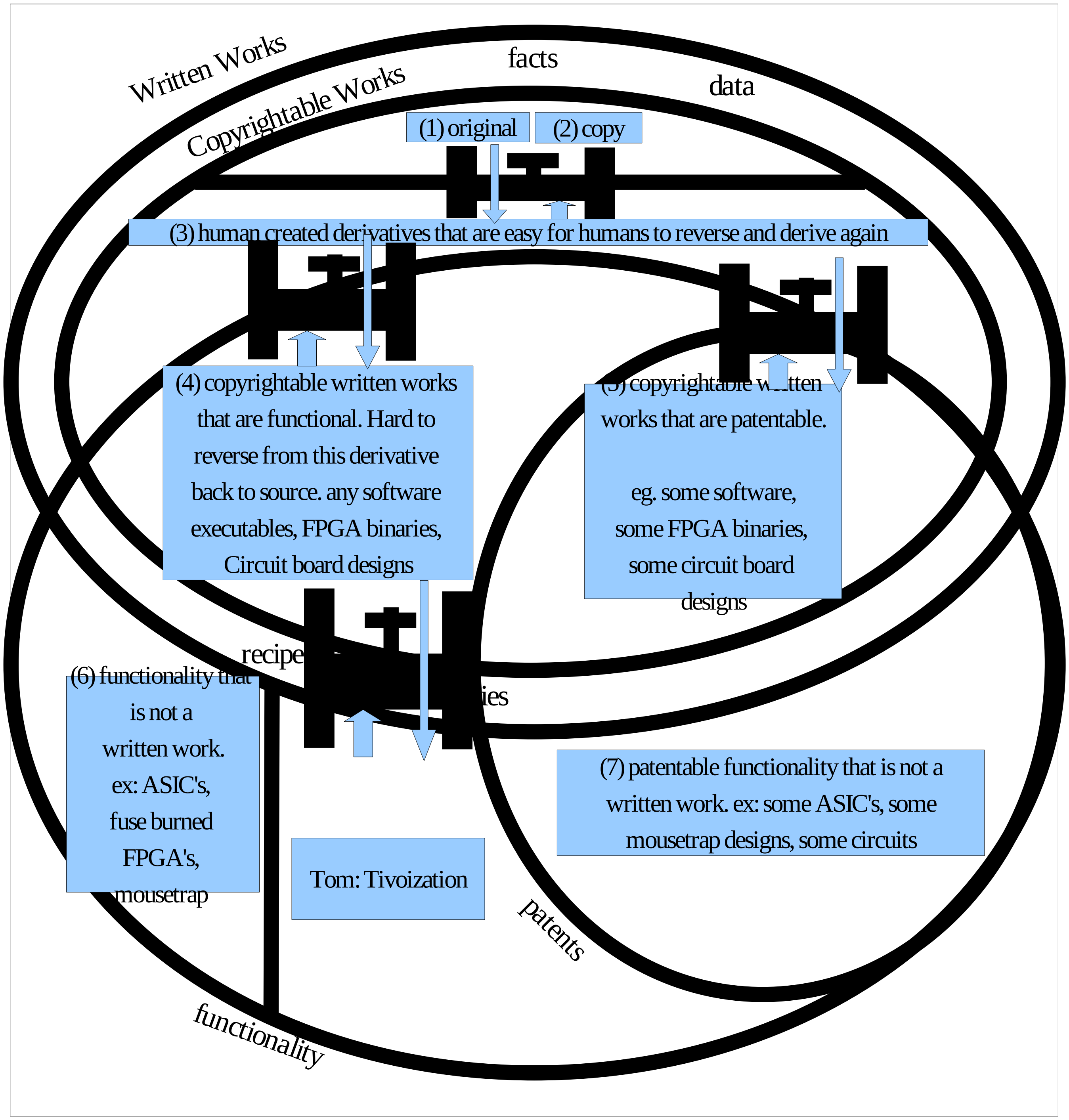
Labyrinth Overlay of Public Domain Style Licenses

Since public domain style licenses have no source code requirement, the overlay is completely inactive. Nothing can trigger the source code requirement because there is no source code requirement to trigger. Even a distributed executable can be licensed All Rights Reserved, so not even that block is active.

Original Function1		Original Function2	
Copy of Original Function1		Copy of Original Function2	
Source Code of Derivative of F1, nondistributed executable	Source Code of Derivative of F1, distributed executable	Source Code of Derivative of F2, nondistributed executable	Source Code of Derivative of F2, distributed executable
NonDistributed Executable Derivative of F1 and F2		Distributed Executable Derivative of F1 and F2	

Labyrinth of All Rights Reserved

The labyrinth of All Rights Reserved is shown below for comparison. Alice creates the original work and Alice controls access to all the rooms in the labyrinth.


Written Works
Copyrightable Works
facts
data
(1) original
(2) copy
(3) human created derivatives that are easy for humans to reverse and derive again
(4) copyrightable written works that are functional. Hard to reverse from this derivative back to source. any software executables, FPGA binaries, Circuit board designs
(5) copyrightable written works that are patentable.
eg. some software, some FPGA binaries, some circuit board designs
recipe
ies
(6) functionality that is not a written work. ex: ASIC's, fuse burned FPGA's, mousetrap
(7) patentable functionality that is not a written work. ex: some ASIC's, some mousetrap designs, some circuits
Tom: Tivoization
patents
functionality

Labyrinth Overlay for All Rights Reserved

All Rights Reserved works have no source code requirement, so the overlay is completely inactive.

Original Function1		Original Function2	
Copy of Original Function1		Copy of Original Function2	
Source Code of Derivative of F1, nondistributed executable	Source Code of Derivative of F1, distributed executable	Source Code of Derivative of F2, nondistributed executable	Source Code of Derivative of F2, distributed executable
NonDistributed Executable Derivative of F1 and F2		Distributed Executable Derivative of F1 and F2	

An Assessment of the Labyrinths

Some interesting questions emerge after mapping several of the more common copyleft licenses. The first question is “Why?” Why are the licenses designed the way they are designed? What is the goal of these licenses? Is there any pattern that emerges from all these different licenses such that we could create a common constitution simply by looking at the laws that a government implements? Can we discern an overall commitment based of these specific actions?

It is this meta discussion where all the debates in Free, Libre, and Open Source debates occur. People disagree on whether or not a license should have an anti-tivoization clause, and they immediately go to meta-arguments that are essentially answering “why” they believe the license exists. The reason the debate continues is partly because people don't really discuss WHY they want a FLOS license, they discuss the direct practical effect of WHAT they want the license to do.

All Rights Reserved versus FLOS

The first thing that is immediately obvious is that there is a commitment to FLOS licenses over All Rights Reserved licenses. It's sometimes easy to forget amongst the squabbles and flamewars about some aspect of a free license, but even those who disagree on HOW a FLOS license should be worded, both sides generally that FLOS is better than All Rights Reserved (ARR).

The different commitments that might source this include a commitment to contribute works to the general public, to do some good deed, to give a gift in the sense of a gift economy. Other commitments would probably include the notion of creating outstanding, high quality works that can't be done with All Rights Reserved works. All bugs become shallow with enough eyes, and FLOS is about enabling as many eyes as are interested in being able to see the bugs.

If nothing else, the labyrinth of All Rights Reserved versus the labyrinth of any other license highlights the fact that ARR basically focuses all its control on the original creator of the work. All the doors are situated such that only the original author may make a derivative, may fix a bug, may add a new feature.

There are also some political motivations. Some folks think that Copyright is wrong, that it shouldn't exist, or it shouldn't exist in it's current form, and these folks might be interested in using FLOS as a way to make a political statement that they are against the notion of Copyright and/or Patents.

Copyleft versus Public Domain style licenses

Once you choose a commitment to FLOS in some way, the first question is whether you should go with a Public Domain style license, such as MIT or BSD licenses, or whether you should go with some sort of copyleft license, such as GNU-GPL or one of the many other copyleft licenses.

Again, there are some who choose a Public Domain style license over a copyleft license for political reasons, to make a statement, to make a point, whatever. But we're interested in whether there are any functional differences between the two licenses such that one would be measurably better in some way than the other.

One measurable difference between Public Domain style licenses and Copyleft licenses is that Public Domain style licenses allow downstream developers to create a derivative, keep that derivative behind a closed door, and use that derivative to compete against the original work and the original creators. If Alice releases some work under a PD license, and if Dave creates a derivative of Alice's work, Dave can license his version All Rights Reserved, and Alice cannot access Dave's version, can't access Dave's source code. If Dave patents some of his functionality, Alice will have to wait 20 years to implement that functionality into her application. Dave can use all these private rooms to compete directly with Alice for users or customers or whatever.

As Alice improves her code and keeps it Public Domain, Dave can integrate her changes while keeping his version proprietary if he so chooses.

A Public Domain license installs some one-way doors that give Dave the means to compete against Alice. Copyleft licenses, on the other hand, remove some number of the one-way doors, such that Dave can't make a proprietary derivative, and can't compete against Alice.

The choice boils down to who is more important, who has more rights: the original author (Alice) or some downstream derivative maker (Dave).

If you choose a copyleft license, the answer to that question is Alice. If you choose a Public Domain style license, the answer is Dave. If you choose a copyleft license, the priority is that Alice and Dave remain on a level, or equal, playing field. Dave is allowed to make derivatives, but Alice is guaranteed access to those derivatives. If Dave wants to create a derivative, he must do so in some way that does not compete against Alice. Alice makes the original work, the original contribution, but only allows Dave to take it into a different room only if Alice can follow Dave into that same room and get his version of the work.

Alice choose copyleft because she wants to leverage the community in a way that encourages cooperation for mutual benefit. Public Domain style licenses allow the members of the community to compete against one another. Not that Dave must necessarily take a Public Domain style work, create a derivative, and make it Proprietary, but Alice is choosing up front that she'd rather Dave not contribute at all if Dave won't contribute to the whole community.

If Alice chooses a Public Domain style license, Alice is creating a work and licensing it in a way that indicates she isn't concerned with competition from downstream derivatives.

If Alice chooses a copyleft license, Alice is indicating that the focus of her license, the basis of her commitment, is to the original authors and the community of contributers who want to contribute Freely to the work the same way as Alice did.

Public Domain: (1) Downstream Derivatives then (2) Original Community

Copyleft: (1) Original Community then (2) Downstream Derivatives.

Copyleft A versus Copyleft B licenses

The biggest flamewars in the FLOS community seem to erupt over questions revolving around different versions of copyleft licenses. Should it have an anti-tivoization clause? Should it have a patent protection clause? Should it have a source code requirement? Etc. There are different copyleft licenses that have different answers to these same questions.

Different people support different licenses for different reasons. But it seems that some folks choose a particular flavor of copyleft and forget the main commitment that comes from choosing a copyleft license over a Public Domain style license: the original authors and the community of contributers they wish to hold as equals.

If the priority is not Alice and the community of equals, then a Public Domain license could be chosen instead, and the whole problem of licensing issues would be greatly simplified. Public Domain licenses don't have anti-tivoization, don't have patent protection clauses, don't have source code requirements.

Once a copyleft license is chosen, Alice and her community of equals is given priority over Dave and his proprietary derivatives, Pete and his patented derivative, and Tom and his tivoized private room.

The pattern that emerges when looking at all the copyleft Labyrinths is an attempt to pull all the doors off the hinges so that the community may travel freely from one room to another, so that the community is protected from someone using the community's work to compete against the community.

When a question arises over what a copyleft license should do about some particular thing like source code requirements, patents, tivoization, DRM, DMCA, or any other angle, the commitment that seems clear is that the work should be allowed to travel wherever people want it in the labyrinth, as long as, no one is allowed to close the door behind them and keep the rest of the community from following.

If a room is closed off, it is closed off completely. The door is welded shut and the door handle is removed. One-way doors are generally discouraged (source code requirements for derivatives, patent protection clauses, etc). And when people argue for one-way doors in a copyleft license, it seems they generally are sacrificing the commitment to Alice and the community in favor of allowing Dave to have some private and proprietary room that Alice and the community cannot enter.

If someone can argue for allowing Tom to create a private room via Tivoization and do so from some generic commitment that also supports the generic commitment of copyleft over public domain licenses, I have not heard it. If the commitment is to prioritize downstream derivatives over the original community, then a Public Domain license would follow through on that commitment far more than a copyleft license with source code requirements and patent protection clauses and so on.

Some argue against the anti-tivoization clause on the grounds that it violates the Freedom to Use the software any way a person wants. This freedom comes out of the older licenses that used to say stuff like “you're free to use this software any way you want as long as it isn't used for nuclear power, or military equipment, or in South Africa” or similar. These clauses create problems for the community by limiting who can join the community, and by permanently disallowing people to join that might otherwise join. It also creates problems because the license tries to code up “good political causes and bad political causes” into a license.

My reading of the “Freedom to Use” clause is that it occurs more as a “statement against politically motivated licensing” decisions. South Africa's government of apartheid, the reason some older licenses prohibited use in South Africa, no longer exists. No one argues that the “Freedom to Use” should override any Patent Protection clause. Yet some argue that the “Freedom to Use” clause should allow Tom to tivoize Alice's code.

In a copyleft license, when the “Freedom to Use” commitment bumps up against the commitment to “community of equal access”, the community of equal access should win out. Obviously, not everyone agrees with this. But it seems clear when looking at the labyrinths for the different licenses that the goal is to remove as much as possible any one-way trap doors. To allow the original authors equal access to any downstream derivatives.

A Public Domain style license has a “Freedom to Use” commitment as well, but in the Public Domain style licenses, that commitment overrules the “Community of equal access” commitment.

Public Domain: (1) Freedom of Use (including private derivatives) then (2) Community of Equals

Copyleft: (1) Community of equals then (2) Freedom of Use.

Overlay A versus Overlay B

The different overlays are a bit more difficult to extract a common commitment. And it is probably in the license overlay that a number of different commitments get mish-mashed up into different overlays with different priorities.

The most common copyleft license with no source code requirement at all, and therefore no overlay at all, is the Creative Commons-Share Alike license. This license is universally regarded as not recommended for software projects, so we'll simply ignore this overlay for comparisons.

The remaining licenses have overlays which can be simplified into two binary choices.

A source code requirement in a copyleft license may be triggered on (1) any distributed derivative, (2) any derivative whether distributed or not. And the copyleft license may also define derivative to mean (A) the maximum definition of derivative recognized by copyright or (B) a subset of derivatives limited by some functional boundary.

This creates a number of permutations:

(1A) distributed derivative, maximum derivative: GNU-GPL

(1B) nondistributed derivative, maximum derivative: Apple licence

(2A) distributed derivative, functional derivative: GNU-LibraryGPL

(2B) nondistributed derivative, functional derivative: proposed Open Hardware License

Triggering Based On Distribution or Always On

It seems that triggering the source code requirement off of distribution would show a commitment to a right to privacy of sorts. Dave could make his own private derivative, and as long as he kept that derivative to himself and didn't use it to compete against the community, he didn't have to show anyone his source code. This is the argument given for the GNU-GPL which is triggered by Distribution.

The GNU-GPL was clearly created to address a number of issues in the FLOS software community, namely a bunch of different and incompatible licenses. The GNU-GPL created a license that could be universally adopted, was not political in nature, didn't have restrictions against commercial use, and created a community of equals that contributers could join.

The thing is the GNU-GPL was created in 1986. And in 1986, a 1200 baud modem was probably a restricted phone line. Websites didn't exist, although science fiction books had started talking about something called “cyberspace” in 1982. And the “right to private derivatives” didn't show it's shortcomings until much later when the World Wide Web came into being, websites sprang into being, and people started using Linux to host those sites. Up until this point, there wasn't much possibility of someone having a private derivative that they did not distribute, but were still able to use it to compete against the community of equals that had created the original version.

By 2003, however, this loophole was commonly available. And it was in 2003 that the Apple Public Source License came out, with a source code requirement that wasn't triggered by Distribution of the Work.

By this time, though, the GNU-GPL had been around for almost 15 years, and had quite a bit of inertia behind it that would prevent changing the license. This inertia comes in part from the political process needed to change the license and to get buy in from all the contributers who had used the previous license. Not everyone chose GNU-GPL for the same reason. Not everyone chose GNU-GPL based on the same commitments. And rewriting the license to close a loophole that took advantage of one commitment might cause some flame wars. This can be seen with the anti-tivoization debate around GNU-GPL v3. So, it isn't too surprising that GNU-GPL never changed its source code requirement.

The proposed Open Hardware license demonstrates that the source code requirement can NOT be triggered by distribution of the work, because in hardware, the final product distributed is silicon, which may be outside of any definition of “derivative”.

The commitment here seems to be that downstream derivatives are allowed to be kept private, but only if they are not used, in any way that competes with the community that created it.

Public Domain style license: (1) Right to Privacy then (2) Community of Original Contributers.

Copyleft: (1) Community of Original Contributers then (1) Right to Privacy of Downstream Users

Limiting License to Functional Boundary or Maximum Definition

The second question of overlays is whether to limit the source code requirement such that it is triggered only when a modification is made inside some functional boundary, or whether it should be applied to the maximum allowable boundaries.

This is a question of how “close” Dave's code can get to Alice's copyleft code before Alice's license requires that Dave provide source files for his code.

The law is somewhat fuzzy on what level constitutes a legally recognized “derivative work”. Linking may or may not be a derivative, depending on what you're linking. If Alice releases some copyleft code that is a bunch of generic library functions, then Dave might be able to link Alice's code into his code and may not trigger a source code requirement, because it may not be a “derivative”.

Technically, a license can place restrictions such as “You are free to use this work anywhere but in South Africa” or “You are free to use this work anywhere but in military equipment” or “you are free to use this code anywhere but in life threatening situations”. So, it is conceivable, that a license could say something like “You are required to provide the source code of any application that runs on this operating system”, even if the application is not a copyright “derivative” of the copylefted operating system in anyway. It would then be up to a court to decide if such a license is enforcable.

So, the question is should the license try to maximize it's area of affect? Should it try to have as wide a range as possible to the works which the source code requirement applies? Or should it minimize the source code requirement as much as possible?

Well, completely minimizing would simply be eliminating it completely. The range would be “zero”. The original work would have source code provided because Alice provided it. But any derivative, no matter how close would be considered “far enough away” that no source code requirement would be triggered.

So that doesn't seem to hold much water.

The commitment behind the source code requirement seems to again boil down to preventing someone from creating a private derivative that competes against the community's version.

And in that light, the question is “Who is the competition?”

Worst case, a software company such as Microsoft might decide to roll in the code from a copylefted software project and use it to create an operating system that competes directly with the community's version. In this hypothetical case, if copyright law were to say that the “range” of what defines a copyright derivative was a functional boundary, then the copyleft community could, if it chose to do so, put hooks in place such that they could leverage Microsoft's code for their own purposes. Users would have to purchase a copy of Microsoft's operating system, but then they could somehow link in the FLOS software and it wouldn't be considered a derivative, so Microsoft couldn't prevent it.

The thing is current Copyright Law might consider such a hook to be a derivative, and therefore would give Microsoft the right to forbid that derivative.

So, it would seem that the “maximum” range for the source code trigger of what defines a derivative might be somewhere in the neighborhood of what someone like Microsoft can use as a definition of a derivative to prevent someone from hooking into their code. What's good for the goose, is good for the gander, and all that.

If Copyright law were changed tomorrow such that linking was never considered a derivative work, then a copyleft license would only need to trigger up to a functional boundary as defined by linking. As long as copyright law defines a derivative work to include things beyond linking, to include things beyond functional boundaries, then to protect the community from someone using that legal loophole to compete against the copyleft project, the maximum boundary of what triggers the source code requirement should be at least as wide as the legal definition of “derivative”.

With the proposed Open Hardware license, copyright law does not apply to the functional blocks. so a copyleft project could use functional boundaries as the limit of what triggers the source code requirement. For example, if Dave were to release a Digital Video Recorder (DVR) that used some copylefted hardware blocks, and then he added his own blocks, then if someone in the copyleft community purchased that hardware, they could use those blocks to do whatever they wanted with it. Copyright won't prevent someone from buying Dave's hardware and then rewiring it for other purposes.

Therefore an open hardware license doesn't need to be triggered beyond a functional boundary. Dave can add his own functional blocks and keep them private, but the original copyleft community can rewire it if they wanted. (Note this assumes an anti-tivoization clause applies to the whole platform so that any FPGA's or firmware or whatnot can be rewired or reprogrammed by the copyleft community)

The question is then what of hardware patents and can they be used to compete against a copylefted hardware community? That's a bit more difficult question. For the most part, with hardware, users have to purchase the hardware to use it. With software, users could conceivably download software for free if it is freely available and not restricted by a patent, etc.

With hardware containing patented logic, it would generally be logic that is built into custom hardware. Which means you have to purchase the hardware to use it. Maybe a DVR contains a patented method of recording video in a super-compressed way. Well, to get that hardware, to get that DVR with its patented logic, you'll have to purchase the DVR. There isn't much way around that as we don't yet have massive but generic FPGA systems in every household that people download different bitstreams to, to program the generic system to perform some specific function. The way hardware costs works, that may never happen.

So, if Alice creates some copylefted hardware design, and Pete uses Alice's code in his design and adds his own patented logic, then it is assumed that Alice will have to purchase the hardware from Dave to use it. The question is whether Alice, having paid Dave for his hardware, will be able to use his hardware and rewire it to her own purposes. Patents aren't really the source of potential competition. The source of possible competition is whether Dave throws in some sort of DRM or anti circumvention logic that in some way (legally or technically) prohibits Alice from rewiring the hardware she purchased from Dave.

So, with the commitment of preventing someone from using a copylefted work to create some derivative and compete directly against the original community that created the work, the question of the maximum boundary that should trigger a source code requirement seems to be answered with another question of “To what extent can outside people compete?” The answer to this depends on whether the project is software or hardware.

This would set the maximum range. The minimum range of course is “zero” which means no source code requirement at all.

An interesting scenario to test a “middle minimum”, as it were, would be to have the “open hardware license” written in such a way that it wasn't limited to just hardware, but could be applied to any copyrightable written work that had some functional boundary.

This might end up being quite similar in effect to what the GNU-Library GPL does.

Imperfect License Choices Still Produce Good Results

Another point to keep in mind is that even imperfect licenses produce good results. If one takes on the idea that the most important commitment of copyright is the original author and contributers over any downstream derivative being pulled into a private room, then the GNU-GPL is flawed in that it's source code requirement is triggered by distribution. This allows people to create derivatives and compete directly with the gift economy community that created the work in the first place.

And yet, the works licensed under GNU-GPL continue to thrive.

Not all copyleft licenses have anti-tivoization clauses, but those projects continue to thrive.

Some open hardware projects use GNU-GPL, and even though GNU-GPL has issues that don't fully protect those open hardware projects, they continue to receive contributions.

At the extreme end of the spectrum, some projects use a Public Domain style license, which would allow all manner of competition from downstream derivatives being taken private. And yet those projects thrive while using a Public Domain style license.

The thing is that while there may be a “perfect” license that emphasizes its commitment to the original developers over down downstream derivatives being taken private, imperfect licenses can still produce good results.

A license is an “enabler” for a gift economy. The license doesn't guarantee a gift economy will emerge around your project simply because you used one license over another. Nor does a license guarantee that no one will try to compete against you in the first place.

The thing to always remember is that FLOS projects are HUMAN projects. They involve people. And people will contribute to a good cause whether it's protected by a perfect license or not. It's quite possible that folks might see proprietary competition as an inspiration to make the FLOS version as good as any proprietary version. It's possible that flaws in a license that allow proprietary competition will be fixed by a community of contributers who refuse to be beaten out by that competition. If that's the case, then the only issue is whether patents or other monopolization laws would forbid the FLOS version to keep up with the proprietary version.

This is a good time to segue into the next section of this book, namely a look at the human aspects of a FLOS project.

The Human Aspects of a FLOS Project

Choosing a FLOS license is no guarantee you'll see an entire community blossom around your initial work. People will have to find something interesting about your project to freely give their time and energy to contribute to it. And “interesting” has widely different definitions from one person to the next. There will need to be an easy way for people to contribute to the project. The invention of “wiki” software was a major enabler for wikipedia and other text-based (as opposed to code-based) projects. And the level of “scalability” of your project might have an impact.

The Scalability of a FLOS project

Scalability is a measure of how well a project grows as you add more people and more resources. Some projects are naturally more scalable than others.

If your project were “Making a baby”, then it wouldn't be very scalable. You could add as many people as you wanted to the project, but you'll still have to wait nine months before you make a baby. This is because pregnancy is a “sequential” process. You can't do it in parallel by adding more people and reducing the total time. It's nine months no matter what.

Other projects are less sequential and more “parallelizable”, meaning you can split the work among many people, and reduce the total time to finish the project. Software can be split up into functional blocks, each block can be assigned to a different person, and under ideal circumstances, a project that would take a single person a million hours to complete can be finished by one-thousand people contributing one-thousand hours of work each.

For projects that are parallelizable, there is a second question of how much overhead each person requires. If a software project could be split up into functional blocks processing data in a fixed sequence, then one person might be assigned one block, and they would only have to talk to two other people, the person sending them data, and the person they're sending data to. The overhead would be relatively small. two people would have to agree on what data format they will pass between their blocks, and that's it. No one will care what any other block does in the project other than the two blocks they are directly connected to.

The overhead for such a project would be relatively low, which would mean you could theoretically add blocks and people without affecting the schedule or final quality.

Brooke's Law

On the other hand, some projects are of such a design that each block needs to know a lot about what's going on everywhere else in the project. Worst case, everyone must talk with everyone else. As you add more people, you add overhead that each person must deal with in communicating with everyone else on the project.

The number of communication channels for such a project is equal to

$$n(n-1)/2$$

If a project has 100 people in it, and each person must talk to everyone else on the project, then the total number of communication channels in that project is 100*99/2 = 4950.

10 people = 45 channels

20 people = 190 channels

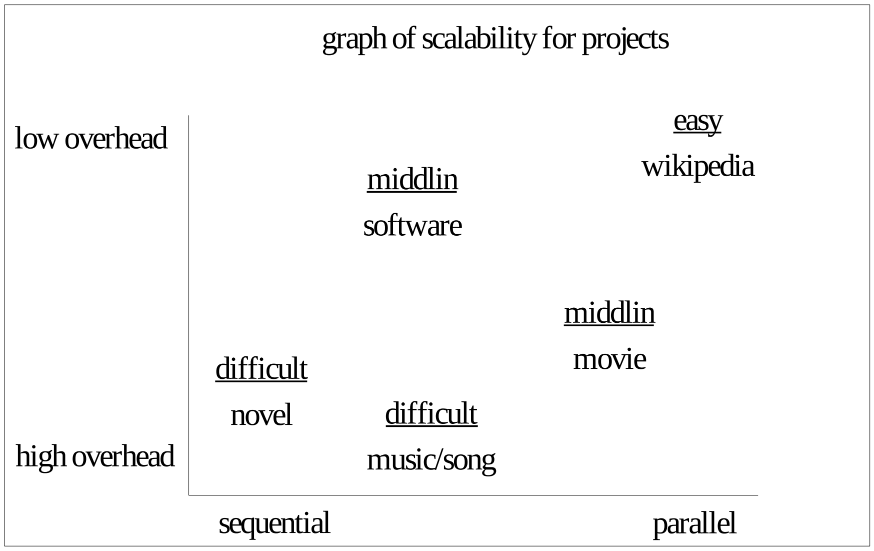
30 people = 435 channels

40 people = 780 channels

As you can see, parallelizing a project with a lot of communication overhead will eventually add so much overhead that it collapses under it's own weight. This lead to Brooke's Law which states “Adding people to a late project will make it later.”

Graphing Scalability

Some projects are mostly sequential, and some can be broken up and performed in parallel chunks. Some parallel chunks have little overhead with any other chunk in the project. And some chunks have a LOT of overhead with everything else in the project. If we graph sequential/parallel on one axis and "overhead" (communication, training, any overhead that's a function of the number of people on the project), then we can see some different projects and where they land on the graph. The closer to the upper right, the more scalable the project is. The closer to the lower left, the less scalable the project is.



Wikipedia is extremely scalable. People can work on one paragraph of one article and not know anything about any other article in the encyclopedia. It is parallelizable because each article can have a different person working on it, each paragraph even. And it has low overhead. For the most part, editors don't need to know what's going on anywhere else in the encyclopedia to edit an article. There may be some overhead as far as figuring out whether information is already duplicated in a related article and should it be put in another article as well, and so on. But for the most part, the natural scalability of an online encyclopedia such as wikipedia is extremely high. The more people you add to the project, the more potential to produce content, to produce more articles, to fill in existing articles, etc.

Software is somewhat scalable. It is parallelizable along functional boundaries. usually a person or a team of people are assigned to create subroutines. The overhead is based on how many people call the subroutine you're coding and how many subroutines are you calling that someone else coded.

A CGI movie (something that uses computer graphics or machinima or similar) is somewhat scalable. many scene can be worked on in parallel. Once the characters are in the computer and the script is ironed out, a scene can be assigned to a person or team and they can work on that scene fairly independently of any other scene. Overhead will be caused by issues of character personalities remaining consistent or developing in a believable manner, plot consistency, etc.

A fiction novel is extremely non-scalable. It usually takes one person to write the entire novel from start to finish to maintain a consistent voice. The number of different plots in the world are actually fairly small. Why people like a particular novel is more a function of the narrative voice, literary tics, and other matters of very subjective qualities, than it is a function of plot, story, character development, or other things that could be parallized.

Composing and recording a new song is fairly non-scalable as well. Like a novel, a song is more about it's “voice” than about it's literal content, it's lyrics, the story it tells, the notes it hits. Composing might be parallelized into different instruments and tracks, but everyone must stay in sync with everyone else.

So, when considering whether to turn your project into a FLOS project, consider how scalable it is. This isn't to say a hard-to-scale project can't succeed as a FLOS project, it's just saying to be aware of the task you're taking on.

For example, software is middling scalable, but Linux and its applications have been successfully deployed as FLOS projects. The thing to keep in mind is that it took many years to get to that point. In 1983, Richard Stallman started the GNU project with the goal of creating a Unix-compatible operating system and applications. It wasn't until 1991 that Linus Torvalds posted his Linux kernel on the web. In 1996, the KDE windowing system was started for Linux. In 2000, the source code for Open Office (applications including word processor, spreadsheet, etc) was released under the GNU-LGPL license. So, Stallman's original goal ended up taking 20 years to achieve. Patience can overcome hard to scale FLOS projects.

If you're project isn't easily scalable, but you're patient, you may still see your project succeed. It helps to have smaller, intermediate goals, that contributers can set their eyes on to feel some sense of accomplishment as partial progress is made. A project with a long term, hard-to-reach goal and no intermediate goals, may discourage people from contributing at all.

People generally want to contribute their time and energy to things that will make a difference. If your project doesn't have any intermediate achievable goals, then people may view their contributions as going into a black hole and never making any difference.

On the other hand, if your project is easily scalable, that isn't a guarantee of success either. Your project may appeal only to a small number of people as potential contributers, and the natural parallelism of your project won't matter if only a few people are interested in contributing. If your goal is too small or too specific, you may not find enough people to contribute to your project to see the effects of scalability. In which case, it may be better to find a way to somewhat generalize your project, and turn your original project into an intermediate goal.

i.e. if your project is to design a FLOS hardware fifo, that may be too specific to get people to contribute their time and energy. However, if you expand your project to have a final goal being a complete FLOS hardware processor, and your fifo is just one block, then that might appeal to a wider group of people, and get a wider group of contributers. Having a bigger goal can be a draw for people to contribute. The important thing is to have intermediate goals so that people don't feel like their spinning their wheels for a long time before they see some milestones accomplished.

Scalability in Theory, Scalability in Practice

Just because your project is highly scalable in theory doesn't mean you can't make it unscalable in your implementation.

Wikipedia, in theory, has an amazingly high scalability. It is an online encyclopedia that anyone with a computer and internet connection can access and can contribute to. It is extremely parallelizable since people can work on individual articles without having to know much about any other article in the encyclopedia. People simply contribute about topics they know. And there is little communication overhead to talk with other people working on other articles.

Wikipedia in practice, though, is becoming a bureaucratic nightmare that would curdle Kafka's milk. The amount of overhead required to deal with wikipedia is becoming a huge function of dealing with rules and regulations put in place by a bureaucracy who seems more interested in establishing their importance, their power, their control of articles, their various empires, than in enabling a highly scalable encyclopedia that anyone can edit without worrying about every administrator who might happen by and without knowing every nuance of administrative rules.

The bar to entry to edit wikipedia is being raised by cliques of administrators and bureaucrats and editors who want to keep the riff raff out and are more interested in maintaining and building their allies to exert power on whatever pet article they've decided to adopt as their personal property. The number of people I've run into who've told me they edited some wikipedia articles and then gave up just keeps increasing. About half of those people blamed administrators for abusing their power as their reason for leaving. The other half blamed the process for editing articles being based on “last edit wins” rewarding people for article-squatting and undoing any edit they didn't like. Or they blame a tangle of subjective and open-to-interpretation rules that are gamed by people who know the ins and outs of the rules to keep out people who don't want to spend all their time learning these arcane and subjective rules.

The rules of wikipedia have, in effect, increased the overhead that each person has to deal with to contribute. These same rules have also decreased the parallelism of wikipedia, making it so difficult to contribute that a person can't contribute to an article on Bananas without having to deal with someone who is squatting on that article, without having to deal with someone who is willing to game the rules to control the content of the Banana article, and without having to deal with an alliance of people who work in turn to avoid issues like the Three-Revert-Rule (3RR), and so on.

The solution would be to redesign the rules of wikipedia to encourage scalability, by reducing the communication overhead, and by encouraging more people to contribute. I proposed some rule changes in my article “Saving Wikipedia, Scaling Wikipedia” which are intended to leverage the massive scalability that is potentially available to a project like wikipedia. “Saving Wikipedia, Scaling Wikipedia” is aggregated at the end of this book for your convenience.

The point here is not wikipedia itself, but wikipedia as an example. Potential scalability is not the same as implemented scalability. How you implement your project can greatly affect overhead and can affect how parallel your process actually is. Just like a perfect license isn't required to produce results, a perfect set of rules isn't required to produce results either. Wikipedia might survive even if it doesn't change any of its process. However, if you're designing your project, the license you choose and the way you implement your project can affect the end result, the overall success of your project.

Summary

Hopefully, the Venn Diagrams and Labyrinths are a useful way of understanding, explaining, and comparing all the different FLOS licenses out there. The goal was to come up with a clear and concise way of explaining the fundamental differences between the basic varieties as well as explaining some of the more subtle differences between some of the copyleft licenses. And we were able to do so without relying on subjective terms like Free, Libre, or Open Source.

When discussing something like an anti-tivoization clause, one no longer has to revert to arguments about Freedom this versus Freedom that. One can simply describe the effect of tivoization as a room in a Venn Diagram. And whether the license has an anti-tivoization clause or not simply determines what sort of door connects that room with the rest of the diagram.

It's also fairly easy to see from the Venn Diagrams and Labyrinths that all licenses essentially boil down into three basic types: All Rights Reserved, Public Domain style, and some flavor of copyleft.

If you are interested in creating a community of contributers that will have the same equal rights as any other contributer, then a copyleft license of some sort is recommended. Such a license would ideally contain patent protection, anti-tivoization, and some form of source code requirement. These extra clauses attempt to remove any potential one-way, trap doors in the labyrinth, and should prevent any downstream derivative from being used to compete directly against the original creators.

If you're not concerned about competition, then you could use a simple Public Domain style license.

One observation from generating all the labyrinths is that restricting a copyleft license to only address issues of copyright law, while ignoring issues of patents, functionality, DRM, usage, and other structures outside copyright law, is an artificial restriction and could potentially allow non-copyright structures to be used to create one-way trap doors where a person could create a proprietary room from which they could compete against the community itself. If you choose a public domain license, then you can safely ignore these issues. If you're trying to protect the original community of contributers, then these concerns should be addressed.

The other important piece is that picking a license and slapping it on your work isn't a guarantee of success. An imperfect license may still yield good results if the contributers are willing to overcome them. However, choosing a bad license could kill an otherwise potentially successful project.

Licenses have nothing to do with the natural scalability of your project. Your project is either naturally easily scalable or not. (Or somewhere in between.) Choosing a FLOS license won't change this. The easiest project to scale in a FLOS environment is very parallelizable, very non-sequential, and very little overhead per person added. A difficult to scale project is very sequential in nature and requires a lot of overhead per person working on the project.

A good license and a naturally scalable project isn't a guarantee of success. You can't make people contribute to your project. And they're likely to not do that if your project doesn't inspire them to contribute their time and energy.

But an imperfect license and hard to scale project isn't a guarantee of failure either. Licenses protect against potential competition using legal means to remove one-way doors. This isn't a problem if people aren't interested in competing against you in the first place, or if they'd get bad publicity for competing against you using proprietary forks. And a hard to scale project can still succeed with enough patience.

Attachments

This article “Saving Wikipedia, Scaling Wikipedia” is aggregated with the “Libre Labyrinth” article for your convenience. The “Saving Wikipedia, Scaling Wikipedia” article is referenced in the “Libre Labyrinth” article in the section titled “Scalability in theory. Scalability in practice.”

“Saving Wikipedia, Scaling Wikipedia”

Wikipedia is in the process of slowly imploding. Vandals and POV pushers are messing with the articles. Admins have to clean up the mess. But the job of administration requires a lot of oversight, which wikipedia does not have. Which makes selecting admins difficult and monitoring admins a difficult job. The problem is that while the vandals multiply easily and rapidly, the number of administrators does not scale easily to keep up.

The way Wikipedia is currently designed, the position of Administrator requires a lot of overhead for each person who is made an admin. The election process is rather lengthy, and once elected maintaining oversight on each admin invokes Brook's Law, increasing the amount of work that wikipedia has to deal with. As wikipedia continues to add admins, it could potentially reach a point where the total amount of effort put into creating and maintaining every administrator position and dealing with abusive editors, POV pushers, and so on, will become greater than the effort put into creating good encyclopedia articles.

Some might argue that this point has already been reached.

The problem is that the position of administrator does not scale well as the number of editors increase. The solution would be to make the application of administrative duties scale easily as the number of editors increase.

The solution is to redesign administrative duties so that these powers require relatively little overhead to hand out to people, so that there's little cost in adding admins to keep up with the increase in editors. The current role of administrator has a lot of overhead in the election process and a lot of overhead in the oversight process. If the roles were redesigned such that monitoring these admins was made easier by making it easier to spot abuse, by limiting the amount of potential abuse that any individual admin could potentially inflict, and by discouraging those individuals who would tend to be drawn to the role of administrator due to the power it gives them, then the overhead per admin would greatly drop as it would be easier to create new admins and monitor them for abuse. The number of admins could then easily scale to keep up with new editors coming to wikipedia.

This became the topic of conversation over at Making Light. With the kicking around of some ideas, I think we came up with some workable solutions. This is a write up of one piece of it, dealing specifically with two new types of admins called “Janitors” and “Jurists” and a ticketing system for Jurists. There are some more suggestions on the Making Light thread regarding editor subtypes and some other ideas, but this document just got way too long to include everything.

First, there would be a new type of administrator called a “Janitor”. A Janitor can block users for two reasons: obvious vandalism and violations of a strict interpretation of the 3RR rule. Why these two violations? Because they are easy for people to quickly spot a violation, and they are easy to spot when a Janitor misuses their privileges.

Currently, the violations that Janitors would take care of are vandalism and 3RR. This could possibly expanded to other violations, but it is very important that the job of Janitor be limited to rules that are obvious to see and obvious to detect misuse. This will allow scaling, which means a lot of Janitors can be easily added, and oversight will be minimal. Abuse will be obvious and the Janitor would have their privileges revoked.

Becoming a janitor should be relatively easy. It might not even need a nomination process or a vote for consensus. Which is also good, since these take a lot of time, and create an opportunity for quid pro quo among users. Instead, anyone with some minimum number of edits could become a janitor by request. Removing a janitor’s privileges should be equally easy. An obvious misuse of privileges is an automatic and permanent revocation of privileges.

With an easily expandable group of Janitors dealing with Vandals and 3RR violations, the task left for the remaining Admins is one of dealing with POV pushing, violations of NPA policy, and other issues which generally require more of a subjective interpretation of events to see if a violation occurred.

This is more difficult to scale, because an admin can abuse their blocking privileges by applying a creative interpretation of what is a NPA violation, what is POV pushing, etc. Which means an abusive admin is harder to detect. The current solution is to fix this with a large up front cost of selecting admins through a nomination process, with the assumption that once nominated, the admin will operate without abusing their power.

However, this ignores the age old adage that power corrupts, and absolute power corrupts absolutely. While an admin may have the best of intentions at the time they are running for election, the current process for elections, with all of its massive overhead required to sort out good candidates, does absolutely nothing to deal with an admin who, with all manner of good intentions, begins to abuse their position and power after election. This situation is dealt with via the equally massive overhead process of taking an issue to the Arbitration Committee. This is why wikipedia doesn't scale easily.

What is needed is a way to redesign the position of Admin such that the possible ways of abuse are minimized, so that oversight isn't needed. The arbitration committee can still deal with abusive admins, but by redesigning the role of admins, it would be possible to reduce the potential for abuse in the first place, making it easier to add admins without worrying about abuse, and without adding massive oversight and overhead per admin.

The most immediate abuse is an admin who is editing an article and then uses admin privileges to block an editor who disagrees with their point of view. Wikipedia attempted to fix this by declaring that administrators cannot use their privileges on articles they are editing. An abusive admin easily circumvents this with allies who are admins. The admins work on their articles they are interested, and when a dispute arises with a regular non-admin editor, the admin calls in their ally to block the editor using some creative interpretation of policy to find an excuse for a block. The admin repays this favor by going into the other admin’s article upon request and returning the favor.

Note it's very important to distinguish the problem of “abuse” from the issue of “breaking the rules”. Currently, “abuse” isn't the same as “breaking the rules” until after a long, lengthy, tiring process before the arbitration committee. The idea here is to reduce to potential for “abuse” so that the overhead of per admin is reduced, and the job of admin can scale to keep up with editors.

Wikipedia's current system of rules easily and almost naturally encourages and creates an environment of quid pro quo among abusive admins. Even the process of nomination generally requires the support of existing admins. And that support may be the first “scratch” that the new admin feels indebted to repay.

This whole system of quid pro quo among the abusive admins is diffuse and subjective. It is diffuse because the guilty parties are spread out among several admins working as allies, and working slowly over time. And it is subjective because an abusive admin can use a subjective interpretation of events to declare some innocent behavior to be a violation and hand out a block. This diffusiveness and subjectiveness makes it extremely difficult to point to any one specific incident and say it was admin abuse. This in turn makes it extremely difficult to catch admin abuse. This difficulty makes any case before the Arbitration Committee extremely long and exhaustive. Which, in the end, makes it nearly impossible to scale the number of admins to meet the needs of wikipedia.

What wikipedia needs to be able to easily scale the number of admins is a way to short circuit the quid pro quo and the reward that comes from creative interpretation of rules. Requests for Deadminship are a lot of work, take a lot of time and energy, and simply do not scale to the number of admins needed.

The solution to this is another new type of admin called a “Jurist”. The title “Jurist” reflects the fact that the job of this particular admin is to judge the evidence, to bring in the subjective interpretation. A Jurist can block users for any rules violation, including NPOV and NPA, as well as vandalism and 3RR.

The difference, however, is that a Jurist cannot edit any content articles while they are a jurist. People who become Jurists must give up the ability to edit the content of any mainspace article. Jurists may edit talk pages, user pages, wikipedia policy pages, but not main content.

This breaks the first payoff that an abusive admin might use their privileges for. Editing an article and blocking anyone who gets in their way. While wikipedia’s rule that no admin may use their privileges on a page they are editing, the response of an abusive admin is simply to build a group of allies and develop a system of quid pro quo so that an admin edits one page, and their friend hands out any blocks, and the admin returns the favor on the friend’s page.

By disallowing the editing of main page articles, the set of rules make it much more difficult to create a system of allies and a system of quid pro quo with other jurists.

However, this still allows a Jurist to potentially misuse their privileges in two ways. First the jurist might develop ally editors. The editor works on the article sympathetic to the POV of the Jurist, and the Jurist hands out blocks to anyone who opposes that editor. Second the editor might develop a list of admins known to be friendly to his POV, and specifically ask for their help when dealing with the application of subjective rules.

To alleviate the natural tendency for allies to build between editors and Jurists, the system of Jurists will be handled with a Ticket Request system, or a docket system. A “request jurist” page will be created, and editors who witness another editor violating the rules will submit a request to the system.

When a Jurist has some time to work on a ticket, they will go to the Ticket Request system and ask for a case. The ticket request system will randomly assign a ticket to the Jurist. The Jurist can accept or decline the ticket. If they decline, that information is recorded publicly, and the Jurist is presented with another ticket. If the Jurist accepts the ticket, they then read the information given, render a judgment, and hand out a block.

This means that a Jurist cannot go into a page they are interested and hand out blocks to editors who oppose their POV. This also means that an editor cannot build a list of friendly Jurists and start a habit of making sure all requests for help aren’t first directed to the friendly admin via email so that admin will be on the case. The ticketing system uses random assignments to prevent a jurist from selecting a page they have a particular personal POV about.

It might help to build into the system a record of all the users that a Jurist has to interact with on a ticket. And then any further requests that Jurist deals with should avoid those users. This should prevent any alliances from building.

So, Jurists have full admin privileges for blocking users for any policy violation. However, they are prohibited from editing article pages, and they can only operate on tickets handed out to them randomly via the ticket request system. This means that many of the incentives that encourage admin abuse have been removed from the system.

It also means that becoming a Jurist requires the user give something up.

Currently, becoming an Administrator grants the user many privileges without requiring the editor give anything up. This makes the job of administrator a coveted spot specifically for those who are inclined to abuse the position. While wikipedia states that being an admin is "no big deal", the privileges handed out to admins with no privileges removed from the admin, mean that becoming an admin can only be seen as an increase in power. Whether that power is a "big deal" is a matter of each individual administrator, and oversight on that involves a lot of overhead.

By making Jurists give up the right to edit main page articles, those most likely to abuse an admin position are less inclined to apply for a Jurist position. Abuse is still possible, but it is limited to randomly selected incidents assigned to the Jurist. While a Jurist may have an intention to POV push on some political topic, the ticketing system would guarantee that they would only act as a Jurist on any particular page only once, making it hard for a Jurist to push a POV on a specific page.

All told, the job of Jurist then becomes something that can scale more easily than the job of Admin. Since most of the biggest incentives for potential abuse are removed, becoming a Jurist should attract people interested in contributing to a job that needs to be done on wikipedia, not people who want the power to assert their position on articles. This should make the nomination process much easier.

Being disallowed from editing main page articles also means that being a Jurist is more likely to be a temporary gig. Currently, being an admin on wikipedia gives an editor all the perks of being an editor with all the powers and privileges of being an admin. There is no incentive for admins to step down over time, and this can encourage extremely diffuse systems of quid pro quo, and admins extremely good at gaming the system to push their pov, as admins sit in power for several years.

If nothing else, the number of editors who become administrators, compared to the number of administrators who give up their admin powers and become regular editors again, should be sufficient evidence that being an admin is, in fact, a "big deal". If it wasn't a big deal, then giving up the job of admin would be as common as taking on the job of admin.

A Jurist position might be more like a six month, nine month, or possibly one year gig. Editors give up the privilege to edit main page articles to volunteer for a job that wikipedia needs done. The Jurist serves their job for a while. And then the editor moves on and returns to editor status.

Wikipedia has a slogan that being an admin should be “No big deal”. By redesigning the system into Janitors and Jurists, this actually becomes a true statement.

The whole point of this idea of splitting the job of Wikipedia Aministrator into Janitors and Jurists is to transform the job into something that can easily scale to keep up with the vandals and revert warriors and the POV pushers.

The job of Janitor is relatively easy for anyone to do, and relatively easy and quick to spot abuse, which both discourages abuse and makes spotting a bad Janitor and removing their privileges quick and easy too.

The job of Jurist is more complicated, but by removing the biggest incentives that would attract an abusive editor to want to become an admin, the Jurist system makes the abuse of admin privileges to push POV extremely difficult. Even if a Jurist wanted to abuse their position, the randomness of the ticket system limits the damage.

The intent is that by creating these two new positions, wikipedia can be saved from the onslaught of vandals and POV pushers without collapsing under the weight of editors looking to abuse admin privileges to push their own POV.

By scaling wikipedia, we can save wikipedia. By changing the job of admin to that of Janitors and Jurists, the role of policing wikipedia can be easily scaled, with little overhead, to keep up with ever growing onslaught of trolls and vandals.