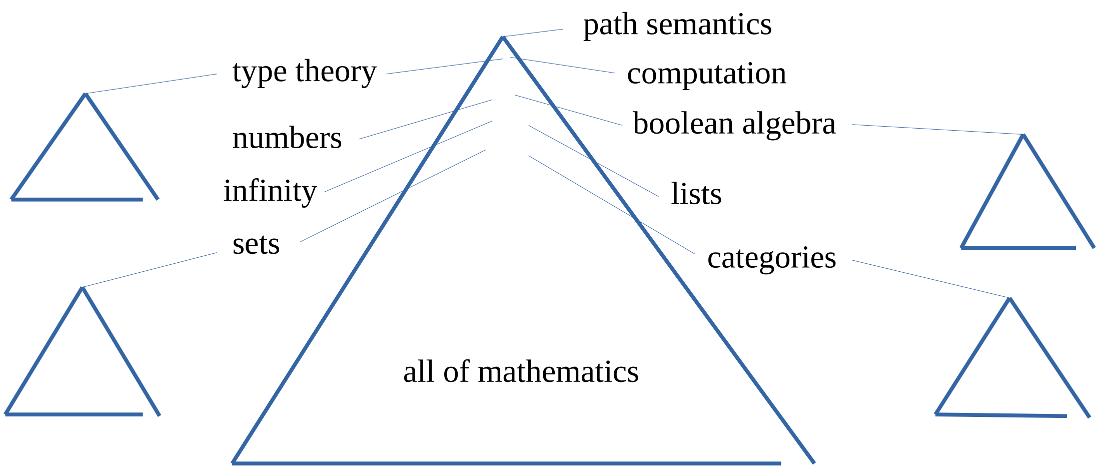
Path Semantics

by Sven Nilsen, 2016-2023

Abstract:
Path semantics is used to study foundations of mathematical and programming languages. It is useful because it defines an informal, yet precise semantics enough to "bootstrap" into existing mathematical languages, without requiring a specific language implementation.

Formal languages[1] that can express sentences conforming to path semantics[2] give a formal way of checking for meaningful statements.



This illustration is meant to point out that there are multiple theories of mathematics, that have different strengths and weaknesses. Path semantics is just one way of looking at the world of math.

The foundation of mathematics has been traditionally associated with First Order Logic, but in the past century, Type Theory has emerged as an alternative. However, Type Theory is relatively complex compared to First Order Logic. It is desirable to keep the foundation of mathematics as simple as possible.

Another desirable property of the foundation is to be able to reason about data structures. Data structures are not predicates and can hide information, which is problematic in First Order Logic, and substitution by propositional equality is not always sound.

For example, if a data structure models randomness depending on some type, it is only sound to substitute under tautological equality, not mere propositional equality.

Path Semantics solves these problems, by lowering the foundation to propositional logic and data structures. Instead of inventing a more complex language, like First Order Logic or Type Theory with dependent types, Path Semantics uses a smaller and simpler language that is more trusted. It might seem at first that Path Semantics adds a lot of new complexity, but all this complexity is reduced to a simpler language. The trick is how to make this possible.

The perspective that Path Semantics takes on mathematics, is high dimensional. This means, from the rules alone it is not easy to tell how it works in practice, because applied mathematics is a low dimensional perspective. The rules that work across many dimensions are harder to think about.

For example, when reasoning about function inverses, one can define a data structure `inv` which takes a type as argument. So, if `f` is a function, then `inv(f)` represents the inverse of `f`.

Now, from an applied perspective, it is easy to think of the inverse of a function as a concrete function. However, the rules of theorem proving with the inverse often holds, even though there is no concrete function that corresponds to the inverse. This means, the foundation needs to handle the subtle difference between rules, that work in the absence of a solution of the inverse, and rules that require the solution.

For example:

    f(a) == b

Can not prove:

    inv(f)(b) == a

Without a proof that `inv(f)` has a solution.

In Path Semantics, this solved by using a “qubit” operator `~`:

    ~inv(f)   =>   inv(f)(b) == a

The `~` operator is defined using a data structure. It can also be defined in classical logic using randomness. Here, `~inv(f)` means that `inv(f)` has some solution, so it is safe to use the right side of the implication. The whole axiom looks like this:

    (inv(f)(b) == a) ∧ ~inv(f)   =>   inv(f)(b) == a

This axiom is not part of the foundation, because there are many ways to use Path Semantics. There are many perspectives of mathematics and Path Semantics does not tell you which is the correct one. However, the idea is to provide a foundation which can be built upon to explore mathematics.

Now, you might question: Why provide a foundation if it is not complete?

The answer is that mathematics can never be complete. It is only possible to provide a foundation that helps people to reach longer, but there is no foundation that helps people to reach all mathematical knowledge. Yet, the foundation should also help people question their assumptions, by not providing single answers when there are more than one answer.

The approach Path Semantics uses, is that through propositional logic and data structures, one can construct logical languages that are interesting on their own, yet also fit together like building blocks. Instead of adding new features to the language, like predicates, that might have undesirable hidden assumptions, one can use existing programming languages without modification. By following the instructions of Path Semantics, people can build their own extended mathematical foundations in the programming language of choice. This works when the language is:

- Statically typed
- Supports generics

With other words, many mainstream programming languages, like Rust, can be used to implement Path Semantics. This approach makes it possible for people to learn about foundations by scratch.

When starting to build a mathematical foundation, the first question one should ask is:

What is a symbol?

The answer to this question is not trivial.

In Path Semantics, one answers with 3 letters: PSI.

The 3 letters stands for **P**ath **S**emantical **I**ntuitionistic Propositional Logic. Intuitionistic Propositional Logic (IPL) is the basis for PSI and this corresponds to static types in programming:

PSI = IPL + `~` + `<` + core axiom

This means, one starts with a statically typed language, with support for generics, and program in it the path semantical qubit operator `~` and the path semantical order operator `<` using data structures. At the end, one needs to introduce the core axiom.

The core axiom is the most central axiom in Path Semantics, hence the name. It is very, very hard to understand all the consequences that follow from the core axiom (there are literally infinite new theorems) and most of these consequences have no practical applications in mathematics. However, this is Path Semantics' non-trivial answer to the question "What is a symbol?". Path Semantics is kind of like a genie (or, jinni) that grants wishes, whether this is a good idea or not. Furthermore, nobody has proved that PSI is the correct answer. The only thing I know is that, when you combine PSI with other logical languages from Path Semantics, it seems to give you the ability to do mathematics.

To many people, a "just do it yourself and see what happens" approach to the core axiom is not satisfactory. They would like to know what it means. It seems as if, an alien with unimaginable knowledge came to Earth and just wrote down the answer and it works, while people struggle to figure out why. This is how Path Semantics is like. The reason is that the work required to understand why, is immensely huge. The probability is basically zero that you arrive at this solution by chance (ironically, it kind of was discovered by chance, although not in correct form). It took several years, just to train my brain thinking about it, before I could build logical languages that pointed me in the direction of a version that worked. This is why I offer this approach: So far, it is the best approach to it that I can give. Someday, people might figure out what it means and when they do, please tell me! I also would like to know what it means.

However, I have an idea of what it *might* mean: The core axiom provides a way to propagate knowledge along path semantical levels of propositions. In physics, one can think about path semantical levels as moments in time. For each moment, there is a set of theorems one can prove. The core axiom makes it possible to change one moment based on the previous one, without polluting the set of theorems that are provable within the previous moment. In Type Theory, these "moments" are like places where cumulative type hierarchies live.

PSI is basically a logical language needed to make it possible to say "A is of type B", or `A : B`.

Wait a minute! We already have a statically typed language! Why do we need PSI?

The reason we need PSI is because we need to simulate aspects of dependent types. PSI bootstraps statically types, as they are built into most mainstream languages, into a logical framework that is better equipped to reason about mathematics. So, now you have types inside types! Hooray!

The core axiom does not handle all the information needed for types. There are 2 parts: One part that is handled by the `<` operator (path semantical order) and one part handled by the core axiom (path semantical quality). Together they correspond to an operation on types if and only if one uses the following representation:

$$(a : b) == ((a => b) \land (a < b))$$

The reason is that the answer to "What is a symbol?" is not merely "Types!". Path Semantics as a foundation needs to handle more cases, most of which are still unknown to mankind. The idea of using PSI to model types, is just one perspective out of many.

The core axiom in PSI models how symbols are used, not just types. This is a very important insight. If one studies PSI in isolation, one get hints at how to build other building blocks. This is where people often loose track of what Path Semantics means. It is like, PSI is a vast landscape that can be explored and which contains hidden treasures. These treasures does not give you the full answer, only a clue. It is like finding puzzles and by solving these puzzles, you get your answer.

Now, it is important to remember that simple rules in mathematics can give rise to enormous complexity. In the game of Chess, there are more possible games than atoms in the observable universe. However, Chess is just a speck of sand in the desert of PSI. Just to understand how complex PSI is: We can never even compute the basic binary operators one by one. The amount of colors in RGBA (Red, Green, Blue and Alpha channels), has each a binary operator in PSI of homotopy level 2. In level 18, you need a 4TB hard drive just to write down the number of binary operators in decimal form. There is so much to explore about PSI that we will never in practice complete it. This vast desert of mathematics is inexhaustible, even it is just one building block.

Yet, Path Semantics itself, which is much larger than just PSI, does not define all of mathematics. PSI describes how symbols are used, but it is not the only mean to learn how symbols are used. By interpreting how existing tools and techniques in mathematics use symbols, one can develop a deep intuition of mathematics.

Path Semantics is not a set of doctrines. It is a framework which provides tools to look at mathematics from certain perspectives. If you want to explore mathematics by other means, then there is nothing wrong in doing that. Either way, you will learn something.

However, Path Semantics also have perspectives about what kind of perspectives are out there. It is a theory that "looks outwards", as if with curiosity, because you will not find complete answers here. You will find puzzles, which needs to be solved, before you get satisfied. It is more like an abstract landscape to find puzzles and there are some tools helping to solve them:

- Puzzles are discovered
- Solutions are designed

When people say "all of mathematics" they usually imply something as if mathematics exists prior to the discovery of it. "All of mathematics" is not meaningful to talk about until all of mathematics is defined, which is impossible. Path Semantics is used to "focus on" parts of what we mean when we use mathematics. This focus is always there, secretly hidden, so there is no fixed universal solution.

From Path Semantics, you can construct any formal language in any way you want, as long there is an interpretation of symbols that does not violate the core axiom.

There is a lot of evidence that path semantics is powerful. Yet it might be difficult to convince some people that meaning of symbols can be understood through path semantics. This confusion results usually from not understanding the difference between formal and informal languages used for theorem proving[3]. The distinction between "formal" and "informal" does not necessarily mean that an informal language is less rigorous than a formal one.

	Rigorous	**Non-rigorous**
Formal	Logic, e.g. PSI	-
Informal	Path semantics	English

PSI is a formal language, yet Path Semantics as a whole is not. Path Semantics is not possible to formalize fully, because as more building blocks are invented, the more stuff is discovered that needs to be formalized. This process never ends.

One common argument that people use, is that Set Theory models most of mathematics. However, this is not accurate. Set Theory does not model, e.g. randomness. In foundations of physics, the deterministic nature of Set Theory only holds in a perspective of a multiverse, which is not accessible in practice. So, it is more accurate to say that Set Theory models most of mathematics, *but from a certain perspective*.

Once you add stuff like "from a certain perspective" in mathematics, it becomes so complex that it gets ridiculous to believe that all of mathematics can be understood to any significant degree. It is like, when observing the Earth is flat from a certain perspective, this does not imply that you can understand all that happens on Earth as globe. Only in some languages, like First Order Logic, that models relations as a perfect information environment, one can pretend to know everything. This means that First Order Logic is biased, because it *pretends* to know, when in practice we can not know everything. Path Semantics is an approach were one admits to not know everything and where it is impossible to know everything in practice. The "not knowing" is built into the framework itself.

Therefore, Path Semantics can not provide answers to many deep questions. This is because when we attempt to answer them, we make design choices. These design choices introduces bias that shapes how we think. To understand how this bias works, one needs experience and for now there is no other way than to explore on your own, or together with other people.

However, the very least Path Semantics can do, is to point out some flaws in other languages, such as First Order Logic. This is done by constructing alternative langues, like Avatar Logic. Once you compare First Order Logic and Avatar Logic, you will notice the tradeoffs in design. There is no "correct" answer, and Path Semantics does not tell you these answers when there are only puzzles.

To the extent that we have languages that give us answers to problems, we have already introduced so much bias into the language design that it is meaningless to use the language as a doctrine for reasoning about all of mathematics. Path Semantics is not a naive framework of mathematics. On the other hand, it handles nuances to a much higher degree than e.g. Set Theory can express. To claim to not know everything, does not make one less mature. It is better than lying about it. The irony is that Path Semantics is often used to design theories where some object "lies" to the other objects. However, as a framework, there is completely honesty about the lying going on, formally.

All formal languages are rigorous, but not all rigorous languages are formal. The rigor that underlies a rigorous informal language is often not shown directly, but assumed from the context that the language is used for informal reasoning. For example, to keep a paper somewhat comprehensible and readable, it is common to leave out steps that are convincingly easy to prove, or possible to prove under various assumptions that are pointed out, or it might be assumed that a proof exists as a part of the argument. This practice is used in mathematics in general.

The practice of mathematics is so similar to path semantics that the only difference is a stricter usage of identity of functions, plus that Path Semantics develops new syntax for expressing ideas. This is a good thing, because it justifies the way informal theorem proving is done in practical mathematics and programming. One might learn Path Semantics to improve informal theorem proving skills.

Since Path Semantics is informal, it does not claim that everything in a paper can be proven in e.g. Set Theory. It has less guarantees: You might not even be able to prove something in practice, perhaps it is too vague, maybe it is not well understood, too complex etc. However, the proof itself might be crystal clear of what it means: Vagueness can be pragmatically powerful to express ideas. Path Semantics tries to explain how this can happen without making it too complex or too vague.

The benefit of Path Semantics is that it can describe what we mean when e.g. using equations to describe something else. A common combination is to use Path Semantics to express new ideas while working on methods to formally reason about them in various ways.

- Use path semantics to express new ideas
- Use formal languages to model parts of a new idea formally (or the whole if possible)

Path Semantics relies on a technique called “bootstrapping”:

1. Define an axiom of quality that holds for all Path Semantics
2. Construct a reflective language in Path Semantics
3. Describe formal languages in the reflective language

The purpose of constructing a reflective language is to get better tools for expressing ideas.
Even if this reflective language might be hard for formalize, it is possible to check various specific proofs against the core axiom of quality. It is also a super-set of existing languages that are possible to formalize, e.g. dependent typed languages. This means the only place where intuition is needed, is where an existing formal language is not powerful engouh to express some idea.

In this paper, I will use the core axiom to construct a language, in an informal way, just simple enough to describe logical gates (AND, OR, NOT etc.).

Step 1: Axiom of quality (core axiom)

The interpretation of formal languages is about an arrangement of symbols, where a collection of symbols `F` are associated with another collection of symbols `X`, written `F => X`. `F < X` expresses a non-circular definition (2021 standard order[4]). For all such collections of symbols, a quality expressed as `$F_0$ ~~ $F_1$` is uniquely associated with a quality `$X_0$ ~~ $X_1$`:

$$\frac{(F_0 \sim\sim F_1) \land (F_0 => X_0) \land (F_1 => X_1) \land (F_0 < X_0) \land (F_1 < X_1)}{X_0 \sim\sim X_1}$$

This is the axiom of quality, that holds for Path Semantics.

The word "quality" refers to "Path Semantical Quality"[5] (see reference for more information). Quality is a partial equivalence relation that lifts biconditionals with symbolic distinction. This makes it possible to express structure between symbols in language, which is not provable from an empty context by definition. The motivation is that mathematics is a language where one always puts something in, whether it is data or assumptions in the form of code.

The axiom is a modification of Leibniz' law[6] where equality of arguments to predicates is replaced by quality between data structures as propositions. Predicates can be modelled using a data structure for function application and an axiom for substituion in arguments of the application using normal equality of propositions. The core axiom does not talk about function application directly. Instead, it provides a mechanism in which new axioms can use stronger assumptions for correct behaviour.

In informal theorem proving, one can use other formal languages, but in order to determine whether the reasoning works when translating between languages, Path Semantics expresses predictions that corresponds to particular equations. The semantics of these predictions are "bootstrapped" from the core axiom without relying on the semantics of equations.

The corner stone of Path Semantics is to define how one talks about other things that might be too complex to formalize. Existing tools, syntax and techniques from standard mathematics is then used to do the theorem proving, while preserving meaning. Path Semantics' power is that it interprets into the language that is suitable for a particular domain of mathematics. This means that people learning Path Semantics can use existing tools without needing to translate into another language.

Inquality

It follows from the axiom of quality that there is a similar theorem of inquality:

$$\frac{\neg(X_0 \sim\sim X_1) \land (F_0 => X_0) \land (F_1 => X_1) \land (F_0 < X_0) \land (F_1 < X_1)}{\neg(F_0 \sim\sim F_1)}$$

Inspiration of the core axiom

I used normal paths[7] to do some theorem proving that I found useful, but it was difficult to use them, with the same ease as informal theorem proving, in Coq[8] and Idris[9]. To justify the informal technique I used, I wanted to learn more about foundation of mathematics. Vladimir Voedvodsky[10] held some recorded lectures about the foundations of mathematics, which I watched. The way he used paths from Homotopy Theory to talk about mathematics was inspiring, so I named my idea of extracting hidden functions from within functions after this word: Path. After a while, more and more "paths" were formalized, so I started thinking about them as ways of associating collections of symbols with other collections of symbols. One day I realized that most of mathematics depends on definitions. If I could find an informal axiom that explained how symbols are used, I could use arbitrary definitions to construct more mathematics. To me it was important to express mathematical ideas concisely in the language of functions, so I did not need a formal axiom. I made it part formal and part vague, formal enough so I could “bootstrap” into functional programming.

As time went on, there were several stages of improvements, but overall the philosophy of the core axiom has been preserved. It provides a background for theorem proving, but does not complete the steps required to fully reason about e.g. functional programming. Instead, one considers multiple logical languages as building blocks for extended foundations, where functional programming is just one possible direction out of many.

A normal path is basically an open box in Cubical Type Theory of 2 dimensions, where the solution of the normal path is the missing edge (filled in with Kan filling operations). However, the syntax of normal paths makes it possible to express it in a form where it can be substituted with solutions. The normal path syntax generalises to functions of arbitrary number of arguments, such that a symmetric normal path can be used when there is symmetry in orthogonal edges.

f[g] <=> h	Symmetric normal path
$f[g_0 \times g_1 \rightarrow g_2]$ <=> h	Asymmetric normal path

The use of normal paths for theorem proving was the motivation for developing Path Semantics. First, I tried to model normal paths in dependent types. One problem was that one could not simply compose normal paths:

f[g][h] <=> f[h · g]

I had help from an expert in Lean to get as close as possible, however it was not fully formalized. However, the syntax worked and one could understand normal paths formally when a solution exists. I realized that I needed an imaginary inverse operator `inv` to model total normal paths:

f[g] <=> (g · f · inv(g))

Without this form of composition fully formalized, one requires adding a proof, which is not necessary, that results in friction in the theorem proving process. The imaginary inverse makes theorem proving work painless, but uses syntax for normal function composition. This syntax is less readable for humans when functions have multiple arguments.

Since I was not able to formalize normal paths fully, I looked for ways to develop a logical foundation that could be used to check inference rules used in eventually new programming languages supporting normal paths. Dependent types generalises First Order Logic, so when dependent types was insufficient, it meant that First Order Logic was also insufficient and there was no existing mathematical foundation to formalize normal paths.

There are many inference rules required to support a substantial subset of Path Semantics. Therefore, I looked for an approach where I could prove as much as possible using as few axioms as possible. I thought “What if I provide a mechanism for connecting two symbols together?”. The first inference rule I wrote down was the first draft of the core axiom.

Considering that I had very little knowledge of logic at the time, it must have been one of the luckiest guesses in the history of mathematics, like hitting bulls-eye at the bar from the moon.

Functional completeness

Bits are the simplest collection of symbols that satisfy the quality axiom. Consider the axiom as a computer circuit to prove soundness of free variables of bits when:

$$\frac{F_0 < X_0}{F_0 = 0, X_0 = 1}$$

This corresponds to a binary logical gate equivalent to NOR:

F_1	**X_1**	**F_0=F_1**	**X_0=X_1**	**Meaningful**	**NOR**
0	0	1	0	0	0
0	1	1	1	1	1
1	0	0	0	1	1
1	1	0	1	1	1

By changing the `<` into `>`, this corresponds to a binary logical gate equivalent to NAND:

F_1	**X_1**	**F_0=F_1**	**X_0=X_1**	**Meaningful**	**NAND**
0	0	0	1	1	1
0	1	0	0	1	1
1	0	1	1	1	1
1	1	1	0	0	0

Although the axiom of quality is simple, it contains enough complexity to model NOR or NAND. It is therefore functional complete (any computation can be done by connecting NOR or NAND gates together). In the tables above, `~~` (quality) is lowered to `==` (equality). It means this way of proving function completeness is not fully formal, but contains some amount of vagueness.

Historic Revisions

The deep symmetry of functional completeness, together with experience from constructing languages from the axiom, was used to refine the axiom of quality from the first form

$$\frac{F_0(X_0), F_1(X_1), F_0 = F_1}{X_0 = X_1}$$

to the next form that gives symbols in formal languages a more constructive character (2016):

$$\frac{F_0(X_0), F_1(X_1), F_0 > X_{0,} F_0 = F_1}{X_0 = X_1}$$

However, more research and getting rid of ambiguous syntax resulted in the final form (2021):

$$\frac{(F_0 \sim\sim F_1) \wedge (F_0 => X_0) \wedge (F_1 => X_1) \wedge (F_0 < X_0) \wedge (F_1 < X_1)}{X_0 \sim\sim X_1}$$

Step 2: Construct a reflective language in Path Semantics

Let us start informally with a more complex collection of symbols: Words.

In this language we associate a logical gate \`not\` with an input type \`bool\`:

not(bool)

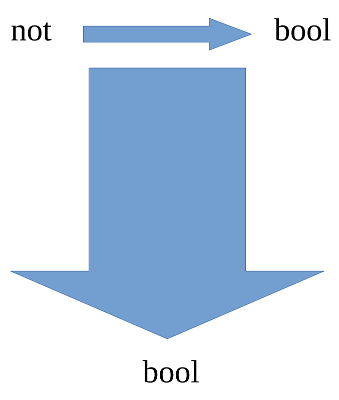
This choice is arbitrary and done simply because we know how \`not\` gates behave. It makes sense to associate the symbols this way because it follows the axiom of quality.

If somebody says \`a ~~ not\`, then we know \`a\` also takes the input type \`bool\`.

To describe the output type, we associate \`not(bool)\` with \`bool\`:

not(bool) → bool

This is just creating a collection of symbols out of the two associated symbols, and then associate this new collection with something else.



This operation can be done in two ways, either by creating \`not(bool)\` or \`bool → bool\` first. Some languages write this as:

not : bool → bool

Then we do a simple, but powerful trick: Associate each value with a type, and return itself.

true(bool) → true
false(bool) → false

This trick might not be taken literally, but rather to use the analogue syntax for sub-types:

x : [true] true <=> x ~~ bool

Or simply:

x : [:] true <=> x ~~ bool

This syntax makes it possible to “lift” variables into types through the language of sub-types.

Now, describe `not` by associating lifting variables into sub-types:

```
not([:] true) → [:] false
not([:] false) → [:] true
```

Testing that `not(not(X)) ~~ X`:

```
not(not([:] true)) ~~ not([:] false) ~~ [:] true
not(not([:] false)) ~~ not([:] true) ~~ [:] false
```

The `[:]` symbol is symmetric, so we can refactor it:

```
not [:] (true) → false
not [:] (false) → true
```

It gives us the ability to compute with values like in a normal programming language.

Introducing a wildcard notation and shorthand version for cases:

```
and(bool, bool) → bool
[:] (true, true) → true
[:] (_, _) → false
```

This is sufficient for deriving Boolean algebra in a short and nice syntax.

Step 3: Describe formal languages in the reflective language

Deriving a formal language is one thing, but how can we describe them reflectively?

It turns out that the same mechanism that associates values with their type can be used more than once. For example, the `not` gate can be used like this:

```
and([not] [:] false, [not] [:] false) → [not] [:] false
and([not] [:] false, [not] [:] true) → [not] [:] true
and([not] [:] true, [not] [:] false) → [not] [:] true
and([not] [:] true, [not] [:] true) → [not] [:] true
```

Or written like this:

```
and [not] (bool, bool) → bool
[:] (false, false) → false
[:] (_, _) → true
```

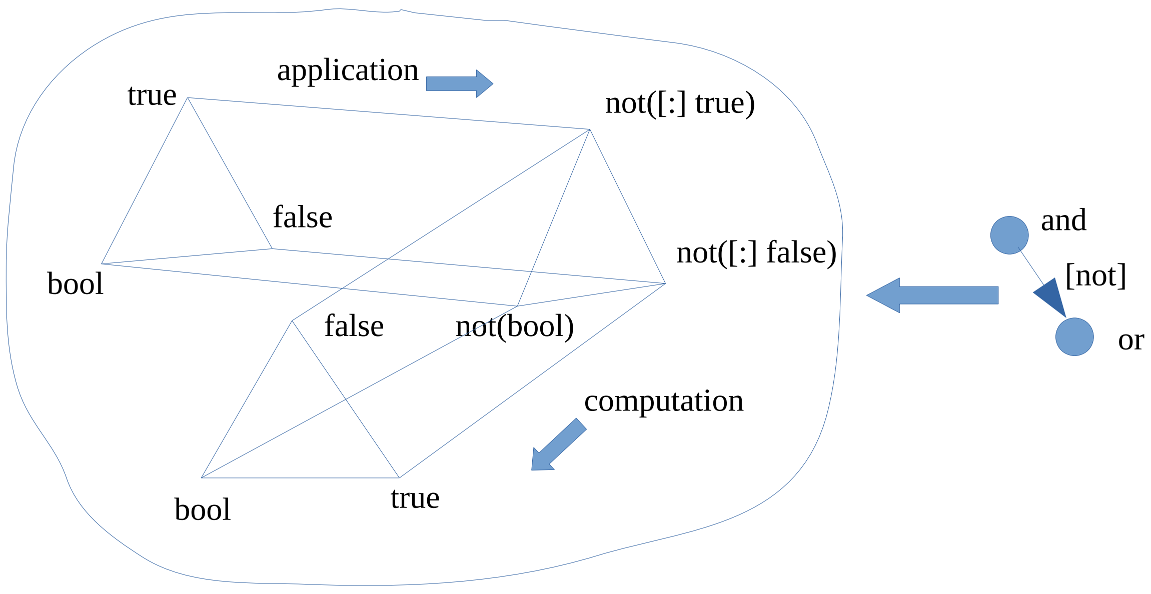
Notice that this is the definition of `or`, such that:

```
and [not] <=> or
```

Which is the same as writing the equation:

```
not(and(A, B)) = or(not(A), not(B))
```

The hidden meaning is a geometrical structure that can be reflected over and over, infinitely times. Words like “application” and “computation” are labels we put on the dimensions of thought space in this geometry:



The meaning of symbols are restricted by the ways we use them. The way it is restricted in formal languages is according to the axiom of quality.

References:

[1] “Formal language”
Wikipedia
https://en.wikipedia.org/wiki/Formal_language

[2] “Path Semantics”
AdvancedResearch, Sven Nilsen
https://github.com/advancedresearch/path_semantics

[3] “Informal Theorem Proving”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/informal-theorem-proving.pdf

[4] “New Standard Order for Levels”
Sven Nilsen, 2021
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip2/new-standard-order-for-levels.pdf

[5] “Path Semantical Quality”
Sven Nilsen, 2021
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip2/path-semantical-quality.pdf

[6] “Identity of indiscernibles”
Wikipedia
https://en.wikipedia.org/wiki/Identity_of_indiscernibles

[7] “Normal Paths”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/normal-paths.pdf

[8] “The Coq Proof Assistant”
https://coq.inria.fr/

[9] “Idris: A Language with Dependent Types”
https://www.idris-lang.org/

[10] “Vladimir Voevodsky”
Wikipedia
https://en.wikipedia.org/wiki/Vladimir_Voevodsky