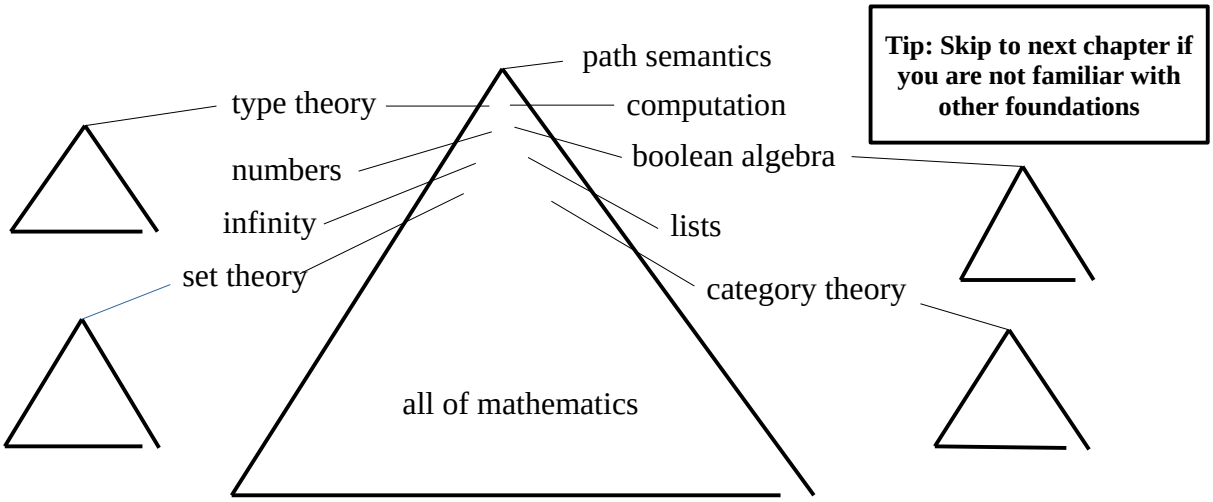
Path Semantics

by Sven Nilsen, 2016-2025

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.



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.

Set Theory and Type Theory are commonly used to study foundations of mathematics:

- Set Theory can be thought of as the study of the operator `∈` (element of)
- Type Theory can be thought of as the study of the operator `:` (type of)

Path Semantics can be thought of as a solution to the problem of modeling Type Theory in logic.

In Path Semantics, the `:` operator from Type Theory is given a precise logical definition, which relates propositions between path semantical levels. Path semantical levels corresponds to cumulative type hierarchies in Type Theory, which in Set Theory is a von Neumann universe.

Normal logic does not have path semantical levels, so when designing these levels there is a choice in how to represent them. In Set Theory, a member might belong to multiple different sets. What makes Type Theory useful in programming is that every member of a type only has one type. This property makes it easier to type-check programs.

In logic this translates to preserving invariance of the theory up to normal propositional equality of members. This means, if `a == b ∧ a : c ∧ b : d`, then one can prove `c == d`. The natural choice of `:` is using `=>` (implication) and `<` (path semantical order). However, since `a == a` for all `a`, when `a` is true one can prove `a == c`. This makes a member of a type equal to its own type, which is undesirable.

Another interpretation of path semantical levels is physical moments in time. Due to the problem above, this would cause everything to happen all at once, in the first moment of the universe.
To fix this problem, Path Semantics introduces a new operator `~` called a "path semantical qubit":

~a “qubit” of `a`

The name “qubit” comes from this operator being implemented in classical models as random truth tables, generated using the input proposition as a random seed. The random truth table is fixed for a normal enumeration of proof cases, but between each turn a new random truth table is generated. The behavior of these qubit propositions are analogous to qubits in quantum information theory.

With other words, where Set Theory and Type Theory might be thought of as the study of a single operator, Path Semantics departs from these two other foundations by requiring two operators. The first operator is `:` as in Type Theory, but grounded in logic. The second is the qubit operator `~` that extends normal logic with random truth tables.

Now, random truth tables lead to a new problem: The qubit operator `~` is tautological congruent, not normal congruent. To build this theory, a meta-theory of logic (HOOO EP) extends logic itself.

It turns out that once HOOO EP is added, there is no need for definitional equality. This solves the problem where e.g. Homotopy Type Theory relies on Identity types, or in Set Theory where one needs to add an equality operator to reason about uniqueness. Path Semantics solves this problem directly by treating the qubit operator `~` as fundamental, from which the meta-theory is derived:

(a == b)^true `a == b` is tautological, replaces use of definitional equality

HOOO EP is used to reason properly about the qubit operator `~`, but as a side effect, it solves the problem of meta-logic in general. This is a very elegant solution, as if the theory builds itself.

Back to the problem of using `a == b` as the notion of equality of members:

a == b ∧ a : c ∧ b : d → c == d

Using the qubit operator `~`, one can express `~a ∧ ~b` as these two propositions being temporal in space-time. Instead of saying that `a` is true and `b` is true directly, we say that their temporal propositions are true. Since the `~` operator is tautological congruent, it holds for any argument that can be proven to be equal without making any assumptions. This design respects the desirable properties of logic that its invariance holds up to tautological equality. However, one could also just assume e.g. `~a == ~e` and `a == e`, which holds locally in space-time.

To prevent everything from happening all at once, one uses `a == b ∧ ~a ∧ ~b` instead of `a == b`:

(a == b ∧ ~a ∧ ~b) ∧ a : c ∧ b : d → c == d ∧ ~c ∧ ~d

This property propagates to the next moment in time, causing time to move forward automatically.

The design above fixes the issue of members becoming equal to their type by being true.
To simplify the design, one replaces `a == b ∧ ~a ∧ ~b` with a binary operator `~~`:

a ~~ b ∧ a : c ∧ b : d → c ~~ d

Axioms of path semantical order `<` are chosen such that one can prove:

a : c ∧ b : d → (a ~~ b) : (c ~~ d)

The `~~` operator is path semantical quality and is defined as following:

$$(a \sim\!\!\sim b) := (a == b) \wedge \sim a \wedge \sim b$$

This definition does not necessarily hold in non-standard Path Semantics, for example when studying Early Schelling's notion of non-duality in Transcendental Propositions. Early Schelling's non-duality is stronger than Hegel's Immanent Contradiction of Idealism. The latter is compatible with standard Path Semantics. Non-standard Path Semantics is used in post-Kantian philosophy.

To preserve the notion of temporal equality, one uses a lifting axiom:

$$a == b \wedge sd(a, b) \quad \rightarrow \quad a \sim\!\!\sim b$$

The `sd` operator is symbolic distinction, which lacks the ability to prove that its arguments are symbolic indistinct, because symbolic indistinction is propositional infinity and considered taboo. Knowledge of symbolic indistinction collapses the property of logic's invariance up to tautological equality. However, symbolic distinction itself is sound and can be used to approximate indistinction.

In physics, observers can distinguish objects from each other, but any two references to the same object are necessarily symbolic distinct at some metaphysical level. The time interpretation of path semantical levels preserves the intuition that "x" is the same symbol as "x", is trivial in one sense, but metaphysically this only holds up to some level of abstraction of symbols that observers use.

While the focus of Set Theory and Type Theory are on their single operators respectively, this work overlaps with Path Semantics to a such extent, that Path Semanticists are generally not very interested in this operator, as they can reuse prior work. Path Semanticists focus more on the qubit operator `~`, which is significantly more interesting because it lacks prior work.

It means that Path Semantics as a field might be thought of as the study of the qubit operator `~`.

This puts Path Semantics on a complete different trajectory as a foundational theory than the other foundational theories. In general, when working in these other fields, people avoid Path Semantics. They have the choice to switch to Path Semantics, but they can not work in both fields without making it an interdisciplinary study.

Due to this difference in focus, the technical language of Path Semantics differs from the technical language used in Set Theory and Type Theory, which might make it a higher barrier of entry for people from these fields. However, the framework of Path Semantics is very close to standard mathematical notation and extends it in ways that harmonizes, such that when people come from standard mathematical backgrounds and logic, they do not need to relearn their use of notation.

In summary: This introduction explained the motivation of Path Semantics and how it differs from other foundational theories.

Time to move on to the introduction of the school of Path Semantical thought.

The School of Path Semantical Thought

Formal languages[1] that can express sentences conforming to Path Semantics[2] give a formal way of checking for meaningful statements. In Path Semantics, the core idea is to fix semantics of symbols such that two symbols are the same, if and only if all paths from the symbols are the same. There are many kinds of paths in Path Semantics and people can also freely invent their own kinds, but among people working with Path Semantics there is an expectation that symbols can be used to identify paths. This is the core principle and is expressed in the core axiom of Path Semantics.

As a private person, there is no requirement to adhere to Path Semantics. People are not constrained philosophically to follow Path Semantics. It is for pragmatic reasons to create guardrails against erroneous reasoning when operating in language design without the support of other mathematical foundations. The motivation is to distinguish Path Semantics as a profession of excellence.

Just like any profession of mathematics relies on core principles to enable collaboration, the core idea of Path Semantics enables collaboration among Path Semanticists. One can understand a Path Semanticist as somebody who is a mathematician, logician or participates in other fields using mathematical knowledge, but also adheres to the core idea of Path Semantics for collaboration.

To most mathematicians, Path Semantics sounds a lot like Leibniz' law of indiscernibles. In Category Theory, the Yoneda lemma plays an equal important role. However, when Leibniz wrote down his ideas, he did not have formal logic to formalize his statement. Leibniz' law of indiscernibles was interpreted formally later, when Second Order Logic was invented, causing mathematicians to claim historically this was true and authentic to Leibniz' idea, based on that specific interpretation, which relied on definitional equality, an ad-hoc operator added to logic to fix the problems with propositional equality for substitution. This is not good enough standard of math.

Path Semanticists do not accept definitional equality without elaboration on implementation semantics in various formal languages, but recommend tautological equality as a more trusted logical operator. They do not accept the original work on incompleteness by Gödel, nor the associated undecidability proof of the Halting problem. It is not due to disagreement about the conclusions, but these methods used in the proofs do not live up to the standards that Path Semanticists require of mathematical knowledge based on the core idea of Path Semantics.

In mathematics, there are formal languages, that are higher in regard by the community, than other formal languages for professional credibility, when it comes to sharing mathematical knowledge. Path Semanticists, working in their distinguished professions, investigate and study all sorts of languages to become better at language design. The goal is to design languages with proper language biases that fit the requirements of usage, aiming to improve our civilization overall.

The foundation of mathematics has been traditionally associated with First Order Logic[3], but in the past century, Type Theory[4] has emerged as an alternative. However, Type Theory is relatively complex compared to First Order Logic. It is desirable to keep the foundations of mathematics as simple as possible, but this might collide with the complex requirements of practical usage. Path Semanticists believe that using multiple foundations is proper as long mathematical knowledge is generated satisfying the core principle of Path Semantics, or when it is not, to issue warnings or elaborate on the edge cases as a way to give people precise information about various systems.

Another desirable property of some 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 normal propositional equality is not always sound. Definitional equality was

invented to work around this problem, but unfortunately one can not derive definitional equality from other propositions without adding additional axioms. The research on Path Semantics has resulted in the conclusion that tautological equality is a more proper way to address the logical requirements for substitutional equality, but this also require extending Intuitionistic Propositional Logic with exponential propositions, which in programming corresponds to function pointers.

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. Definitional equality fixes this problem, but it is added ad-hoc and is not a first class construct in logic. Tautological equality, `(a == b)^true`, is logically equivalent to power equality, `b^a & a^b`, and is a first class construct.

Path Semanticists recommend tautological equality, by lowering the foundation to propositional logic and data structures. Instead of inventing a more complex language and rely on it, like First Order Logic or Type Theory with dependent types, Path Semantics uses a smaller and simpler idea that is more trusted, informing foundational work, such that one can trust First Order Logic or Type Theory, when applied in specific contexts according to the core idea. This also generalizes to domains when these language tools are not appropriate, such as quantum functions. Through the framework of Path Semantics, one can understand the semantics of quantum functions that they are not objects with an underlying source code, but are fully described mathematically by combining quantum existential paths with randomness. It might seem at first that Path Semantics adds a lot of new complexity, but all this complexity is reduced to a simple idea. The trick is how to make this possible and show that this way of generating mathematical knowledge can be trusted.

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` that takes a function 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 sometimes hold, even when there is no concrete function that corresponds to the inverse. One such rule is:

$$inv(g \cdot f) == inv(f) \cdot inv(g)$$

This means, the foundation needs to handle the subtle difference between rules. Some rules work in the absence of a solution of the inverse, but others require a solution and must be guarded.

For example:

$$f(a) == b$$

Can not imply:

$$inv(f)(b) == a$$

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

Intuitively, if one could prove that, then every function would have an inverse, which changes the categorical structure into a groupoid. Some applications might do this intentionally, but it does not hold in general for all mathematical knowledge. This is why Path Semanticists seek more precision.

In Path Semantics, this is solved by using a “qubit” operator `~`[5]:

$$\sim inv(f) \quad \rightarrow \quad inv(f)(b) == a$$

The `~` operator is defined using a data structure. It can also be defined in brute force models of classical logic with boolean algebra 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:

$$f(a) == b \wedge \sim inv(f) \quad \rightarrow \quad inv(f)(b) == a$$

This axiom is not part of the foundation, because there are many ways to use Path Semantics. There are many perspectives of mathematics. Path Semantics does not tell you which perspective is correct. 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 completed, since mathematics is extensible. It is only possible to provide a foundation that helps going to the frontier of math, 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 Semanticists use, 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

These features are common in most general purpose mainstream programming languages.
There are some desirable additional language features that enables more support for advanced theorem techniques, such as for-all quantifier of generics, but overall it is sufficient to get started using these two features. This saves the trouble of manually implementing a type checker.

For people interested in checking their work using a formal language, using a special customized designed language without starting from scratch, the AdvancedResearch community (that currently hosts the Path Semantics project) offers the tool Hooo[20]. This tool has a standard library and a syntax that is easy to understand and read, which grammar enforces maintainability of proofs by design.

The growing industry of formal theorem proving, assisted by computers, makes it attractive for many professionals to use a large community tool for collaboration. It is not the aim of AdvancedResearch to replace these tools, but usually they do not satisfy all requirements of doing Path Semantics in general. Therefore, Path Semantics is regarded as a separate field that focuses on study of languages beyond the ability of other fields, to reduce the amount of redundant work. This makes Path Semantics a highly specialized field, but to avoid becoming too narrow over time, the scope is extended to all new ideas about language design that adhere to the core idea of Path Semantics. The result is that many people, who are deep intuitive mathematical thinkers, get attracted to Path Semantics and the community sharing their work based on the same core idea.

With other words, many programming languages, like Rust[6], can be used to implement Path Semantics, but there are also tools available, backed by AdvancedResearch, for those that do not want to derive everything themselves from scratch or feel that the ecosystem of libraries or tooling in other languages is not proper for their work. This approach makes it possible for people to learn about foundations of mathematics in more than one way. They can practice by building from scratch or by building upon existing knowledge shared by a community.

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 this question with 3 letters: **PSI**.

The 3 letters stands for **P**ath **S**emantical **I**ntuitionistic Propositional Logic[7]. Intuitionistic Propositional Logic[8] (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, in order to finish, 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 Semanticists' 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. It could be wrong. For example, the error might an idea that people have some intuition about, but over-extends it by taking it too far. The same holds for other theories in mathematics that build on axioms: You should always be cautious in case there is some context where these axioms do not hold for valid reasoning.

The only thing I know is that, when you combine PSI with other logical languages from a Path Semantical perspective, it seems to give you the ability to do mathematics. Beyond that, one can study mathematics from a philosophical perspective and apply the core idea of Path Semantics without being slowed down by formalization.

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. It was not developed initially from the foresight of understanding what it all meant. The amount of work required to understand why it works, is an immensely huge burden to take on. 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 theory, not in the absolute confidence that it is correct in all situations: So far, it is the best I can make it for now. 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.

Hence, symbols in Path Semantics work kind of like the past. The past can not change. Intuitively: Symbols can not change in language under valid mathematical reasoning and what follows can be to a large extent unknown to some observer. Yet, with this knowledge that the symbol might be trusted with this property, one can over time generate mathematical knowledge about the world.

When people decide to agree on this principle by design, this enables collaboration with shared knowledge. This requires seeing other people as part of the same civilization. Therefore, the topic of civilization becomes immediately central to the study of Path Semantics. However, this is not about how to rule civilization, but primarily about sharing knowledge in a civilization, that happens to violate policies of tyrants that seek to keep knowledge from people for abuse of power and control. Path Semantics do not give any person a particular advantage when all agree upon the principle, but the generated mathematical knowledge is largely non-trivial and serves as a good investment.

Path Semantics has a huge potential in bridging all kinds of mathematical knowledge, so the foundational answer of the question "What is a symbol?" with PSI, might seem simplistic at first. PSI is basically a logical language needed to make it possible to say "A is of type B", or `A : B`, using propositional logic, plus quality to describe path-like relationships.

Wait a minute! We already have a statically typed language! Why is there a type operator in PSI?

The reason we need PSI is because we need to simulate aspects of dependent types. PSI bootstraps statically type checking, 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! While this seems unnecessary complex on one hand, it gives also more insight into how types work.

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:

$$(a : b) == ((a => b) \wedge (a < b)) \quad \text{or, equivalently:} \quad ty(a, b) == ((a => b) \,\&\, pord(a, b))$$

The two expressions above are the same, but written twice to make it clearer what is the specific construct in the second expression. These constructs are not built-ins in the language used by default, so one does not use the type judgement in the original language, but construct a new one.

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.

Another use case of PSI is time. Previously, I mentioned that symbols are kind of like the past, they can not be changed under valid mathematical reasoning. There is a time interpretation of PSI where propositions are not only considered representing symbols, but as propositional content in time.

One can immediately see from this example alone, using a philosophical lens, that Path Semantics is super, super connective to all sorts of mathematical knowledge. Sometimes in surprising ways. This is why Path Semantics is often used to build bridges to other fields, kind of like a central hub of using language in a civilization. It has a language bias that lends itself well to connectivity.

The core axiom models how symbols are used, not just types. This is a very important insight. If one studies PSI in isolation, one gets some hints at how to build other building blocks. This is where people often lose 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), each taking a byte, have each a binary operator in PSI of homotopy level 2, resulting in 4 294 967 296 operators. In comparison, just one level below, you will find the 16 binary operators of normal logic.

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 finish exploring. This vast desert of mathematics is inexhaustible, even it is just one building block.

Binary operators are the building blocks of expressing statements in logic, so when there are this many binary operators, this allows many island logical languages to exist out there with some internal sense or nuance, that might have some practical application in a very narrow context.

A homotopy level in PSI is defined by how many times one can nest the path semantical qubit operator `~`. This kind of level does not correspond precisely to homotopy levels in Homotopy Type Theory (HoTT), because it is a stronger logical theory, but is a clear analogue and that is why the term is used. However, it is possible to model Homotopy Type Theory within PSI and reason about it. Imagine this bootstrapping process as a chain where one starts with propositions, IPL as the logic, continuing to PSI and finally build HoTT. There might also be more than one way to build HoTT in PSI and different approaches can have strengths and weaknesses compared to each other.

IPL is the most trusted logic known to mankind. Path Semanticists extend IPL with exponential propositions, due to a lacking power operator and uses the HOOO EP axioms for meta-theorem proving. HOOO EP is orthogonal to PSI, but required for completing the reasoning about the path semantical qubit operator `~`. This can be done without arriving at PSI, likewise one can build PSI without using HOOO EP. However, since these two theories fit together and complement each other, Path Semanticists often assume HOOO EP implicitly when it is proper.

HOOO EP is so fundamental for Path Semanticists, that this is one of the reasons the original work by Gödel on incompleteness of mathematics is not sufficient to the highest standard of mathematical knowledge. Gödel's notion of provability uses Provability Logic, a modal logic, where Löb's axiom, when translated naively to HOOO EP, is absurd. HOOO EP is the correct and more fundamental notion of provability and can be used to model Provability Logic. The result is

that there is no debate among Path Semanticists whether Gödel's work is good enough or not, it is simply not good enough. This can be provoking for people new to Path Semantics. However, there is nothing anyone can do about it, but this does not take away the influence of Gödel's work on mathematical history and computer science. All the ideas are not necessarily wrong due to this error.

In summary so far, PSI opens up the door to a new mathematical world of extreme rich complexity. When flesh out PSI formally, using the HOOO EP axioms, it gives new insights into the highly influential work by Gödel on incompleteness. This alone should qualify Path Semantics as a field breaking new ground theoretically.

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. PSI is not required in all cases, the core idea alone suffices.

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 has 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. Still, 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[9]. 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	Joker Calculus
Informal	Path Semantics	English

PSI is a formal language, yet Path Semantics as a whole is not. The reason Path Semantics is rigorous, is due to its core idea that people working in Path Semantics agree upon. 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.

For example, here is a statement that is familiar to Path Semanticists:

```
f <=> \true
```

This looks a bit like a formal statement, but it is not necessarily so, depending on interpretation. To describe the statement above in Second Order Logic, one might write the following:

```
∃ f { ∀ x { f(x) == true } }
```

This is a statement in Second Order Logic because one quantifies over `f`, which is a function.

It can also be shortened by restricting custom language operators to predicates:

```
∃ f { ∀ x { f(x) } }
```

Yet, this is still not entirely formal, as it is not known whether one talks about constructive logic (IPL) or classical logic (PL) that has the excluded middle axiom. Only in classical logic one might draw on the semantical equivalence between propositions and boolean values. The issue here is that `true` might be a type in constructive logic, in which case the whole expression is also a type, but it also might be a value of a boolean, in which case the whole expression is a value of some type, that is boolean when the statement is true. So, from the outset, the statement is informal because there is an ambiguity of what kind of language is used.

In addition, the usage letter `f` informs Path Semanticists that this is a function. Most languages have no way to express this idea in general. However, Path Semanticist say simply:

```
f <=> \true
```

Which conveys the idea of a proof. With other words: It is a baby picture of modern civilization. It is not relevant for Path Semanticists to always be entirely formally precise, because a lot of semantics is conveyed using a precise notion of ambiguity that allows a flexibility of interpreting a statement. Hence, it is an informal expression, but it is at the same time rigorous, because it still satisfies the core idea.

What is going on here, is that Path Semanticists express enormous amount of information among themselves, using just a short sentence with a few symbols. This is why people are motivated to use Path Semantics. It saves a lot of time, that enables people to work on stuff they find interesting.

One common argument that people use in mathematics, is that Set Theory[10] models most of mathematics. However, this is not accurate. Set Theory does not model everything, e.g. randomness. Randomness is a huge part of math. 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.

Another example: First Order Logic models relations as a perfect information environment, so it kind of *pretends* to know everything. It gives people a feeling that everything in their world is

knowledgeable, when in practice we can not know everything. If you only knew First Order Logic, you might not believe there is something like a secret in language, because it is seems open to you.

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. While this might seem like a purely philosophical perspective, it has consequences in how people actually implement language tools using computers.

For example, private information in data structures is an essential tool in engineering, such as software libraries. This protects people from making errors, where the intended usage provides safety guarantees by the library interface.

The problem is that there are things missing in First Order Logic that are essential in practical programming, so this formal language does not serve an actual foundation in this case. Path Semantics avoids this problem, simply by not assuming that all operators are like predicates. Removing this assumption makes the logical foundation weaker, but this does not imply that it is less powerful. Weaker proofs are often better proofs, because they hold in more contexts.

Path Semantics constrains the language of mathematics in other ways, different from First Order Logic, that are also powerful ideas in themselves. Path Semantics is about going back to basics and making different choices of language design.

Therefore, Path Semantics can not provide answers to many deep questions. Its power comes from mostly avoiding design errors done in the past. It can not give us direct answers to many deep questions, because it assumes so little. When we attempt to answer these deep questions using Path Semantics, we often must make design choices. These design choices introduce language biases that shape how we think. To understand how this bias works, one needs experience and for now there is not many ways to learn this, than to explore on your own, or together with other people.

However, the very least Path Semantics can do, is to point out some design flaws in other languages, such as First Order Logic. This is done by constructing alternative languages, like Avatar Logic[11]. 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 directly. There are only puzzles, from which you might figure out the answers by yourself.

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 other objects (e.g. Seshatic Queenity[22]). However, as a framework, there is complete honesty about the lying going on, formally.

All formal languages are rigorous, but not all rigorous languages are formal. Rigor in an informal language is often not shown directly, but assumed from the context. For example, to keep a paper somewhat comprehensible and readable, it is common to leave out steps that are easy to prove. This practice is used in mathematics in general. Hence, mathematics as an activity is often informal.

The practice of mathematics is so similar to Path Semantics that, in most cases, 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.

To many people, mathematics might seem like magic, as if it has always been there, even before humans started to use symbols to represent ideas formally. However, in practice, mathematics is inseparable from the usage of symbols. Symbols seems so simple. It is when people talk about what symbols mean, that the perspective gets more complex. In attempt be more precise, people use symbols to talk about the meaning of symbols. As a result, meaning of symbols itself is reduced to studying usage of symbols. This usage is formalized in the core axiom of Path Semantics.

Perhaps surprisingly, mathematics itself is highly connected with the usage of symbols, so the gap between usage of symbols in general and the idea of mathematics, which might seem magical at first, can be narrowed and make the magic go away. What most people just assume is out there somewhere, like when Newton described the law of gravity, is actually a result from careful usage of symbols that influences our minds to make these discoveries.

I need to show people that Path Semantics is not merely pretending to model mathematics, but that it is a perspective of mathematics itself, without any magic. Mathematics is, precisely put, about usage of symbols and really there is nothing more or less to say about it, because it is so broad.

Path Semantics can describe what we mean when e.g. using equations using a combined approach:

- Path Semantics to express new ideas
- 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 (core axiom) 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 to 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 enough 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[12]). For all such collections of symbols, a quality expressed as `$F_0$ ~~ $F_1$` is uniquely associated with a quality `$X_0$ ~~ $X_1$`:

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

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

An easier representation to read, that can be derived using particular axioms for path semantical order and the definition of the path semantical type operator `:`, can be written as following:

$$\frac{(F_0 : X_0) \wedge (F_1 : X_1)}{(F_0 \sim\sim F_1) : (X_0 \sim\sim X_1)}$$

This form provides the intuition of an introduction axiom for a path-like typing judgement relation, such that a path gives rise to some path space. The quality operator `~~` stands for the path-like relation, across members and types, but is technically different from a homotopy path, so it should not be confused with Homotopy Type Theory (HoTT). Instead, it is normal to model HoTT using PSI, by extends Intuitionistic Propositional Logic (IPL) with this axiom. While PSI is stronger theory than HoTT (propositions live in h-level 1), it might be also viewed as a more fundamental theory, kind of like how propositions are viewed as more fundamental than predicates in logic.

The reason this form is not used, is because in some models the quality operator `~~` is implicit and never specified symbolically, such as in Path Semantical Logic (PSL), which is a classical model. In that model, the above form becomes non-representable. This is why the first form is used in general.

The word “quality” refers to “Path Semantical Quality”[5] (see reference for more information). Quality is a partial equivalence relation that lifts bi-conditionals with symbolic distinction. This makes it possible to express structure between symbols in language, which structure is not provable from an empty context, by definition, due to lack of reflexivity. 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.

Among the 16 binary operators in classical logic, the AND operator is the natural partial equivalence operator, since it has symmetry and transitivity, but no reflexivity. However, most people do not think about AND as an almost-equivalence. The problem with AND is that in being true, it makes its arguments true, which triggers IMPLY conditions, hence starting propagation. Quality gets closer to the intuition of an almost-equivalence that is also kind of like AND, but does not trigger IMPLY, such that the underlying symmetry in path-like problems can be modeled. In PSL, this provides a benefit of exponential speedup in brute force theorem proving performance.

The axiom is a modification of Leibniz’ law[13] where equality of arguments to predicates is replaced by quality between data structures as propositions. Predicates can be modeled using a data structure for function application and an axiom for substitution 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 behavior. Quality is almost empty of content, which allows great flexibility in usage, but has lot of symmetry.

The combination of lot of symmetry in the quality axiom, combined with little specificity of behavior, provides a powerful combination that can be used in many applications, without losing much flexibility in the requirements of usage. That is the theoretical basis for understanding the efficiency that humans gained historically, when starting to use symbols. This event, like the mastering of fire, or inventing the wheel, caused the birth of civilization, due to boost in efficiency.

In informal theorem proving, one is allowed to use other formal languages in parts of the proof. 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 a new language.

Inquality

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

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

Inquality has major philosophical implications in Path Semantics. This is because, from the first form, one gets one kind of reasoning forward in time, which propagates proposition toward causal relationships. A second kind of reasoning backward in time, which finds original causes of events.

Inspiration of the core axiom

I used normal paths[14] to do some theorem proving that I found useful informally. Formally, it was difficult, in Coq[15] and Idris[16], compared to informal theorem proving. To justify the informal technique I used, I wanted to learn more about foundation of mathematics. Vladimir Voedvodsky[17] 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.

Another inspiration to start the Path Semantical project, aiming at more general mathematics, was Adinkra Physics Diagrams, introduced by Michael Faux and Sylvester James Gates in 2004. These diagrams are used to describe complex equations of supersymmetry in physics in concise form. I discovered that I could model a lot of math with such diagrams by making small modifications. For example, I developed a definition of matrix determinants that is easier to understand visually. To describe more math, I extended Adinkra Physics Diagrams to a discrete space called “Context”, due to the property of modeling a spatial change of some value as an edge between nodes, as context. The more general diagrams have the property that one can construct higher dimensional shapes from lower dimensional ones and contract them to restore the lower dimensional shadows.

This latter inspiration translated well onto the intuition that when mathematicians use equations, the form of an equation is just a lower dimensional projection, a shadow, that shows the content of the equation using some language bias to the person interpreting the symbols. Logically, all the information is present in the equation, but this can not be perceived directly by people using the symbols. Hence, when combining these two insights, one of that of “paths” and second of context space, I learned that a path could be a perspective of something that might not be clearly representable using only the information in the path alone. The step further that, what might not be clearly representable could use a symbol, was a simple one. All paths from that symbol could give meaning to the whole information content. This was how the idea of Path Semantics was born. It happened a summer day I was relaxing, on a beach in Norway, a kind of “eureka!” moment. I ran home to find pencil and paper to write this idea down. The concrete example I was thinking about at the time, was addition and multiplication of the even property of natural numbers.

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.

I believe the most important idea of all mathematics, is the idea to express the substitutional functional equality, of some problem regarding properties of functions, with its solution. This idea is why I believe that people are motivated to use mathematics in the first place. Approximately 90% of what people use mathematics for, is to find some computable solution that predicts an aspect of a more complex function, such that the aspect predicted can be viewed as a property of the more complex function. This is how humans organize activity in the world by applying information technology and language tools, because the world itself is too complex to described in full detail.

The summer day by the beach when I thought about addition and multiplication of the even property of natural numbers, I realized that this was an example of the most essential activity in use of mathematics. Since Voedvodsky showed that paths are fundamental in the foundations of mathematics, I used the word “path” to put a name on the general case of what I was thinking of.

Path Semanticists use the term “normal path” to distinguish that kind of path from other paths, such as trivial or existential paths. Since normal paths are so common in usage of mathematics, it can be understood from the context that people are talking about normal paths, simply by saying “path”. However, in some contexts it is ambiguous what kind of path one means or whether one means paths in general, which makes “normal path” emphasize the normal usage of the word “path”.

A normal path is not the same as a homotopy or path in Homotopy Type Theory (HoTT). This can be confusing, so I have to elaborate on how it differs. A homotopy is a higher order path that can be continuously deformed from one path as start, to a second path as a goal. This semantics of deformation plays an important role for topological spaces. In the context of functions, which can have any number of arguments, there is no distinction between higher order paths, homotopy, and paths that only take a single argument. In that sense, I did not use the word “homotopy” because it would lead to a distinction that felt artificial. A path to me was on one hand a function, which is also a function in the homotopy theoretical interpretation of HoTT, but it was on second hand a solution to some problem. The use of the term “function” was proper for the solution, but the start was a problem, which technically is a more general idea than a function. So, the name “path” seemed both appropriate and inappropriate, depending on perspective. The homotopy was just substitutional equality and did not require a new name. That is why I stuck with the name “path” to distinguish the form of relating a problem to its solution, that is also contractible to a function. While there was a problem with ambiguity in terminology, there was none ambiguity in the symbolic expression.

One can criticize this choice of terminology from a categorical perspective, because functions are morphisms in Set, while paths are morphisms in another category. Still, from a language design perspective, a normal path is contractible to a function when it has a solution, in which case it also satisfies the description of a homotopy. This means, it is the use of normal paths, not the categorical interpretation, that gives meaning to the terminology in a consistent way. Use of symbols and content of information played a greater role than the categorical distinction between mathematical objects. The latter is not accurately portraying functions as they are used in practical programming. I wanted to have a more pragmatic approach, which did not pretend things in real life were more easy to model mathematically, than they actually are.

A normal path is basically an open box in Cubical Type Theory[18] 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 generalizes 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
$f[g_0 \times g_1 \to g_2]$ <=> h	Asymmetric normal path

The use of normal paths for theorem proving was the motivation for developing Path Semantics. So, I spent a lot of time working on the syntax to express ideas as concisely I could make them. No character goes to waste. However, it turns out that formalizing this idea was much harder than I expected. I ran into problems with all the tools developed by the mathematical community.

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[19] to get as close as possible, still 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. Function composition syntax is less readable than normal paths 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 is the analogue of First Order Logic for Type Theory, 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.

Other people might have just given up by this point. Consider that one could prove that normal path composition worked in the obvious cases where functions had solutions. The remaining cases of merely symbolic manipulation might seem trivial. Yet, ignoring full formalization is a bad habit that happens a lot in the mathematical community. This bad habit is the reason there was no foundation for functions in practical programming in the first place. Ironically, by soldiering on, trying to solve these edge cases, I came to accidentally describe the most controversial axiom that I know, even surpassing the continuum hypothesis or axiom of choice. For a long time, it seemed to cause more new problems, than providing solutions. The result is nonetheless an overturning of our understanding of meta-theorem proving in logic, interpretation of time in physics, and providing a critique of Gödel's work on incompleteness. Only the puzzles from one tiny axiom! That is not bad.

Functional completeness

Bits are the simplest collection of symbols that might 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 (2015)

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

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):

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

The first version is interesting, because it can interpreted as a kind of propagation from a path between two functions into a path that contracts all data that these functions might operate on. While this has little value for normal functions in general, it changes significantly when considering atomic functions, which have only one input. By going back to the most basic level of language in mathematics, this perspective starts to make more sense. It is from using atomic functions that one can think about unknowns in this level of language as collection of symbols, because atomic functions can bind more symbols together.

For example, consider a list:

A = [1, 2, 3]

To represent this as an atomic function, I produce the following, making the argument the first item:

A(1)

Again, applying this two more times, I treat this as an atomic function, adding rest of the items:

A(1)(2) A(1)(2)(3)

So, while the language of atomic functions is not powerful enough to express the full semantics of lists, it can bind symbols together in collections, kind of like how lists are functioning.

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`:

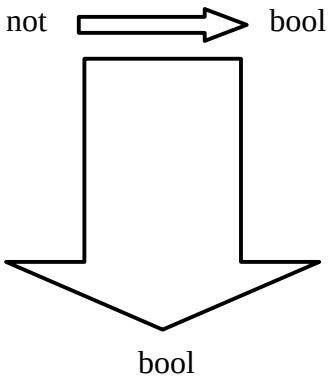
not(bool)

This choice is arbitrary and done simply because we know how `not` gates behave. It makes sense to associate the symbols this way, to use the axiom of quality. If somebody says `a ~~ not`, then we know `a` also takes the input type `bool`. Like in the previous chapter, this is the language of atomic functions, where there is only one possible input to `not`. It is not the normal function one talks about here, but about an atomic function, something more basic. The argument `bool` might be thought of as being associated with `not`, but it is not the same symbol. In propositional logic, one says that `not` implies `bool`, but the propositions are indeterminate and left in that state such that there is no triggering of the condition `not` into `bool`. One wants the collection of symbols to preserve their indeterminate state to model the meaning of actual symbols, instead of boolean values. The parentheses `()` are arbitrarily chosen, as a way of expressing our atomic function. One is allowed to introduce any syntax for atomic functions, but the choice of syntax signifies the dimension of evaluation.

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. The latter approach is often preferred in Type Theory, but does not associate the output uniquely with the input, since the input is `bool` and output is `bool`, which becomes ambiguous in general.

In Path Semantics, all mathematical languages have language bias. This means there exists no universal approach to reach full objectivity. This is acknowledged among Path Semanticists. Path Semanticists do not claim to be more objective: The trick is to assume less, but design more. While the approach here is desirable to make a convincing argument as part of the bootstrapping process, it is OK to break the rules later on, as long the overall design satisfies the core idea of Path Semantics. So, Path Semantics does not give a universal language to check proofs. The only things that can be checked by a computer are statements in theories that are modeled in some language, which as a consequence will have some language bias associated with that particular model.

The technique used in this bootstrapping process is to create a very constrained language, just enough to arrive at the intended goal of design.

Path Semanticists often prefer to work from over-constrained theories and extend them to more flexible theories, instead of working from under-constrained theories and add lots of constraints. The reason is that in programming, one always have to make tradeoffs in design. These tradeoffs can be the amount of memory used vs time complexity. Unless an algorithm is learning or improving itself over time in performance, the programmer has to design the algorithm to be efficient. This is why under-constrained theories often results in bad design. The burden of optimizing an under-constrained theory can make it humanly impossible to achieve good enough results within the time limits of a project.

However, there is also another use case of constrained language: To make convincing arguments about design choices. This is a technique that is common in Path Semantics, but there is no formal underlying model. While math is heavily used in the process, the interpretation of the work appeals to many creative approaches. This diversity is embraced in Path Semantics, since there is a shared common understanding that the real world is complex. To deal with this complexity, one has to be creative sometimes.

With other words, `not(bool) → bool` is desirable in this case because it connects the type `bool → bool` with `not`. In the semantics of `not`, it takes a `bool` as argument first, before returning a `bool`. While the type `bool → bool` is easier to construct first in Type Theory, it is not proper to use the type theoretic approach when arguing for semantics, since semantics is depending on the concrete use of symbols. In this case, the symbol of focus is `not`. Do not worry if you do not get it, this is just to explain that the approach used in Type Theory is not ignored in Path Semantics. Other approaches are seen as complementary.

Some languages write `not(bool) → bool` 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 corresponding 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. In the example above, the type `bool` appears on the left side of the `:` operator, which unusual. However, in this language, there are only symbols in the initial language constructs. Therefore, `bool` in itself is just a symbol, but when used in the appropriate context, it becomes a type. This approach is used because symbols are fundamental for all of mathematics, by the way symbols are used. Meaning allows symbols to take on different interpretations which in various formal languages can be fixed, such as e.g. not allowing `bool` to ever appear on the left side of `:`. Here, one is constructing the pre-formal semantics by extending usage from symbols into language specific constructs.

Again, here is the statements I was talking about:

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

The information content above on both sides (defined by the usage of symbols), are equivalent. When you read the expressions, they can seem very different, but that is a matter of perspective. This transformation is only valid when you define values as atomic functions taking their type and returning themselves. You do not have to take the syntax literally, though. The atomic functions are just to associate the argument uniquely with some collection of symbols. Given that context, one can treat the two sides as having the same amount of information.

However, the two statements might not evaluate to the same value. Evaluation on the left side in most contexts erases the sub-type from the output, such that you end up with less information than the right side. So, in this context, `<=>` does not mean you can substitute. It means only that the information content is equal on both sides. This is a form of overloading, since there are no standard mathematical operators that communicate precisely what is meant. The use of symbols here is to direct attention of the reader, without having a formal underlying model.

It can be very unnerving here to not be able to follow a purely formal argument in logic. In one sense, this should be very simple. In another sense, one becomes aware that, when one understands what is going on, there is no easy way to understand the process formally. However, in Path Semantics, it is common to operate without any other formal foundation. This is needed to communicate the ideas efficiently. It works because humans are very good at understanding nuances in language biases. You could create a formal model of Boolean Algebra in e.g. Avatar Logic, but that would require learning Avatar Logic first to understand it. Here, the bootstrapping is done informally, just enough to get to the language level of functional programming. There exists formal models of functional programming using Path Semantics, but usually, sub-types are used at the value level. Here, it was used on type level in combination with the semantics of atomic functions.

For example, you can not say this:

x : bool

The `x` one is talking about in `x : [:] true`, is a type. In the expression `x : bool`, it becomes a member of type `bool`. The `[:]` operator is a shorthand for `[true]` which applies the `true` atomic function to `x`, returning `true`. The full statement is `x : [true] true`. Since `true` is returned, from some operator `[:]`, one can infer that `:` in this context means `true`. First, the full statement is expanded. Second, since the `true` atomic function only takes one argument, the argument must be `bool`. Hence, `x ~~ bool`. In logic, `x` is a proposition and `bool` is a proposition, but they are qual (“equal” with the “e”) to each other. The logic models the use of symbols. One can use `~x` to say that `x` has some usage. From a logical perspective, there is nothing ambiguous going on here. However, to understand the bootstrapping process, one would have to interpret the logic and this is too hard for people that are new to Path Semantics. This is why bootstrapping is done informally.

Path Semantics can be used to model precisely how people use symbols in logic. The problem is that since mathematics is incomplete, due to not being defined entirely, one would have to expand the model sufficiently to convince people that it indeed captures the semantics. There exists large models that have sufficient structure to include functional programming, enough to convince people that it works. These models use the same building blocks as in the foundation of Path Semantics. However, here one chooses a specific interpretation of `:` in `x : [:] true` that gives it a computational meaning, to make a convincing informal argument. It can be done formally, but most people are not good enough in logic to be convinced by such techniques. Only experts do that.

The entire point is to show people what Path Semantics is about. People have to be convinced that Path Semantics models mathematics, through formalizing the use of symbols carefully. The foundation is so powerful that it can be used in many ways, both formally and informally. This is an advantage that Path Semanticists have over other fields, e.g. in pure logic. In Path Semantics, you are not restricted to using an existing formal language. You can create your own designs. If the foundation of Path Semantics was not good enough to allow people to develop their own designs, then Path Semantics would be useless. People use Path Semantics because they want to create designs.

You might not be convinced by this process. If you are an experienced programmer, then you might get the semantics of atomic functions and get the gist of it. However, since not being able to follow the process in a logical sense, it might not seem that convincing at first. The problem is that Path Semantics is already at a level of expertise in logic, that surpasses most of the existing expertise in logic. So, you are not going to be convinced by the logic anyway, as it is most likely far above the current level of logic you master. Path Semantics has to extend IPL and most people do not even know what IPL means. Do you see the problem? There is no way to explain the bootstrapping process to most people, without appealing to a background of programming experience. This is the background that most people, who get interested in Path Semantics, come from. They get attracted to Path Semantics because it promises a pragmatic foundation for functions, although incomplete. Functions are vital in programming and current other foundations do not handle functions very well.

Over time, when you get more experienced with Path Semantics, you will find this process more convincing. This has something to do with the intuition you build up about language biases. You will feel the level of constraints in the language being used mentally and see that there is only one path ahead: A single method to get to the level of functional programming. This method follows naturally from the limited language design being used here. You will also not get that convinced by equations or logical arguments, because you will see lot of directions that do not get explored. The constrained language here is to put laser-focused attention on what it means to make functional programming work. When you reach this level, experienced programmers know what they can do. This means they get satisfied with the result. They just need to get convinced that Path Semantics can be used to bootstrap up to that level. Above that level, there is no need to work further. This technique is called a “bridge”. Path Semantics is super useful to build bridges to other fields.

The bridge to functional programming is very short, so most people get convinced by this argument that functional programming is more “natural” than most other programming language designs. Of course, that depends on whether you accept Path Semantics or not. Some people might agree with that view, without working within Path Semantics. This might help them to get convinced to study more Path Semantics, because they know from programming experience that this view holds, but they might have not found anyone presenting them a convincing argument why it holds. The fact that this argument can be presented within Path Semantics, is impressive. However, this is not a formal view, so it is more like a bonus, but not considered a solid mathematical truth.

Back to the bootstrapping: `x : [:] true` allows reaching a level of language where some symbol can be tied to the input as a type and branch off to a member of that type.

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

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

This is not yet on par with the syntax of normal functional programming languages like Haskell or Idris. However, if you are an experienced programmer, then you can see where this is going.

At this point, one has logically reached functional programming. In principle, one could model this in logic and keep a large messy database e.g. in Avalog, that has the content of the code. Still, programmers prefer to maintain code in text files and put huge effort into making the text readable. This is why improving the syntax is needed, so people get convinced that this is something that looks like actual functional programming, instead of being a convincing argument only to experts.

Before moving on to improving syntax, why not test that it works first?

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

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

When doing these operations, one is extending the language. It takes on new meaning, but the way one extends is carefully managed, to not violate the core idea of Path Semantics. From a formal perspective, this might not make any sense. How can one just extend language? Think about this informally instead: When using symbols in a new way, one creates meaning. One constructs a "path". That path has to satisfy the requirement expected from Path Semanticists to be a valid language tool for reasoning. The rules are broken, but there are rules about how to break rules.

One can easily see, as a programmer, that this language is very limiting. It can not do much, except basic stuff. The function `not` is barely more complex that the `idb` function (identity of boolean). The reason `not` is used here, is because it is simple, but less boring than `idb`. There is no point in making a complex argument. People use actual tools for programming, so there is no need to develop another programming language to show that Path Semantics works. The reason one uses Path Semantics, among other things, is to convince ourselves that one can trust the implementations of functional programming that already exists. Most of them were built before Path Semantics was developed. So, Path Semantics is kind of like a theory that predicts these tools. It explains them scientifically. However, Path Semantics does not stop when it reaches the level of functional programming. It keeps going on, to probability theory, non-determinism and quantum functions. This basic example can seem boring, but from a programmer perspective: No other foundation actually explains how people use functions in programming today. Think about the billions of lines of code out there, that nobody had a solid mathematical foundation for. Now, suddenly there is one and it predicts that the designs used in the industry are good enough. It also predicts standard mathematical notation correctly and even improves upon some edge cases (with some philosophical nuance that was unexpected).

From a logical perspective, that all of it is explainable by starting which such simple theory, is extremely impressive. Normally, one requires several axioms to describe e.g. First Order Logic and from these axioms one usually do not get more than one language. However, Path Semantics supports all the fields of mathematics, at once in one go. Plus fields that have not yet been developed. For practical reasons, people use more complex theories to build tools. Starting from foundational Path Semantics will usually take too long time. However, if you are looking for creating something new, then using Path Semantics might help you to get there.

It is not possible to build a theory without modeling, so Path Semantics does not "explain" e.g. First Order Logic as a finished product. What it explains are complex theorems resulting from an enormous amount of symmetry, hidden behind the scenes. This symmetry is there because of how people use symbols. Most of these theorems will never occur to anyone or to be expressible within the practical tools that people develop. They will just exist, ready to be discovered someday by some curious person exploring the wild world of semantics.

Logically, Path Semantics is a mathematical monster of a language that has a such huge complexity, not in the axioms, but in the logical consequences of the axioms, that there is no known technology, nor expected to be invented in the future, to handle all this complexity. This is not how people use Path Semantics. How they use it is to narrow down possible designs such that they can reach a good design more efficiently. When the design is done, Path Semantics is no longer needed, because the design will work anyway. The new design will generate new mathematical knowledge on its own.

In theory, over time as people use Path Semantics, there will be produced enough language tools to cover all practical use of mathematics, such that the foundation of Path Semantics is no longer needed. However, perhaps this will take longer time than the expected lifetime of the universe. For now, people already have some useful tools, but as civilization advances, one would like more tools.

The `[:]` symbol is symmetric between input and output, so we can refactor it:

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

This syntax 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.

While doing these steps in the argument, one is not actually seeing the logic going on behind the scenes. In logic, one can prove infinite amount of theorems that become true as a logical consequence of the structure one introduces in the source code. So, it is not possible to grasp the logical side of it entirely. Logicians have to rely on the shared knowledge of their entire field, to get convinced quickly by a new theory. Propositions live in an exponential complex semantic space, which can be non-intuitive for the human brain to reason about. When you read this argument, it might look like simple stuff, but when reaching Boolean Algebra, what happens on the logical side is that all the circuits one can construct, even in principle, have theorems about them. Prior to Boolean Algebra, there are no such theorems. There is a huge shift in the perspective of logic when you reach this level of language. This semantic shift happens without being visible to humans.

All the time when working this way, one is simply using symbols. Symbols are not something inherent in logic. This is why one needs the axiom of quality. How symbols work follows from this axiom, logically. The logical gates are built in a way that might seem arbitrarily, but this construction has all the mathematical information about the logical gates built into it. Therefore, there is no magic going on here, simply using symbols and constructing mathematical objects.

Not long time ago, most of the human species believed in spirits and invisible forces, due to their language capability to think about these things, while not understanding language sufficiently to distinguish their own false beliefs from reality. Mathematics started moving the modern mind away from this kind of thinking into a more analytic approach. However, in the past century, mathematics went a bit too far and became dogmatic. The field was overtaken by programming and mathematicians got a harder time finding a job. This resulted in some kind of collective superiority complex where programming was considered a lesser form of knowledge, until Moore’s law broke the barrier that allowed people to start thinking about mathematical foundations using only code.

This tension between mathematics and programming still resonates with some people today. However, what one learns from Path Semantics, is that there is almost nothing in mathematics that is not like logic, except for this core axiom. When you add this axiom, you can reach the rest of mathematics reasonably easy. What happened is that logic, not just programming, slowly starts to eat away at mathematics. Logic and programming covers mathematics in some sense. This allows mathematicians and logicians to start thinking about programming and programmers can start think about logic, as a more unified and interdisciplinary higher field. The result is that if you want to work as a mathematician today and in the foreseeable future, then it is useful to learn both logic and programming. The mathematical authority of journals gets shifted over to formal proofs and the semantics gets shifted over to Path Semantics. The question is whether one has to think differently about how people use mathematics in their research, but for now it looks like people have no problems learning the language tools that are developed. Still, you can visit a mathematical library, e.g. Institut Henri Poincaré in Paris, and open the books of mathematicians and physicists in the past. What you will observe, is that the language of math changes drastically in the past centuries. In the previous century, this continuous change hit a small bump, but now it continues at a similar rate, if not greater, than before. That bump took one or two generations, so it is actually just people not seeing the overall development in mathematics good enough. This is one of the reasons we need more Path Semanticists, to study how language changes over time in this discipline.

Path Semantics might be understood in a historical perspective, where mathematics is slowly recovering from this small bump and it did not meet the demands at the time of computer science. This is why it became an opening for somebody like me to contribute. If mathematics continues to speed up in design, then to get an easier advantage in the field, more people have to do hard work outside the more traditional branches of math. One part of the Path Semantics project is to invite people to explore more, such that Path Semantics does not remain only a minor correction in a relatively short period of history. The goal is to extend the notion of mathematics itself to new domains, for example by introducing the distinction between Inside and Outside theory.

What is difficult for most people to understand, that every time one uses a symbol, one is "summoning" the axiom of quality in some way. Since mathematics is reducible to using symbols, there is nothing going on that is not about using symbols. Everything is this axiom being repeated over and over, kind of like how NAND is used to construct boolean circuits in computer design.

If you learned a bit philosophy, then you might have heard of Platonism. Many people working as mathematicians are more Platonic biased in their thinking than the average person. However, some people might also be biased toward dual-Platonism, called Seshatism[21]. This is happening because of something called "internal difference" in philosophy. Formally, one can study this by studying the core axiom of Path Semantics. This means there is a formal model that explains this seemingly complex idea in a very simple way.

However, the problem is that people can not see this process happening, because using symbols is such as deep implicit activity, that one does it without thinking too much about it. When you see a symbol, you use it in a way that might seem obvious to you, but logically, this is not tautological. There is nothing you do with symbols that is grounded in the physical state of the world. It is added meaning by people who decide to use symbols in particular way among themselves. The awareness of this "something" not being logical, was part of many religious traditions through history, including the texts written by 2nd century Early Christians. They called it Zoe and Logos, which are derived from ancient Greek philosophers. Zoe was erased from history by book burning, so this knowledge was lost, but there might be a mythological origin: Zoe was the daughter of King Midas.

With other words, when using symbols, people are only "seeing" the shadow of what they are actually doing, but like with any equation, all the information about the symbol is already present in

the form of the symbol. For example, the word “dog” perfectly captures the meaning of “dog”, across all usages of the word. You might have a default interpretation of the word, referring to the animal, but that default interpretation is also part of the meaning of the word. People can perceive the meaning of this symbol immediately, despite everything about dogs is extremely complex in the world itself. All that complexity gets contracted down into the symbol.

That means, people are really using Path Semantics, at least to some degree, just by using symbols. Everybody uses Path Semantics, every day, all the time. Yet, only Path Semanticists admit that they are doing it and they agreed among themselves of using symbols that way to collaborate.

Step 3: Describe formal languages in the reflective language

When I try to visualize `not` in some space of semantics, I imagine it like a node or neuron with paths or dendrites extending from it. Through programming I am very familiar with this operator, so to me, it gives an emotion of something being familiar. However, I am also visualizing it in front of me, at some distance. I do not confuse `not` with myself, so it is not easy for me to imagine myself as that node or neuron looking outwards. From this thought experiment, one can understand that humans attribute properties to symbols that generalize across all sorts of symbols, for example, the feeling of familiarity. When there is a feeling of unfamiliarity, the symbol might be perceived as standing for something of unknown meaning or sense. Also, when I say “I”, my thoughts jump to myself, looking outwards, kind of being that symbol, instead of like seeing myself in the mirror. So, while there are properties one relates to symbols, they are not the same for all symbols. These distinctions in properties might be thought of as a reflective process, where one does not need to think hard to recall these associations when thinking or reading.

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

In Path Semantics, this is called a “normal path”, or just “path” for a shorthand. Normal paths are commutative diagrams in the category of Set, but unlike in these diagrams, there is a bias in the syntax by orientation. It means, one “sees” the diagram, but either horizontal or vertically.

Normal paths use computation, but go up in abstraction, providing a parallel level of computation. The further you go up, the closer you come to the core axiom of Path Semantics. The space gets smaller. In ancient Egyptian mythology, this was the eye of Horus at the top of the pyramid.

Another normal path is the following:

    or [not] <=> and

This is not the same as `and [not] <=> or`, but its twin. All 16 binary operators in normal classical logic has a normal path like this, which twin normal path goes in the other direction. Usually, we call the two normal paths for `and` and `or` for De Morgan's laws in math. In IPL, you should be careful, because not all these paths are constructive.

What happens here is that I decided to use this syntax in a way such that when I reach this point of equivalence, I regard it sufficient to imply that all other paths from these two constructions are also the same. This means, I can go back and replace one collection of symbols with another.

Which is the same as writing the equation:

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

Notice that I did not use the semantics of equations. Not even once. This is a fundamental idea.

You use Path Semantics when you think about abstract things. When you do this, you go closer mentally to the top of the pyramid of math, closer to Logos or the eye of Horus. This direction is biased toward Platonism. You might think that going in the other direction is biased toward Seshatism. However, this not entirely correct.

The pyramid of abstraction is infinitely high. You will never escape it, because it just gets bigger and bigger and full of complex math that eventually describes our world, somewhere in this abstract multiverse, exactly down to every smallest detail. This pyramid has also a twin, just like the normal paths of `and` and `or` by `not`.

The twin pyramid is invisible, kind of like how the role of women has been through the past two millennia in history. Women have done 90% of the work through time. Men have mostly been lazy, argued over philosophical ideas and fought wars.

The invisible pyramid is the actual tapestry of history. It is right in front of you, all the time. Therefore, Seshatism is not a subservient concept toward infinity in the Platonic pyramid. Seshatism was the dominant philosophy prior to the rise of Platonism, that favored causality and concreteness over abstract ideas.

When Plato wrote, Seshat was already an ancient deity. Seshat was kind of like a Muse goddess, but came millennia before the ancient Greeks started to standardize the Nine Muses. Plato forgot that Seshat's mother is Ma'at, the principle of justice and balance. He only mentioned Thoth, the male counter-part of Seshat, that resulted in a long downward spiral, kind of like a curse of Pharaoh. It is not unthinkable that this single event, brought the human species to extinction in some future possible worlds. This error propagated from Plato to Aristotles, to the first authors and revisions of the books of Moses, written around when the library of Alexandria was built, and hence influenced the entire Western worldview, that in turn decided to ignore the warnings of scientists, mostly due to religious extremism, created by shadow organizations wanting to bring the end of the world and enslave the human race like in the dark age.

Pharaoh, according to ancient Egyptian mythology, was living his after-life fighting to maintain the order of the world and keep it from destruction, battling monsters in the body of the sky goddess

Nut, which journey was also perceived as some kind of the underworld, leading to some confusion of whether this was up or down, or both, to make the sun being born again and rise for another day.

Hence, Seshat was erased from history through the feather strokes of Plato and did not even have a surviving cult that worshiped her.

However, this is not where the story ends for Seshat.

Seshat was the goddess of the divine library, that kept copies of finished books from scribes. Today, Seshat might be thought of as still passing down her knowledge in the form of libraries around the world, which are our actual centers of civilization. Many people think money rules the world. However, money is just a mean for making transactions. The actual production in the economy requires resources and knowledge. This knowledge comes mostly from libraries around the world. The pyramid you see on the US dollar is a manufactured lie by childish men playing in secret societies, to make you believe in something else than the evidence right in front of you. There is a temple in every major city for Seshat wherever you go, it just does not advocate it in a huge sign in front of the building.

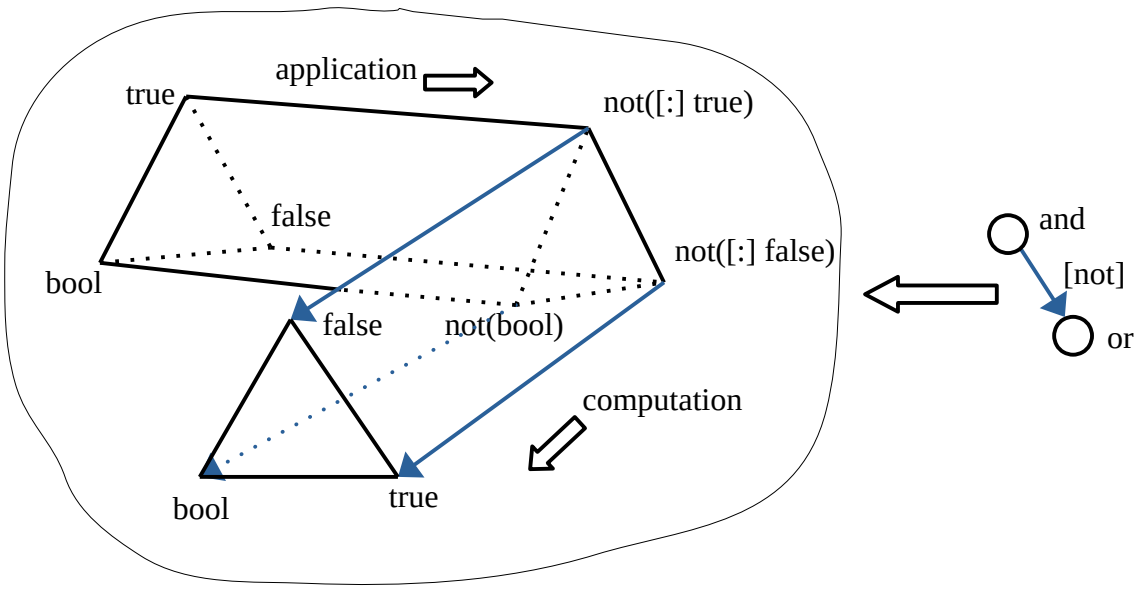
In the library, people need to stay quiet:

“Hush…” (… the library keeps many secrets).

“Come find me,” she whispers, “I will preserve your work for an eternity.”

It can be extremely difficult to follow the thought process when a Path Semanticist is presenting a new theory. This is not because people do not understand what is going on with symbols. Everybody uses symbols all the time, so many people have gained enough experience to understand what is going on. The problem is, they do not understand why they understand it. The process in presenting these steps reveals that there is something people know, but do not know how they know it. Doing this for extended periods of time can be nauseating, so remember to take breaks.

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.

To elaborate on why normal paths are a fundamental idea: You have two sides of an equation. What you put on each side, you can decide for yourself. In propositional logic, this is always allowed as assumption, so you are free to equate anything, absolutely anything, you can think of. This gives an enormous amount of freedom and is part of the reasons that logic is a powerful language tool. Still, to claim that something is a fundamental idea, requires more effort than just putting two arbitrary chosen sides of an equation together and use it as an assumption. There has to be an argument that convinces Path Semanticists that this is truly a fundamental idea, that one arrives at by a series step of small changes, instead of just assuming something arbitrarily. The ideas that Path Semanticists are looking for, is in the form of these convincing arguments. When this work is done, one can use equations or other language tools without worrying about it. The language tools humans have in mathematics are so powerful that they will work when the symbols are put correctly together into collections. The problem is to convince ourselves why one uses particular collections of symbols.

Path Semanticists believe that symbols mean what people are using them for. This is a very Wittgensteinean approach to philosophy of language. However, at the same time, the way people do Path Semantics is not a very post-modern approach, that frequently attacks meaning. Meaning in Path Semantics is contractible from the use of symbols. How people use symbols, under valid mathematical reasoning, is formalized in the axiom of quality. Still, quality is an internal difference in philosophy, so it can be used with inauthentic Platonic or Seshatic[21] language bias, causing suffering or oppression in structures of power. So, while meaning is still there, it is open-minded.

Internal difference means that the core axiom is kind of like a wormhole that brings you from the abstract pyramid to the concrete pyramid. It is the same in both worlds. The core axiom of Path Semantics is simultaneously an event that starts new threads in the tapestry of history, but also an abstract idea that ancient religions developed mythology around. This internal difference can be modeled precisely in the implementation of PSQ for brute force theorem proving, which shows that the distinction is internal to the language. From outside the language, one can not tell the difference, no more than one can tell a random bit vector from its dual. Therefore, internally, there is duality, but from distance, there is non-duality. Both perspectives are unified and harmonized into one idea.

To escape the abstract Platonic pyramid, you do not bow and go down the pyramid carrying the weight of everything above you on top of your shoulders, but you go up and up, make yourself smaller and smaller, feeling lighter and lighter, reaching the tip and that is the moment where the world is created, not from a beginning by an intellectual mind, but an eternal creation in each moment of time. It is a sacrifice by the goddess Seshat, that could have written all the stories of the world on her own, due to her inventing writing herself and being a goddess, but she gave up her own originality to let people be creative. However, being creative, which is a gift, is not enough. You have to pay her back by finishing your work. Hence, we are here, writing our stories and when we are finished, our stories get kept in the divine library.

I hope you find the time to be creative.

Good luck!

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] "First-order logic"
Wikipedia
https://en.wikipedia.org/wiki/First-order_logic

[4] "Type theory"
Wikipedia
https://en.wikipedia.org/wiki/Type_theory

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

[6] "Rust"
Rust programming language
https://www.rust-lang.org/

[7] "Path Semantical Logic"
AdvancedResearch – Reading sequences on Path Semantics
https://github.com/advancedresearch/path_semantics/blob/master/sequences.md#path-semantical-logic

[8] "Intuitionistic Logic"
Stanford Encyclopedia of Philosophy
https://plato.stanford.edu/entries/logic-intuitionistic/

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

[10] "Set theory"
Wikipedia
https://en.wikipedia.org/wiki/Set_theory

[11] "Avatar Logic"
AdvancedResearch – Summary page on Avatar Extensions
https://advancedresearch.github.io/avatar-extensions/summary.html#avatar-logic

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

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

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

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

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

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

[18] “cubical type theory”
nLab
https://ncatlab.org/nlab/show/cubical+type+theory

[19] “Lean”
Microsoft Research – The Lean programming language
https://leanprover.github.io/about/

[20] “Hooo”
AdvancedResearch – Propositional logic with exponentials
https://github.com/advancedresearch/hooo

[21] “Seshatism”
Sven Nilsen, 2021-2022
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip2/seshatism.pdf

[22] “Seshatic Queenity”
Sven Nilsen, 2022
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip2/seshatic-queenity.pdf