Overview of Path Semantical Logics

by Sven Nilsen, 2024

In this paper I give an overview of logical languages in Path Semantics that is non-ambiguous and explains how they are implemented in different logical systems. This overview is meant for logicians such that they can be confident what they mean when they refer to a Path Semantical logic.

IPL = Intuitionistic Propositional Logic

IPL is the most important logical language in the world, since it is the basis for many propositional logic systems and simply typed programming languages. It is also called “constructive logic”.

The name “intuitionistic” was originally intended by its creators to describe the logic as intuitive, as opposite to classical logic as seemingly non-intuitive. However, nobody today actually thinks about this logic as intuitive. Most people consider it more difficult than classical logic.

The underlying “intuition” of IPL is that it follows the semantics of proofs. For example, if I have a proof of `A` and a proof of `B`, then I have a proof of `A & B`. Here, I did not explain what I mean by proofs. One could say `p : A` and `q : B` implies `(p, q) : A & B`, however, this depends on how IPL is implemented in some logical system. IPL itself does not specify how to represent proofs and neither does it have a `:` operator by default. It only describes the rules that covers the semantics of proofs, without referring explicitly to any proof in general.

PL = Propositional Logic

PL = IPL + excluded middle

When you extend IPL with the excluded middle, you get classical propositional logic. When normal people talk about the excluded middle, they usually usually say “something is either true or false”. However, this is not correct and misleading, since the statement “either true or false” in isolation is provable in IPL without excluded middle. The reason it is misleading is because “something is” changes the meaning of the entire sentence.

A typical logician might express excluded middle using `⊢` from Sequent Calculus:

```
⊢ a | !a          for all `a`
```

Path Semanticists might express the excluded middle using `^` from HOOO EP:

```
(a | !a)^true     for all `a`
```

With other words, `a | !a` can be proved from `true`, which is true everywhere, so `a | !a` is always true. This holds for any proposition, such that one can prove e.g. `b | !b` or `(a & b) | !(a & b)` etc.

Constructive theorem proving vs brute force

A constructive proof is a program that when its type is checked, one has proved the type of the program as a statement or proposition. Constructive proofs are usually faster than brute force. The latter requires stepping through every case and counting true and false cases.

Monotonic solvers vs linear solvers

In proof search, one might use a monotonic solver or a linear solver.

A monotonic solver has a list of facts that have been discovered so far and iterates through rules to derive new facts from existing facts. Sometimes, a monotonic solver substitutes variables in a rule with variables it is trying to prove something about. For example, when one has excluded middle, which is a rule that holds for any proposition, a solver might substitute for a concrete variable which generates a new rule that only holds for a particular case. This is what “for all” means in logic. “For all” means the rule can be applied to some variable or constant. Logic usually has two constants `true` and `false`. When a monotonic solver reaches some goal, it stops. If a goal is impossible to solve, then the monotonic solver might stop when it can prove when the goal is impossible and reports an error. In other implementations, a monotonic solver might run out of memory. If no new facts are added after checking all rules, then it is common for the monotonic solver to terminate.

A linear solver is similar to a monotonic solver, except that a rule can consume facts. This is like making a choice where some facts are resources. This means, a linear solver can not make every possible choice and therefore it usually orders the rules in a sequence where some rules are always evaluated before others. The priority of rules is important for the user, so the outcome can be predicted. Linear solvers are less used in logical systems, but there exists a linear logic that is weaker than IPL. However, linear logic is not considered more important than IPL due to the language being in less use. It still has many users, not just the mainstream usage of IPL.

What makes automated theorem proving difficult

If proof search was cheap, then automated theorem proving would be easy. A user might simply use a solver, whether it was monotonic or linear, and specify the initial condition and the goal to be reached. However, since there can be many variables or constants, there might be many ways to apply rules to these variables and constants. This is how the combinatorial complexity grows, usually exponentially and makes automated theorem proving difficult.

Another reason is that a term of some type can be very complex, for example when writing a function that handles a specific case of the proof. It takes creativity and intelligence to figure out how to solve various cases in complex proofs. The trick is not only to solve a single case, but to figure out how to split the proof up into easier steps such that together all the steps cover all cases. This is very hard, but through training, people or AI can get very good at it.

Theorem proving assistants

Most of math is usually done informally by sketching out proofs or designs on paper or chalkboard. Mathematics is usually divided into people who work on proving theorems and people who work on designing theories, languages and logical systems. Path Semanticists are usually of the latter category where design is more important than proving a specific theorem. However, in both cases, people might use a computer to implement a design or to assist in theorem proving.

A theorem proving assistant is a computer program that often uses a formal language to assist the user in proving theorems. A computer can store a library of existing proofs that have been checked formally. An interface allows the user to import existing proofs or tactics to find proofs. When people prove a new theorem, they can usually share this as a library or module in the logical system. Computers are useful when many people need to collaborate on difficult proofs. Otherwise, more hours are required to write papers, publish them in journals and reviewing other people’s work, something that is already solved when integrating mathematical knowledge in a single system.

Path Semantics

Path Semantics is a wide field that spans logical foundations of mathematics to quantum functions. The emphasis is on design where tools are developed to allow a person to derive good designs and implement them in a separate system than the system where the design was developed. With other words, Path Semantics is about transferring designs across domains and make reasonable trade-offs to achieve goals of usability and efficiency.

It might be a bit counter-intuitive, but math might be used to find good designs, not just to prove theorems. This often requires high level of creativity. However, it is not easy to reduce such work into a simple set of rules. Still, Path Semantics is built on a foundation of logical languages that extends normal logical languages and grounds semantics of the theory. Since these logical languages are built up from scratch to serve this purpose, they can be shockingly different than what logicians are used to work with. Very hard problems had to be solved in Path Semantics to satisfy requirements of the theory. Standard practices in math and logic, such as the use of Sequent Calculus, might be insufficient for Path Semanticists.

HOOO EP

HOOO EP = IPL + exponential propositions

HOOO EP is one of the most important contributions to propositional logic. It introduces a new operator to logic that is written `^` or `→`. In programming languages, the type of this operator corresponds to a function pointer. Function pointers can not capture variables from the environment. A function pointer can be created at compile time, possibly with the use of other function pointers.

The function pointer is an exponential object, like a closure or lambda that represents implication in logic. The difference is that with implication e.g. `a => b`, this statement does not say in which context it holds. One can assume many things first before one says `a => b`. There is no way to prove what is assumed. Now, a function pointer e.g. `a → b` or `b^a` says in HOOO EP that if one has a proof of `a`, then this is sufficient in any context to produce a proof of `b`. It does not depend on the context.

HOOO EP is empty

It is not possible, or at least very difficult, to create a language with HOOO EP where programs preserve meaning in the sense of executable environments. This is because the rules in HOOO EP are axioms, or unimplemented. When using HOOO EP, the programs become non-executable. With other words, HOOO EP is empty.

In Continental Philosophy, there is a distinction between Negative Philosophy and Positive Philosophy. Negative Philosophy is like before the world is created and therefore a thought operates on emptiness. This is kind of how HOOO EP works. HOOO EP can prove statements about the world in advance that are impossible to prove inside the world itself when it exists over time. If you find this confusing, then think about it as philosophers coming up with this idea before it could be formalized as a logical language. It does not mean that there is an absolute difference. Philosophy can be a bit strange since we might not know what is possible to do yet, but it can produce ideas that occur in design of logical systems. Path Semanticists are unusually tolerant toward this form of vagueness and learn to live it, because the foundation is basically existential horror, yet it works.

From here on, it is going to get more and more existentially horrible, as I will show how to construct other logics for Path Semantics.

Tautological equality and power equality

In normal classical logic, or PL, a proposition is either true or false, or as I said earlier, the excluded middle can be expressed in HOOO EP as `(a | !a)^true` for all `a`. Under brute force theorem proving, one can assign 0/false or 1/true to every argument and check the proof case by case. When a theorem holds for every case, one says that the proof is `true`. Now, programmers are used to think about `true` and `false` as booleans, but the semantics of booleans is different from what we mean when we say that a proof is `true`. When a proof is `true`, we mean that it is tautological equivalent to the type `true`, a type that has a proof that can be constructed everywhere.

Tautological equality of `a` and `b` is written `(a == b)^true`. This means, in every context, `a` is equal to `b`. Tautological equality is not the same as `a == b`. In logic, `a == b` is a shorthand for `(a => b) & (b => a)`. In programming, these are two lambdas that can capture variables from the environment. When we say `(a == b)^true` we say that `a == b` can be proved from `true`, or an empty context. With other words, it holds everywhere. It turns out that in HOOO EP, one can express tautological equality in another form which is `b^a & a^b`. This is called “power equality” and the shorthand version is `a =^= b`. Power equality is logically the same as tautological equality.

PSQ = Path Semantical Quantum Propositional Logic

PSQ = IPL + excluded middle + qubit = PL + qubit

PSQ introduces a new unary operator to logic `~` called a “qubit”. Some logicians use `~` for NOT, but here I will use `!` for NOT and `~` for qubit.

The qubit operator `~` is tautological congruent. It means, when `(a == b)^true`, `~a` can be substituted with `~b`. Most operators are normal congruent that only requires `a == b`. All normal congruent operators are tautological congruent, but not all tautological congruent are normal congruent. The qubit operator is only tautological congruent, therefore not normal congruent. This is the weirdest thing that happens to logic, ever, and completely blows out standard thinking about congruence of operators. This is why HOOO EP is needed, just to deal with this weirdness.

Under brute force theorem proving, the qubit operator `~` is implemented as taking a bit vector and using it as a seed to a deterministic random generator that changes between every case. With other words, for a single case, `~` behaves consistently, but between cases it changes. As a result, there are some statements that can not be proved to be `true`, since one needs to run the check infinite times in general. Brute force never gets to prove `true` in PSQ, only the opposite when it fails for any case. Yet, one can not say when it fails that the proof is `false`, since in HOOO EP a statement `false^a` is when `a` is false in every case. The correct statement is `!(a^true)`, but people might say the proof is `false` even though this is technically wrong. It is just easier to say.

With other words, PSQ can only prove things that are false (followed with a debate about what this actually means) under brute force. However, when using HOOO EP it is possible to prove `true`.

PSQ and HOOO EP

When putting PSQ and HOOO EP together, one gets the following:

PSQ + HOOO EP = IPL + excluded middle + qubit + exponential propositions

HOOO EP is very powerful and used for lots of stuff besides PSQ. However, it was created to handle PSQ properly through the perspective of Negative Philosophy.

PSQ as sequences of bits

There is way to interpret PSQ as proving theorems about sequences of bits. Every argument to the proof, e.g. `a, b, c, ...` give the first bit of each sequence. The next bit is accessed by using the qubit operator `~`: `~a, ~b, ~c, ...`. The third bit is `~~a, ~~b, ~~c, ...`. There is no limit to how many times the `~` operator can be applied, so PSQ forms a infinite-valued logic.

The number of binary operators in PSQ increases super-exponentially with how many times `~` is applied. One often says “homotopy level” when referring to how many times it is used, although Homotopy Type Theory is a weaker theory that goes beyond propositions. A proposition is a type which any two proofs are equal. It is commonly thought of as PSQ being a simpler theory that one might get by adding axioms to Homotopy Type Theory, but this is not usually done in practice. It is more common to just use a language designed particularly for PSQ, either brute force or by using HOOO EP. Just like linear logic is a weaker theory than IPL, Homotopy Type Theory is a weaker theory, a kind of mathematical giant in the background, that PSQ lives in the shadow of. There is plenty to explore in PSQ alone, so separating these languages is often a good idea.

The Sesh axiom

In PSQ, one has an axiom called “Sesh”:

(!~a == ~!a)^true for all `a`

It means that when under brute force the random generator flips the bits by reading the value of a particular fixed positioned bit. It flips the resulting new bit vector back to preserve Sesh. One can think about Sesh as a form of excluded middle but “orthogonal” to it, which seems a bit mystical.

The word Sesh comes from the ancient Egyptian goddess Seshat, a goddess of writing, wisdom and knowledge and is related to “spirit” and “female divinity”. Negated qubit `!~a` is often related to a formal dual-Platonistic language bias that to Path Semanticists has a positive meaning in the expression of Seshatism.

'Sesh' sign of ancient Egypt - Min. of Tourism & Antiquities

Sesh and the aesthetics of spirituality

When a Path Semanticist enters a domain of language design that is so existentially that there seems very difficult to provide a clear explanation of why one choice is preferrable to another choice, one often uses spirituality to explain the situation. Spirituality has its own form of aesthetics can can not be easily explained in terms of something practical, yet there are some choices in this form that seems to make more sense than others. The Sesh axiom is a such choice, where its meaning lies beyond what can be expressed as a pure necessity. The goal is to achieve completeness, to find a balance in the language, in order to avoid having users stumbling upon very surprising features.

When `a` is the first bit in a sequence and `~a` is the next, most people do not think consciously, that this way of reasoning, is biased toward the positive value of `a`. There is no reason why not one could choose e.g. `~!a` as the next bit. In fact, this works for all proofs without Sesh if the user chooses to do this consistently. The fact that people do not do this, is because the mental distance from `a` to `~a` is smaller than the mental distance from `a` to `~!a`.

However, with Sesh, `a` to `~!a` does not work, because it is the same as going from `a` to `!~a`, the negative value of the next bit. So, Sesh provides an intuition for why users normally do `a` to `~a` as the right choice. In the sense that users do this anyway, but it would be surprising for them to discover they could do something else, removing this surprise from the language improves the aesthetic balance, even though this happens mostly as an invisible gentle touch. It is the only axiom of this kind that is required for qubit to complete PSQ as a finished product.

Path semantical quality

With PSQ it is possible define path semantical quality `~~`:

a ~~ b <=> (a == b) & ~a & ~b for all `a, b`

The quality operator is binary and a partial equivalence. There is no partial equivalence operator among the 16 normal binary operators in logic. So, the quality operator is the most important binary operator that departs from standard view of logic. Whether one adds qubit or quality first, does not matter, because one can prove the same theorems using one or the other, so one might try to write:

PSQ = PL + qubit = PL + quality

However, Path Semanticists distinguishes “quality” from “qubit” using an extra axiom:

(a == b) & sd(a, b) → a ~~ b for all `a, b`

This axiom lifts biconditions (`==`) into quality (`~~`) when the arguments are symbolic distinct. Symbolic indistinction is kind of like infinity in logic, where `a == a` is known by the reader to have symbolic indistinct arguments, yet from within logic this is not known. Logic can not “see” that `a` is the same symbol as `a`. It only treats propositions in terms of truth values.

The reason Path Semanticists think this way is because they want quality to behave like equality dynamically, such that one does not need to introduce quality explicitly over time. This leads to the interpretation of path semantical layers as moments of time and what the core axiom of path semantics really means. The core axiom can not work with a total equivalence operator that has reflexivity, symmetry and transitivity. It will only work with a partial equivalence that has symmetry and transitivity. In physics, one might think of `a == b` when they are symbolic distinct as a branch of the multiverse that adds a new assumption to history. Prior to this moment, the branch was entangled with the future branches that share its past and are metaphysical identical.

Symbolic distinction in various languages

While symbolic indistinction is somewhat a taboo in logical design, being propositional infinity and therefore “outside language”, one can use symbolic distinction to approach propositional infinity from the other side, through its negation. Symbolic distinction is by default undefined when its arguments are symbolic indistinct. There are possibly infinite number of valid axioms to introduce symbolic indistinction, for example `a` is symbolic distinct from `a & b`, so Path Semanticists just add them by need because it is a weird corner of logic that few people ever enter. One can immediately see from this that mathematics and logic can never be complete in this sense of having all the axioms.

In some logical systems, it is possible to prove that two symbols are symbolic distinct, without even producing a fact that relates it to symbolic distinctness. For example, in Avalog, one can use inequality in the rule. There is no need for an explicit symbolic distinction operator. The following is from a formalization of PSI:

```
(X, q'(Y)) :- (X, Y), (Y, X), X != Y.
```

Translated:

```
(x == y) & sd(x, y)     →       x ~~ y          for all `x, y`

(X, Y)          =>              x => y
```

Since `a == b` is the same as `(a => b) & (b => a)`, you can read `(X, Y), (Y, X)` as `x == y`.

Avalog uses Avatar Logic, where `q’` is a 1-avatar that relates X to Y. Avatar Logic uses extended binary relations instead of predicates, so it can take a while to get familiar with the notation.

Core Axiom of Path Semantics

The pinaccle of Path Semantics is its core axiom, that relates everything so far together while introducing a new idea to logic: Path semantical levels. A path semantical level is like a floor in a building, or a moment in time, depending on how one whishes to interpret it. Within each level, one can reason with normal logic. However, between the levels, to transition from one level to the next without collapsing the state of propositions by implication, one uses the core axiom:

```
(a : c) & (b : d) & (a ~~ b)     →       c ~~ d
```

Here, `:` is defined in a way such that `a` is a proposition that exists on a lower level than `b` when `a : b`. One can think about it as associating each level with some natural number and when arguments are introduced, they are introduced in a specific level so that an order can be determined:

```
a : b           <=>             (a => b) & (a < b)
```

One way to interpret it is as `a` being in the previous moment of `b` and they are related by some metaphysical identity such that things that are said about `a` might be partially said about `b`. However, they do not have the same propositional state. If `a` is `true` then `b` is `true`, since it follows from implication. So in this sense, `a` and `b` being the exact same constant `true` is like giving `true` some transcendent meaning over time. There is no known physical attribute in the world that has this property, so reality can be thought of some logical language without the constant `true` by default. This means, change over time propagates propositionally in indeterminate states.

Why quality is partial equivalence

To preserve the propositional indeterminate state while decoupling propositional values over moments that share metaphysical identity, one must operate on at least two arguments. This means, the operator that propagates across moments must be at least binary. However, this binary operator can not be reflexive, since reflexivity would propagate instantly from the earliest moment and cause things to happen without sufficient control. This is easier to see by reducing the core axiom:

(a : b) & (a : b) & (a ~~ a) → b ~~ b

This is the same as the following using the qubit operator `~`:

(a : b) & ~a → ~b

If `~a` is somehow true by default, then `~b` becomes true by default. Time has no meaning if it all happens by default as an eternal space-time without anyone experiencing the individual moments. From what we know about quantum physics, it does not have to be something that causes another thing to happen explicitly, but that each moment is still undefined in terms of what will happen next. The universe makes a choice and the observer follows moments along a branch. With other words, it is like making different choices of assumptions and by this choice, one causes propagation to new moments. Path Semantics is not our particular universe, just the simplest and dumbest possible to describe in logic with this property. The trick is to use Path Semantics by encoding knowledge into this format, exploiting the structure between path semantical layers.

To encode the observer this way is existential horror!

Yes, it is existential horror. The human brain, despite its complexity, experiences the world in a way that one can approximate reasoning as coherent moments where each moment is a kind of room for reasoning. The internal states of memory and sensory experiences are linked to give previous states metaphysical identity, transitioning into new states, not having total conservation, but only through our qualitative personal experience. Nobody actually understands Path Semantics well enough to say whether it can cover the phenomena of conscious experience, but that is not the point: The application of Path Semantics is as a language for powerful theorem proving. It is a logic.

PSI = Path Semantical Intuitionistic Propositional Logic

PSI = IPL + quality + core axiom

By removing excluded middle from PSQ, swapping qubit with quality (that means to add the extra lift axiom from equality to quality by symbolic distinction), plus the core axiom, gives the full glory of PSI. This is a constructive logic. One can use HOOO EP with PSI to prove things about stuff from the perspective of Negative Philosophy. However, there might also be languages that satisfy Positive Philosophy. It is common to use brute force with PSQ, by adding core axiom cases manually where it is needed. One can also use a logical system with HOOO EP support, e.g. Hooo.

By spending so much effort explaining what goes into the building blocks, I do not need to say much more about PSI. The rest is about explaining how to build from PSI to fill out the family of logical languages for Path Semantics. This is pretty straight forward.

PSL = Path Semantical Classical Propositional Logic

PSL = IPL + excluded middle + quality + core axiom + sd = PSI + excluded middle + sd

PSL is PSI with excluded middle and automatic symbolic distinction between propositional arguments in the same layer. In notation for PSL, one uses e.g. `(x y) (a b)` where `x y` are propositions in level 0 and `a b` are propositions in level 1. The core axiom of path semantics holds between layers in PSL. It is common to not use quality in PSL but only equality.

There is an efficient algorithm that provides exponential speedup for brute force theorem proving, by exploiting symbolic distinction and symmetries in the core axiom to check fewer cases.

Theorems in PSL can be translated into theorems in PSQ using the following technique:

(x y) PSL		**PSQ**
f(x, y) & f(x, x) & f(y, y)	=>	f(x, y)

Using PSL as if it was PSQ directly is dangerous, because it can propagate counter-intuitive things. If one does not know how to use PSL safely, then it is recommended to use PSQ instead.

EL = Existential Logic

EL = IPL + excluded middle of non-existence

EL is a logic that is stronger than IPL, but weaker than PL. It has the following axiom:

(!!a | !a)^true for all `a`

EL, or Existential Logic, has an impact on Existential Philosophy, as the name suggests. The reason is that existence can not be affirmative. I can not write on a note “I exist!” and keep it as a proof for my own existence. Am I? Am I conscious? I can write on a note “I am not dead!” and this holds true because whether I am a philosophical zombie or not, at least I am not dead like corpse when writing the note. The idea is that there is a lack of affirmation of existence under some conditions that pose some problems, but the propositional states resemble a bit the classical propositional states.

PSEL = Path Semantical Existential Logic

PSEL = IPL + excluded middle of non-existence + quality + core axiom

When adding excluded middle of non-existence to PSI, one gets PSEL. This language is at the boundary between the safe PSI and unsafe PSL and is largely unexplored today. What is out there in the fog, in the darkness? Do you dare heading into the unknown? Can this language shine some light on our own existence?

MEL = Middle Exponential Logic

MEL is not a logical language in itself, but a way to talk about what is meant by things that are neither fully true nor fully false, using HOOO EP. This is the definition of Up, Down and Mid:

up(a)	<=>	!!a | !(a^true)
down(a)	<=>	!a | !(false^a)
mid(a)	<=>	up(a) | down(a)

Understanding HOOO EP

Some things from HOOO EP that are commonly used:

tauto(a)	<=>	a^true	`a` is a tautology
para(a)	<=>	false^a	`a` is a paradox
uniform(a)	<=>	a^true | false^a	
theory(a)	<=>	!uniform(a)	

Tautologies vs paradoxes

A tautology is a statement that is always true under all circumstances. When we say that a thing is true, we usually mean that the current conditions makes it true, not that the thing is true under all circumstances. When we say that a proof is true, we usually mean that the proof holds under all conditions, so it is a tautology. People use these different meanings ambiguously, so they have to learn when one meaning is applied versus another.

A paradox is a statement that is always false under all circumstances. When we say that a thing is false, we usually mean that the current conditions makes it false, not that the thing is false under all circumstances. When we say that a proof is false, we usally mean that the proof fails for at least one case, which is not a paradox. People do not usually think carefully about what paradoxes mean and thus get confused by not applying it consistently.

You might be familiar with the Liar's paradox:

> This sentence is false

This sentence above does not make any sense whether one interprets it as a truth or a lie. Notice that we interpret the sentence accordingly to whether somebody is saying something meant to be true, or somebody is saying something meant to be false. People joke all the time, so they have to learn to interpret sentences in more than one way. Most people think about the above sentence as a paradox because it does not make sense either way. Formally: There are two cases and in either case it does not make sense, so it is a paradox. A paradox is not a statement that does not make sense. Sensibility is only one way to interpret logic. A paradox is a statement in which every case it is false.

Now, consider the following sentence:

> This sentence is true

Whether you interpret it as a truth or a lie, it makes sense! What it says affirms the interpretion either way. Some people struggle thinking about this as a lie, because when they notice an affirmation, they take it as a truth. This is a language bias that many people have. The trick is to think of the person saying this as a sarcastic joke. In either case, it makes sense, so it is a tautology.

Why nuances of tautologies and paradoxes matter

When people use natural language, under most everyday scenarios the nuances of tautologies and paradoxes do not matter. However, because people use these nuances inconsistently, the apparence of paradoxes might seem mysterious. A paradox should not be mysterious when we refer to them, but something of a clear meaning. While the thing we talk about as a paradox might in itself be complex, this is not very different than e.g. talking about a horse. Horses are complex too in ways

we can not fully express. However, when we say “this is a paradox” or “this is a horse” the meaning should be clear to the listener.

Paradoxes are often presented in philosophy classes as some kind of mystery boxes and this confused people. What makes a paradox is not a particular example of a paradox, but what all paradoxes have in common. All paradoxes have in common that there are more than one way to try understanding them, but no matter how one tries, it fails to make sense or be true. Path Semanticists learn to accept a strict definition of tautologies and paradoxes that settles these debates by not diving deep into particular examples, but applying these ideas formally in a practical way. The fascination of paradoxes does not need to go away, but one can separate usefulness of paradoxes into its own layer in language, that gets isolated from the particular structure of examples.

Uniformity and theories

When a proposition is either a tautology or paradox, it is called a “uniform proposition”. In some languages uniformity can cause an ambiguity. In ancient times, people philosophized about the rebirth of the universe and this might be thought of a sudden transformation from one extreme state to the other. The ambiguity of uniformity in some languages is an uncertainty occuring by taking the very last step, the overall state might become in superposition of the beginning and the end.

A theory is opposite to uniformity. In order to say something interesting, one must assume something that is neither always true nor always false, but somewhere in the middle. This is because from `true` one can only prove things that already are true and from `false` one can prove any statement. Neither of these cases give an interesting context to talk about. A theory is not interesting unless it is possibly false or possibly true. In science, one says that a theory must be falsifiable. However, to be more precise, nobody wants a scientific theory that is never true either, so a theory must also be true under some conditions to be useful.

Theoretically possible vs existential possible

In modal logic, it is common to add axioms for `□` (necessity) and `◇`. In some logical systems, these operators are not defined but combined with various axioms for modal logic.

When one says that something is theoretically possible, it must be either a tautology or a theory:

◇a <=> a^true | theory(a)

This definition of theoretical possible turns out to be a stronger statement than what is common in most modal logics. The more common notion of possibility is the following:

!!◇a <=> !((!a)^true)

Defining symbolic distinction up to tautological equality

I previously mentioned that there is an axiom that lifts equality into quality by symbolic distinction:

(a == b) & sd(a, b) → a ~~ b for all `a, b`

It turns out that there is a way to express this in a more general way:

(a == b) & theory(a == b) → a ~~ b for all `a, b`

To prove the first axiom as a theorem, one can define symbolic distinction as following:

sd(a, b) <=> ◇(¬(a == b))

With other words, `a` and `b` are symbolic distinct when it is theoretically possible that they are not equal. This weakens the meaning of symbolic distinction to hold up to tautological equality only. One has to be careful with a such definition since there are cases where two symbols are distinct yet they might be tautological equal, specially under some assumptions in HOOO EP.

Some logical systems might choose this approach while others might use another approach. The nuances matter logically, but whether they make sense as trade-offs depends on what the motivation and goal is for a particular logical system. Path Semanticists learn to live with this complexity.

Summary

Listing up the logical languages for Path Semantics:

PL = IPL + excluded middle

HOOO EP = IPL + exponential propositions

PSQ = IPL + excluded middle + qubit = PL + qubit

PSI = IPL + quality + core axiom

PSL = IPL + excluded middle + quality + core axiom = PSI + excluded middle

EL = IPL + excluded middle of non-existence

PSEL = IPL + excluded middle of non-existence + quality + core axiom

HOOO EP is usually added to any language where meta-theorem proving is needed.

From within languages, there are various trade-offs, however, from the way these languages are listed here, one can be 100% confident in what is meant by these languages. They have a precise meaning, which involves very few axioms in total.

Implementations can vary and differ by strengths in provability. There might be monotonic or linear solvers searching for proofs. There might be brute force or constructive theorem proving. Modal logic might be handled differently between systems. Symbolic distinction might also be handled differently.

The complexity of these languages can be overwhelming, despite their simple design. Trying to philosophize about them in detail can feel like existential horror. Yet, it works!