Qualitative Implication

by Sven Nilsen, 2025

In this paper I introduce a transitive binary operator for Path Semantics that is neither reflexive nor symmetric. It provides a weaker pre-quality operator, implying material implication, that is important for bridging the foundation of Path Semantics to Group Theory and Discrete Geometry. This operator gets associated with a standard symbol like the qubit and quality operators.

In Logic, it is a good idea to construct weaker theories relative to useful strong ones, because this enables more control in language design. People can use axioms to arrive at stronger results using a weaker model, such that the models cover both the weak and strong results.

The path semantical quality operator (e.g. `a ~~ b`) is useful because it implies propositional equality, yet it is not reflexive. It is symmetric and transitive, which is partial equivalence. Partial equivalence is important because when identifying objects in Category Theory, identity morphisms on every object in some category assumes that the object is in the category. With reflexivity, this pushes the identity morphisms to every possible category, which is not desirable. Therefore, partial equivalence allows more control in the language design process. Otherwise, it gets difficult to build logical models that are useful in Category Theory. While the core idea of Path Semantics is modeled by the core axiom, the solution to the problem that the core axiom presents results in useful math.

Due to the usefulness of the of the quality operator, in the past decade since the beginning of Path Semantics, something corresponding to material implication was so far ignored in the research. However, a late discovery in improved language design is better than never. The job of Path Semantics is not to develop new mathematical theories, but to provide building blocks such that various mathematical theories can be formalized with a minimal set of dependencies and instructions for how to implement them in programming, or guide the Path Semanticist toward good designs for custom mathematical languages. Hence, it is mainly about design, not about theorem proving. Theorem proving is often part of various disciplines. The theorem proving done in Path Semantics is about design of mathematical languages, but it also provides a framework for reasoning about programming. Path Semantics formalizes and extends standard mathematics.

The quality operator in Path Semantics is not necessarily fundamental in standard Path Semantics, but defined by the qubit operator. Logically, each can be derived from the other, so ontologically both qubit and quality can be seen as fundamental. In practice, though, Path Semanticists often prefer working in weak logics to prove results, where it is easier to start with the qubit operator. Hence, quality can be viewed as merely syntactic sugar. In some theories, such as classical models of PSL (Path Semantical Classical Propositional Logic), the quality operator is implicit in the model of the core axiom between path semantical layers.

Therefore, it is understandable that the path semantical quality operator has played a major role in the focus of the research on foundational Path Semantics since the beginning. Yet, when looking at the definition of path semantical quality, it seems surprising in hindsight, that …:

(a ~~ b) := (a == b) & ~a & ~b

… was not used as inspiration to explore this definition:

(a ~> b) := (a => b) & ~a & ~b

The syntactic change from the definition to the second, is such a minor change, that it seems as if one ought to make this discovery at some early point in the research. Still, a decade has passed without any people noticing or suggesting this direction. Sometimes, it can be the smallest steps that are hardest to find in good language design.

The `~>` symbol was also used previously, but in the form:

(a ~> b) := (a => b) & (~a => ~b)

Still, this operator was never written up properly in a paper. But, added in an ad-hoc way.

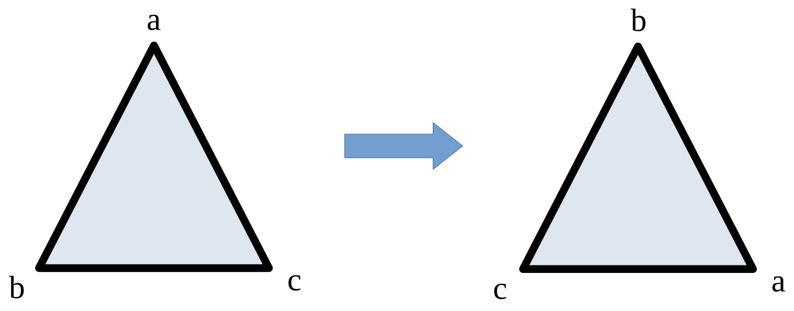
It becomes immediately clear (once motivated by Group Theory and Discrete Geometry), why the following should be used instead:

(a ~> b) := (a => b) & ~a & ~b

This operator should use the symbol `~>` as part of standard Path Semantics, while the previous used definition should be removed. The name of this new operator is baptized “Qualitative Implication”, based on the terminology of “Material Implication” for `=>` in normal logic.

Here is a reconstruction of the argument that led to this discovery:

Imagine that an edge between two points in geometry is given a propositional value. The desirable property of this proposition is that one can e.g. rotate a triangle by adding an axiom that generates the symmetry group of rotation. However, the triangle is not rotated in some embedded space. Instead, the rotation of the triangle is a self-rotation of three points `a, b, c` where point `b` moves to the previous position of point `a`, `c` to `b` and `a` to `c`:



In Group Theory, this is just the generator of rotation of the triangle. This kind of rotation is not new in mathematics. On the contrary, it is a basic idea that is also very well understood.

The problem is how to model this in logic. For example, it is desirable to use the proposition of the generator itself in the interpretation of theorems. Hence, once the generator has been proved, one “knows” that this is the rotation of some triangle. This means one can also use it as an assumption.

At the same time, it is also desirable that one can not prove e.g. that two corners of the triangle are swapped, while the third corner stays fixed. There are two sub-groups of the permutation group of triangles. A permutation group gives all possible states. Depending on which starting configuration one uses, one is limited to one of two sub-groups by the rotation of the triangle. A swap of two points while the third stays fixed would result in the whole permutation group when combined with rotation of the triangle. Hence, every possible state would be reachable from every state.

For example, starting in state `012`, one gets `120`, which repeated gives `201` and finally `012`.

The notation used here is that `012` is modeled by a list `x := [0, 1, 2]`.. This is also the identity generator, mapping every state to itself, such that `x[i] == i`. It might seem confusing at first, but lists of ordered incremented indices can be used as both states and generators at the same time.

You can also use functions of type `nat → nat` to model infinite generators. The identity function `$\text{id}_{\text{nat}}$ : nat → nat` maps every natural number to itself. By manipulating such functions, one can construct all kinds of geometry without any underlying space. A function of type `nat → T` can map from the index to the space `T`, such that rotations by indices can be thought of as being projected down to something that looks like rotation in an embedded space. When the resolution gets increased high enough, this will looking indistinguishable to the human eye from a normal rotation.

Whether you use finite lists or infinite lists, e.g. using functions of type `nat → nat`, does not matter. The essence here is that you can model lots of things with the support of Group Theory.

In the standard definition of Group Theory, one uses a binary product, which can make it difficult to imagine discrete groups in this way shown above. However, if this binary product is using the symbol `*`, then `a * b` can be thought of as applying `(* b)` as an operator to the state `a`:

(* b)(a) <=> (\(x) = x * b)(a) <=> a * b

It is also possible to use the left argument as the operator and the right argument as the state. Here, the path semantical notation of `(* b)` makes it more obvious to the reader.

When translating this to functions of type `nat → nat`, all of a sudden the semantics of Group Theory seems to make no sense. Here, a state is just a natural number and the operator maps natural numbers to other natural numbers. Still, one can use theorems from Group Theory and interpret them in this setting. The fact that it is kind of non-sense to do this is not an obstacle in mathematics.

In summary, `012` is the identity generator on the triangle. It maps every corner to itself. To describe rotation of the triangle, it is sufficient to say `120`. This has all the information required to construct the operator that rotates one step. From this operator one can construct any rotation, but only rotations in the rotation group, when not given access to other operators. So, in this sense, any generator that results in the same two sub-groups for all permutations “is” the rotation.

There are two such generators for a triangle, that gives the rotation group: `120` and `201`. Either can be derived from the other. You can define one operator in terms of the other and vice versa.

In Combinatorics, the rotation `012, 120, 201` is called a “necklace”. Necklaces might be thought of as rotations of polygons in the flat plane. For hypercubes, you need spatial rotations per axis. In this sense, there is no non-ambiguous description of the underlying geometrical figure. The advantage is that one can prove more general theorems for all symmetric groups on sets using permutations, but to interpret the theorems, one requires an interpretation of the state. Therefore, when you design a customized mathematical language, the motivation is often to get away from the abstract theory and boil it down to something that is understandable by the user. This process is applied mathematics and language design. Path Semanticists work mostly on the mathematics of what makes something understandable, by proving things about language design.

Now, with this knowledge under your belt, there is not much more to say about this part of the argument. For example, hypercubes require an interpretation of the state and you can figure out the rest from that interpretation. You can construct all kinds of operations on hypercubes. This should be seen as just practical knowledge about applying Group Theory. However, to make further progress, one has to move away from this perspective into something completely different.

The next part of the argument, which makes it more complex than just talking about Geometry, involves modeling of hyper-graphs in logic. A hyper-graph is a graph where there are not just edges, but can also contains edges from one edge to a node, or any edge to any edge. Hyper-graphs generalizes graphs. Most practical use of Geometry is for graphs, because it is very intuitive for the human brain: There are only points and lines and the rest, surfaces and cubes etc., is kind added ad-hoc onto that structure. Hyper-graphs, on the other hand, does not translate as well intuitively to how most people think. They might seem a bit strange.

For example, people are used to think about a triangle as 3 points with 3 edges. On top of this structure one adds a surface. The surface might have a front and back side, depending on from which perspective the triangle is viewed in some embedded space. This is how triangles in computer graphics work. You might have played a game where you can go “through” an object and see through the surface from one side, but not the other side. So, a triangle can have just “one side”. In practice, the rendering engine uses an algorithm that determines whether the triangle is clockwise or counter-clockwise on the screen, before deciding whether to render just one side, or use different materials depending on perspective. Obviously, there is a “front” and a “back” of a triangle, but to make such decisions, all one needs is to calculate 1 bit of information.

To model a triangle using a hyper-graph, one can start with two points `a, b` and create an edge between them `edge(a, b)`. From this edge, one can create a hyper-edge to the point `c`:

    edge(edge(a, b), c)

The hyper-graph is the entire expression. Now, there are many ways to construct such triangles. One of the first things you might notice is that one can permute the points `abc` in any order. From a state `012`, one can “interpret” this triangle as `edge(edge(0, 1), 2)`. Still, you can swap any arguments to `edge`, which creates another permutation group. When swapping `edge(0, 1)` to `edge(1, 0)`, this does not move outside the permutations `01, 10`. It is in the same group. However, swapping `edge(edge(1, 0), 2)` with `edge(2, edge(1, 0))` moves beyond the group. Therefore, there is another higher group for hyper-graphs than for permutations.

The key insight of the argument is what 1 bit of information describes the “front” or “back” when modeling triangles in hyper-graphs. At first, you might think that since there are two forms of expressions, `edge(edge(_, _), _)` and `edge(_, edge(_, _))`, this corresponds to that 1 bit. However, this is wrong. It is not essential to keep the same amount of sub-groups in the higher group for hyper-graphs as for the two sub-groups for triangles in the permutation group. Extra symmetry in higher dimensions is mostly harmless. The actual essence here is that one is not allowed to flip any two points, while a third point stays fixed. When you swap from the first form to the other, this is actually a swap of arguments, which are kind of like points for a hyper-graph. For a moment, just forget that `edge(a, b)` can operate on edges. Think about `a, b` as points. This technique makes it much easier to reason about hyper-graphs. You can always generalize the result later.

An advantage of hyper-graphs as a language is that all expressions are propositional. This means that when modeling hyper-graphs in logic, it suffices to simply use propositions! There is a one-to-one correspondence between hyper-graphs and constructive logic. Now, since the Curry-Howard correspondence gives a one-to-one map between Type Theory and logic, it means that one can use Type Theory to model hyper-graphs and vice versa. You can start with one thing and model the other. Therefore, hyper-graphs are very interesting from a type theoretic perspective.

The question now becomes: How do we model hyper-graphs in logic such that they can be used in Type Theory, in a similar way to use of language that corresponds to Group Theory, Geometry and Topology? This is a complex question, where Path Semantics produces a simple answer: `a ~> b`.

Consider that quality implication `~>` is defined using the qubit operator `~`:

(a ~> b) := (a => b) & ~a & ~b

All the complexity of Group Theory, Geometry and related fields such as Topology, is reducible to normal constructive logic extended with the qubit operator! This is a huge amount of knowledge packed down to a very simple operator. So, there is really just one operator and the rest is sugar.

You see, the past decade spent on researching Path Semantics was entirely missing this direction. It is not like Path Semantics is without useful results: There is so much stuff coming out of Path Semantics that very few people are able to keep it all in their head. There is simply too much theory.

Yet, Group Theory and Topology are two major fields that inspired the work on Path Semantics. The idea that Path Semantics was useful for understanding these fields was not missed at all.

A triangle `abc` is modeled by:

(a ~> b) ~> c

To give a rotation of a triangle, one uses:

(a ~> b) ~> c → (b ~> c) ~> a

Under HOOO EP, this statement is an exponential proposition.
Therefore, one can express “rotation of a triangle” as a propositional value.

For a quad `abcd`, one can use the following:

(a ~> b) ~> (c ~> d)

Just like for the rotation of a triangle, one uses the generator for quad rotation:

(a ~> b) ~> (c ~> d) → (b ~> c) ~> (d ~> a)

There is another way to give the same amount of information as a quad:

((a ~> b) ~> c) ~> d

However, since this connects a triangle with a point, one can think of it as a tetrahedron instead. Most people think about tetrahedrons as very different from quads, but since it is basically the same amount of information in there, you can choose how to interpret that information by yourself.

I think that it is more intuitive design to stick with quads and tetrahedrons in their respective obvious forms and not try to make it more difficult and confusing by conflating them. The fact that there is a natural interpretation is very interesting, particularly from a Path Semantical perspective.

To model a triangle where two points `a, b` can be swapped, while `c` stays fixed:

(a ~~ b) ~> c

This follows from `(a ~> b) → (b ~> a)` can prove `a ~~ b`. When you assume this and the above, you can prove `(a ~> b) ~> c` and `(b ~> a) ~> c`. We can do this as a useful exercise.

The path semantical qubit operator `~` is tautological congruent, so one should use this assumption:

```
all(a ~> b → b ~> a)^true                A
```

Here, `all` lifts all sub-expressions of free variables (those that are not symbols) to for-all.

This makes it possible to instantiate this rule for all `a, b` using A:

```
a ~> b → b ~> a
b ~> a → a ~> b
```

One can prove the following theorems without making any assumptions:

```
all(a ~> b → b ~> a) → (a ~> b) == (a ~~ b)
all(a ~> b → b ~> a) → (b ~> a) == (a ~~ b)
```

Using HOOO EP and the above, one can easily prove the tautological equivalences from A:

```
((a ~~ b) == (a ~> b))^true
((a ~~ b) == (b ~> a))^true
```

The proof uses power transitivity, `b^a & c^b → c^a`, which is a bit tricky to prove from the HOOO EP axioms, but since it holds, one can just use this as part of the step in the proof.

Now, one can prove the following without any assumptions:

```
all((a ~> b) & (a == c)^true → (a ~> c))
```

In Hooo, you just prove a function with the type `(a ~> b) & (a == c)^true → (a ~> c)`:

```
fn f : (a ~> b) & (a == c)^true → (a ~> c) { … }
```

Notice that Hooo at the point of writing has no support for `~>`, so you can use `qi'(a, b)` instead. The symbol `qi` stands for "Qualitative Implication" and must declared using `sym qi;` before use.

If you want to make the proof clearer, you can lift it into `all`:

```
let f2 = f() : all((a ~> b) & (a == c)^true → (a ~> c));
```

There is no need for explicit lifting in Hooo, because you can just instantiate, but the point is this:

```
let f3 = f() : ((a ~~ b) ~> b) & ((a ~~ b) == (a ~> b))^true → ((a ~> b) ~> c);
let f4 = f() : ((a ~~ b) ~> b) & ((a ~~ b) == (b ~> a))^true → ((b ~> a) ~> c);
```

This enables substitution using the earlier tautological equivalences, so you get two triangles:

```
let tri1 = … : ((a ~> b) ~> c);
let tri2 = … : ((b ~> a) ~> c);
let r = refl(tri1, tri2) : ((a ~> b) ~> c) & ((b ~> a) ~> c);
return r;
```

That's all! Also, by enabling swapping `a ~> b` with `b ~> a` for all `a, b`, you get all permutations.

With other words, one can add axioms for various group generators to the theory of `~>` and prove theorems about them. For example, that they are indeed groups. This can be non-trivial. The identity element of a group obtained from a generator `g` is the fix-point `$g^n(x) == x$`. Usually, one uses `e` for the identity element, so `$e <=> g^n$`. When `n` is known, one can find the inverse of `g`, simply by using `$g^{n-1}$`. The generator is both a state and an operator, but also a representation of the entire group, the identity element and its inverse. The binary product is simply function composition:

f * g	<=>	f . g	left handed version
f * g	<=>	g . f	right handed version

If you are working with permutations in Dyon, then you can use `sift`:

f * g	<=>	sift i { f[g[i]] }	left handed version
f * g	<=>	sift i { g[f[i]] }	right handed version

In any case, you have to pay attention whether you use left or right language bias.

Usually, there is no point in formalizing everything in e.g. Hooo. That would be very hard when the groups become very large. You have to figure out how you want to work with them efficiently.

If you use `$g^{n-1}$` for the inverse in computations, then this is going to be very slow for large groups. Instead, the language, to be usable, should make this transition by substituting for a more efficient operator. Most of the time, people are not interested in doing this manually. This is part of the challenges of mathematical language design.

Qualitative implication is neither symmetric nor reflexive. It is only transitive. This fits a hierarchy:

a ~> b	transitivity	
a ~~ b	symmetry + transitivity	partial equivalence
a == b	reflexivity + symmetry + transitivity	equivalence

Transitivity means that if you have `a ~> b` and `b ~> c`, then you can prove `a ~> c`.
By adding axioms for symmetry and reflexivity, you can prove `((a ~> b) == (a == b))^true`.
So, you can walk down the hierarchy from qualitative implication toward equality in steps.

The problem with material implication is that it is reflexive, which is a bit too strong, sometimes.

By using qualitative implication you can create a weaker theory and let users add their own axioms. You give them more control over whether they want to model something that e.g. rotates, how it rotates and what symmetries the mathematical objects in their theory should have. Good language design is not about making all future decisions on behalf of the users. It is about letting users decide what they want to do.

The role of Path Semantics is not to put a dogmatic burden on people, like Set Theory where everything must be a Set. It can provide building blocks where people can add a few axioms here and there to shape the theory into almost anything they would like to model. So, Path Semantics is trying to get out of the way of the users, but providing support and assistance where needed.

There is no point in proving all theorems about e.g. Group Theory in the foundation of Path Semantics. That would take too long time and not be as useful as building efficient tools to actually just work with Group Theory for various domains. However, you should not just accept Group Theory without questioning it. That is why Path Semantics is needed: To show it makes sense.