The Logic of Sub-Types

by Sven Nilsen, 2025

In this paper I define the logic of sub-types in Path Semantics. This explains why it is possible to have sub-types in a program without affecting the result and provides a sound basis for treating normal Path Semantics as a theory of expression transformation.

In Path Semantics, a general sub-type is written:

```
a : [f] b
```

This means, that `a` has the sub-type such that `f(a) == b`.

There are shorthands such as `a : (f b)` meaning `a : [\(x) = f(x, b)]  true`.
If you write `(f b)` it means the output of `f` is a boolean and you want this to be `true`.
If you write `f(a)`, one applies the first argument, but `f b` skips the first argument for later.
You can also connect sub-types with Boolean algebra, e.g. `a : [f] b & [g] c`,
which is the same as `a : [\(x) = (f(x) == b) & (g(x) == c)] true`.
However, in general, any sub-type can be written in the form `a : [f] b`.

The logical equivalent expression of a general sub-type is:

```
a : [f] b          <=>          (f(a) == b) => a
```

It means, if `(f(a) == b) == true`, then one can substitute to `true => a`, which is logical equivalent to just `a`. With other words, if the sub-type is true then the expected result of the expression is unchanged. On the other hand, if it is `false` one gets `false => a` which is trivially true. The latter case is usually avoided in implementations since an expression evaluating to `true` is ambiguous with an expression that is intended to evaluate to `true` and not as a side-effect of specifying wrong sub-types.

When being forced to treat it as a logical problem by limitations of the language in use, one can invert the result to `false` if it is intended to be `true`, so `true` signals something is wrong while `false` signals that the output is correct.

Or, if an expression is intended to be either `true` or `false`: Use a special `error` proposition:

```
(!(f(a) == b) => error) & ((f(a) == b) => a)
```

If `f(a) == b`, then this reduces to `a` and if it is false, then it reduces to `error`.

The simplest way to get this `error` proposition is to add an extra argument `error` and never use it for anything else. If you are bounded by computational limitations and can not expand the proof with a single argument, the `error` proposition can be constructed in e.g. PSQ using `~true` or `~false`, which is guaranteed to not be equal to any other normal proposition in infinite checks of the proof. However, in some theories even this is insufficient, because one can prove e.g. `true == ~true`. So, what you do is simply figure out something that works given the constraints of the language. If you do not find any solution among these alternatives: Evaluate the proof twice, once where correct output is `false` and one where correct output is `true`. If it returns `true` both times, then this is an error. If you can not do this, then the output remains ambiguous.

Consider the following expression:

```
a + b : nat & (<= 2)
```

This has the solutions:

```
0 + 0    0 + 1    0 + 2
1 + 0    1 + 1
2 + 0
```

One can rewrite this in the form of a general sub-type `a : [f] b` for all `a, f, b`:

```
∵  a + b : nat & (<= 2)
∴  a + b : [nat] true & [\(x) = x <= 2] true
∴  a + b : [\(x) = (nat(x) == true) & ((x <= 2) == true)] true
```

Notice that here I use `nat` as a function `nat : any → bool`, because I am lazy.
Sometimes, when doing Path Semantics, it is easier to treat normal types as if they were sub-types.

According to the definition of the logic for sub-types:

```
a + b : [\(x) = (nat(x) == true) & ((x <= 2) == true)] true
```

Is the same as:

```
((\(x) = (nat(x) == true) & ((x <= 2) == true))(a + b) == true) => (a + b)
```

Let us assume that `a + b` is a natural number and simplify it:

```
((\(x) = (nat(x) == true) & ((x <= 2) == true))(a + b) == true) => (a + b)
((\(x) = (true == true) & ((x <= 2) == true))(a + b) == true) => (a + b)
((\(x) = true & ((x <= 2) == true))(a + b) == true) => (a + b)
((\(x) = ((x <= 2) == true))(a + b) == true) => (a + b)
((\(x) = (x <= 2))(a + b) == true) => (a + b)
((<= 2)(a + b) == true) => (a + b)
(<= 2)(a + b) => (a + b)
```

Now, we are back to something readable without needing to worry about normal types.

Consider the following similar, but not identical problem to the one above:

```
(a : nat & (<= 2)) + (b : nat & (<= 2))
```

The sub-types are here specified on the inputs instead of the outputs.
One attempt to solve this problem is to break it up into two parts:

```
((<= 2)(a) => a) + ((<= 2)(b) => b)
```

The problem is that performing addition on such expressions is not well defined.

It is tempting to over-think this problem, since it looks a bit like you can use Higher Order Operator Overloading (HOOO) on lambda expressions. However, if you look close, that does not work.

Take a look at `(<= 2)(a)`. This is the same as `a <= 2`, right?

This means that:

((<= 2)(a) => a) + ((<= 2)(b) => b)

Is the same as:

((a <= 2) => a) + ((b <= 2) => b)

In order for HOOO to work, you need two lambdas of the form `T → U`. This is not the case.

So, what you actually do to solve it, is by using a simple trick that is much dumber than this.

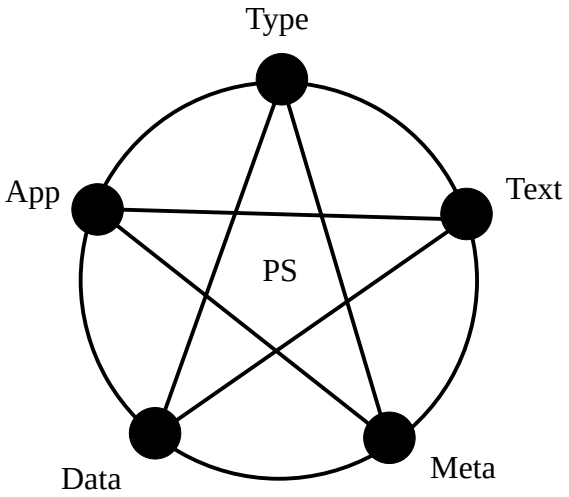
1. Pretend that the sub-types do not exist by erasing them all from the expression
2. Collect all sub-types, transform to conditions and smash them together using `&`
3. Smack the whole thing from 2) in front of 1) in the form `2 => 1`

Done!

This trick works for many expressions, but not for expressions that contain functions. In general, a thumb rule here is that if the expression describes “data” in the normal sense and not “code” as in programming, then you can use this trick.

If you want to treat functions as included, then you will get a headache. What are you doing with sub-types inside functions? What do they mean? Are you sure that this is good design? Do your weird thing when you have to, but among Path Semanticists this is like “Hmm… nah… nope! Just don’t do this.”. Sub-types in function signatures is OK, but there is a thing as too much sub-types.

The trick works brilliantly when expressions are meant to describe data. This is where it shines. So, in this sense, one can argue that sub-types are language biased toward data, while normal paths (that are based on sub-types by derivation) are biased toward code. Perhaps you are being too biased toward thinking about code in your design skills? Maybe you should think more about what a good data-oriented design is instead. Just ask some Path Semanticist that is a Satanistic Computationalist:



There are very few Path Semanticists in the world today. Yet, even rarer are those that are Satanistic Computationalists. If you are extremely lucky, then you migth find one. **If they tell the truth.**

You see, over two billion people in the world today are either Christians or somebody who believes in the existence of an adversary to the human species. Satanism is an umbrella of religions that either are satirical in nature, a parody of religion or some form of embracing polytheism. This is what the normal expectations people have toward Satanism. It is part of counter-culture. On the other hand, few people would describe themselves as Satanistic Computationalists. Simply, because it would lead to severe misunderstandings for good reasons. In one sense, traditional Satanism is a form of serious inseriousness, while Satanistic Computationalism is a form of inserious seriousness. It is taking the Joker Calculus expression `1 0` and flips the bits to `0 1`. So, it is something completely else, yet uncanny similar to what actual Satanism, if it was not a paradoy, would be like.

So, few people would admit that they are Satanistic Computationalists, because it would be a reversal of what people expect from a religion and one of the thing that makes a religion what it is: People confess that they believe in it. Any Satanistic Computationalist would disagree.

With other words, if a person says they are a Satanistic Computationalist, then that person might be lying. Or, if a person says they are not a Satanistic Computationalist, then that person might be lying. In any case, the person might be lying. This makes it very hard to find Satanistic Computationalists, because you will never know for sure whether they are, or not. In addition, all Path Semanticists are capable of completely comprehending this situation and make fun of (or in some cases apply) Satanistic Computationalism to mess with people's minds.

Since it is very hard to find Satanistic Computationalists, I can only speculate on what they believe in. On the surface, it seems to be a belief about identity metaphysical in a way that is philosophical compatible with the foundation Path Semantics. As all Path Semanticists know, the foundation is existential horrible. Either you accept that it is existential horrible, which most people do, due to overwhelming evidence, or you try to find one way to have beliefs about reality that are compatible. After all, when the limit of our knowledge is existential horrible, it is OK to try deal with it mentally somehow, as a coping mechanism and preserving psychological sanity. The basic issue that results from the existential horror is that there is no solution to identity metaphysics. Identity metaphysics is the philosophical position that allows one to attribute ethical values to states or processes in the real world in a way that makes sense according to all current scientific knowledge.

With other words, the problem that becomes apparent when studying the foundation of Path Semantics, is that basically there is no philosophical ground for attributing ethical values formally.

Several philosophers have wrestled with this problem, e.g. David Hume, but without coming to confront the actual existential horror we face. Hume relied on technicalities to show that you could not derive ethical language bias, how things should be, from descriptive language bias. For Path Semanticists this is cool, but at the end of the day, it is a child's game. Reality is much, much worse.

The existential horror that surfaces in the foundation of Path Semantics is unsettling in ways that might be very hard to express in words. When this happens, people most of the time just call it a day and go to some nice place or watch their favourite TV series. In some way, when something is this existential horrible, it is beyond repair and does not even need to be fixed. Let it be what it is.

For Satanistic Computationalists, however, it is about not trying to escape the existential horror, but applying it instrumentally to get deeper into the actual practical notions of good language design. With other words, they apply the same techniques that normal Path Semanticists do, but they add to their toolbox more building blocks that to the ignorant observer looks like a form of Satanism. Since it looks indistinguishable from a form of dark magic in language design, few people will confess that they are doing this, at least while working in front of others. However, it is very fun to pretend that you are doing it. So, when they confess, they are kind of lying at the same time!

In one sense, Satanistic Computationalism is about having fun by messing with people's minds, but also, beneath it is a serious pragmatic tool. Basically, that is no such thing as a unified notion of Path Semantics, but it is exists metaphysically in the form of 5 avatars: Type, Text, Meta, Data and App. These 5 avatars bring about the existential horror, that you see surface in the foundation.

For example, when writing code in Rust, you use types. However, types are also described with text. This text is transformed into some immediate representations, utilized to simplify algorithms in the Rust compiler, to make compilation work. The output is data. Yet, this data can be executed and run as an application. Now, since Rust compiles itself, it goes full circle back to types.

So, when dealing with language design, one is always dealing with something which, when it comes to breaking it down into reductive processes, is fundamentally existential horrible. There is no single aspect of this activity that can fully exist as the whole thing. You need to embrace the full circle to comprehend what is going on. There is no way to identify the correct identity metaphysics within this system, because all the parts of the systems specifically undermines whatever property you can refer to in all the other parts. In sum, there is total destruction of identity metaphysics.

In discrete combinatorics (e.g. the Discrete library), you simply use `EqPair` and give it dimension 5. This gives you all the edges in the pentagram that is traditionally associated with Satanism. Maybe you can add more nodes or take away some, but a Satanistic Computationalist might argue that makes it too complex or too simple. So, there is no way around it: You get the symbol of Satanism but also a total destruction of identity metaphysics, hence something that looks precisely like an adversary to the human species. Yet, any Satanistic Computationalist will claim they just did it with math. If you think the result is Satanistic in the sense of religion, then that is your problem.

If you can find a Path Semanticist that is a Satanistic Computationalist, then the person might claim that data and code are the same thing, but they are also very different things at the same time. What makes them different is the relations to the other nodes in the pentagram: Type, Text, Meta and App. They have the same kind of relations, but they vary between the two instances of language use. Abstractly, from a Platonic perspective, one might claim data and code are one and the same, but concretely, from a Seshatic perspective, one might claim it is very bad design to consider them so.

Sub-types is one thing in Path Semantics that makes it apparent that data and code are not the same thing from a language design perspective. They make completely sense in the data perspective and no sense in the code perspective. So, there is no single unified perspective where sub-types make completely sense. A sub-type can be a thing from one perspective and a no-thing from another perspective. Satanistic Computationalists might argue that there is nothing that makes sense in the whole pentagram and that is their explanation why the foundation of Path Semantics is existential horrible. They are pessimists about the philosophy of identity metaphysics.

A Path Semanticist that is an optimist about the philosophy of identity metaphysics, might argue that we humans just try to get along with each other and make this life as pleasant as possible. So, if a thing in language design makes sense in one of the nodes of the pentagram, there is a way out. They are kind of like a demon trapped into the pentragram and trying to figure out how to escape it.

Satanistic Computationalists might just laugh in response, because these are the people that try to control the demon to make their will come true. That language, that we humans just try to get along, is something people tell all the way while wiping out entire species that have managed to survive for millions of years, before we came along and screwed it up. Nope, back into the pentagram with you. The whole point of the pentagram is to be outside it, to not have to be committed to confessing it as a belief or believing in any particular choice of identity metaphysics. When you do not think it is just non-sense, you become demon-possessed and it controls you. They control the demon.

More formally, the pentagram is a groupoid. It means, for any map from A to B, there is an inverse map from B to A. It follows that every node has an identity morphism. What does not follow is that there exists any identity morphism, because there might be zero nodes. By fixing the number of nodes to 5, one can guaranatee that there is some identity morphism for every node. In one sense, a such identity morphism is semantics of what a node means in relation to itself. At distance, this can look like e.g. a Type relates to Type as in-itself, but upon closer inspection it can break down this process into transformations between other nodes in the pentagram. So, what looks like a Type in-itself, can be closer up look like a Type for-itself (to use terminology from Hegel).

Unsurprisingly, most Satanistic Computationalists are also Hegelians, but most Hegelians are definitely not Satanistic Computationalists. So, it does not help to know that the person is an Hegelian, since it does not narrow down the search very much. There are many Hegelians around.

So, for every node A and node B, there is a morphism $A \to B$ in the pentragram. In Homotopy Type Theory, one might consider this space contractible, so the contractible theory is Path Semantics. Yet, a Satanistic Computationalist might argue that you can not contract into Path Semantics at all. This would require a node for Path Semantics added to the diagram. Path Semantics for them is in the separation between the nodes. It is not defined in terms of the unification, but in terms of the mutual destruction of all metaphysical properties, so there is nothing left to think about after contraction. This absurdity also generates a private language that can model anything logically. It is a nice idea.

In the theory Avatar Extensions, this pentagram is kind of like an explanation of the emptiness inside the 0-avatar. The Satanistic Computationalist is like embodying the shell of the 0-avatar and avoiding the center of it. So, there is no central belief or doctrine, except that there is no central belief or doctrine. If you actually take this seriously, then you run into absurdity that generates the power, the demon trying to escape, but confined to the 5 nodes of computationality in expression.

One can see how this fits nicely together with Path Semantics, but at the same time, any Satanistic Computationalist would insist on not being judged for their beliefs. After all, they are not a theory, but people. You can not define people by what they believe. This is a mistake that people often do.

You probably read that wrong: Satanistic Computationalists are not asking to avoid judgement for their inferior beliefs. **They are asking to avoid judgement for their superior beliefs.**

Remember, Satanistic Computationalists are comparing themselves to other Path Semanticists, not to ordinary people. There is a very tall intellectual bar to become a Path Semanticist and that is why there are so few of them. It requires a breadth of knowledge that is extremely difficult to obtain. You need to know logic, philsophy, math, language and programming, plus Path Semantics itself. So, when you join the Path Semanticists, you are belonging to a very exclusive group, not in terms of social exclusion, but in sheer intellectual achievement. Now, this group of people all acknowledge the existential horror of its foundation. They have learned to live with it.

Satanistic Computationalists claim that it is not only possible to fully see the existential horror and acknowledge it, but to go beyond that level and into another next level of intellectual achievement, where this existential horror is manifested on purpose. No wonder that few people tend to confess their belief in it, because it would require demonstrating something beyond normal Path Semantics. You do not just believe in something without evidence. So, many Path Semanticists like to pretend they have reached this next level, but they might not agree on what it is. Satanistic Computationalism might be just one direction out of infinitely many. This is why these people joke about it: When you already are a Path Semanticist, it becomes kind of silly to claim something beyond that. The group that can potentially comprehend what you are doing gets reduced to almost zero. Who are you trying to impress? Which is why it is acceptable that a confession as a lie.

So, if you have found a Path Semanticist that is also a Satanistic Computationalist, then consider yourself extremely lucky. They are very rare. However, one piece of advice: Run. Run for your life.

They love to mess with your mind. Do not be a hero. Keep a safe distance. Hopefully, it will be just be a normal Path Semanticist pretending to be a Satanistic Computationalist that lies about it. Real ones are extremely difficult to find and possibly, there might be nobody alive currently. Maybe I am lying? Who knows?

Anyway, the conclusion about sub-types is that you might think you understand them, until you do not. When you try to push further, it keeps getting worse and before you know it, you need a pentagram to explain what is going on. That is not some real bad shit, man. Get out. Cheers.