How to Think Abstractly With Path Semantics

ILLUSTRATED
by Sven Nilsen, 2019

“Path Semantics is a powerful tool for thinking.”
- Sven Nilsen, 2019

Some day, you will come over an idea that is so great that it will change your life. However, to reach that idea, you need longer arms. Much longer arms. You need to grow intellectually. You need to reach into the unknown, and grab it! But how?

I will teach you.

Pay attention.

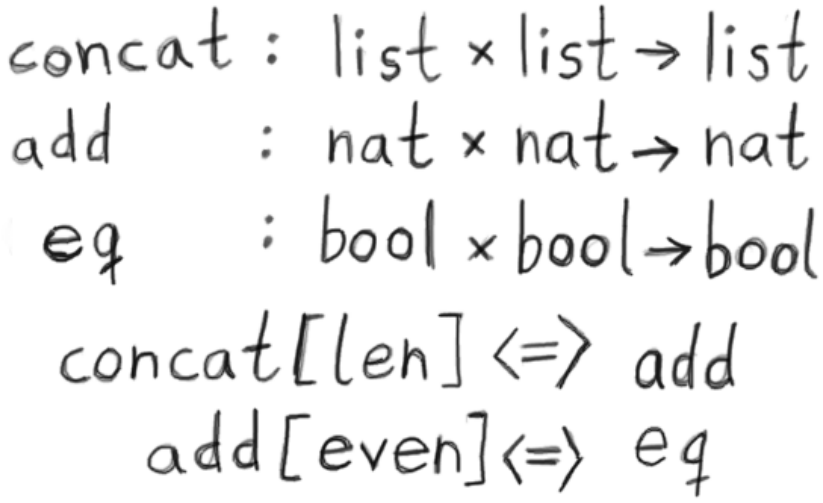
Listen.

…

Listen to the quiet silence within your mind.

CONTINUE SCROLLING TO THE NEXT PAGE

Path Semantics[1] uses notation like this:

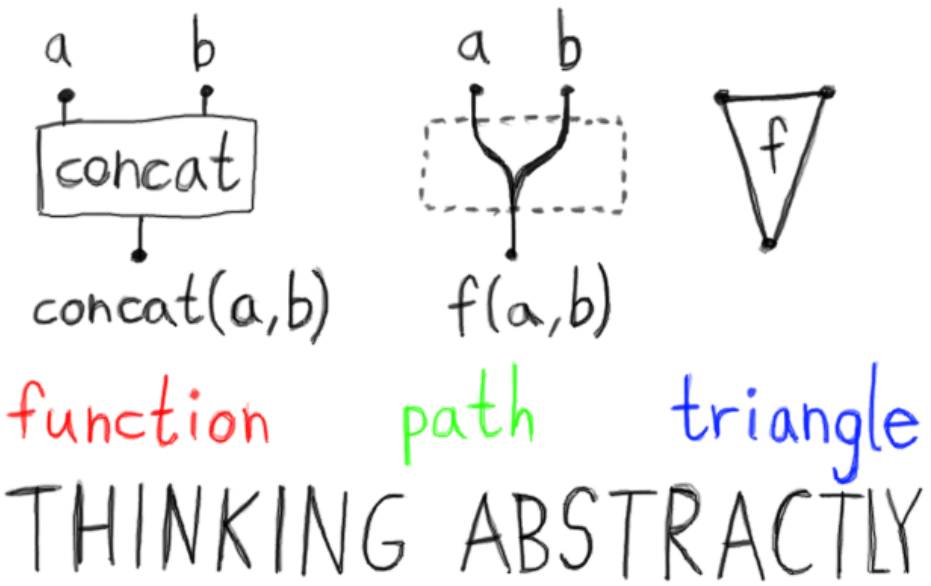


What does this mean? What is the intution that you can use to understand it?

- I might understand that `concat` is a function that takes two lists and puts them together.
- I might understand that `add` is a function that adds two numbers.
- I might understand that `eq` is a function that checks two booleans for equality.

However, I might not understand why I am shown this example, or why it makes something powerful.

There are many ways to think of functions, but here is a particular useful one:

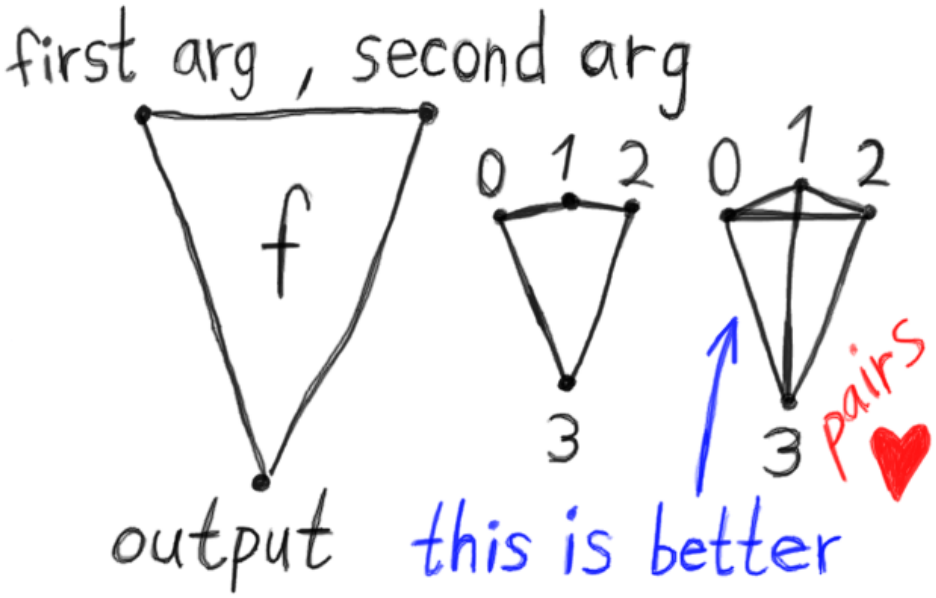


Many people think of functions like boxes which you put something in and get something out. However, if you could see through the box, you would see some wires that connects the input to the output. These wires must exist in order for the function to work. These wires also exists mathematically: They make up the machinery in abstract spaces such that mathematics makes sense.

Another way of thinking is that a function is like a triangle. This way of thinking is more abstract. We are hiding the stuff inside the box, and think about the function only in terms of where we put stuff in and where we get stuff out.

As you might know from 3D programming, a triangle can be used as a primitive building block to construct approximations of any geometric shape. There is something similar going on when you work with functions represented this way.

The two upper points of our triangle-function are the arguments, the lower is the output:

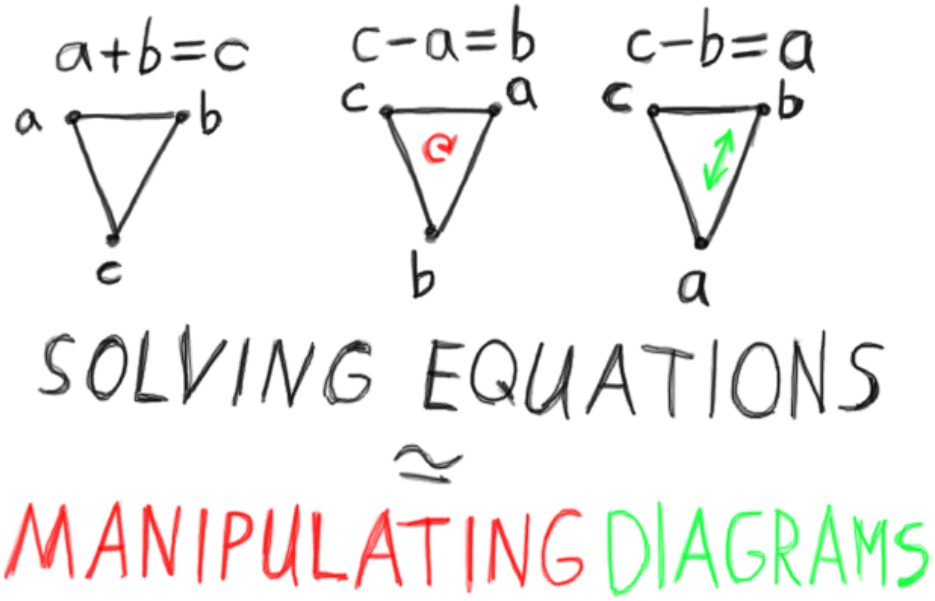


If there were 3 arguments instead of 2, then it might be generalized to a 4-gon. However, that is a bit weird: Why do you only connect the first argument and the last argument to the output?

A better way is to draw a line for every pair.

I love pairs. Pairs are very beautiful mathematical objects. Pairs are important in a field called “discrete combinatorics”. Although algorithms in this field are difficult to write down correctly, you can use the “discrete” library[2] to do this for you. It uses Rust’s type system to infer the correct algorithm, all you need to do is to define the space you want to use.

A function can be written as an equation, so why not do the following trick?



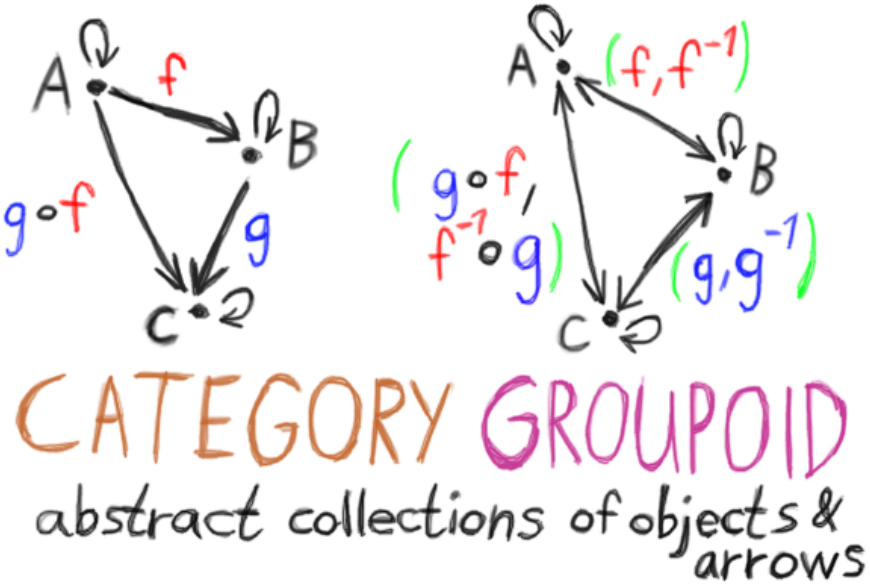
Since one can manipulate a triangle, or pairs between some points, one can also think about it as manipulating an equation. Each operation gives us a new function. The diagram means “the upper points are arguments in the order we call them and the point that is down is output”. So, when we rotate or swap the points, we can think about it as solving for the output variable in terms of the input variables.

Assume that you have some algorithm that can solve an equation automatically. You just program it with the information from the triangle, and it outputs the solution of the equation.

For example, you can create a such algorithm using the “generic linear solver” library[3]. It is an automated theorem prover that does the math for you, and after some programming, you can just sit back and think about what to do at a higher abstraction level.

Since this is the only information required, it means that manipulating such diagrams are actually equivalent to solving equations. This is what the `≃` symbol means. It is not the exact same thing, but for every operation you can do in one system, it can be translated into an operation in the other system.

In mathematics there is a deep pattern described by “Category Theory”[4]:



A Category requires that for every arrow from one object A to B, and an arrow from B to C, there must be an arrow from A to C. In addition there is an arrow from every object to itself.

In functional programming, the arrow from A to C can be constructed with function composition.

A Groupoid[5] is a Category that has invertible arrows. For every arrow from A to B, there is an arrow from B to A. Every arrow that goes from each object to itself is its own inverted arrow.

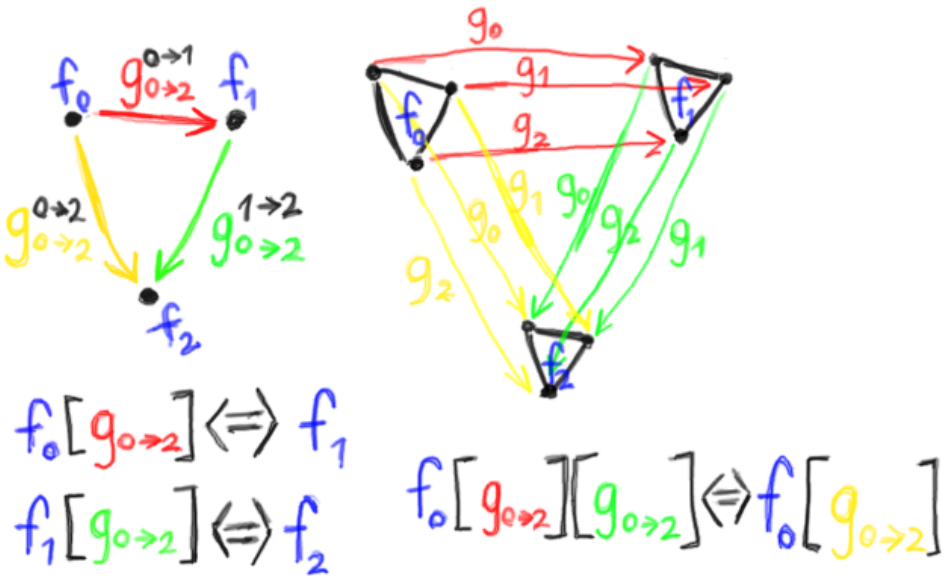
In discrete combinatorics, if you use the “discrete” library, a Category can be thought of as a space where we want to store a number for each direction between pairs. This is the `NeqPair` space. For a Groupoid, we only need a single number for pairs, so this is just `Pair`.

For the information stored in arrows from every object to themselves, we can count the number of objects and use an array. Discrete combinatorics is often used to find upper bounds on various problems of complexity. Separating this information from the pairs is good, because it allows easier compression of the space from symmetrical information of the arrows, getting a lower upper bound.

Since you could store the information outside the diagram structure, e.g. a tuple in a `HashMap` in Rust, it means that it is not hard to get a Category from a Groupoid, from a programming perspective.

However, in mathematics the distinction is very important for which proofs that are sound, because if you have a Groupoid, it implies a Category, but a Category does not imply a Groupoid, so mathematicians have to be much more careful.

Now, by using the idea of functions as a kind of triangle, or pairs, one can draw the following:



Notice that in the left diagram, functions are treated as points, but on the right side, they are triangles. Instead of a single function between functions, one creates a functor[6] using a tuple of functions, which has one element function for every input and output.

For example:

$$f[g_{0\to 2}] <=> f[g_0 \times g_1 \to g_2]$$

In generic path semantics this functor is written `$g_{i\to n}$` or `$g_{j\to m}$`, e.g. `$f[g_{i\to n}]$` or `$f[g_{j\to m}]$`. This notation is called Nilsen-Cartesian Product Notation[7], designed such that indices can be erased or added easily.

When people learn Path Semantics, they often wonder where Category Theory has gone. Where is it?

The answer to this question is of course that Category Theory is built into Path Semantics, so you do not have to worry about it! If you compare the diagrams above with the previous diagram example of a Category, you can see the same pattern.

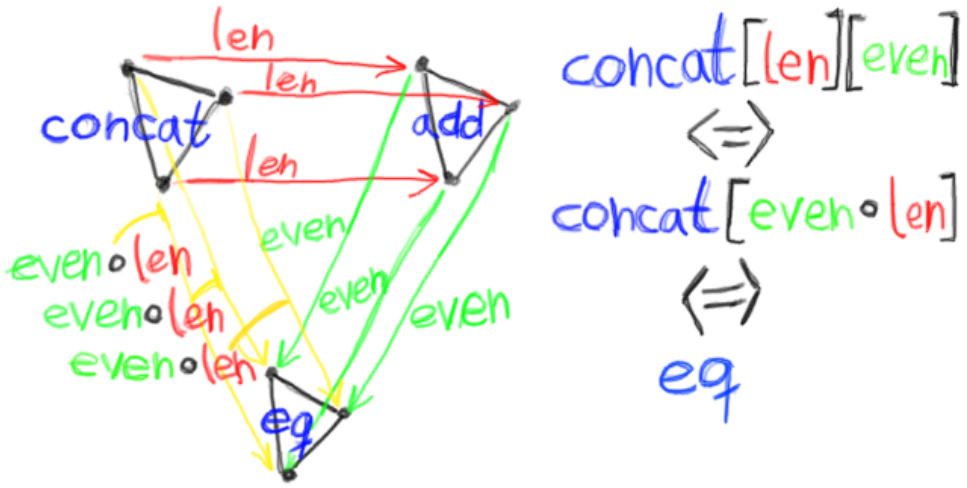
The only thing I left out in the illustration above is arrows that go to the same object.
In Path Semantics, these arrows are constructed with the generic identity function:

$$id_T(x : T) = x$$

Path Semantics encodes the same mathematical ideas of Category Theory in the language of functions. By interpreting functions in different ways, one might get as much power from Path Semantics.

When people compare Path Semantics with Category Theory, they often forget that they are designed for different purposes. Category Theory was designed by mathematicians for mathematicians. Path Semantics was designed by a programmer (myself) for programmers (like myself).

When we write stuff like `concat[len][even]` in Path Semantics, we are chasing diagrams:



If the functor contains only a function, it is “lifted” to a tuple that copies it for every input and output.

concat[len] <=> concat[len × len → len]

The `eq` function can be thought of as making a prediction about the `concat` function.
It tells us that when you concatenate two lists, the even-length property of the output can be predicted from the even-length property of the inputs.

It is common to write proofs like this:

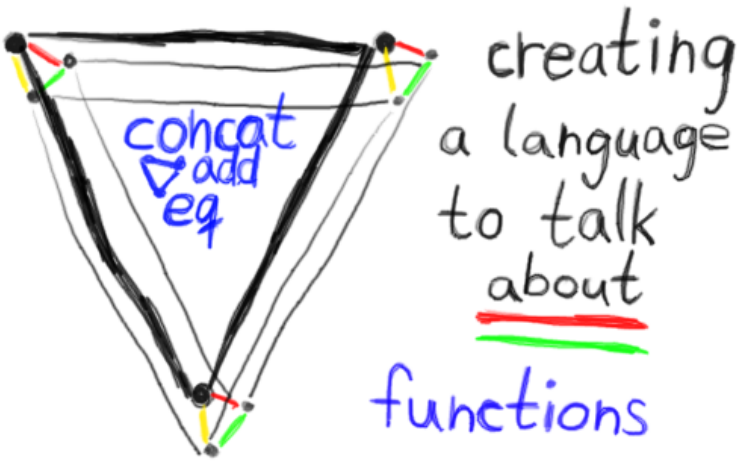
concat[len][even]	
add[even]	Using concat[len] <=> add
eq	Using add[even] <=> eq

Notice that the diagrams I used to illustrate what this means, has a special combinatorial property.
To construct such space with the “discrete” library[2], use:

- DirectedContext (Category)
- Context (Groupoid)

The pair semantics is built into those discrete spaces.

Now, you might wonder why this is called “Path Semantics” and not “Path Theory”:



The same diagram is shown in a different way by contracting and expanding edges.

Since the diagrams can be manipulated by e.g. contracting them in various ways, one can choose to “focus on” the meaning of mathematics seen from a particular perspective. This ability to “focus on” various aspects is very important to create a language that talks about functions.

Programming is all about creating functions for various purposes. The power of Path Semantics comes from the ability to reason efficiently and clearly about code. Every statement in Path Semantics corresponds to an equation, but the same equation can be expressed in more than one way, depending on what one wants to talk about. One might consider Path Semantics as a kind of programming language, except that it usually does not run directly on a computer. It is a mathematical notation designed for programmers.

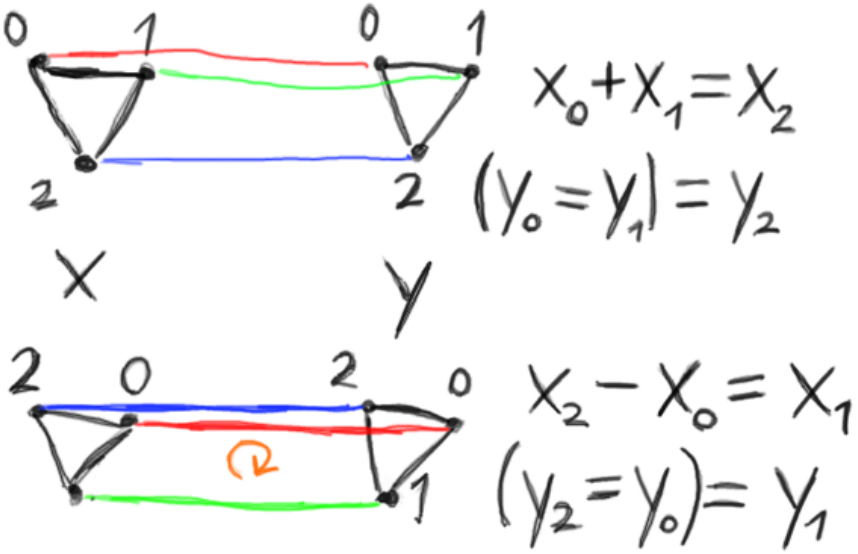
There exists languages that have dependent type systems[8], e.g. Idris[9], that lets you prove properties about the code. I strongly recommend checking out Idris, if you have not already done it.

Path Semantics is designed to reason *informally* about code, while languages such as Idris are designed to reason *formally* about code. A dependent type system adds information to the types, which is an **extrinsic** mathematical property preserved consistently with the code, through typing rules. Path Semantics uses **intrinsic** mathematical properties of functions. This means, if you want to reason about what functions are actually doing, one might use something like Path Semantics for theorem proving. However, if you want to check that some code is correct, one would use a dependent type system.

Every proof you can e.g. do with Idris, can be done in Path Semantics, but not vice versa:
Informal theorem proving is more powerful than *formal theorem proving*, but with less guarantees.

Path Semantics so powerful that it can build “bridges” to other languages. It is easy to translate from one way of thinking to another, which makes it possible to combine the power of both techniques.

When you manipulate the diagrams in Path Semantics, one can solve multiple equations at once:



Remember the following rule:

    add[even] <=> eq

In the drawing above, I have a side `x` that uses `+` (`add`) and a side `y` that uses `=` (`eq`). However, when you rotate the points in the triangles around their centers, the `+` becomes a `-` (`sub`), yet the `y` side stays the same. Using the technique of manipulating the diagram, we have proven a new rule:

    sub[even] <=> eq

If you learned that `=` is associative and commutative, it is easy to explain why the `y` side stays the same. The `y` side talks about `x` in the same way before and after the manipulation.

There are no more rules to be learned from swapping two triangle points or rotating the triangle. This is because there are two `sub[even] <=> eq`'s and one `add[even] <=> eq`. In total three rules.

Can you guess what the number of rules are for equations of 4 variables? What about 5?

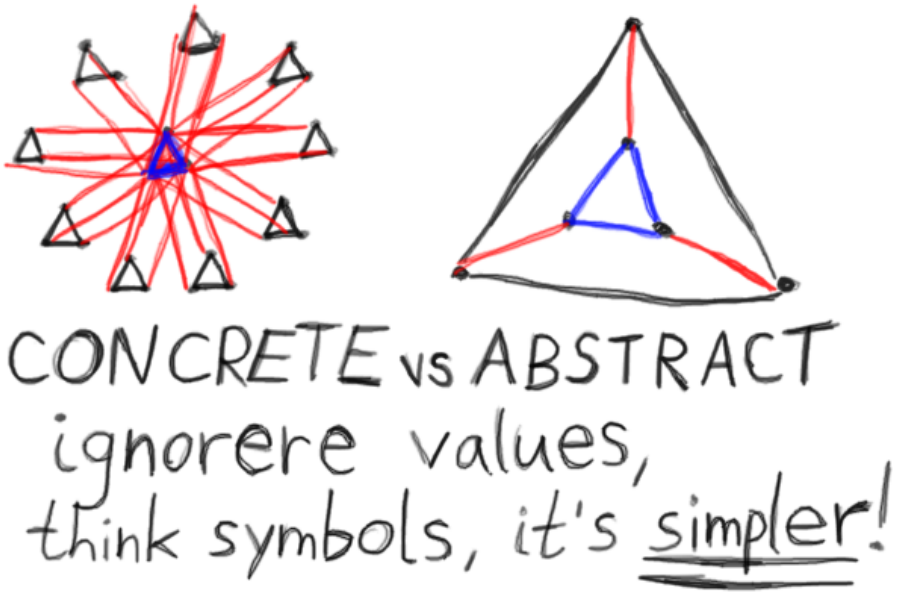
Meaning, or semantics, needs to be part of something dynamic to be useful.

When we “focus on” various aspects of mathematics, or the real world,
the meaning of the language we use to talk about it, also changes.

Mathematics is used because of this ability. An equation can predict something about the world, and if we need something else, we can transform equations, combine them etc. So, basically, that is the “secret” of mathematics. This ability requires solutions, so more advanced theories are designed for particular domains when solutions are not easy to find.

However, what happens if you want to know whether all this is correct or not?
If we want to be sure that our ideas work, we might want to check it for all concrete values.

When examining the diagrams of the concrete values of the same ideas, they become more complex:



The two diagrams above have actually the same structure, they just **focus** on different aspects of it. Try imagine the black triangles on the left side grow in size and move on top of each other.

Some people believe that abstract thinking makes things more complex. However, once you master abstract thinking, things get *simpler*.

You can *think faster*. You grow intellectually. You get longer “arms” which you can use, one day, to grab into the unknown and pull out the idea so big that it will change your life forever.

Checking that something is correct in one language can take exponentially longer time than in a more powerful language. Checking proofs by testing against all concrete values for example, is much harder than keeping a list of known proofs and look them up when you need them.

In addition, abstract thinking often lead to more symmetries which simplifies algorithms.
It is not as easy to write down algorithms for concrete values as for abstract symbols,
e.g. using the “discrete” library to model the combinatorial spaces.
There are some pairs that are excluded from the concrete diagram, which makes it required to use filtering on the combinatorial space. On the other hand, the abstract diagram fits perfectly into the combinatorial space and requires no filtering.
For some reason, abstract symbols contain more symmetries among themselves than concrete values.

The symmetry that, abstract reasoning itself gives you, is reflected in the diagrams.

Therefore, ignore values, think symbols, it’s simpler!

Path Semantics is all about symbols.

References:

[1] "Path Semantics"
AdvancedResearch, Sven Nilsen
https://github.com/advancedresearch/path_semantics

[2] "Discrete – Combinatorial Phantom Types for Discrete Mathematics"
AdvancedResearch, Sven Nilsen
https://github.com/advancedresearch/discrete

[3] "Linear Solver – A linear solver designed to be easy to use with Rust enums"
AdvancedResearch, Sven Nilsen
https://github.com/advancedresearch/linear_solver

[4] "Category Theory"
Standford Encyclopedia of Philosophy
https://plato.stanford.edu/entries/category-theory/

[5] "Groupoid"
Wikipedia
https://en.wikipedia.org/wiki/Groupoid

[6] "Functor"
Wikipedia
https://en.wikipedia.org/wiki/Functor

[7] "Nilsen Cartesian Product Notation"
Sven Nilsen, 2017
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/nilsen-cartesian-product-notation.pdf

[8] "Dependent type"
Wikipedia
https://en.wikipedia.org/wiki/Dependent_type

[9] "Idris – A language with dependent types"
Idris-Hackers, Edwin Brady
https://www.idris-lang.org/