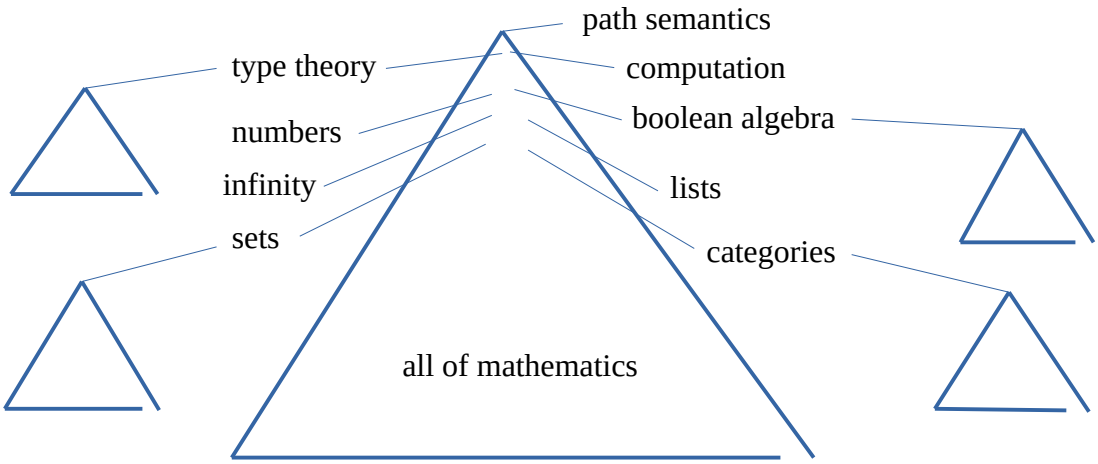
Path semantics

by Sven Nilsen, 2016-2019

Abstract:
Path semantics is used to study foundations of mathematical and programming languages. It is useful because it defines an informal, yet precise semantics enough to "bootstrap" into existing mathematical languages, without requiring a specific language implementation.

Formal languages[1] that can express sentences conforming to path semantics[2] give a formal way of checking for meaningful statements.



This illustration is meant to point out that there are multiple theories of mathematics, that have different strengths and weaknesses. Path semantics is just one way of looking at the world of math.

Yet, path semantics itself does not define all of mathematics. It just describes how symbols are used. By interpreting how existing tools and techniques in mathematics uses symbols, one can develop a deep intuition of mathematics.

"All of mathematics" is not meaningful to talk about until all of mathematics is defined, which is impossible. Path semantics is used to "focus on" parts of what we mean when we use mathematics.

From path semantics, you can construct any formal language in any way you want, as long there is an interpretation of symbols that does not violate the core axiom of equality.

There is a lot of evidence that path semantics is powerful. Yet it might be difficult to convince some people that meaning of symbols can be understood through path semantics. This confusion results usually from not understanding the difference between formal and informal languages used for theorem proving[3]. The distinction between "formal" and "informal" does not necessarily mean that an informal language is less rigorous than a formal one.

	Rigorous	**Non-rigorous**
Formal	Logic	-
Informal	Path semantics	English

All formal languages are rigorous, but not all rigorous languages are formal. The rigor that underlies a rigorous informal language is often not shown directly, but assumed from the context that the language is used for informal reasoning. For example, to keep a paper somewhat comprehensible and readable, it is common to leave out steps that are convincingly easy to prove, or possible to prove under various assumptions that are pointed out, or it might be assumed that a proof exists as a part of the argument. This practice is used in mathematics in general.

The practice of mathematics is so similar to path semantics that the only difference is a stricter usage of identity, plus that path semantics develops new syntax for expressing ideas. This is a good thing, because it justifies the way informal theorem proving is done in practical mathematics and programming. One might learn path semantics to improve informal theorem proving skills.

Since path semantics is informal, it does not claim that everything in a paper can be proven in e.g. Set Theory. It has less guarantees: You might not even be able to prove something in practice, perhaps it is too vague, maybe it is not well understood, too complex etc. However, the proof itself might be crystal clear of what it means: Vagueness can be pragmatically powerful to express ideas. Path semantics try to explain how this can happen without making it too complex or too vague.

The benefit of path semantics is that it can describe what we mean when e.g. using equations to describe something else. A common combination is to use path semantics to express new ideas while working on ways to formally reason about them in various ways.

- Use path semantics to express new ideas
- Use formal languages to model parts of a new idea formally (or the whole if possible)

Path semantics relies on a technique called “bootstrapping”:

1. Define an axiom of equality that holds for all path semantics
2. Construct a reflective language in path semantics
3. Describe formal languages in the reflective language

The purpose of constructing a reflective language is to get better tools for expressing ideas.
Even if this reflective language might be hard for formalize, it is possible to check various specific proofs against the core axiom of equality. It is also a super-set of existing languages that are possible to formalize, e.g. dependent typed languages. This means the only place where intuition is needed, is where an existing formal language is not powerful to express an idea.

In this paper, I will use the core axiom to construct a language, in an informal way, just simple enough to describe logical gates (AND, OR, NOT etc.).

Step 1: Axiom of equality (core axiom)

The interpretation of formal languages is about an arrangement of symbols, where a collection of symbols `F` are associated with another collection of symbols `X`. This associative connection must not lead to a circular definition, which is expressed as `F > X`. For all such collections of symbols, an equality expressed as `F = F` is associated with an equality `X = X`:

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

This is the axiom of equality, that holds for path semantics.

The axiom is a modification of Leibniz' law[4] where propositions are replaced by truth values as collection of symbols. In informal theorem proving, one can use other formal languages, but in order to determine whether the reasoning works when translating between languages, path semantics expresses predictions that corresponds to particular equations. The semantics of these predictions are "bootstrapped" from the axiom without relying on the semantics of equations.

The corner stone of path semantics is to define how one talks about other things that might be too complex to formalize. Existing tools, syntax and techniques from standard mathematics is then used to do the theorem proving, while preserving meaning. Path semantics' power is that it interprets into the language that is suitable for a particular domain of mathematics. This means that people learning path semantics can use the tools without needing to translate into another language.

Inequality

It follows from the axiom of equality that there is a similar axiom of inequality:

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

Inspiration of the axiom

I used normal paths[5] to do some theorem proving that I found useful, but it was difficult to use them with the same ease as informal theorem proving in Coq[6] and Idris[7]. To justify the informal technique I used, I wanted to learn more about foundation of mathematics. Vladimir Voedvodsky[8] held some recorded lectures about the foundations of mathematics, which I watched. The way he used paths from homotopy to talk about mathematics was inspiring, so I named my idea of extracting hidden functions from within functions after this word: Path. After a while, more and more "paths" were formalized, so I started thinking about them as ways of associating collections of symbols with other collections of symbols. One day I realized that most of mathematics depends on definitions. If I could find an informal axiom that explained how symbols are used, I could use arbitrary definitions to construct more mathematics. To me it was important to express mathematical ideas concisely in the language of functions, so I did not need a formal axiom. I made it part formal and part vague, formal enough so I could "bootstrap" into functional programming.

Functional completeness

Bits are the simplest collection of symbols that satisfy the equality axiom. Consider the axiom as a computer circuit to prove soundness of free variables of bits when:

$$\frac{F_0 > X_0}{F_0 = 1, X_0 = 0}$$

This corresponds to a binary logical gate equivalent to NAND:

$\mathbf{F_1}$	$\mathbf{X_1}$	$\mathbf{F_0=F_1}$	$\mathbf{X_0=X_1}$	**Meaningful**	**NAND**
0	0	0	1	1	1
0	1	0	0	1	1
1	0	1	1	1	1
1	1	1	0	0	0

Although the axiom of equality is simple, it contains enough complexity to model NAND and is therefore functional complete. Any computation can be done by connecting NAND gates together.

This deep symmetry of functional completeness, together with experience from constructing languages from the axiom, was used to refine the axiom of equality from the first form

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

to the final form that gives symbols in formal languages a constructive character:

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

Step 2: Construct a reflective language in path semantics

Let us start with a more complex collection of symbols: Words.

In this language we associate a logical gate `not` with an input type `bool`:

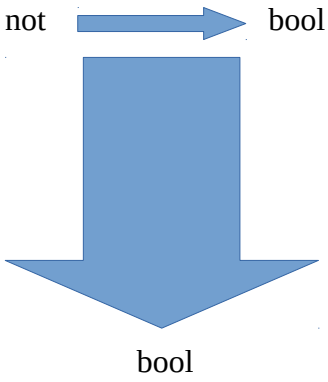
not(bool)

This choice is arbitrary and done simply because we know how `not` gates behave. It makes sense to associate the symbols this way because it follows the axiom of equality. If somebody says `a = not`, then we know `a` also takes the input type `bool`.

To describe the output type, we associate `not(bool)` with `bool`:

not(bool) → bool

This is just creating a collection of symbols out of the two associated symbols, and then associate this new collection with something else.



This operation can be done in two ways, either by creating `not(bool)` or `bool → bool` first. Some languages write this as:

not : bool → bool

Then we do a simple, but powerful trick: Associate each value with a type, and return itself.

true(bool) → true
false(bool) → false

This trick might not be taken literally, but rather to use the analogue syntax for sub-types:

x : [true] true => x : bool

Or simply:

x : [:] true => x : bool

It makes it possible to “lift” variables into types through the language of sub-types.

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

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

Testing that `not(not(X)) = X`:

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

The `[:]` symbol is symmetric, so we can refactor it:

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

It gives us the ability to compute with values like in a normal programming language.

Introducing a wildcard notation and shorthand version for cases:

```
and(bool, bool) → bool
[:] (true, true) → true
[:] (_, _) → false
```

This is sufficient for deriving Boolean algebra in a short and nice syntax.

Step 3: Describe formal languages in the reflective language

Deriving a formal language is one thing, but how can we describe them reflectively?

It turns out that the same mechanism that associates values with their type can be used more than once. For example, the `not` gate can be used like this:

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

Or written like this:

```
and [not] (bool, bool) → bool
[:] (false, false) → false
[:] (_, _) → true
```

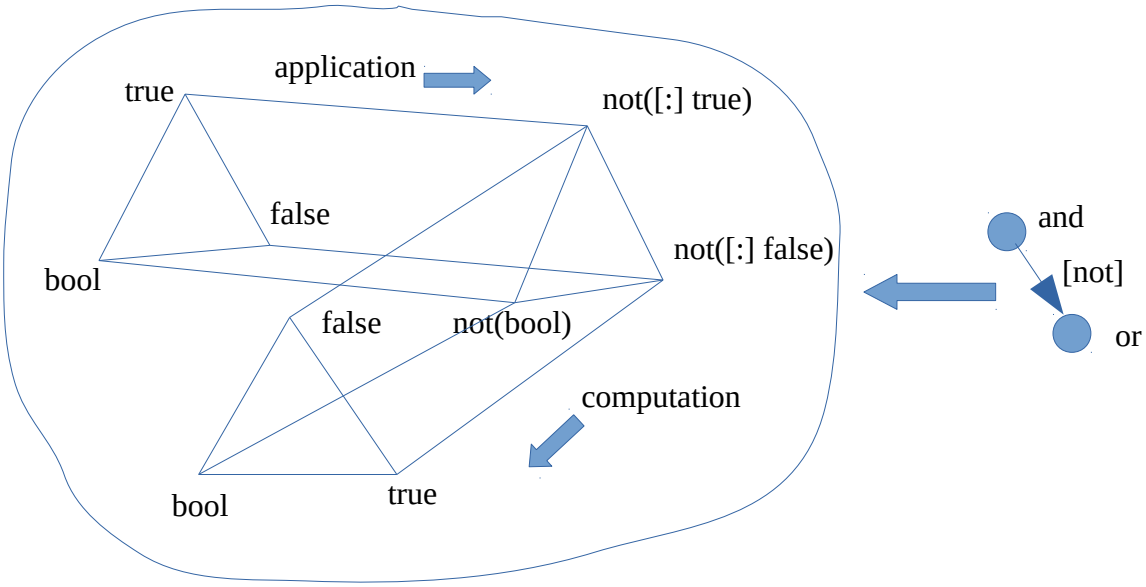
Notice that this is the definition of `or`, such that:

```
and [not] <=> or
```

Which is the same as writing the equation:

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

The hidden meaning is a geometrical structure that can be reflected over and over infinitely times. Words like “application” and “computation” are labels we put on the dimensions of thought space in this geometry.



The meaning of symbols are restricted by the ways we use them. The way it is restricted in formal languages is according to the axiom of equality.

References:

[1] “Formal language”
Wikipedia
https://en.wikipedia.org/wiki/Formal_language

[2] “Path Semantics”
AdvancedResearch, Sven Nilsen
https://github.com/advancedresearch/path_semantics

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

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

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

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

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

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