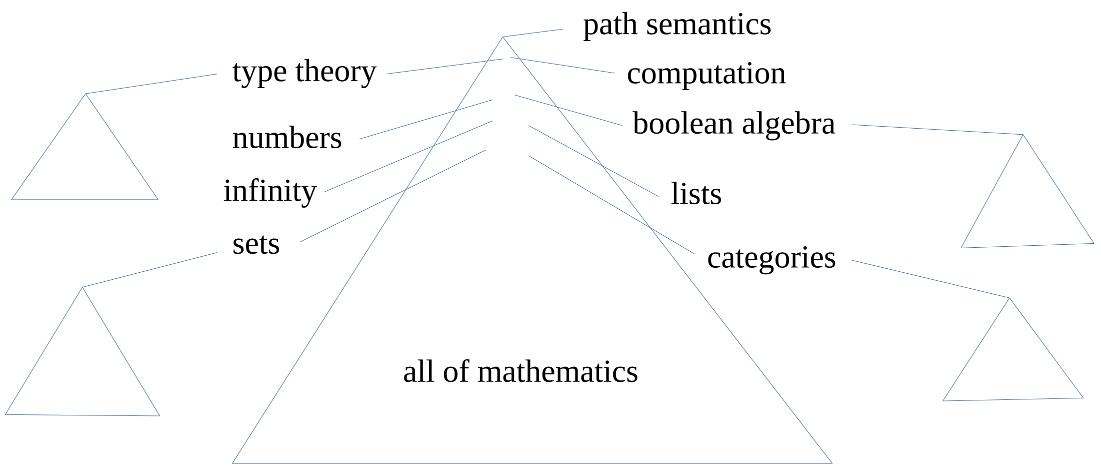
Path semantics

by Sven Nilsen, 2016

Abstract:
Path semantics is used to study foundations of mathematics and programming languages. It is useful because it defines a precise semantics without a language. Problems that originate from language construction can be analyzed without depending on a formal language. It provides an intuitive starting point for analysis.

It is believed that all of mathematics is optimized path semantics for various domains. All formal languages that can express sentences conforming to path semantics give a formal way of constructing meaningful statements.



No formal language with axioms that can express arithmetic can prove its own consistency and be complete. How does path semantics work around this limitation?

Path semantics only defines how the semantics should behave in formal languages. Since you can construct any formal language you want, there is no way to prove the consistency of path semantics without ending up with a circular argument. A such proof is also believed to be invalid by path semantics.

There is a lot of evidence that path semantics is powerful. It is believed informally that path semantics defines meaning of all mathematics.

Why path semantics is believed to describe all mathematics follows from three steps:

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

Step 1: Axiom of equality

The interpretation of all 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 all path semantics.

Inequality

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

$$\frac{F_0(X_0), F_1(X_1), F_0 > X_{0,} X_0 \neq X_1}{F_0 \neq F_1}$$

Functional completeness

Bits are the simplest collection of symbols that satisfy the equality axiom. We 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:

F_1	**X_1**	**F_0=F_1**	**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

NAND is 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 is believed to give any formal language its 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`:

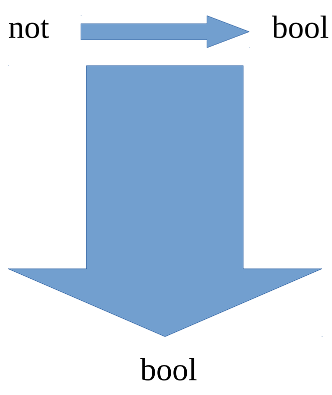
not(bool)

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 associated `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 writes this as:

not → bool → bool

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

true(bool) → true
false(bool) → false

For all such constructs, the argument can be inferred from the return value. Therefore, the type can be inferred from a variable. To not mix it up with other ways to organize symbols, we write

bool $\rightarrow_{\text{true}}$ [true] true
bool $\rightarrow_{\text{false}}$ [false] false

to annotate which path was taken from `bool` to `true` or `false`.

Now we describe `not` by associating it with a path of how it behaves for variables:

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

This is actually mixing the collections of symbols, since `[true]` can be thought to be a symbol `not__true`:

```
not([true] true) = not__true(true)
not([false] false) = not__false(false)
```

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

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

Now we replace the brackets with `:` because it can be inferred from the variable:

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

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

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

It gives us the ability to compute with variables like in a normal programming language. We have now derived Type theory.

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 variables 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

This is symmetric, so we can refactor out `not` and write down the equality:

> and [not] = or

Which is the same as writing the equation:

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

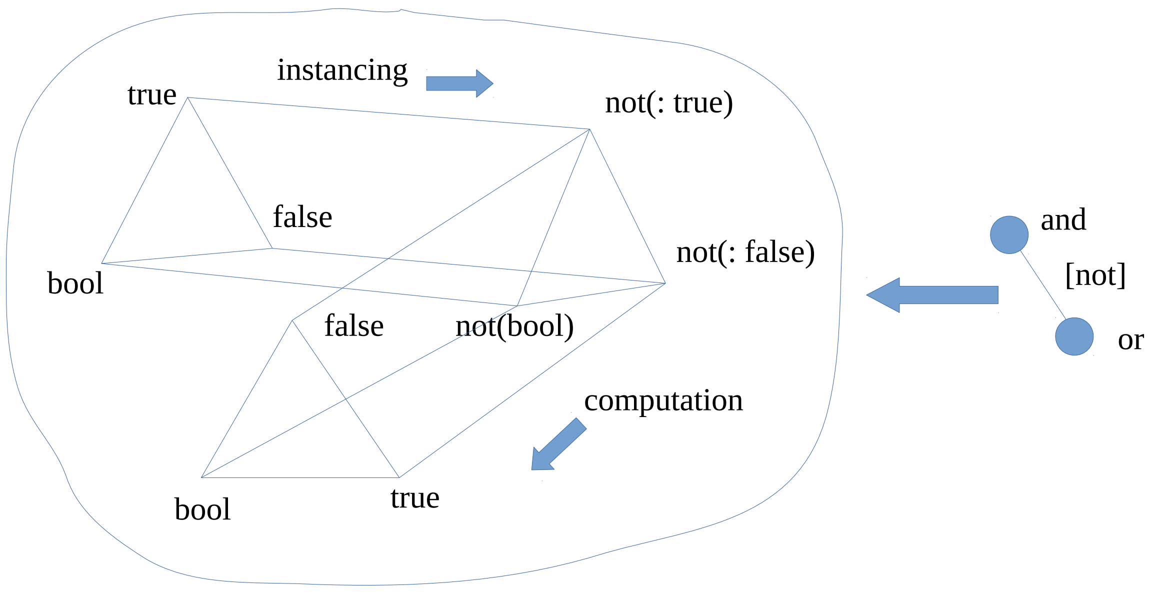
What happened here is a sequence of associations:

$$\text{bool} \rightarrow_{\text{not}} [\text{not}]\ \text{bool} \rightarrow_{\text{false}} [\text{not}]\ [\text{false}]\ \text{false}$$

This sequence of associations, called a “path”, allows us to book keep what we are doing. It prevents us from violate the axiom of equality when inserting it into a different collections of symbols. The meaning of the symbols must confirm to the strict semantics along all axis of transformations. As long as we do this, the only limit is our imagination.

These “paths” is why the theory is called “path semantics”.

The hidden meaning is a geometrical structure that can be reflected over and over infinitely times. Words like “instancing” 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 is according to the axiom of equality.