Semantics of Propositions

by Sven Nilsen, 2022

In this paper I show how to derive the semantics of propositions.

The semantics of propositions is derived from four fundamental concepts:

- Product
- Sum
- Function
- Type

Product and Sum are dual to each other. Function and Type are orthogonal to each other.

The terms "dual" and "orthogonal" comes from the language of Category Theory[1].

Dual means that in some category, one obtains the dual concept by reversing every morphism. Orthogonal means that two morphisms are connecting a morphism with a parallel morphism in a commuting square. A morphism is the same as an "arrow".

This means that one can reduce these four fundamental concepts into three higher concepts:

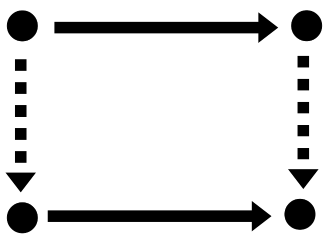
- Arrow
- Duality
- Orthogonality

First, one should investigate these three higher concepts to make sure they are understood clearly.

An arrow is simply something that points from one object to another object:

The dual arrow points in the reverse direction:

Orthogonality means there are two kinds of parallel arrows that are related in the following way:



This is called a "commuting square". Notice that when you reverse all arrows, the square still commutes. This means the commuting property of orthogonality is invariant under duality.

I will not go further into these three higher concepts, but it is nice to know they exist.

Both `true` and `false` are related to Product and Sum respectively:

$\top := (,)$	The empty Product is `true`
$\bot := (\|)$	The empty Sum is `false`

Notice that `true` and `false` are types, not proofs of types:

$\top$: type	type_of($\top$) = type
$\bot$: type	type_of($\bot$) = type

As proof of `true`, one can use `()`, which is an empty tuple:

() : $\top$	type_of(()) = $\top$

When having a proof of `false`, one can prove anything because there are no cases to match against:

absurd := \(B : type, a : ⊥) → B = match a {}

absurd[type_of] <=> \(B : type) → type = B

Notice that `a : ⊥` vanishes in the normal path[2].

This happens with all arguments that are values, due to the property of `type_of`:

type_of : $\text{type}_n \to \text{type}_{n+1}$

Since the codomain of `type_of` does not include all of its domain, some terms will vanish.

To construct a Product, one can use the function `prod`:

prod := \(A : type, B : type, a : A, b : B) → (A, B) = (a, b)

prod[type_of] <=> \(A : type, B : type) → type = (A, B)

To construct a Sum, one can use the following:

sum_left := \(A : type, B : type, a : A) → A | B = left(a)
sum_right := \(A : type, B : type, b : B) → A | B = right(b)

sum_left[type_of] <=> \(A : type) → type = A | B
sum_right[type_of] <=> \(B : type) → type = A | B

Now, `not` can be defined as a type constructor that produces a proof of `false`:

¬ : \(A : type, a : A) → ⊥

¬[type_of] <=> \(A : type) → type = ⊥

Notice that it is impossible to construct `not`. It is a type constructor, not an actual function.

That's it! Everything else is derived from these definitions. This calculus satisfies Intuitionistic Logic (IPL)[3]. From this one gets new kinds of logical languages that are used in path semantics[4].

References:

[1] "Category Theory"
Wikipedia
https://en.wikipedia.org/wiki/Category_theory

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

[3] "Intuitionistic logic"
Wikipedia
https://en.wikipedia.org/wiki/Intuitionistic_logic

[4] "Path Semantics"
Sven Nilsen, 2016-2021
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/path-semantics.pdf