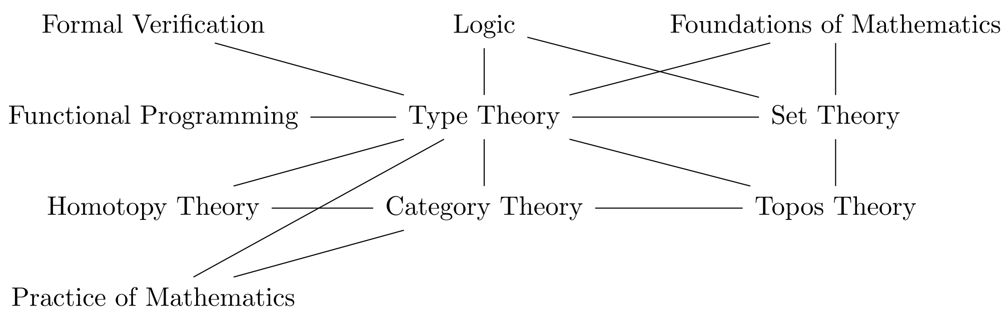
Type Theory with Paige North

Jason Schuchardt

July 1, 2019

Type theory is a relatively new branch of math. It cannibalizes a lot of other areas of mathematics. In particular it has connections to logic, set theory, and functional programming. It is the basis of a lot of modern functional programming. It also takes a lot of inspiration from category theory, and its subfield, topos theory. It's also related to homotopy theory.

The map:



1 Basics

Basic objects are *types* and *terms*. Every term belongs to exactly one type. When the term t belongs to a type T, then we write $t : T$.

Example 1.1. In set theory, one writes $5 \in \mathbb{N}$. In type theory, one writes $5 : \mathbb{N}$.

In type theory, every term belongs to exactly one type. This is not the case in set theory.

Example 1.2. In set theory, one can think of 5 by itself. It is itself a set. One can then say that $5 \in \mathbb{N}$, and $5 \in \mathbb{R}$.

This is not the case in type theory. In type theory, one can only consider a term, like 5, as being part of a type, $\mathbb{N}$.

The terms $5 : \mathbb{N}$ and $5 : \mathbb{R}$ are completely different things. We might be able to compare them in a few classes, but they aren't immediately comparable. They are distinct.

History: The first person to come up with a sort of type theory was Bertrand Russell in 1902. He invented it to solve Russell's paradox (Is there a set that contains all the sets that don't contain themselves). However, it was very different from what we'll be talking about.

2 A first type theory: The Simply Typed Lambda Calculus

The simply typed lambda calculus (Church 1940) is the simplest type theory along the lines of what we're thinking about, and a model for computation (functional programming).

Definition 2.1. A simply typed lambda calculus with $\implies$ consists of the following:

- Atomic types $T_1, \ldots, T_n$.
- A type $S \implies T$ (S "implies" T) for every two types S and T.
- For each type T, we have variables
$$x_1^T, \ldots, x_m^T : T,$$
and for any term $t : T$ and variable $x : S$, we have the term $\lambda x.t : S \implies T$.

  We say we are *abstracting x from t*. The term t might involve x (if it doesn't then the "function" we are defining is a constant function).
- For every term $f : S \implies T$, and every term $s : S$, we have a term $fs : T$. We call this term the *application of f to s*.

These are subject to the equations

- For every term $t : T$, variable $x : S$, term $s : S$,
$$(\lambda x.t)s = t[s/x],$$
where $t[s/x]$ is the term obtained from t by replacing every instance of x with s.
- For each $f : S \implies T$ and variable $x : S$ which doesn't occur in f (so we don't have variable naming conflicts basically). Then we have
$$\lambda x.(fx) = f : S \implies T$$

Another example of a type is $\mathbb{N}$. $\mathbb{N}$ is a type with terms $0 : \mathbb{N}$ and $Sn : \mathbb{N}$, where $n : \mathbb{N}$ is a term. 0 is a *closed term of* $\mathbb{N}$.

T_1 is an atomic type, but $T_1 \implies T_2$ is nonatomic. $(T_1 \implies T_2) \implies T_3$ is also nonatomic.

2.1 Things we can do!

Example 2.1. For every term $t : T$ not containing all variables of type S, there is a term of $S \implies T$. We need to supply a variable $x : S$ and our term $t : T$ to get such a term. For example $\lambda x_j^S.t : S \implies T$, where x_j^S doesn't occur in t.

Then
$$(\lambda x_j^S.t)s = t[s/x_j^S] = t.$$
This is a constant function that sends everything in S to our term t.

Example 2.2. For every type T, there is a term $T \implies T$. Let $x : T$ be a variable. We can build the identity function:
$$\lambda x.x : T \implies T.$$

The first equation tells us that
$$(\lambda x.x)t = x[t/x] = t,$$
which is why we can call it the identity function.

Question: Do we have a different identity function for each variable?

Answer: We can prove that they are the same. In type theory we distinguish between syntax and semantics. The formulas $\lambda x.x$ and $\lambda y.y$ are *different* formulas, in a sort of naive sense. However their semantics are somehow the same.

For the proof, we can use the function variable replacement rule (the second equation)

$$\lambda x.(fx) = f.$$

If x and y are variables, then let $f = \lambda y.y$.

$$\lambda y.y = f = \lambda x.(fx) = \lambda x.((\lambda y.y)x) = \lambda x.x.$$

This is a basic theory of functions. We have a type for functions, $S \implies T$, and an identity function, $\lambda x.x : T \implies T$.

Note! The only functions we have are ones for which we can write down an explicit formula. E.g. $\lambda x.x$, $\lambda x.t$.

This is *not* a theory of *mathematical* functions (in the sense of Set Theory), but *computational* functions!

What is a "mathematical" function. Officially, in set based math, a function $f : X \to Y$ is defined as a graph $G \subseteq X \times Y$. Intuitively, this set is something like a table, that records for each $x \in X$, the value $f(x) \in Y$. For example, for the function

$$\lambda x.x^2,$$

(the function $f(x) = x^2$) from $\mathbb{R}$ to $\mathbb{R}$. The set theoretic representation of this function will be as the set of pairs

$$\{(0,0),(1,1),(2,4),(\sqrt{2},2),\ldots\}$$

Even though the functions we consider in math usually have nice and finite formulas, a mathematical function doesn't have to have a formula, and might only be describable by such an infinite table.

Neither humans nor computers can work with infinite descriptions, so if we're interested in studying functions in a computational setting, we need a finite formula for every function.

In type theory, we have the interpretation

$$\text{Functions} \longleftrightarrow \text{Programs}.$$

Under this interpretation

$$\begin{aligned} \text{Types} &\longleftrightarrow \text{Specifications of programs} \\ \text{Terms} &\longleftrightarrow \text{Programs fulfilling the specification} \\ S \implies T &\longleftrightarrow \text{Programs which take input } s : S \text{ and return } t : T \end{aligned}$$