A brief survey on Type Theory

YEE-JIAN TAN, National University of Singapore, Singapore

This is a survey article on Martin-Lof's Type Theory.

CCS Concepts: • **Theory of computation → Type theory**; *Constructive mathematics*; *Higher order logic*.

Additional Key Words and Phrases: Intuitionistic Type Theory, Calculus of Inductive Constructions, Martin-Lof Type Theory

1 INTRODUCTION

1.1 Background

Type Theory lies in the intersection of logic and computer science. It was first introduced by Russell in 1903 as a solution to the famous Russell's Paradox in set theory [8], that the set of all sets (also the class of all classes) cannot exist. Type theory solves this problem by establishing a hierachy of objects, predicates, predicates of predicates etc. in a hierachy now known as the extensional hierachy. These schema is proved to solve the fundamental paradox lying at the roots of set theory raised by himself. At this point, type theory was still just a mathematical theory about logic.

The connection between the Type Theory and computer science is made possible after Alonzo Church invented the Lambda Calculus as an investigation into the models of computation in the 1930s [5]. The Lambda Calculus (LC) is a small system of computation which consists only of functions and their applications. Since lambda calculus is untyped by design, the investigation of assigning types to lambda calculus terms and their rules led to what was known as Type Theory in Computer Science now.

In this paper, we will focus more on the systems that derived from the attempts to type different variations of lambda calculus, from the Simply Typed Lambda Calculus (STLC) to the most recent Homotopy Type Theory (HoTT). This survey paper will assume basic knowledge about propositional and predicate logic, as well as some familiarity with programming types, or even better, abstract data types such as pairs, functions and functionals (higher-order functions). Although we will go through the fundamentals of lambda calculus briefly, prerequisite knowledge about lambda calculus is not required but recommended.

1.2 Overview

Section 2 will give a brief introduction on lambda calculus and type systems, before looking at their combination: typed lambda calculus. Furthermore, we will establish the correspondence between logic and type theory. After this section, all analysis will look at the two sides of typed lambda calculus, type theory as a basis for functional programming languages, and type theory as a logical system due to its correspondence to logic.

Section 3 will look at the different systems of typed Lambda Calculus chronologically, and discuss their roles in these 2 aspects: as functional programming language and as a logic system.

Section 4 will look at the current proof assistants and programming languages, as well as dicussing the type theory systems behind the hood. This provides an insight into how type theory is used in practice.

In the last section, Section 5 we will look at current research trends and what progress has been made, as well as what are the possible directions for future research.

Author's address: Yee-Jian Tan, yeejian.tan@u.nus.edu, National University of Singapore, Singapore, 138595.

2 INTRODUCTION TO TYPE THEORY

2.1 Lambda Calculus

The Lambda Calculus is, in modern terms, a system consists of purely "functions" and "function calls", on a **untyped** system without any data structures. The specification for objects in Lambda Calculus, called *items*, is extremely short:

(1) Every variable (usually $x, y, z, t, \ldots$) is a Lambda Calculus term (called *term* from here on).
(2) If M is a term, then the abstraction (function) $\lambda x.M$ is a term.
(3) If M, N are terms, then MN (application) is a term.

The syntax can be written succintly as $e ::= x \mid \lambda x.e \mid e\ e$, each branch corresponding to the written rule above. The simple syntax and the sole idea of **abstraction** provided a way to **create terms based on terms**, a feature which be extended and discussed in Section 2.5.

2.2 Simply Typed Lambda Calculus (SDLC)

The *SDLC* was invented in 1940 by Alonzo Church, not long after he formulated the (untyped) lambda calculus ([6]). The typing rule fairly simple: the types start by having a base types B, then types are either in B or is an "arrow type":

$$\tau ::= \tau \to \tau \mid T \quad \text{where } T \in B.$$

2.3 Types in Computer Programming

In computer programming, the term "types" usually refer to the categories of different data structures that programs compute, such as real numbers, floating-point numbers (finite-precision real numbers), integers and more. These are examples of concrete types in programming.

On the other hand, types without a contrete representation is called **abstract types.** Since the lambda calculus does not come with constructs such as numbers, characters or strings, the typing of it is inevitably abstract. This led to the investigation of typing on lambda calculus, of which the most intuitive form is known as "Simply Typed Lambda Calculus" (SDLC).

It is important to note that abstract programming types have a similar formulation from the type in Type Theory or logic, and the implementation of such mathematical types differe in programming languages and proof assistants.

2.4 Curry-Howard correspondence

There are 3 main findings in the Curry-Howard Correspondence ([13],[10]), the most relevant to this paper being:

Natural Deduction corresponds to Typed Lambda Calculus.

The correspondence requires an isomorphism, ie., it is proven that

(1) Every statement in the logical system can be represented as a term in typed lambda calculus.
(2) Every valid typed lambda calculus term corresponds to a statement in the natural deduction logical system.

It is paramount to get an intuivite idea of this correspondence because it forms the backbone of the rest of the paper. We outline it on Table 1:

Table 1. Curry-Howard Correspondence

Natural Deduction	Lambda Calculus
lambda calculus terms	proofs
types	propositions
p has type P	p is a proof of P
$p \implies q$	$(\lambda x.y) : P \to Q$

This idea of "proposition as types, proofs as programs" sees extensions with the invention of different extensions of the *Simply Typed Lambda Calulus*, with its limited feature, corresponds to *propositional logic*.

2.5 The Lambda Cube

Typed Lambda Calculus has two main constructs — terms (or objects) and types. Untyped Lambda Calculus allows terms to depend on terms (via the abstraction rule), hence typed lambda calculus as well. A natural question to ask is: can we generate terms/types using other terms/types, or conversely, have term/types depend on other terms/types?

All the 2×2 possibilities are well-investigated by different type systems. We list their formal names below:

Table 2. Definition of Typed Lamda Calculus features

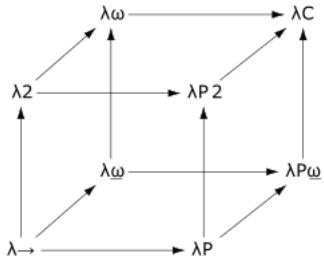
Generate	Using	Known as	Example
Terms	Terms	**Abstraction**	Identity Function: $\lambda x.x$
Terms	Types	**Polymorphism**	Polymorphic Identity Function: $\Lambda\alpha.\lambda x.x$
Types	Terms	**Dependent Types**	Class of types indexed by A: $\prod_{x:A} B(x)$
Types	Types	**Type Constructors**	$\mathsf{TREE} := \lambda A : *. \prod B.(A \to B) \to (B \to B \to B) \to B$

We know that *Simply Typed Lambda Calculus* only has **abstraction** since it is just contains naively typed lambda calculus and nothing more. On top of *STLC*, a Type Theory can have any combination of other 3 features, creating $2^3 = 8$ possibilities.

The Lambda Cube is a visual representation of the inclusion of these 4 features in different type theories, introduced by Henk Barendregt[4] in 1991. It has three axes:

- x-axis($\to$) represents **dependent types**, allowing **types to depend on terms**. Examples in the next section all rely on dependent types, other than *Simply Typed Lambda Calculus* and *System F*.
- y-axis($\uparrow$) represents **polymorphism**, allowing **terms to depend on types**. The most direct example is $\lambda 2$, also known as *System F*, which is just *Simply Typed Lambda Calculus* with *Polymorphism*.
- z-axis($\nearrow$) represents the use of **type Constructors**, allowing **types to depend on types**. An example is the *Calculus of Inductions* (λC).

Fig. 1. The Lambda Cube. By Tellofou - Own work, [CC BY-SA 4.0], https://commons.wikimedia.org/w/index.php?curid=76344034



It covered all 8 possibilities as analyzed above. In this paper, we will not take an exhaustive look at all the type systems corresponding to all 8 vertices of this cube, but highlight some prominent type systems that have been invented.

3 TYPE THEORY DEVELOPMENT

3.1 System F

System F is a Typed Lambda Calculus system that is the *Simply Typed Lambda Calculus* but extended with **polymorphism**, meaning terms such as functions can be created from types. A classical example is the polymorphic identity function, which first accepts on the type of the argument given:

$$\Lambda\alpha : *.\lambda x : \alpha.x \ : \forall\alpha.\alpha \to \alpha$$

The typing rules are not described here for simplicity. An interesting result is the Girard-Reyolds Isomorphism, named after the two researchers who independently discovered System F, where Girard proved the Representation Theorem and Reynolds, the Abstraction Theorem. These two theorems can be seen as the two directions of the isomorphism between *second-order polymorphic lambda calculus ($\lambda 2$)* and *second-order intuitionistic predicate logic (P2)*.

The idea of polymorphism is useful in programming, where a function can "change" its behaviour according to the type of the given input. Therefore, System F and its extensions $F_\omega, F_{:>}$ which add **type operators** and **subtyping** respectively, are often used in the implementation of functional programming languages. For example, the Glasgow Haskell Compiler (GHC) uses an extension of F_ω in its implementation.

3.2 Calculus of Constructions

The **Calculus of Constructions**[9] is a high-level purely functional programming language, and also a type theory that is most well known for sitting on the top of the Lambda Cube, ie., it has all the properties of **polymorphism, type constructors and dependent types** and hence a superset of **System F**. This system underlies the implementation of the influential theorem prover **Coq**. The nice properties of this system includes **strong normalization**, showing that all programs terminate, and additionally, it is logically consistent.

There has been many extensions for **Calculus of Constructions**, one notable example the *Calculus of Inductive Constructions (CIC)*, which is fomulated as the **Coq** theorem prover was implemented. One feature of CIC is it improves on pure *Calculus of Construction*'s drawbacks when implemented as a theorem prover or programming language, namely that the inductive definitions made computing functions efficiently and proving some natural properties impossible [14].

3.3 Logical Framework

Logical Framework (LF) is different in its purpose from other type theories that it is designed to be a "general theory of logical systems that isolates the uniformities of a wide class of logics"[11]. It is designed to be a formal system as weak as possible, but with nice properties that allow the representation of different type theories by guaranteeing some normalization. On the Lambda Cube, it has the **dependent type** feature which allows types to be created using terms.

Some central features to note about Logical Framework are on **Definitional Equality**, which is defined between all entities (called *Kinds*, which are types of *Families*, which in turn are types of *Objects*) as the transitive clousure of the **parallel nested reduction**, a rule based on the β-reduction in Lambda Calculus. It inherits what is known as the Church-Rosser Property, which shows that if a Lambda Calculus term can be reduced to two different terms, then there exist a term where these intermediate terms can eventually converge to under reduction. As a type system, **LF** enjoys the **strong normalizing** and **recursively decidable** properties, making it suitable for embedding different logics.

3.4 Martin-Lof Type Theory

Martin-Lof Type Theory is also known as Dependent Type Theory or Constructive Type Theory. Invented by Martin-Lof in the 1970s, it is originally a framework designed to rigorously formulate constructive mathematics, a branch of mathematics that focuses on the philosophy of *constructivism*, asserting that any proof of existence of any mathematical object needs an explicit example to be consider valid.

The logic behind this philosophy known as *intuitionistic logic* or *constructive logic*, which is classical logic without the law of excluded middle. The *Law of Excluded Middle (LEM)* asserts that every proposition is either true or false:

$$\frac{}{P \vee \neg P}\,\text{LEM}$$

Which leads to non-constructive existence proofs which first assume the non-existence of the desired object, and reach a contradiction. Even though this is sound in classical logic, it does not provide any evidence for this newly proved existence, hence non-constructive.

Martin-Lof type theory first start off with only 3 ground types: the *null type* **0** which has no object in the type, the *unit type* **1** which has only one object in the type, and the *boolean type* **2**, which consists of `{True, False}`. All other types are constructed via a the Σ *type* and Π *type* inductively.

Via the Curry-Howard isomorphism, the proposition claiming that two proofs a, b for the same proposition A are identical, $Id_A(a, b)$ is then a type. The objects $p : Id_A(a, b)$ residing in $Id_A(a, b)$ are proofs that the proofs a, b are identical. An important question in mathematics was the **Uniqueness of Identity Proof (UIP)** problem:

For any type A, terms $a, b : A$, can $Id_A(a, b)$ have more than 1 element?

The provability of **UIP** in Martin-Lof Type Theory was the biggest open problem in Type Theory during its time. It was finally solved by Martin Hofmann and Thomas Streicher[12] in the negative, using an interepretation of Types as Groupoids. This approach was later independently used by Vladimir Voevodsky, Steve Awodey and others to create Homotopy Type Theory.

3.5 Homotopy Type Theory

It started with the problem in martin-lof's type theory about the identity type and how many terms can inhabit the type. Since each term is a proof, this translates to UIP.This is first proven by Hofmann and friend in 1998 or something.

Hofmann and Streicher had a Groupoid model for intensional Martin-Lof type theory.

In 2007, Awodey and Warren presented a novel connection between model categories, a category with three distinguished subcategories: "weak equivalences", "fibrations" and "cofibrations", which is a structure studied in homotopy theory, a branch of algebraic topology. The explanations on homotopy theory and category theory is out of the scope of this paper, but it is important to recognize that the slogan of "Proposition as Types" from the Curry-Howard Correspondence can be interpreted in such systems as "Fibrations as Types", shedding new light to Type Theory especially in terms of identity types. In particular, equalities are viewed as paths, dependent types are viewed as fibrations, giving a rare geometry interpretation to the area of logic, where diagrams are scarce.

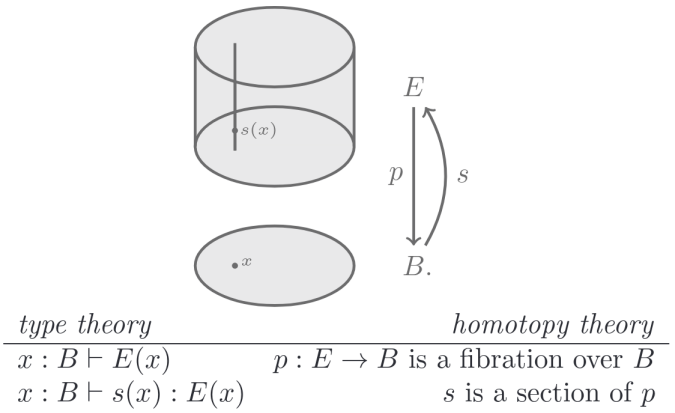


Fig. 2. Homotopy theoretic interpretation of dependent types and terms (taken from [15])

In simpler terms, we have another interpretation of Type Theory through homotopy theory in Table 3:

Table 3. A simplified interpretation of Type Theory in Homotopy Theory[3]

Type Theory	Homotopy Theory
Type	Space
Objects	Point in space
$a : A$	a is a point in the space A
$a = b : Id_A(a, b)$	a, b are path connected
Id_A the type of identities	the path space of continuous maps $[0, 1] \to A$
$f : A \to B$	continuous function from A to B

The standard reference for this new innovation is the HoTT Book[16], by the Univalent Foundations Program of the Institute of Advanced Study.

4 PROMINENT SYSTEMS USING TYPE THEORY

Table 4. Systems and their Type Theory

Type Theory	Systems
System F	ML family (OCaml, Standard ML), Haskell
Logical Framework	Edinburgh Logical Framework (LF), Isabelle, Twelf
Calculus of Constructions	Coq, Agda [1]
Homotopy Type Theory	Cubical Agda[7]
Dependent Type Theory	F*, Lean, Idris, Isabelle, Agda etc.

5 FUTURE RESEARCH DIRECTIONS

As Homotopy Type Theory is the most recent system, there are many open problems, mostly hidden behind the mathematical requirement to understand them. The Homotopy Type Theory Summer School 2019 (HoTT2019) has collated a list of open problems from the speakers, and the nLab website, an initiative to promote the use of Category Theory, has a good collection of open problems on HoTT as well ([2], [1]).

REFERENCES

[1] 2019. Homotopy type theory HOTT2019 summer school open problems list. https://ncatlab.org/homotopytypetheory/show/HoTT2019+Summer+School+open+problems+list

[2] 2019. Homotopy type theory open problems. https://ncatlab.org/homotopytypetheory/show/open+problems

[3] Steve Awodey, Álvaro Pelayo, and Michael A. Warren. 2013. Voevodsky's Univalence Axiom in homotopy type theory. arXiv:1302.4731 [math.HO]

[4] Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (1991), 125–154. https://doi.org/10.1017/S0956796800020025

[5] Alonzo Church. 1932. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33, 2 (1932), 346–366. http://www.jstor.org/stable/1968337

[6] Alonzo Church. 1940. A formulation of the simple theory of types. *The journal of symbolic logic* 5, 2 (1940), 56–68.

[7] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: a constructive interpretation of the univalence axiom. arXiv:1611.02108 [cs.LO]

[8] Thierry Coquand. 2018. Type Theory. In *The Stanford Encyclopedia of Philosophy* (Fall 2018 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

[9] Thierry Coquand and Gérard Huet. 1986. *The calculus of constructions.* Ph.D. Dissertation. INRIA.

[10] H. B. Curry. 1934. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences* 20, 11 (1934), 584–590. https://doi.org/10.1073/pnas.20.11.584 arXiv:https://www.pnas.org/content/20/11/584.full.pdf

[11] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *Journal of the ACM (JACM)* 40, 1 (1993), 143–184.

[12] Martin Hofmann and Thomas Streicher. 1998. The groupoid interpretation of type theory. *Twenty-five years of constructive type theory (Venice, 1995)* 36 (1998), 83–111.

[13] William A Howard. 1980. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), 479–490.

[14] Christine Paulin-Mohring. 2015. Introduction to the calculus of inductive constructions.

[15] Álvaro Pelayo and Michael Warren. 2014. Homotopy type theory and Voevodsky's univalent foundations. *Bull. Amer. Math. Soc.* 51, 4 (2014), 597–648.

[16] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study.

[1] Agda uses the Unified Theory of Types, which is a combination of Calculus of Constructions and Martin-Lof's Type Theory.