Introduction to dependent type theory

Ali Caglayan

October 31, 2018

1 Introduction

Simply typed lambda calculus (STLC) has been well documented and studied by type theorists and mathematicians, and it's features have been used by many programming languages [NEED REFERENCE].

In [2] it is noted that "Research monographs on dependent and inductive types are lacking." This will essentially be one of the goals of this thesis, to provide a guide for mathematicians and computer scientists about the use of dependent type theory. As this doument is written there is no single account of all approaches to ıdependent type theory.

Awodey [1] made an observation that Dybjer's [3] categories with families (CwF) is a presheaf category with a representable natural transformation (it's fibers are representable). He then proceeds to show conditions needed to model a dependent type theory with Π, Σ and Id types.

This thesis will have three main goals.

To present a dependent type theory

To model the semantics of such a type theory using categorical methods

To discuss the applications to mathematics and computer science (proof assistants, programming languages and foundations)

Finally we may also discuss recent developments of something called "Homotopy type theory" and how that fits into the general picture.

1.1 Lambda calculus

We recall that there are 3 kinds of expressions in lambda calculus: variables, abstractions and applications. These are defined inductively on themselves. A variable is simply a string of characters from an alphabet. A lambda abstraction looks like $\lambda x.y$ where x is some variable and y is some expression. There are alternate ways of writing this, allowing us to drop the need for naming x, for example de Brujin indices. Finally an application is simply the concatenation ab of two expressions a and b. We will assume that This fully describes the syntax of this type theory. We will now introduce some rules that tell us which expressions we can derive from other expressions. Firstly we have β-reduction which tells us if we have an expression of the form $(\lambda x.y)z$ this can be reduced

to an expression where all occurrences of x in y are replaced with the expression z. We also have α-conversion which I would argue isn't really a rule as naming of variables can be completely avoided in the first place using de Brujin indices or even combinators. [2, 4]

1.2 Contexts

In mathematics we work with contexts implicitly. That is there is always an ambient knowledge of what has been defined. Mostly due to the nature of how we read mathematical papers. We can make this explicit using contexts. We will not however, use contexts in our discussion of type theory but we will provide a formal exposition in the appendix.

2 A formal simply typed lambda calculus

Our judgements:

Γ ctx	Γ is a well-formed context.
$\Gamma \vdash A$ Type	A is a type in context Γ.
$\Gamma \vdash x : A$	x is a term of type A in context Γ.

Type theory "will be about" deriving judgements from other judgements. Which can be concisely summarised in the form of an inference rule

$$\frac{A_1 \quad A_2 \quad \cdots \quad A_n}{B}$$

which says that given the judgements $A_1, \ldots, A_n$ we can derive the judgement B.

2.1 Structural rules

We now look at the rules that govern contexts and the structure of our type system.

We begin with a rule stating that the empty context (which as contexts are sets or lists is well-defined) is well-formed. Which is another way of stating that the context was grown in a specified way and is not just an arbitrary list or set of variables.

$$\frac{}{\varnothing \text{ ctx}} \text{ empty-ctx}$$

We also want the concatenation of two well-formed contexts to be well-formed.

$$\frac{\Gamma \text{ ctx} \qquad \Delta \text{ ctx}}{\Gamma, \Delta \text{ ctx}}$$

We omit rules about repeating or removing repeated elements and ordering lists (think of them as finite sets).

A variable is a statement of the form $x : A$ where x is known as the term and A its type.

2.2 Function types

We introduce a formation rule for the function type.

$$\frac{\Gamma \vdash A \text{ Type} \qquad \Gamma \vdash B \text{ Type}}{\Gamma \vdash A \to B \text{ Type}} \ (\to)\text{-form}$$

We now need a rule for producing terms of this new type. We introduce the introduction rule for the function type.

$$\frac{\Gamma, x : A \vdash y : B}{\Gamma \vdash (\lambda x.y) : A \to B} \ (\to)\text{-intro}$$

We will sometimes call this lambda abstraction. We next introduce a way to apply these functions to terms in their domains. We introduce our elimination rule for the function type.

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B} \ (\to)\text{-elim}$$

This is essentially usless unless we have a way to compute (or reduce) this expression. This is where our computation rule comes in. The computation rule will tell us how our elimination rule and introduction rule interact.

$$\Gamma, x : A, y : B, (\lambda x.y) : A \to B, a : A \vdash (\lambda x.y)(a) : B$$

3 Syntax

4 Rules

5 Symantics

Definition 1. Let C be a small category. The **category of elements** $\mathrm{el}(F)$ of a functor $F : C \to \mathbf{Set}$ is the following pullback in **Cat**:

$$\begin{array}{ccc} \mathrm{el}(F) & \xrightarrow{\rho_F} & \mathbf{Set}_* \\ {\scriptstyle \pi_F}\downarrow & & \downarrow{\scriptstyle U} \\ C & \xrightarrow{F} & \mathbf{Set} \end{array} \tag{1}$$

where U is the forgetful functor from the category of pointed sets $\mathbf{Set}_*$ to the category of sets **Set**.

Thus $\mathrm{el} : [C, \mathbf{Set}] \to \mathbf{Cat}$ is a functor. TODO: Prove this.

Definition 2. A **category with families (CwF)** consists of:

- A small category C
- A terminal object $1 \in C$
- Two presheaves $\mathbf{Tm}, \mathbf{Ty} \in [C^{\mathbf{op}}, \mathbf{Set}]$
- A morphism of presheaves **of** : $\mathbf{Tm} \to \mathbf{Ty}$
- An algebraic representation of the map **of** or in other words a right adjoint to $\mathrm{el}(\mathbf{of}) : \mathrm{el}(\mathbf{Tm}) \to \mathrm{el}(\mathbf{Ty})$ (TODO: Don't link this definition so much need rephrasing).

References

[1] S. Awodey. Natural models of homotopy type theory. *ArXiv e-prints*, June 2014.

[2] Henk Barendregt. *Lambda calculus with types*. Perspectives in logic. Cambridge University Press, Cambridge, 2013.

[3] Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 120–134, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

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