Introduction to dependent type theory

Ali Caglayan

November 16, 2018

0.1 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. [1, 5]

0.2 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.

1 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.

1.1 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.

1.2 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 useless 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.

$$\frac{\Gamma, x : A, y : B, (\lambda x.y) : A \to B, a : A \vdash (\lambda x.y)(a) : B}{\Gamma \vdash y[x/a] : B} \ (\to)\text{-comp}$$

We will describe what is known as a simply typed lambda calculus. There is a lot of literature on type theory, and it doesn't seem that there are many authors in agreement of ways to present it.

In [1] a more type theoretic approach, analysing the type theory mostly in the synctactic world. This gives us a good starting point for how we want our type theory to be presented however it may not be so easy to keep an eye on how the categorical semantics (the ways we model types in mathematics) behave. In order to do this we will use references such as [2, 3, 4]. This will be from the more categorical logic school of thought, which will study type theory that is "generated" by certain categories in interest.

We start by describing a general class of simple type theories as outlined in [3]. Firstly we introduce the notion of a *signature*. Similar accounts can be found in [2]. This will essentially consist of "generating" a category from some signature (which can be thought of as a stripped down type theory syntax), and then studying the functors from that category into other categories. This allows nice properties from the second category to be "pulled back" onto our type theory giving it features we desire.

Definition 1.1. A **signature** is a pair $(\mathbf{Typ}, \mathcal{F})$ where **Typ** is a finite set of **basic** (or **atomic**) **types**. And a functor $\mathcal{F} : \mathbf{Typ}^\star \times \mathbf{Typ} \to \mathbf{Set}$. Where $\mathbf{Typ}^\star$ is the Kleene-Star operation on a set (or the free monoid over **Typ**), defined as $X^\star := \bigcup_{n \in N} X^n$ whose elements are finite tuples of elements of X for a set X. We have **Set** for the category of finite sets. Note that the sets in the domain of the functor are realised as discrete categories.

We will usually write a signature as $\Sigma := (\mathbf{Typ}, \mathcal{F})$, denote $|\Sigma| := \mathbf{Typ}$ and write $F : \sigma_1, \ldots, \sigma_n \to \sigma_{n+1}$ when $F \in \mathcal{F}((\sigma_1, \ldots, \sigma_n), \sigma_{n+1})$.

Definition 1.2. Let **Var** be a countable set. Elements $x \in \mathbf{Var}$ are called **variables**.

Note this style of variables is essentially de Brujin indices. But allows us to have a set of names for our variables, which allows future annoyances like α-equivalence to be sorted out easily due to the plentiful existence of bijections from $\mathbf{Var} \to \mathbf{Var}$.

Definition 1.3. A **variable declaration** is a pair $(x, \sigma) \in \mathbf{Var} \times \mathbf{Typ}$ usually written as $x : \sigma$. This can be read as "the variable x has type σ. We will define $\mathbf{Dec} := \mathbf{Var} \times \mathbf{Typ}$.

Definition 1.4. A **context** Γ is an element of $\mathbf{Con} := \mathbf{Dec}^\star$. In other words, a context is a finite list of variable declarations. We will usually write a context Γ as $v_1 : \sigma_1, \ldots, v_n : \sigma_n$. Note that the Kleene-Star has a monoid structure with operation ",". We can thus give **Con** a monoid structure and write, for contexts Γ and Δ another context Γ, Δ which is the concatenation of two contexts. The notation here allows the "expanded version" to coincide, as in Γ, Δ can be written as $v_1 : \sigma_1, \ldots, v_n : \sigma_n, w_1 : \tau_1, \ldots, w_m, \tau_m$.

We also note that there is a canonical inclusion $\mathbf{Dec} \hookrightarrow \mathbf{Con}$ given that **Dec** freely generates the monoid **Con**. This will allow us to write $\Gamma, x : \tau$ for $v_1 : \sigma_1, \ldots, v_n : \sigma_n, x : \tau$.

We now denote the basic statements of our language. These statements are called **judgements** and we will derive

References

[1] Barendregt, H., 2013. *Lambda calculus with types*, Perspectives in logic. Cambridge: Cambridge University Press.

[2] Crole, R.L., 1993. *Categories for types*. Cambridge: Cambridge University Press.

[3] Jacobs, B., 1999. *Categorical logic and type theory*, Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland.

[4] Lambek, J., 1986. *Introduction to higher order categorical logic*, Cambridge studies in advanced mathematics ; 7. Cambridge: Cambridge University Press.

[5] Univalent Foundations Program, T., 2013. *Homotopy type theory: Univalent foundations of mathematics*. Institute for Advanced Study: `https://homotopytypetheory.org/book`.