Introduction to dependent type theory

Ali Caglayan

November 27, 2018

Barendregt [1] (or B for short) introduces *simply typed lambda calculus* by introducing three versions $\boldsymbol{\lambda}^{\mathrm{Cu}}_{\to}$, $\boldsymbol{\lambda}^{\mathrm{Ch}}_{\to}$, $\boldsymbol{\lambda}^{\mathrm{dB}}_{\to}$.

1 Type theory

1.1 Untyped lambda calculus

Definition 1.1. Let **Var** (what B calls V) be a set of variables perhaps defined as $\mathbf{Var} := \{x, x', x'', \dots\}$. We will use B's inductive notation and write this as

$$\mathbf{Var} ::= x \mid \mathbf{Var}'$$

which is read: elements of **Var** are of the form x or an element of **Var** with a $'$.

We then define a set **Tm** (what B calls Λ) of terms (what B calls lambda terms).

Definition 1.2. Elements of **Tm** are defined as follows

$$\mathbf{Tm} ::= \mathbf{Var} \mid \lambda\ \mathbf{Var}\ \mathbf{Tm} \mid \mathbf{Tm}\ \mathbf{Tm}$$

where a **term** is either a **variable**, a **lambda term** (usually of the form $\lambda x.t$) or an **application** of two terms.

B goes ahead and eases the notation slightly, which we also do. This is for readability mostly.

Remark 1.3. We introduce the following notation:

(i) Letting $x, y, z, \dots, x_0, y_0, z_0, \dots, x_1, y_1, z_1, \dots$ denote arbitrary variables.

(ii) $M, N, L, \dots$ denote arbitrary lambda terms (elements of **Tm**).

(iii) Application of terms is left-associative i.e. $A(B(C\cdots)) \equiv ABC\cdots$

(iv) Lambda terms are right-associative i.e. $\lambda x_1.(\cdots(\lambda x_n.M)\cdots) \equiv \lambda x_1 \cdots x_n.M$

If we were to choose not to introduce these notational simplifications, it would be very tedious to write all the brackets and not very helpful to the reader.

We will now introduce the notion of a **free variable**.

Definition 1.4. Let $M \in \mathbf{Tm}$.

(i) The set of **free variables** of M, written $\mathrm{FV}(M)$. Variables that are not free are called **bound**.

(ii) If $\mathrm{FV}(M) = \varnothing$, then M is called **closed** or a **combinator**. The set of combinators can be written as

$$\mathbf{Tm}^{\varnothing} = \{M \in \mathbf{Tm} \mid \mathrm{FV}(M) = \varnothing\}$$

We can define $\mathrm{FV} : \mathbf{Tm} \to P(\mathbf{Var})$ by induction on M which we can do due to the inductive definition of **Tm**. So we have three cases:

$$\begin{aligned} M \equiv x, &\quad \mathrm{FV}(M) := \{x\} \\ M \equiv \lambda x.N, &\quad \mathrm{FV}(M) := \mathrm{FV}(N) - \{x\} \\ M \equiv NL, &\quad \mathrm{FV}(M) := \mathrm{FV}(N) \cup \mathrm{FV}(L) \end{aligned}$$

Example 1.5. Some well known combinators are $\mathbf{I} :\equiv \lambda x.x$, $\mathbf{K} :\equiv \lambda xy.y$ and $\mathbf{S} :\equiv \lambda xyz.xz(yz)$. These are well studied however we will not discuss them much here. For a comprehensive study of various combinators and their uses see [2].

We now define (untyped) lambda calculus. B does this by defining what they call an equational theory on **Tm**. This is where the calculus has a notion of equality. We will simply say that this equality is an equality from the metatheory (the logic used to define the calculus). For all intents and purposes our logic is first order logic with ZFC. Although it is very unlikely we will use choice anywhere.

Definition 1.6. The symbol $\equiv$ denotes the equality in the metatheory. This will have all the usual properties of an equivalence relation and also play nicely with our terms. For example $M \equiv N \implies \lambda x.M \equiv \lambda x.N$.

This means that we will not have to define properties like reflexivity and transitivity as they essentially come for free from our metatheory. This also has the advantage that we can comfotably add equalities (forcing two things to be equal) without having to define it in our calculus.

We go onto define $\boldsymbol{\lambda\beta\eta}$ as the terms **Tm** modulo the equivalence relation of the equality in our metatheory. To which we will add the following equalities:

$$(\lambda x.M)N \equiv M[x := N] \qquad (\boldsymbol{\beta}\text{-rule})$$

$$\lambda x.Mx \equiv M \qquad (\boldsymbol{\eta}\text{-rule})$$

Note that when we write terms from now on we are really talking about the representative of the equivalence class of terms in the set of terms modulo our definitional equality. B talks about reductive theories where we have essentially inference rules giving

$$\frac{(\lambda x.M)N}{M[x := N]}\,(\boldsymbol{\beta})$$

$$\frac{\lambda x.Mx}{M}\,(\boldsymbol{\eta})$$

Remark 1.7. It is here that B talks about α-equivalence. We will go ahead and do the same by adding in equalities for α-conversion of terms. Thus our terms modulo definitional equality will be up to α-equivalence too.

Remark 1.8. B also talks about properties of the reduction defined such as satisfaction of the Church-Rosser theorem. This is not entirely relevent here but may be important that it holds.

1.2 Simple types

So far we have been working in untyped lambda calculus, which in itself has been the basis of many functional programming languages. However for our purposes we could argue it is uninteresting.

We will now try to classify our terms in such a way that we assign a type to them. Then we will restrict our lambda terms' applicability by checking the type. This may seem restrictive but it is a very useful notion that will be prevelent in the theory to come.

Definition 1.9.

References

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

[2] Smullyan, R., 2012. *To mock a mocking bird.* Knopf Doubleday Publishing Group. Available from: `https://books.google.co.uk/books?id=NyF1kvJhZbAC`.