Introduction to dependent type theory

Ali Caglayan

November 28, 2018

Contents

1 Type theory

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

1.1 Untyped lambda calculus

Definition 1.1.1. 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 **Term** (what B calls Λ) of terms (what B calls lambda terms).

Definition 1.1.2. Elements of **Term** are defined as follows

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

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 1.1.3. 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) $M, N, L, \ldots$ denote arbitrary lambda terms (elements of **Term**).

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

(iv) Nested lambda terms have their inner lambdas dropped i.e. $\lambda x_1.(\lambda x_2.M) \equiv \lambda x_1 x_2.M$. Although this will almost never be used.

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 1.1.4. Let $M \in$ **Term**.

(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

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

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

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

Example 1.1.5. 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 **Term**. 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 1.1.6. 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.

Definition 1.1.7. (Term substitution) We define term substituion by induction on . Let $a, b \in \mathbf{Var}$ and $M, N, K \in \mathbf{Term}$

$$\begin{aligned} a[a := M] &:\equiv M \\ b[a := M] &:\equiv b, \qquad b \not\equiv a \\ (KN)[a := M] &:\equiv (K[a := M])(N[a := M]) \\ (\lambda x.N)[a := M] &:\equiv \lambda(x[a := M]).(N[a := M]) \end{aligned}$$

Note in the last case if M is not a variable and replaces x we may get a nonsense term. We will explicitly disallow this but it doesn't matter too much.

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

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

$$\lambda x.Mx \equiv M, \quad x \notin \mathrm{FV}(M) \tag{$\boldsymbol{\eta}$-rule}$$

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

$$\frac{(\lambda x.M)N}{M[x := N]}\ (\boldsymbol{\beta}) \qquad\qquad \frac{\lambda x.Mx}{M}\ (\boldsymbol{\eta}) \quad x \notin \mathrm{FV}(M)$$

Remark 1.1.8. 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 1.1.9. 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.

[TODO: REMOVE]

Remark 1.1.10. We used the notation $M[x := N]$ which means take all (free) occurances of the variable x inside the term M and replace it with the term N. Note it isn't clear what $(\lambda x.t)[x := N]$ should do when N is a term, however we will try to restrict it's use for free variables only. We also note that in other literature this is written $M[x/N]$ and perhaps sometimes confusingly $M[N/x]$. We will occasionally use the former, but never the latter. It should be read: "replace every free occurance of x with the term N". [TODO: Perhaps make this a definition?? There are many problems with this as it stands, I would hardly think this is a remark.]

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 1.2.1. Let $\mathbb{A}$ be a set. An element of $\mathbb{A}$ is called an **type atom**. The set of **simple types** over $\mathbb{A}$, written **Type** is defined by induction as follows:

$$\mathbf{Type} ::= \mathbb{A} \mid \mathbf{Type} \to \mathbf{Type}$$

that is a simple type is either an atomic type or a **function type** between two atomic types. We define $\to$ to be an operator in the definition. It can be said that **Type** is a tree, with leaves type atoms and braches the operator $\to$ pointing to its arguments. The operator $\to$ will be right-associative, i.e. $A \to (B \to C) \equiv A \to B \to C$.

Example 1.2.2. We give the following examples of type atoms:

(i) Let $\mathbb{A}_0 := \{\mathsf{T}\}$. Then the set of simple types **Type** over $\mathbb{A}_0$ consiststs of the following elements:

$$\mathbf{Type} = \{\mathsf{T},\ \mathsf{T} \to \mathsf{T},\ \mathsf{T} \to \mathsf{T} \to \mathsf{T},\ (\mathsf{T} \to \mathsf{T}) \to \mathsf{T},\ \dots\}$$

(ii) Let $\mathbb{A}_\infty := \{\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3, \dots\}$ be a countable set. Then the set of simple types over $\mathbb{A}_\infty$ is also countable. Note for any natural number n we can count trees with depth n. Then we take the union of all these sets of trees. Giving us a countably indexed union of countable sets, hence countable. This may be a useful fact if we are to iterate over types in the future. [TODO: Rephrase this argument]

(iii) Let $\mathbb{A}_\varnothing := \varnothing$. Clearly **Type** $= \varnothing$. This isn't a very interesting collection of types.

Remark 1.2.3. We will usually reference types as upper case variables and perhaps greek letters for type atoms. And terms as lower case variables. It should be clear from the context which is which. When we write $\mathbb{A}$ we mean a generic set of type atoms.

Definition 1.2.4. Let $,\beta \in \mathbb{A}$ and $A, B, C \in$ **Type**. The operation of type substitution is defined by induction on **Type** as follows:

$$\begin{aligned} \alpha[\alpha := A] &:\equiv A \\ \beta[\alpha := A] &:\equiv \beta, \quad \beta \not\equiv \alpha \\ (A \to B)[\alpha := C] &:\equiv (A[\alpha := C]) \to (B[\alpha := C]) \end{aligned}$$

$$\frac{}{\Gamma \vdash a : A} \text{ if } (a : A) \in \Gamma$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash fx : B} \ (\to\text{-elimination})$$

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

Figure 1: Inference rules of $\boldsymbol{\lambda}_{\to}^{\text{Cu}}$

We now explore three different versions of simply typed lambda calculus, $\boldsymbol{\lambda}_{\to}^{\text{Cu}}$, $\boldsymbol{\lambda}_{\to}^{\text{Ch}}$and $\boldsymbol{\lambda}_{\to}^{\text{dB}}$. Note that the definitions in each section will be local as we will use the same words but subtly change definitions. After we introduce these we will choose one system to develop on.

1.2.1 Curry style simply typed lambda calculus

Definition 1.2.5. We now define $\boldsymbol{\lambda}_{\to}^{\text{Cu}}$

(i) A **(type assignment) statement** is of the form $a : A$ with $a \in$ **Term** and $A \in$ **Type**. The type A is called the **predicate** and the term a is called the subect of the statement.

(ii) A **(typing) decleration** is a statement with subject a term variable.

(iii) A **basis** [context/well-formed context?] is a set of declerations with *distinct* variables as subjects.

(iv) A statement $a : A$ is **derivable** from a basis Γ, written

$$\Gamma \vdash_{\boldsymbol{\lambda}_{\to}}^{\text{Cu}} a : A$$

(usually just $\vdash$ when it is obvious) if $\Gamma \vdash a : A$ can be produced by the inference rules listed in Figure 1. Note $\Gamma, x : A$ denotes concatenation which will be defined later.

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`.