Introduction to dependent type theory

Ali Caglayan

December 6, 2018

Contents

1 Induction

1.1 Well-founded induction

The notion of well-founded induction is a standard theorem of set theory. The classical proof of which usually uses the law of excluded middle [7, p. 62], [3, Ch. 7]. It's use in the formal semantics of programming languages is not much different either [13, Ch. 3]. There are however more constructive notions of well-foundedness [9, §8] with more careful use of excluded middle. We will follow [11], as this is the simplest to understand, and we won't be using this material much other than an initial justification for induction in classical mathematics.

Definition 1.1.1. Let X be a set and $\prec$ a binary relation on X. A subset $Y \subseteq X$ is called **$\prec$-inductive** if

$$\forall x \in X, \quad (\forall y \prec x,\ y \in Y) \Rightarrow x \in Y.$$

Definition 1.1.2. The relation $\prec$ is **well-founded** if the only $\prec$-inductive subset of X is X itself. A set X equipped with a well-founded relation is called a *well-founded set*.

Theorem 1.1.3 (Well-founded induction principle)**.** Let X be a well-founded set and P a property of the elements of X (a proposition). Then

$$\forall x \in X, P(x) \Leftrightarrow \forall x \in X, \quad (\forall y \prec x, P(y)) \Rightarrow P(x).$$

Proof. The forward direction is clearly true. For the converse, assume $\forall x \in X, ((\forall y \prec x, P(y)) \Rightarrow P(x))$. Note that $P(y) \Leftrightarrow x \in Y := \{x \in X \mid P(x)\}$ which means our assumption is equivalent to $\forall x \in X,\ (\forall y \prec x,\ y \in Y) \Rightarrow x \in Y$ which means Y is $\prec$-inductive by definition. Hence by 1.1.2 $Y = X$ giving us $\forall x \in X, P(x)$. □

2 Syntax

2.1 Introduction

We will follow the structure of syntax outlined in Harper [5]. There are several reasons for this.

Firstly, for example in Barendregt et. al. [1] we have notions of substitution left to the reader under the assumption that they can be fixed. Generally Barendregt's style is like this and even when there is much formalism, it is done in a way that we find peculiar.

In Crole's book [4], syntax is derived from an *algebraic signature* which comes directly from categorical semantics. We want to give an independent view of type theory. The syntax only has types as well, meaning that only terms can be posed in this syntax. Operations on types themselves would have to be handled separately. This will also make it difficult to work with *bound variables*.

In Lambek and Scott's book [8], very little attention is given to syntax and categorical semantics and deriving type theory from categories for study is in the forefront of their focus.

In Jacob's book [6], we again have much reliance on categorical machinery. A variant of algebraic signature called a many-typed signature is given, which has its roots in mathematical logic. Here it is discussed that classically in logic the idea of a sort and a type were synonymous, and they go onto preferring to call them types. This still has the problems identified before as terms and types being treated separately, when it comes to syntax.

In Barendregt's older book [2], there are models of the syntax of (untyped) lambda calculus, using Scott topologies on complete lattices. We acknowledge that this is a working model of the lambda calculus but we believe it to be overly complex for the task at hand. It introduces a lot of mostly irrelevant mathematics for studying the lambda calculus. And we doubt very much that these models will hold up to much modification of the calculus. Typing seems impossible.

In Sørensen and Urzyczyn's book [10] a more classical unstructured approach to syntax is taken. This is very similar to the approaches that Church, Curry and de Brujin gave early on. The difficulty with this approach is that it is very hard to prove things about the syntax. There are many exceptional cases to be weary of (for example if a variable is bound etc.). It can also mean that the syntax is vulnerable to mistakes. We acknowledge it's correctness in this case, however we prefer to use a safer approach.

We will finally look at one more point of view, that of mathematical logic. We look at Troelstra and Schwichtenberg's book [12] which studies proof theory. This is essentially the previous style but done to a greater extent, for they use that kind of handling of syntax to argue about more general logics. As before, we do not choose this approach.

We have seen books from either end of the spectrum, on one hand Barendregt's type theoretic camp, and on the other, the more categorical logically oriented camp. We have argued that the categorical logically oriented texts do not do a good job of explaining and defining syntax, their only interest is in their categories. The type theoretic texts also seem to be on mathematically shaky ground, sometimes much is left to the reader and finer details are overlooked.

Harper's seems more sturdy and correct in our opinion. Harper doesn't concern himself with abstraction for the sake of abstraction but rather when it will benefit the way of thinking about something. The framework for working with syntax also seems ideal to work with, when it comes to adding features to a theory (be it a type theory or otherwise).

2.2 Abstract Syntax Trees

We begin by outlining what exactly syntax is, and how to work with it. This will be important later on if we want to prove things about our syntax as we will essentially have good data structures to work with.

Definition 2.2.1 (Sorts)**.** Let $\mathcal{S}$ be a finite set, which we will call **sorts**. An element of $\mathcal{S}$ is called a **sort**.

A sort could be a term, a type, a kind or even an expression. It should be thought of an abstract notion of the kind of syntactic element we have. Examples will follow making this clear.

Definition 2.2.2 (Arities)**.** An **arity** is an element $((s_1, \ldots, s_n), s)$ of the set of **arities** $\mathcal{Q} := \mathcal{S}^\star \times \mathcal{S}$ where $\mathcal{S}^\star$ is the Kleene-star operation on the set $\mathcal{S}$ (a.k.a the free monoid on $\mathcal{S}$ or set of finite tuples of elements of $\mathcal{S}$). An arity is typically written as $(s_1, \ldots, s_n)s$.

Definition 2.2.3 (Operators)**.** Let $\mathcal{O} := \{\mathcal{O}_\alpha\}_{\alpha \in \mathcal{Q}}$ be an $\mathcal{Q}$-indexed (arity-indexed) family of disjoint sets of **operators** for each arity. An element $o \in \mathcal{O}_\alpha$ is called an **operator** of arity α. If o is an operator of arity $(s_1, \ldots, s_n)s$ then we say o has **sort** s and that o has n **arguments** of sorts $s_1, \ldots, s_n$ respectively.

Definition 2.2.4 (Variables)**.** Let $\mathcal{X} := \{\mathcal{X}_s\}_{s \in \mathcal{S}}$ be an $\mathcal{S}$-indexed (sort-indexed) family of disjoint (finite?) sets $\mathcal{X}_s$ of **variables** of sort s. An element $x \in \mathcal{X}_s$ is called a **variable** x of **sort** s.

Definition 2.2.5 (Fresh variables)**.** We say that x is **fresh** for $\mathcal{X}$ if $x \notin \mathcal{X}_s$ for any sort $s \in \mathcal{S}$. Given an x and a sort $s \in \mathcal{S}$ we can form the family $\mathcal{X}, x$ of variables by adding x to $\mathcal{X}_s$.

Remark 2.2.6. The notation $\mathcal{X}, x$ is ambiguous because the sort s associated to x is not written. But this can be remedied by being clear from the context what the sort of x should be.

Definition 2.2.7 (Abstract syntax trees)**.** The family $\mathcal{A}[\mathcal{X}] = \{\mathcal{A}[\mathcal{X}]_s\}_{s \in \mathcal{S}}$ of **abstract syntax trees** (or asts), of **sort** s, is the smallest family satisfying the following properties:

1. A variable x of sort s is an ast of sort s: if $x \in \mathcal{X}_s$, then $x \in \mathcal{A}[\mathcal{X}]_s$.
2. Operators combine asts: If o is an operator of arity $(s_1, \ldots, s_n)s$, and if $a_1 \in \mathcal{A}[\mathcal{X}]_{s_1}, \ldots, a_n \in \mathcal{A}[\mathcal{X}]_{s_n}$, then $o(a_1; \ldots; a_n) \in \mathcal{A}[\mathcal{X}]_s$.

Remark 2.2.8. The idea of a smallest family satisfying certain properties is that of structural induction. So another way to say this would be a family of sets inductively generated by the following constructors.

Remark 2.2.9. An ast can be thought of as a tree whose leaf nodes are variables and branch nodes are operators.

Example 2.2.10 (Syntax of lambda calculus)**.** The (untyped) lambda calculus has one sort **Term**, so $\mathcal{S} = \{\mathbf{Term}\}$. We have an operator App of application whose arity is $(\mathbf{Term}, \mathbf{Term})\mathbf{Term}$ and an family of operators $\{\lambda_x\}_{x \in \mathbf{Var}}$ which is the lambda abstraction with bound variable x, so $\mathcal{O} = \{\lambda_x\} \cup \{\mathrm{App}\}$. The arity of each λ_x for some $x \in \mathbf{Var}$ is simply $(\mathbf{Term})\mathbf{Term}$.

Consider the term

$$\lambda x.(\lambda y.xy)z$$

We can consider this the *sugared* version of our syntax. If we were to *desugar* our term to write it as an ast it would look like this:

$$\lambda_x(\text{App}(\lambda_y(\text{App}(x; y)); z))$$

Sugaring allows for long-winded terms to be written more succinctly and clearly. Most readers would agree that the former is easier to read. We have mentioned the tree structure of asts so we will illustrate with the following equivalent examples. We present two to allow for use of both styles.

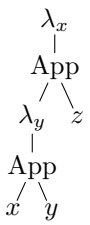

Figure 1: Vertically oriented tree representing the lambda term

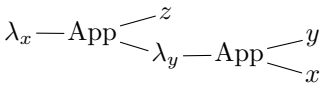

Figure 2: Horizontally oriented tree representing the lambda term

Remark 2.2.11. Note that later we will enrich our notion of abstract syntax tree that takes into account binding and scope of variables but for now this is purely structural.

Remark 2.2.12. When we prove properties $\mathcal{P}(a)$ of an ast a we can do so by structural induction on the cases above. We will define structural induction as a special case of well-founded induction. But for this we will need to define a relation on asts.

Definition 2.2.13. Suppose $\mathcal{X} \subseteq \mathcal{Y}$. An ast $a \in \mathcal{A}[\mathcal{X}]$ is a **subtree** of an ast $b \in \mathcal{A}[\mathcal{Y}]$ [This part is giving me a headache. How can I define subtree if I can't do it by induction? To do it by induction I would have to define subtree.]

[Some more notes on structural induction, perhaps this can be defined and discussed with trees in the section before?]

[add examples of sorts, operators, variables and how they fit together in asts]

Lemma 2.2.14. If we have $\mathcal{X} \subseteq \mathcal{Y}$ then, $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$.

Proof. Suppose $\mathcal{X} \subseteq \mathcal{Y}$ and $a \in \mathcal{A}[\mathcal{X}]$, now by structural induction on a:

1. If a is in $\mathcal{X}$ then it is obviously also in $\mathcal{Y}$.

2. If $a := o(a_1; \ldots; a_n) \in \mathcal{A}[\mathcal{X}]$ we have $a_1, \ldots, a_n \in \mathcal{A}[\mathcal{X}]$ also. By induction we can assume these to be in $\mathcal{A}[\mathcal{Y}]$ hence giving us $a \in \mathcal{A}[\mathcal{Y}]$.

Hence by induction we have shown that $\mathcal{A}[\mathcal{X}] \subseteq \mathcal{A}[\mathcal{Y}]$. □

Definition 2.2.15 (Substitution)**.** If $a \in \mathcal{A}[\mathcal{X}, x]_{s'}$, and $b \in \mathcal{A}[\mathcal{X}]_s$, then $[b/x]a \in \mathcal{A}[\mathcal{X}]_{s'}$ is the result of **substituting** b for every occurrence of x in a. The ast a is called the **target**, the variable x is called the **subject** of the **substitution**. We define substitution on an ast a by induction:

1. $[b/x]x = b$ and $[b/x]y = y$ if $x \neq y$.
2. $[b/x]o(a_1; \ldots; a_n) = o([b/x]a_1; \ldots; [b/x]a_n)$

[Examples of substitution]

Corollary 2.2.16. If $a \in \mathcal{A}[\mathcal{X}, x]$, then for every $b \in \mathcal{A}[\mathcal{X}]$ there exists a unique $c \in \mathcal{A}[\mathcal{X}]$ such that $[b/x]a = c$.

Proof. By structural induction on a, we have three cases: $a := x$, $a := y$ where $y \neq x$ and $a := o(a_1; \ldots; a_n)$. In the first we have $[b/x]x = b = c$ by definition. In the second we have $[b/x]y = y = c$ by definition. In both cases $c \in \mathcal{A}[\mathcal{X}]$ and are uniquely determined. Finally, when $a := o(a_1; \ldots; a_n)$, we have by induction unique $c_1, \ldots, c_n$ such that $c_i := [b/x]a_i$ for $1 \leq i \leq n$. Hence we have a unique $c = o(c_1, \ldots, c_n) \in \mathcal{A}[\mathcal{X}]$. □

Remark 2.2.17. Note that 2.2.16 was simply about checking Definition 2.2.15. We have written out a use of the definition here so we won't have to again in the future.

References

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

[2] H.P. Barendregt. *The lambda calculus: its syntax and semantics.* Studies in logic and the foundations of mathematics. North-Holland, 1984.

[3] J. Barwise. *Handbook of Mathematical Logic.* Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1982.

[4] Roy L Crole. *Categories for types.* Cambridge University Press, Cambridge, 1993.

[5] Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, 2 edition, 2016.

[6] B. Jacobs. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[7] P.T. Johnstone. *Notes on Logic and Set Theory*. Cambridge mathematical textbooks. Cambridge University Press, 1987.

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

[9] Michael Shulman. Comparing material and structural set theories. *ArXiv e-prints*, page arXiv:1808.05204, August 2018.

[10] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., New York, NY, USA, 2006.

[11] Paul Taylor. Intuitionistic sets and ordinals. *The Journal of Symbolic Logic*, 61(3):705–744, 1996.

[12] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2 edition, 2000.

[13] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of computing. Zone Books, U.S., 1993.