Introduction to dependent type theory

Ali Caglayan

February 26, 2019

Contents

1 Introduction

The aim of this thesis is to present to two sorts of audience, the utility of dependent type theorists. The audiences that I have in mind are computer scientists, roughly individuals who wish to write good code, and mathematicians, roughly individuals who wish to write good proofs.

These will be our main aims however we do also wish to develop the machinery formally.

2 Propositions as types

There is a rich interplay between programming and logic known as the Curry-Howard correspondance or propositions as types.

3 What is type theory

Type theory is the study of types systems. That is a system that orginizes data manipulated by programs into types. This has been a very useful concept in computer science. It has allowed the writing of programs taht a more

3.1 Lambda calculus

3.2 Modelling type theory

4 What is dependent type theory?

4.1 What are dependent types?

4.2 Motivation for computer scientists

4.3 Motivation for mathematicians

4.4 Category theory

4.5 Categorical logic

4.6 Future directions

5 Syntax

5.1 Introduction

We will follow the structure of syntax outlined in Harper [6]. 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 [5], 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 [10], 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 [7], 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 [18] 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 [20] 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).

5.2 Well-founded induction

Firstly we will begin a quick recap of induction. This should be a notion familiar to computer scientists and mathematicians alike. The following will be more

accessible to mathematicians but probably more useful for them too since they will be generally less familiar with the generality of 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 [8, p. 62], [4, Ch. 7]. It's use in the formal semantics of programming languages is not much different either [21, Ch. 3]. There are however more constructive notions of well-foundedness [16, §8] with more careful use of excluded middle. We will follow [19], 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 5.2.1. 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 5.2.2. 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 5.2.3 (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 5.2.2 $Y = X$ giving us $\forall x \in X, P(x)$. □

We now get onto some of the tools we will be using to model the syntax of our type theory.

5.3 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 5.3.1 (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 5.3.2 (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 5.3.3 (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 5.3.4 (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 5.3.5 (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 $\mathcal{X}_s$.

[[Wording here may be confusing]]

Definition 5.3.6 (Fresh sets of variables)**.** Let $V = \{v_1, \ldots, v_n\}$ be a finite set of variables (which all have sorts implicitly assigned so really a family of variables $\{V_s\}_{s \in \mathcal{S}}$ indexed by sorts, where each V_s is finite). We say V is fresh for $\mathcal{X}$ by induction on V. Suppose $V = \varnothing$, then V is fresh for X. Suppose $V = \{v\} \cup W$ where W is a finite set, v is fresh for W and W is fresh for $\mathcal{X}$. Then V is fresh for $\mathcal{X}$ if v is fresh for $\mathcal{X}$. By induction we have defined a finite set being fresh for a set $\mathcal{X}$. Write $\mathcal{X}, V$ for the union (which is disjoint) of $\mathcal{X}$ and V. This gives us a new set of variables with obvious indexing.

Remark 5.3.7. 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 5.3.8 (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 5.3.9. 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 5.3.10. An ast can be thought of as a tree whose leaf nodes are variables and branch nodes are operators.

Example 5.3.11 (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.

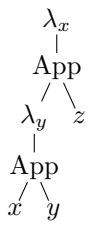

Figure 1: Vertically oriented tree representing the lambda term

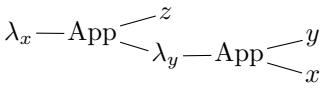

Figure 2: Horizontally oriented tree representing the lambda term

Remark 5.3.12. 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 5.3.13. 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 5.3.14. 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 5.3.15. 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 5.3.16 (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 5.3.17. 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 5.3.18. Note that 5.3.17 was simply about checking Definition 5.3.16. We have written out a use of the definition here so we won't have to again in the future.

Abstract syntax trees are our starting point for a well-defined notion of syntax. We will modify this notion, as the author of [6] does, with slight modifications that are used in [12, 13], the Initiality Project. This is a collaborative project for showing initiality of dependent type theory (the idea that some categorical model is initial in the category of such models). It is a useful reference because it has brought many mathematicians together to discuss the intricate details of type theory. The definitions here have spawned from these discussions on the nlab and the nforum.

We want to modify the notion of abstract syntax tree to include features such as binding and scoping. This is a feature used by many type theories (and even the lambda calculus). It is usually added on later by keeping track of bound and free variables. [CITE]. We will avoid this approach as it makes inducting over syntax more difficult.

Definition 5.3.19 (Generalized arities)**.** A **generalised arity** (or signature) is a tuple consisting of the following data:

1. A sort $s \in \mathcal{S}$.
2. A list of sorts of length n called the **argument sorts**, where n is called the **argument arity**.

3. A list of sorts of length m called the **binding sorts**, where m is called the **binding arity**.
4. A decidable relation $\triangleleft$ between $[n]$ and $[m]$ called **scoping**. Where $j \triangleleft k$ means the jth argument is in scope of the kth bound variable.

The set of generalised arities **GA** could therefore be defined as $\mathcal{S} \times \mathcal{S}^\star \times \mathcal{S}^\star$ equipped with some appropriate relation $\triangleleft$.

Remark 5.3.20. In [6] there is no relation but a function. And each argument has bound variables assigned to it. But as argued in [13] this means arguments can have different variables bound even if they are really the same variable. To fix this, bound variables belong to the whole signature. Which confidently makes it simpler to understand too.

This definition is more general than the definition given in [13] due to bound variables having sorts chosen for them rather than being defaulted to the sort tm. It is mentioned there however that it can be generalised to this form (but would have little utility there).

We will now redefine the notion of operator, taking note that generalised arities are a super-set of arities defined previously.

Definition 5.3.21 (Operators (with generalized arity))**.** Let $\mathcal{O} := \{\mathcal{O}_\alpha\}_{\alpha \in \mathbf{GA}}$ be a **GA**-indexed family of disjoint sets of **operators** for each generalised arity α. An element $o \in \mathcal{O}_{\alpha \in \mathbf{GA}}$ is called an operator of (generalised) **arity** α. If α has sort s then o has **sort** s. If α has argument sorts $(s_1, \ldots, s_n)$ then we say that o has **argument arity** n, with the jth argument having **sort** s_j. If α has binding sorts $(t_1, \ldots, t_m)$ then we say that o has **binding arity** m, with the kth bound variable having **sort** s_k. If the the scoping relation of α has $j \triangleleft k$ then we say that the jth argument of o is in **scope** of the kth bound variable of o.

Remark 5.3.22. We overload the definitions of arity and operator to mean generalised operator and operator with generalised arity respectively.

Now that we can equip our operators with the datum of binding and scoping we can go ahead and define abstract binding trees.

[[Lots of concepts for asts have been redefined for abts, perhaps its worth making note of that back in the asts definitions]]

Definition 5.3.23 (Abstract binding trees)**.** The family $\mathcal{B}[\mathcal{X}] = \{\mathcal{B}[\mathcal{X}]_s\}_{s \in \mathcal{S}}$ of **abstract binding trees** (or abts), of **sort** s, is the smallest family satisfying the following properties:

1. A variable x of sort s is an abt of sort s: if $x \in \mathcal{X}_s$, then $x \in \mathcal{B}[\mathcal{X}]_s$.

2. Suppose `G` is an operator of sort s, argument arity n and binding arity m. Suppose V is some finite set of length m which is fresh for $\mathcal{X}$. These will be called our **bound variables**. Label the elements of V as $V = \{v_1, \ldots, v_m\}$. For $j \in [n]$, let $X_j := \{v_k \in V \mid j \triangleleft k\}$ be the set of bound

variables that the jth argument is in scope of. Now suppose for each $j \in [n]$, $M_j \in \mathcal{B}[\mathcal{X}, V]_{s_j}$ where s_j is the sort of the jth argument of $\mathtt{G}$. Then $\mathtt{G}(X; M_1, \ldots, M_n) \in \mathcal{B}[\mathcal{X}]_s$.

Remark 5.3.24. There is a lot going on in the second constructor of Definition 5.3.23. It simply allows for bound variables to be constructed in syntax in a well-defined way that avoids variable capture. This will be useful when defining notions like substitution on abts as we will have the avoidance of variable capture built-in.

[[What is variable capture talk about this and reference this stuff because lots of cleverer people have thought about this too you know.]]

6 Category theory

6.1 Introduction

We will introduce basic category theory. Good references are: [**?** 3, 11, 15]

Category theory will allow us to model the desired behaviour of dependent types.

Definition 6.1.1. A **category** $\mathcal{C}$ consists of:

- A class $\mathrm{Ob}(\mathcal{C})$ (usually simply denoted $\mathcal{C}$ without ambiguity) of **objects**.
- For each object $A, B \in \mathcal{C}$, a set $\mathcal{C}(A, B)$ of **morphisms** or **arrows** called a **homset**. When writing $f \in \mathcal{C}(A, B)$ we usually denote this $f : A \to B$.
- For each object $A \in \mathcal{C}$ a morphism $1_A : A \to A$ called the **identity**.
- For each object $A, B, C \in \mathcal{C}$, and for each $f : A \to B$ and $g : B \to C$ there is a function (written infix or sometimes simply omitted $(gf \equiv g \circ f)$

$$- \circ - : \mathcal{C}(B, C) \times \mathcal{C}(A, B) \to \mathcal{C}(A, C)$$

  called **composition**.

Such that the following hold:

- (Identity) For each $A, B \in \mathcal{C}$ and $f : A \to B$ we have $f \circ 1_A = f$ and $1_B \circ f = f$.
- (Associativity) For all $A, B, C, D \in \mathcal{C}$ and $f : A \to B$, $g : B \to C$, $h : C \to D$. We have: $h \circ (g \circ f) = (h \circ g) \circ f$.

Remark 6.1.2. There are many similar and mostly equivalent definitions of category in mathematics. The mostly fall into two main camps: how they treat their collection of morphisms. The two definitions are equivalent in the usual foundations of mathematics but each has their own advantages. In books such as

[14] a collection of morphisms is used. This approach lends itself more naturally to the notion of an *internal category* which will be an important concept later on. The other definition uses a family of collections of morphisms which lends itself to easily generalise to the notion of an *enriched category*, the definitive reference for which is [9].

The reason it cannot be swept under the rug so easily is because the issue of size is fundamental in category theory. Depending on what definition we chose, it will effect how we can talk about it. For an introduction to category theory, these ideas would mostly confuse the reader, hence we will simply point to [17] for a survey on how size issues are treated in category theory. From here on

We now give some examples:

Example 6.1.3. The **category of sets** denoted **Set** is the category whose objects are small[1] sets and morphisms are functions between sets. Composition is given by composition of functions. This is a very important category in category theory for reasons we shall come across later.

Choosing the direction in which our arrows point was arbitrary, but it does also mean that if we had chosen the other way we would also get a category. So every category we make canonically comes with a "friend".

Example 6.1.4. For any category $\mathcal{C}$, there is another category called the **opposite category** $\mathcal{C}^{\text{op}}$ whose objects are the same as $\mathcal{C}$ however the homsets are defined as follows: $\mathcal{C}^{\text{op}}(x, y) := \mathcal{C}(y, x)$. Composition is defined using the composition from the original category.

[NEEDS REWORDING] Size is a common issue in category theory with many similar ways of dealing with it. It can however cause much confusion and hoop-jumping to be correct. For our purposes we will safely ignore these issues. A formal treatment can be found in the appendix. [TODO: Add formal treatment of size].

Definition 6.1.5. We call a category **small** if its class of objects is really a set.

Definition 6.1.6. Let $\mathcal{C}, \mathcal{D}$ be categories. A **functor** F from $\mathcal{C}$ to $\mathcal{D}$ (written $F : C \to D$) consists of:

- An object $F(A) \in \mathcal{D}$, for all $A \in \mathcal{C}$ (also denoted FA).
- For each $A, B \in \mathcal{C}$, a function $F_{A,B} : \mathcal{C}(A, B) \to \mathcal{D}(FA, FB)$ (also denoted F).
- For each $A \in \mathcal{C}$, $F(1_A) = 1_{FA}$.
- For each $A, B, C \in \mathcal{C}$, $f : A \to B$, $g : B \to C$, we have

$$F(g \circ f) = F(g) \circ F(f)$$

[1] due to Russellian paradoxes we must distinguish between "all sets" and "enough sets". See appendix for details.

Remark 6.1.7. Historically in category theory, one would define covariant, as defined above, and contravariant functors, as a result this terminology has crept into uses of category in certain fields [REFERENCE pretty much any homological algebra book before 80s]. Contravariant functors mean to swap the order of composition when the functor is applied. In modern category theory texts, this is completely dropped as a contravariant functor from $\mathcal{C}$ to $\mathcal{D}$ is simply a covariant functor from $\mathcal{C}^{\text{op}}$ to $\mathcal{D}$. Henceforth, we shall not mention co(tra)variance of functors and refer to them simply a functors.

Remark 6.1.8. Given two functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ we can make a new functor $G \circ F$ called its **composite**, by first applying F then applying G on objects or morphisms. It is simple to check that this is indeed a functor.

Now that we have 'morphisms' between categories we can define another category:

Example 6.1.9. The category of small categories **Cat** has objects small categories and morphisms functors. Composition is given by composition of functors.

Definition 6.1.10 (Definition of natural transformation)**.**

Example 6.1.11. Given two categories $\mathcal{C}$ and $\mathcal{D}$ we can from a category $[\mathcal{C}, \mathcal{D}]$ called the functor category between $\mathcal{C}$ and $\mathcal{D}$. It's objects are functors $\mathcal{C} \to \mathcal{D}$ and morphisms are natural transformations between functors.

Special cases of this example include:

Example 6.1.12. A functor $\mathcal{C}^{\text{op}} \to \mathbf{Set}$ is typically called a **presheaf** in geometry and logic. They live in the functor category $[\mathcal{C}^{\text{op}}, \mathbf{Set}]$ which we will call the **category of presheaves**. This is an interesting construction as it acts like the category $\mathcal{C}$ in some ways with some nice properties from **Set**.

[CHECK THIS] One of the first theorems that is proven in category theory is the **Yoneda lemma**. It says if an object acts like a certain object in every possible way, then it must be isomorphic to that object. Akin to how particles are discovered in particle accelerators by observing how they interact when bombarded with different particles.

Lemma 6.1.13. Let $\mathcal{C}$ be a category. There is an embedding $\mathbf{y} : \mathcal{C} \to [\mathcal{C}^{\text{op}}, \mathbf{Set}]$. Where $\mathbf{y}(A) := \mathcal{C}(A, -)$, maps each object to its contravariant hom functor. Presheaves that arise this way are called **representable presheaves**.

Remark 6.1.14. ![WHAT IS A FULL AND FAITHFUL FUNCTOR?] An embedding is a functor that is full and faithful. We haven't actually proven that the "Yoneda embedding" is an embedding however this is a corollary of the Yoneda lemma which will prove now.

[PICTURES]

Theorem 6.1.15. Yoneda lemma Let $\mathcal{C}$ be a category. For all $X \in [\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$, there is a natural isomorphism between the following functors:

$$[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}](\mathbf{y}(-), X) \cong X(-)$$

Remark 6.1.16. The set of natural transformations between $\mathbf{y}(A)$ and a presheaf X is bijective to the sections of X at A.

7 Theories and models

[NOTE: This is a rough outline of what the document ought to look like, not even worthy of being a draft]

[TODO: Find references for these]

Definition 7.0.1. A theory asserts data and axioms. A model is a particular example of a theory.

For example a model of "the theory of groups" in the category of sets is simply a group. A model of "the theory of groups" in the category of topological spaces is a topological group. A model of "the theory of groups" in a the category of manifolds is a Lie group.

Categorical semantics is a general procedure to go from "a theory" to the notion of an internal object in some category.

The internal objects of interest is a model of the theory in a category.

Then anything we prove formally about the theory is true for all models of the theory in any category.

For each kind of "type theory" there is a corresponding kind of "structured category" in which we consider models.

- Lawvere theories $\leftrightarrow$ Category with finite products
- Simply typed lambda calculus $\leftrightarrow$ Cartesian closed category
- Dependent type theory $\leftrightarrow$ Locally CC category

A doctrine specifies: - A collection of type constructors - A categorical structure realising these constructors as operations.

Once we fix a doctrine $\mathbb{D}$, then a $\mathbb{D}$-theory specifies "generating" or "axiomatic" types and terms. A $\mathbb{D}$-category is one processing the specified structure. A model of a $\mathcal{D}$-theory T in a $\mathcal{D}$-category C realises the types and terms in T as objects and morph isms of C.

A finite-product theory is a type theory with unit and Cartesian product as the only type constructors. Plus any number of axioms.

Example:

The theory of magmas has one axiomatic type M, and axiomatic terms $\vdash e : M$ and $x : M, y : M \vdash xy : M$. For monoids and groups we will need equality axioms.

Let T be a finite-product theory, C a category with finite products

A model of T in C assigns:

1. To each type A in T, an object $[\![A]\!]$ in C
2. To each judgement derivable in T:

$$x_1 : A_1, \ldots, x_n : A_n \vdash b : B$$

   A morphism in C

$$[\![A_1]\!] \times \cdots \times [\![A_n]\!] \xrightarrow{[\![b]\!]} [\![B]\!]$$

3. Such that $[\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$ etc.

To define a model of T in C, it suffices to interpret the axioms, since they "freely generate" the model.

Talk about doctrines

Talk about type theory categories adjunction via syntactic category and complet category. (Syntax-semantics adjunction) Possible to set it up to be an equivalence but may not be needed.

WHY Categorical semantics:

1. Proving things in a D-theory means it is valid for models of that D-theory in all categories
2. We can use type theory to prove things about a category by working in its complete theory (internal language)
3. We can use category theory to prove things about a type theory by working in its syntactic category.

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] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

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

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

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

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

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

[9] M. Kelly. *Basic Concepts of Enriched Category Theory*. Lecture note series / London mathematical society. Cambridge University Press, 1982.

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

[11] Saunders Mac Lane. *Categories for the working mathematician*. Graduate texts in mathematics ; 5. Springer, New York, 2nd ed. edition, 1998.

[12] nLab authors. Initiality Project. `http://ncatlab.org/nlab/show/Initiality%20Project`, December 2018. http://ncatlab.org/nlab/revision/Initiality

[13] nLab authors. Initiality Project - Raw Syntax. `http://ncatlab.org/nlab/show/Initiality%20Project%20-%20Raw%20Syntax`, December 2018. http://ncatlab.org/nlab/revision/Initiality22.

[14] E. Riehl. *Category Theory in Context*. Aurora: Dover Modern Math Originals. Dover Publications, 2017.

[15] J.J. Rotman. *An Introduction to Homological Algebra*. Universitext. Springer New York, 2008.

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

[17] Michael A. Shulman. Set theory for category theory. *arXiv e-prints*, page arXiv:0810.1279, October 2008.

[18] 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.

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

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

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