Introduction to dependent type theory

Ali Caglayan

March 11, 2019

Contents

1 Introduction

The aim of this thesis is to introduce the notion of dependent types to an undergraduate reader. The main idea of dependent types is very simple, and could be understood by a small child. This is deceptively subtle however, since modelling such a formalism is quite tricky. This is evidenced by the fact that there is a lot of disagreement in type theory what has or hasn't really been proven. This however is a familar story in mathematics and is usually remidied by trying to understand what has been done better. Usually with the help of a new persepctive.

Dependent types however, are not only of interest to mathematicians but also programmers. Dependent type theory (much like simply typed lambda calculus) is very much a programming langauge allowing the expression of ideas previously too difficult to express. This is very much facilaiated by its deep connection to predicate logic.

- a[Begin with history and implications of curry howard]
- a[outline the "what they should do" of dependent types]
- a[start to rigoursly model syntax and talk about how bad a job most authors do]
- a[small section about classical inductive definitions]
- a[small section on why categorical semantics]
- a[model simply typed lambda calculus with categorical semantics]
- a[show natural extensions of the idea and why contexts break when dependnet]
- a[outline different approches to solving these problems]
- a[discuss Awodey's natural models]
- a[finally talk about future directions for type theory]
- a[maybe some mention on applications to programming (generalising various constructs, polymorphism, GA data types)]
- a[equality, inductive types, [[[[[maybe a tinsy bit of homotopy type theory]]]]]]

2 Curry-Howard correspondance

2.1 Mathematical logic

At the beginning of the 20th century, Whitehead and Russell pubished their *Principia Mathematica* [14], demonstrating to mathematicians of the time that formal logic could express much of mathematics. It served to popularise modern mathematical logic leading to many mathematicians taking a more serious look at topic such as the foundations of mathematics.

One of the most influencial mathematicians of the time was David Hilbert. Inspired by Whitehead and Russell's vision, Hilbert and his coleagues at Göttingen became leading researchers in formal logic. Hilbert proposed the *Entscheidungsproblem* (decision problem), that is, to develop an "effectually calculable procedure" to determine the truth or falsehood of any logical statement. At the 1930 Mathematical Congress in Königsberg, Hilbert affirmed his belief in the conjecture, concluding with his famous words "Wir müssen wissen, wir werden wissen" ("We must know, we will know"). At the very same conference, Kurt Gödel announced his proof that arithmetic is incomplete [6], not every statement in arithmetic can be proven.

This however did not deter logicians, who were still interested in understanding why the *Entscheidungsproblem* was undecidable, for this a formal efinition of "effectively calculable" was required. So along came three proposed definitions of what it meant to be "effectively calculable": *lambda calculus*, pusblished in 1936 by Alonzo Church [4]; *recursive functions*, proposed by Gödel in 1934 later published in 1936 by Stephen Kleene [10]; and finally *Turing machines* in 1937 by Alan Turing [19].

2.2 Lambda calculus

(Untyped) lambda calculus was discovered by Church at princeton, originally as a way to define notations for logical formulas. It is a remarkaly compact idea, with only three constructs: variables; lambda abstraction; and function application. It was realised at the time by Church and others that "There may, indeed, be other applications of the system than its use as a logic." [CITATION NEEDED][]. Church discovered a way of encoding numbers as terms of lambda calculus. From this addition and multiplication could be defined. Kleene later discovered how to define the predecessor function. [CITATION NEEDED] []. Church later rpoposed λ-definability as the definition of "effectively calculable", what is now known as Church's Thesis, and demonstrated that the problem of determining whether or not a given λ-term has a normal form is not λ-definable. This is now known as the Halting Problem.

2.3 Recursive functions

In 1933 Gödel arrived in Princeton, unconvinced by Church's claim that every effectively calculable function was λ-definable. Church responded by offering

that if Goödel would propose a different definition, then Church would "undertake to prove it was included in λ-definability". In a series of lectures at Princeton, Gödel proposed what came to be known as "general recursive functions" as his candidate for effective calculability. Kleene later published the definition [CITATION NEEDED][]. Church later outlined a proof [CITATION NEEDED][] and Kleene later published it in detail. This however did not have the intended effect on Gödel, whereby he then became convinced that his own definition was incorrect.

2.4 Turing machines

Alan Turing was at Camrbdige when he independently formulated his own idea of what it means to be "effectively calculable", now known today as Turing machines. He used it to show that the Entscheidungsproblem is undecidable, that is it cannot be proven to be true or false. Before publication, Turing's advisor Max Newman was worried since Church had published a solution, but since Turing's approach was sufficiently novel it was published anyway. Turing had added an appendix sketching the equivalence of λ-definability to Turing machines. It was Turings argument that later convinced Gödel that this was the correct notion of "effectively calculable".

2.5 Russells paradox

[Talk about the origin of types and stuff]

2.6 The problem with lambda calculus as a logic

Church's lambda calculus turned out to be inconsistent. [][CITATION NEEDED]. The reason was related to russels paradox, in that a predicate was allowed to act on itself. This led to an abandoning of the use of lambda calculus as a logic for a short time. In order to solve this Church adapted a solution similar to Russell's: use types. What was discovered is now known today as *simply-typed lambda calculus*. [] [CITATION NEEDED, 10 ?]. What is nice about Church's STLC is that every term has a normal form, or in the language of Turing machines every computation halts. [] [CITATION NEEDED] From this consistency of Church's STLC as a logic could be established.

2.7 Types to the rescue

[Talk in detail why typing is good for mathematicians, programmers and logicians]

2.8 The theory of proof a la Gentzen

[Go into the history of the theory of proof e.g. Gentzen's work; take notice of natural deduction]

2.9 Curry and Howard

[Curry makes an observation that Gentzens natural deduction corresponds to simply typed lambda calculus, Howard takes this further and defines it formally, eventually predicting a notion of dependent type.

2.10 Propositions as types

[Overview of the full nature of the observation, much deeper than a simple correspondance since logic is in some sense "very correct" and programming constructs corresponding to these must therefore also be "very correct".]

2.11 Predicates [CHANGE] as types?

[Talk about predicate quantifiers $\forall, \exists$ and what a "dependent type ought to do"]

2.12 Dependent types

[Perhaps expand on the simply typed section]
[talk about pi and sigma types
[talk about "dependent contexts"]

3 Dependent types

Here we will talk about simply typed lambda calculus, adding dependent types and

4 Syntax

4.1 Introduction

We will follow the structure of syntax outlined in Harper [7]. 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 [11], 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 [8], 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 [16] 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 [18] 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).

4.2 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 [9, p. 62], [3, Ch. 7]. It's use in the formal semantics of programming languages is not much

different either [20, Ch. 3]. There are however more constructive notions of well-foundedness [15, §8] with more careful use of excluded middle. We will follow [17], 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 4.2.1. 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 4.2.2. 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 4.2.3 (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 4.2.2 $Y = X$ 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.

4.3 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 4.3.1 (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 4.3.2 (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 4.3.3 (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 4.3.4 (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 4.3.5 (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 4.3.6 (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 4.3.7. 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 4.3.8 (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 4.3.9. 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 4.3.10. An ast can be thought of as a tree whose leaf nodes are variables and branch nodes are operators.

Example 4.3.11 (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:

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

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.

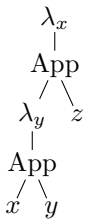


Figure 1: Vertically oriented tree representing the lambda term

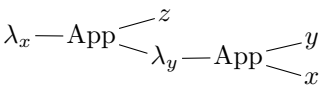


Figure 2: Horizontally oriented tree representing the lambda term

Remark 4.3.12. 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 4.3.13. 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 4.3.14. 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 4.3.15. 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 4.3.16 (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 4.3.17. 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 4.3.18. Note that 4.3.17 was simply about checking Definition 4.3.16. 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 [7] 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 4.3.19 (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 $\lhd$.

Remark 4.3.20. In [7] 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 4.3.21 (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 $j \lhd k$ then we say that the jth argument of o is in **scope** of the kth bound variable of o.

Remark 4.3.22. 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 4.3.23 (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 $\mathtt{G}$ 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 $X_j := \{v_k \in V \mid j \lhd k\}$ 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 4.3.24. There is a lot going on in the second constructor of Definition 4.3.23. 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.]]

5 Simply typed lambda calculus

5.1 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.

5.2 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 syntactic 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 [5, 8, 11]. 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 [8]. Firstly we introduce the notion of a *signature*. Similar accounts can be found in [5]. 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 5.2.1. A **signature** is a pair $(\mathbf{Type}, \mathcal{F})$ where **Type** is a finite set of **basic** (or **atomic**) **types**. And a functor $\mathcal{F} : \mathbf{Type}^\star \times \mathbf{Type} \to \mathbf{Set}$. Where $\mathbf{Type}^\star$ is the Kleene-Star operation on a set (or the free monoid over **Type**), defined as $X^\star := \bigcup_{n \in \mathbb{N}} X^n$ 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 $\Sigma := (\mathbf{Type}, \mathcal{F})$, denote $|\Sigma| := \mathbf{Type}$ 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 5.2.2. 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 5.2.3. A **variable declaration** is a pair $(x, \sigma) \in \mathbf{Var} \times \mathbf{Type}$ usually written as $x : \sigma$. This can be read as "the variable x has type σ. We will define $\mathbf{Dec} := \mathbf{Var} \times \mathbf{Type}$.

Definition 5.2.4. 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

6 Category theory

6.1 Introduction

6.2 Categories

We begin with the notion of a category. This can be thought of as a place mathematical objects live and interact with eachother.

Definition 6.2.1. A *category* $\mathcal{C}$ consists of:

- A set $\mathrm{Ob}(\mathcal{C})$, elements of which are called *objects* of $\mathcal{C}$.
- For each $X, Y \in \mathrm{Ob}(\mathcal{C})$, a set $\mathcal{C}(X, Y)$, called the *homset* from X to Y.
- For each $X, Y, Z \in \mathrm{Ob}(\mathcal{C})$, a function $\circ_{X,Y,Z} : \mathcal{C}(Y, Z) \times \mathcal{C}(XY) \to \mathcal{C}(X, Z)$, called the *composite function*.
- For each $X \in \mathrm{Ob}(\mathcal{C})$, an element $\iota_X \in \mathcal{C}(X, X)$, called the identity map, sometimes written $\iota_X : 1 \to \mathcal{C}(X, X)$.

Such that the following axioms hold:

- *Associativity*: For every $f \in \mathcal{C}(X, Y)$, $g \in \mathcal{C}(Y, Z)$ and $h \in \mathcal{C}(Z, W)$, one has $(h \circ g) \circ f = h \circ (g \circ f)$.
- Identity: For every $f \in \mathcal{C}(X, Y)$, one has $f \circ \iota_X = f = \iota_Y \circ f$.

Remark 6.2.2. One typically writes composition as juxtaposition and omits the symbol $\circ$.

Example 6.2.3. 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.

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

Choosing the direction in which our arrows point was rather arbitrary. This suggests that if we had chosen the other way we would have also gotten a category. So any category we can come up with has an associated dual.

Example 6.2.4. 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.

Remark 6.2.5. It is a simple exercise to check that for any category $\mathcal{C}$

[TODO: Note on commutative diagrams]

6.3 Functors

Definition 6.3.1. Given categories $\mathcal{C}$ and $\mathcal{D}$, a *functor* H from $\mathcal{C}$ to $\mathcal{D}$, written $H : \mathcal{C} \to \mathcal{D}$, consists of

- A function $\text{Ob}(H) : \text{Ob}(\mathcal{C}) \to \text{Ob}(\mathcal{D})$, with notation typically abbreivated to H
- For each $X, Y \in \text{Ob}(\mathcal{C})$ a function $H_{X,Y} : \mathcal{C}(X, Y) \to \mathcal{D}(HX, HY)$

Such that the following diagrams commute in the category **Set**:

- *H respects composition*:

$$\begin{array}{ccc} \mathcal{C}(Y,Z) \times \mathcal{C}(X,Y) & \xrightarrow{H \times H} & \mathcal{D}(HY,HZ) \times \mathcal{D}(HX,HY) \\ {\scriptstyle \circ}\downarrow & & \downarrow{\scriptstyle \circ} \\ \mathcal{C}(X,Z) & \xrightarrow{\quad H \quad} & \mathcal{D}(HX,HZ) \end{array}$$

- *H respects units*:

$$\begin{array}{ccc} 1 & \xrightarrow{\iota_X} & \mathcal{C}(X,X) \\ & {\scriptstyle \iota_{HX}} \searrow & \downarrow{\scriptstyle H} \\ & & \mathcal{D}(HX,HX) \end{array}$$

6.4 Natural transformations

Definition 6.4.1. Given categories $\mathcal{C}$ and $\mathcal{D}$ and functors $H, K : \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\alpha : H \to K$ consists of

For each $X \in \text{Ob}(\mathcal{C})$, a map $\alpha_X : HX \to KX$.

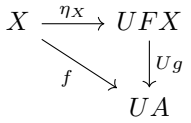
Such that for each map $f : X \to Y$ in $\mathcal{C}$, the following diagram commutes:

$$\begin{array}{ccc} HX & \xrightarrow{\alpha_X} & KX \\ {\scriptstyle Hf}\downarrow & & \downarrow{\scriptstyle Kf} \\ HY & \xrightarrow{\alpha_Y} & KY \end{array}$$

6.5 Having a left adjoint

This is arguably the most important definition we will study in category theory, we will see later on many concepts are special case of the following definition:

Definition 6.5.1. A functor $U : \mathcal{C} \to \mathcal{D}$ has a left adjoint if for all $X \in \mathcal{D}$, there exists an $FX \in C$ and $\eta_X : X \to UFX$ in $\mathcal{D}$ such that for all $A \in \mathcal{D}$ and forall $f : X \to UA$, exists a **unique** map $g : FX \to A$ such that the following diagram commutes:



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

[4] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

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

[6] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

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

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

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

[10] S.C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.

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

[12] nLab authors. Initiality Project. `http://ncatlab.org/nlab/show/Initiality%20Project`, December 2018. Revision 46.

[13] nLab authors. Initiality Project - Raw Syntax. `http://ncatlab.org/nlab/show/Initiality%20Project%20-%20Raw%20Syntax`, December 2018. Revision 22.

[14] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*, volume 1. Cambridge University Press, Cambridge, 1910.

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

[16] 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.

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

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

[19] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

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